UNIVERSIDAD NACIONAL AUTÓNOMA DE MEXICO

FACULTAD DE CIENCIAS

PROGRAMACIÓN EXTREMA Y TECNOLOGÍAS JAVA EN EL DESARROLLO DE UNA APLICACIÓN WEB

T E S I S

QUE PARA OBTENER EL TITULO DE:

LICENCIADAS EN CIENCIAS DE LA COMPUTACIÓN

P R E S E N T A N :

CHANDRA NICHDALI QUINTAS DE LA PARRA

EMILIA BARAJAS RAMÍREZ

DIRECTOR DE TESIS: LIC. EN C.C. FRANCISCO LORENZO SOLSONA CRUZ

2005

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

ACT. MAURICIO AGUILAR GONZÁLEZ
Jefe de la División de Estudios Profesionales de la
Facultad de Ciencias
Presente

Comunicamos a usted que hemos revisado el trabajo escrito:

" Programación Extrema y Tecnologías Java en el desarrollo de una aplicación Web".

realizado por Barajas Ramírez Emilia, Quintas de la Parra Chandra Nichdali

con número de cuenta 097138903
097196103 , quien cubrió los créditos de la carrera de:

Lic. en Ciencias de la Computación

Dicho trabajo cuenta con nuestro voto aprobatorio.

Atentamente

Director de Tesis Propietario	Lic. en C. C. Francisco Lorenzo Solsona Cruz
Propietario	M. en C. María Guadalupe Elena Ibargüengoitia González
Propietario	Dra. Hanna Oktaba
Suplente	M. en C. José de Jesús Galaviz Casas
Suplente	M. en C. Gustavo Arturo Márquez Flores

Consejo Departamental de Matemáticas

Dr. Francisco Hernández Quiroz
Coordinador de la Carrera de
Ciencias de la Computación.

FACULTAD DE CIENCIAS
CONSEJO DEPARTAMENTAL
DE
MATEMÁTICAS

AGRADECIMIENTOS DE CHANDRA

A mi mamita le agradezco que siempre me ha dado su amor, apoyo, respeto, y compañía. A ti que primero piensas en todos tu hijos antes que en ti misma, a ti que siempre has cuidado mi alimentación y mi salud, a ti que siempre me escuchas y me das consejos te dedico esta tesis y mi carrera pues es un producto que alcancé gracias a que siempre me dabas una palmadita en la espalda y alguna palabra de aliento. Le agradezco a la vida por haberme dado una madre como tú, estoy orgullosa de ti. Muchas gracias pues bien se que nos has amado incluso antes de conocernos. Recuerda que siempre contarás conmigo.

A mi hermanita Isis que me enseñó que significa ser una persona generosa y perseverante. Agradezco que en estos últimos años nos dimos la oportunidad de conocernos más a fondo y de apoyarnos mutuamente. Te agradezco que siempre estés para compartir mis penas y mis alegrías mas que como una hermana como una gran amiga. Te deseo lo mejor en esta vida y espero que siempre recuerdes que siempre estaré contigo. Gracias por ser mi hermana.

Ángel te agradezco por todos estos años de amor y comprensión pues bien sabes que eres el amor de mi vida, espero que sigamos nuestro camino juntos como lo hemos hecho hasta ahora. Te agradezco la comprensión que siempre mostraste para la culminación de mi carrera. Te amo.

Emilita te agradezco por toda la amistad y compromiso que siempre has dado, eres un gran ser humano. Gracias a ti concluyo este arduo camino ya que siempre conté contigo en todo momento y nos apoyamos hombro con hombro para llegar a este gran logro. Te agradezco por mostrarme que en esta vida todavía existen personas inteligentes, buenas y comprometidas. Recuerda que siempre contarás conmigo y que este es el principio de una nueva etapa pues siempre hemos de encontrar la manera de seguir cultivando esta hermosa amistad.

Gris te agradezco que hayas sido mi amiga todo este tiempo y que me hayas enseñado a ser mas relajada y reir un poco más. Espero que sigas creciendo en todos los sentidos. Muchas gracias.

A mi papá le agradezco todos sus consejos, pues si pude sobrevivir al ambiente duro de ciencias fue gracias a que él siempre me enseñó a ser fuerte y a no dejarme vencer ante las vicisitudes. Gracias por tu ejemplo de perseverancia.

A Inesita que siempre ha sido una gran amiga, que siempre me ha dado su apoyo, sus consejos y su atención. Espero que esta amistad perdure para toda la vida y que sigamos apoyándonos aunque sea en la distancia. La quiero mucho.

A Oli que nos ha dado su apoyo, que día a día nos da una palabra de aliento para seguir y terminar este proyecto. Gracias por permitir que nuestra relación alumna maestra se convirtiera en amistad, también le agradezco que me haya vuelto al buen camino del ejercicio pues gracias a sus buenos consejos cada día seguiré esforzándome más en mi salud. Muchas gracias.

A Francisco Solsona quien nos brindó su apoyo incondicional sin importar que casi no nos conocía. Te agradezco por confiar en mi y Emilia y gracias por darnos la oportunidad de conocerte, así como la libertad de expresar nuestras ideas. Gracias a ti este proyecto esta llegando a su fin con un muy buen sabor de boca. Muchísimas gracias.

A la Dra. Hanna Oktaba y a la M. en C. Lupita Ibargüengoitioa, por darnos las bases fundamentales de la Ingeniería de Software que como todos saben es el área a la cual me quiero dedicar. Gracias por transmitirme sus conocimientos a través de sus clases y sus consejos personales, así como de brindarme su amistad. Muchas gracias.

Al Dr.Arturo Olvera le agradezco por haberme impartido durante seis semestres ininterumpidos su cátedra excelente, gracias a usted aprendí a querer a las ecuaciones diferenciales. Muchas gracias.

Le agradezco a la UNAM por darme la oportunidad de conocer a tanta gente valiosa, así como de darme todas las herramientas para encontrar mi vocación. Por mi raza hablará el espíritu.

AGRADECIMIENTOS DE EMILIA

A mi familia que siempre me dió su apoyo incondicional, especialmente en los momentos difíciles de la carrera.

A mi mamá que siempre con su fortaleza y con su ejemplo me mostró que uno siempre debe intentarlo. Ella me inspiró a buscar ser alguien en la vida, a no dejar que nada ni nadie me impida conseguir mis metas, a que no debemos conformarnos, siempre debemos buscar lo que queremos sin importar quien está al frente, puesto que no hay nada que no pueda solucionarse. Su dedicación me inspira a seguir adelante y ser mejor cada día. Esta carrera la empecé y terminé por ambas, puesto que gracias al esfuerzo que hemos realizado, juntas hemos podido terminar una carrera (aunque en el título solo aparezca una).

A mi papá que me ha incitado a seguir adelante, a ver siempre todo con optimismo, dejando a un lado el sentido realista y dar paso a los sueños. Él me ha enseñado que uno debe ir a donde sea con alegría y con la idea de tener un buen día sin importar a donde vayamos. Siempre me ha ayudado a relajarme y a olvidar los momentos de presión en la carrera para seguir disfrutando cada momento de mi vida, puesto que cada fin de semana especialmente, nos deleita con su presencia para disfrutar una película o un partido de futbol solo con el afán de estar con nosotros.

A mis hermanos que siempre me aconsejaron y apoyaron con pequeñas y grandes cosas en la carrera y en la vida, pues siempre me enseñaron que debemos tener un equilibrio, no todo es trabajar, ni todo es flojear. Jesús me ha demostrado con su ejemplo que podemos obtener grandes cosas gracias a nuestros propios méritos, mientras que Ángel me ha enseñado la importancia de la lectura para aprender cosas nuevas e interesantes, y tener así una mente más abierta y sobre todo más enterada de lo que sucede alrededor de nosotros.

A mis amigas que me apoyaron en lo que ha sido el reto más difícil de mi vida. A Chandra especialmente porque me enseñó que siempre debemos ser fuertes, a no dejarnos de nadie, puesto que todo es posible con esfuerzo, dedicación y sacrificio. Ella llegó en el momento justo a mi vida, cuando necesitaba no solo una compañera que me apoyara en la carrera, sino más bien una amiga que comprendiera el verdadero significado de la amistad. También le agradezco que me mostrará ese lado tan grandioso que tiene y que a veces los demás no aprecian o no ven, y por el cual es la mejor amiga que he tenido y tendré en mi vida. Esta tesis esta inspirada en ella puesto que siempre me enseñó con el ejemplo que debemos dar el máximo en todo lo que realicemos, puesto que nuestras acciones y nuestro trabajo reflejan quienes somos, además de que me motivó a esforzarme cada día más para satisfacer cada una de nuestras expectativas. A Isis y Aida porque me han dado su amistad y su apoyo sincero. Y a Griselda que me enseñó la gran importancia de ser una persona risueña y relajada, comprometida y dedicada a todas las personas que realmente valen la pena, además de ser mi amiga todos estos años.

A mis sinodales y a mi asesor por sus múltiples enseñanzas y apoyos a lo largo de la carrera y durante el desarrollo de esta tesis. A la Dra. Hanna y a la M. en C. Guadalupe que siempre me enseñaron la importancia de hacer las cosas con el máximo esfuerzo para obtener grandes recompensas, además del importante valor que tiene el aprender a trabajar en equipo, pero sobre todo por la amistad que nos han brindado a Chandra y a mí. A Francisco Solsona por darnos a Chandra y a mí el apoyo incondicional y los estímulos necesarios para desarrollar y terminar esta tesis.

A Papel, Petit, Sam y Mily que me apoyaron con su compañía, sus muestras de afecto, la protección que me dieron respecto a los intrusos que se querían meter a mi casa, además de hacerme pasar momentos muy bellos. Ellos son el más claro ejemplo de como alguien puede dar una amistad incondicional.

ÍNDICE GENERAL

INTRODUCCIÓN

Actualmente la humanidad se encuentra en el inicio de una nueva época, en la cual la tecnología se ha vuelto parte de nuestra vida diaria, en la que las computadoras juegan un papel muy importante para llevar a cabo la automatización y control de procesos tanto para los individuos como para las instituciones.

La Facultad de Ciencias, por su naturaleza, no puede ni debe permanecer ajena a la revolución informática que estamos viviendo, por esta razón se ha creado el proyecto *XFC*.

El proyecto XFC es un proyecto que surgió en diciembre del 2001, el cual pretende introducir el uso de tecnologías de información integrando los distintos sistemas ya existentes y desarrollando los que no se han realizado con el propósito de automatizar y facilitar los distintos flujos de información y tareas desarrolladas en la facultad por los académicos, estudiantes y trabajadores.

Con estas ideas en mente, la presente tesis pretende cumplir los siguientes objetivos:

- Dar una visión general de las tecnologías actuales para la realización de aplicaciones web.
- Ejemplificar el desarrollo de una aplicación web utilizando la programación extrema.
- Unirnos al proyecto XFC a fin de proporcionar una aplicación web para la Facultad de Ciencias de la UNAM mediante la cual, los profesores podrán tener un inventario personal de sus pertenencias.

Nuestro trabajo consta de tres partes:

Primera parte: se enfoca a la descripción de herramientas y tecnologías con las que actualmente contamos para el desarrollo de una aplicación web. Los capítulos que conforman esta parte son:

- Capítulo 1. Programación Extrema: Nos da una breve descripción de las prácticas y valores que sigue esta metodología, así como los elementos necesarios para aplicarla.
- Capítulo 2. Lenguaje de Modelado Unificado: Aquí se explican las características de este lenguaje así como los diagramas más representativos con que cuenta.
- Capítulo 3. Lenguaje Extendible de Marcado: Se describe la estructura de un documento XML, que es una DTD, que significa que un documento esté bien formado.
- Capítulo 4. Java 2 Enterprise Edition: Se explican los componentes de esta plataforma *Aplicaciones Cliente*, *Applets*, *Componentes Web* y *Componentes del servidor*, así como las *APIs* proporcionadas.
- Capítulo 5. Servlets: Básicamente describe la estructura general de un servlet, su ciclo de vida y el contenedor que controla a los servlets.
- Capítulo 6. JavaServer Pages: Nos habla de las características de la tecnología, el procesamiento de un JSP y los elementos de éste.
- Capítulo 7. Java Database Connectivity: Se describen algunos comandos de SQL soportados, la arquitectectura de JDBC, utilización de JDBC, etc.
- Capítulo 8. Tapestry: Explica los conceptos de páginas y componentes, las principales clases de Tapestry y OGNL.
- Capítulo 9. Hibernate: Se muestra la problemática entre el modelo relacional y el modelo orientado a objetos, el mapeo entre estos dos modelos, así como el uso de Hibernate y HQL.

Segunda Parte: Se aborda el desarrollo de la aplicación web para el inventario personal de los profesores de la Facultad de Ciencias mediante la aplicación de las prácticas de la programación extrema y diagramas UML. Para llevar a cabo esta parte se incluyen los capitulos:

- Capítulo 10. Lanzamiento: Da una breve introducción sobre el sistema, el esquema de trabajo a seguir y como se documentarán cada una de las historias.
- Capítulo 11. Iteración 1 -Usuarios-: Explica el proceso de desarrollo de las historias relacionadas al manejo de los *usuarios*.

- Capítulo 12. Iteración 2 -Categorías-: Explica el proceso de desarrollo de las historias relacionadas al manejo de las *categorías* de pertenencia.
- Capítulo 13. Iteración 3 -Pertenencias-: Explica el proceso de desarrollo de las historias relacionadas al manejo de las *pertenencias*.
- Capítulo 14. Iteración 4 -Solicitantes-: Explica el proceso de desarrollo de las historias relacionadas al manejo de los *solicitantes*.
- Capítulo 15. Iteración 5 -Préstamos-: Explica el proceso de desarrollo de las historias relacionadas al manejo de los *préstamos*.

Tercera Parte: Conclusiones.

Anexos: Están conformados por todas las clases, páginas, componentes, archivos de configuración, entre otras cosas que componen a la aplicación Web.

Parte I

Tecnologías, Metodologías y Herramientas

CAPÍTULO 1

PROGRAMACIÓN EXTREMA

1.1. INTRODUCCIÓN

La Programación Extrema (Extreme Programming, XP) fue creada a principios de los noventas por Kent Beck, la cual se define como una metodología ágil para pequeños y medianos equipos de desarrollo de software, y se enfoca a todos aquellos proyectos donde los requerimientos no se encuentran bien establecidos o constantemente sufren cambios. La forma en que se lleva a cabo esta metodología es mediante un conjunto de reglas y prácticas que se realizan en ciclos cortos de desarrollo, los cuales involucran todas las etapas del ciclo de desarrollo tradicional de sistemas (análisis, diseño, desarrollo, pruebas e implantación).

1.2. DESCRIPCIÓN GENERAL DE LA PROGRAMACIÓN EXTREMA

Un proyecto en XP está basado en cuatro valores fundamentales, los cuales se alcanzan mediante la realización de doce prácticas. En las siguientes secciones daremos una breve descripción de estos temas.

1.2.1. VALORES

Comunicación (Communication) En XP se propone que exista una buena comunicación entre los programadores y el cliente, así como entre ellos mismos, para así evitar que se susciten malos entendidos entre cualquiera de las partes y se llegue a un diseño comprensible para ambas partes.

Simplicidad (Simplicity) Este principio se enfoca en la búsqueda de soluciones más simples de diseñar e implementar, pues es más fácil hacer un pequeño

cambio mañana si se necesita, que hacer algo mucho más complicado y que tal vez nunca se use.

Retroalimentación (Feedback) Sugiere presentar de manera frecuente prototipos del sistema al cliente. Esto asegura una verdadera comunicación con el cliente, y un mejor entendimiento de los requerimientos.

Valentía (Courage) Representa el valor para mantener proyectos sencillos, someter el producto a continuas evaluaciones o realizar constantes modificaciones.

1.2.2. PRÁCTICAS

El juego de la planeación (The Planning game) consiste en delimitar los alcances de cada ciclo de desarrollo (conocido como iteración).

En esta práctica interactúan dos equipos independientes: el **equipo de negocios**, el cual está conformado por miembros de la organización que requieren el software, y el **equipo de desarrollo de software**, el cual consiste en todas aquellas personas encargadas de construir el software.

El papel del equipo de negocios es el de establecer que características debe poseer el software que requiere la empresa, además de determinar que funcionalidades se requieren en cada iteración según su importancia y las fechas de entrega de éstas.

Por otro lado, el equipo de desarrollo realiza las estimaciones de tiempo de desarrollo de cada funcionalidad en base a las fechas de entrega establecidas, así como su propia organización, llevando una agenda detallada que determina las prioridades de cada funcionalidad en una iteración en base a estimaciones de tiempo o riesgo.

Pequeñas entregas (Small releases) Los proyectos en XP se llevan a cabo a través de ciclos cortos de desarrollo que contengan las funcionalidades más valiosas, realizándose en las etapas de análisis, diseño, desarrollo, pruebas e implantación. Al final de cada iteración se deberá entregar un avance al cliente para que así se disminuya la incertidumbre y aumente la comunicación con el cliente y permitiendo que éste pueda proporcionar sugerencias de mejora del sistema sobre la marcha.

Metáfora (Metaphor) es la elaboración de una idea general del sistema que prescinda de tecnicismos innecesarios y que permita entender los elementos básicos del sistema y sus relaciones de manera simple, tanto a los involucrados en el proyecto como a las personas ajenas a éste. En otras palabras, es como escribir una historia para definir, estimar y encargar a algún miembro del equipo el desarrollo de un problema, así como presentar una solución y entonces reescribir la historia para ir detallando más los problemas y sus soluciones.

Diseño simple (Simple design) Esta práctica nos dice que un sistema debe contener en todo momento solo aquellas funcionalidades que resultan necesarias y que han sido añadidas conforme las condiciones que el entorno ha requerido, dado que el costo del cambio es mayor en sistemas que cuentan con funcionalidad extra que en aquellos dotados de lo estrictamente necesario.

Pruebas (Testing) Esta práctica recomienda el uso exhaustivo de pruebas para la verificación y detección de errores en el sistema. XP recomienda los siguientes tipos de prueba:

- Pruebas de caja negra, donde importa el aspecto funcional del sistema.
- Pruebas de caja blanca, donde se revisa el funcionamiento interno del software.
- Pruebas automatizadas, las cuales consisten en la codificación de pruebas en un software especial que deben ser superadas por el sistema.
- Pruebas contenidas en el propio sistema, como son los mensajes escritos por los programadores.
- Pruebas de integración, donde se valida que no existan contradicciones entre los componentes escritos por diversos miembros del equipo de desarrollo.
- Pruebas de usuario, las cuales se llevan a cabo al final de cada ciclo para saber si se requieren cambios.

Refabricación (Refactoring) Esta práctica busca la eliminación de funciones obsoletas, adición de comentarios, reescritura de código redundante, enriquecimiento de funciones para uso general o cambio de nomenclatura fuera del estándar, entre otras mejoras, permitiendo así que los desarrolladores puedan responder rápidamente a un cambio en los requerimientos del cliente o en la tecnología.

Programación por pares (Pair programming) Es la práctica en la que dos programadores se sientan a trabajar con una sola computadora para participar

en el desarrollo de un programa. La pareja está compuesta por una persona que escribe el código, mientras que el otro está revisando y pensando en pruebas posibles de lo que el otro escribe.

La proximidad muy cercana y el enfoque de una unidad en la programación promueve el que se compartan ideas y que se realice un mejor diseño, además de que el código resultante está más libre de defectos y toma menos tiempo que el que utiliza un solo programador.

XP demuestra que los programadores no solo trabajan bien en pares, si no que también aprenden mucho mejor a partir de discutir y diseñar el código con otra persona.

Propiedad Colectiva (Collective code ownership) Esta práctica establece que la fase de desarrollo del proyecto es responsabilidad y propiedad de todos los miembros del equipo técnico, por lo cual se permite que cualquier pareja manipule o modifique el código.

Integración continua (Continuos integration) Es el concepto de integrar nuevo código en el código existente y entonces realizar las pruebas que se definieron anteriormente. Esta integración debe realizarse como mínimo una vez al día.

40 horas semanales (40-hour week) La programación extrema propone semanas laborales de 40 horas como estrategia para lograr un buen ritmo de desarrollo y un equipo de trabajo lúcido, motivado y contento. En ocasiones se permite la ruptura de esta práctica debido a factores externos, aunque más de dos semanas consecutivas con horas extra se considera un signo negativo en el proyecto y un motivo para una reunión de replaneación.

Cliente en el sitio de desarrollo (On-site customer) Esta práctica propone que al menos una de las personas que utilizarán el producto acompañe constantemente al equipo de desarrollo para resolver las dudas que surgirán conforme avance el proyecto, así como para aportar ideas y sugerencias para el mejoramiento del sistema.

En caso de que resulte imposible para el cliente, se recomienda que al menos se faciliten medios para que los programadores puedan consultar sus dudas con él en todo momento.

Estándares de codificación (Coding standards) La meta de esta práctica es la de implantar un estándar de codificación que sea aceptado, lo cual permitirá que la comunicación y el entendimiento mejoren sustancialmente.

1.3. APLICACIÓN DE LA PROGRAMACIÓN EXTREMA

En esta sección nos enfocaremos a la descripción de como aplicar la programación extrema en el desarrollo de un proyecto concreto.

1.3.1. PLANEACIÓN DEL PROYECTO

Lo más importante de la planeación es realizar reuniones de entregas en las que se realiza un plan que describe las entregas o iteraciones individuales, cuyas características principales son:

- Lo primero que hay que hacer es identificar y documentar las historias (user story) o requerimientos de los usuarios, las cuales son la descripción de las funcionalidades o tareas que se desea que el sistema realice. Se recomienda que estas historias sean escritas en lenguaje natural y en tarjetas para así poderlas consultar cuando se necesite.

- Posteriormente, el equipo de desarrollo asignará un tiempo de desarrollo en semanas para cada una de las historias pudiéndolas dividir o agrupar si así se requiere.

- A continuación, el equipo de desarrollo ordenará las historias por el grado de importancia de la siguiente manera:

  - Historias esenciales en el sistema.
  - Historias que agregan valor.
  - Historias que agregan funcionalidad.

  o bien, éstas pueden ser ordenadas de acuerdo a su riesgo técnico:

  - Historias con estimación precisa.
  - Historias con cierta certeza de estimación.
  - Historias sin estimación.

- El equipo de desarrollo le informará al cliente sobre el tiempo estimado de desarrollo, para que éste pueda elegir las historias de la siguiente iteración.

El conjunto de historias elegido constituye una iteración, cuya duración se recomienda sea de entre 1 y 4 semanas, ya que con ciclos cortos el número de iteraciones resulta grande, favoreciendo una mayor retroalimentación con el cliente proporcionándole certidumbre sobre el avance del proyecto.

1.3.2. ITERACIONES

Planeación de la iteración

Una vez conformada la iteración, el equipo de desarrollo procederá a dividir las historias en tareas o actividades de programación. El manejo de estas tareas está dado por tres fases que describiremos a continuación:

- **Fase de Exploración**
  - *Escribir las tareas en tarjetas.* Tomar las historias de una iteración y convertirlas en tareas. Las tareas pueden ser más pequeñas que una historia o contener a muchas historias, además de que una tarea no necesariamente está ligada a una historia.
  - *Dividir o combinar las tareas.* Si se descubre que una tarea tomará muchos días, se divide en tareas más pequeñas o si es muy básica, se deberá combinar con otras para formar una tarea más grande.
- **Fase de Compromiso**
  - *Aceptar una tarea.* Un programador acepta la responsabilidad de una tarea.
  - *Estimar una tarea.* El programador responsable de la tarea estimará el tiempo para implementarla, incluyendo el tiempo que puede ocupar en las pláticas con el cliente o con sus compañeros de trabajo.
- **Fase de Manejo**
  - *Implementar una tarea.* Un programador toma una tarjeta de una tarea, encuentra un compañero, escriben los casos de prueba, hacen todo el trabajo juntos e integran el nuevo código.
  - *Registrar el progreso.* Cada dos o tres días, uno de los miembros del equipo preguntará a cada programador cuánto tiempo utilizaron para cada una de sus tareas y si aún no la han terminado, cuánto tiempo les falta.

- *Verificar la historia.* Tan pronto como las pruebas funcionales estén listas y las tareas para una historia estén completadas, las pruebas funcionales deben ser ejecutadas para verificar que la historia funciona.

Estrategia de Diseño

La estrategia de diseño que se propone en XP se basa en 4 pasos fundamentales:

1. Empezar con una prueba, para así saber cuando se ha terminado. Basta con hacer un pequeño diseño para escribir la prueba, identificando cuales son los objetos y sus métodos más evidentes.
2. Diseña e implementa solo lo necesario para hacer que esa prueba corra.Se diseña lo suficiente de la implementación para hacer que esta prueba y todas las anteriores funcionen.
3. Repetir.
4. Si a menudo se ve la oportunidad de hacer el diseño más simple, hacerlo.

Una de las metas que busca la programación extrema es la simplicidad en todos los aspectos del desarrollo de software. En el caso de la estrategia de diseño, se dice que el mejor diseño es el diseño más simple que ejecuta todos los casos de prueba, pero ¿qué se entiende en XP por "un sistema simple"? Un sistema simple en XP es aquel que cumple con las siguientes características:

1. El sistema (código y pruebas) debe comunicar todo lo que uno quiere que comunique.
2. El sistema no debe contener código duplicado.
3. El sistema debe tener el menor número de clases posibles.
4. El sistema debe tener el menor número de métodos posibles.

Pruebas

XP propone el sometimiento intensivo del sistema a pruebas por parte de todas las personas involucradas en el desarrollo del mismo, además de contar con un probador de tiempo completo que evalúe las pruebas existentes, proponga nuevos métodos y traduzca las inquietudes de los usuarios en pruebas palpables.

Los programadores escriben pruebas método por método (pruebas unitarias). Un programador escribe una prueba bajo las siguientes circunstancias:

- Si la interfaz para un método no es del todo clara, debe escribir una prueba antes de escribir el método.
- Si la interfaz es clara, pero se imagina que la implementación será un poco más complicada, debes escribir una prueba antes de escribir el método.
- Si crees que una circunstancia es inusual en la cual el código debe trabajar como es escrito, debes escribir una prueba para comunicar la circunstancia.
- Si te encuentras un problema después, debes escribir una prueba que aisle el problema.
- Si estás a punto de reconstruir un código, y no estás seguro de como se supone que debe comportarse, y no hay aún una prueba para el aspecto del comportamiento en cuestión, debes escribir una prueba primero.

Los clientes escriben pruebas de tipo historia por historia (pruebas funcionales), donde se prueban las funcionalidades del sistema en general.

Con el uso de pruebas intensivas se incrementa la confianza y la velocidad de desarrollo, y el número de errores desciende con el tiempo debido a que las pruebas pueden ser reutilizadas y el cliente puede probar y manipular el producto a su gusto.

Desarrollo

Una vez estimadas y repartidas las diferentes tareas entre los miembros del equipo de desarrollo, se comienza a desarrollar mediante equipos de dos personas (programación por pares).

El desarrollo de cada tarea incluye la codificación específica para la tarea, el diseño de pruebas, la refabricación del código que así lo requiera y la integración de la tarea al proyecto en conjunto.

En XP se propone la integración del código que se vaya generando al menos una vez al día, para de esta forma evitar enormes integraciones futuras, divergencias de programación o fragmentación de código.

XP recomienda para el equipo de desarrollo la realización de reuniones diarias para discutir los problemas que vayan surgiendo, evaluar los avances del día anterior y formar equipos de programación.

Si el equipo observa que el tiempo designado a la iteración es insuficiente, puede solicitar al cliente una reducción en el alcance de algunos requerimientos o la eliminación de tareas innecesarias.

De la misma manera, el cliente puede añadir un requerimiento de mayor valor no contemplado en esta iteración, sustituyéndolo por requerimientos de la actual iteración con tiempos de desarrollo equivalentes al nuevo requerimiento a incluir.

Una vez que las tareas que componen una historia son terminadas e integradas al proyecto, el cliente recibe una nueva versión del sistema que será sometido a pruebas por parte de los usuarios finales para detectar errores no previstos en la fase de desarollo.

Reinicio del ciclo

Concluido el plazo de la iteración y realizados todos los requerimientos de usuario planeados, el equipo de negocios y el equipo de desarrollo se encuentran de nuevo en situación de reunirse, y discutir la próxima iteración, cuyo contenido y duración dependerá en gran medida de la iteración anterior, así como de las circunstancias actuales del negocio que quizá orienten el desarrollo en otra dirección.

Generalmente, una iteración típica incluirá tanto nuevos requerimientos de usuario como requerimientos ya construidos que requieran corrección (cuando no satisfacen las pruebas de los usuarios finales), mejoramiento o adición de funcionalidades.

CAPÍTULO 2

LENGUAJE DE MODELADO UNIFICADO

2.1. INTRODUCCIÓN

Lenguaje de Modelado Unificado (Unified Modeling Language - UML) es un lenguaje de modelado visual orientado a objetos de propósito general que se usa para especificar, visualizar, construir y documentar productos de un sistema de software. Se usa para entender, diseñar, configurar, mantener y controlar la información sobre los sistemas a construir.

2.2. OBJETIVOS DEL UML

- UML es un lenguaje de modelado de propósito general que pueden usar todos los modeladores.
- Mantener la capacidad de modelar toda la gama de sistemas que se necesita construir. UML necesita ser lo suficientemente expresivo para manejar todos los conceptos que se originan en un sistema moderno, tales como la concurrencia y distribución, así 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.
- Es el estándar mundial de la OMG.

2.3. ORIENTACIÓN A OBJETOS

Como se mencionó en la introducción, UML es un lenguaje orientado a objetos, por lo cual en esta sección nos enfocaremos a definir algunos conceptos importantes en lo que se refiere a la orientación a objetos.

2.3.1. OBJETOS

Un objeto es una entidad definida por un conjunto de atributos comunes y los servicios u operaciones asociados. Puede tener un estado que se almacena en una pieza encapsulada de software.

La representación abstracta del objeto en la computadora es una imagen simplificada del objeto real.

Características principales de un objeto

Un objeto posee tres características principales: *estado, comportamiento e identidad.*

- El *estado* se define como el conjunto de propiedades del objeto, donde una propiedad representa una característica específica del mismo. Los objetos poseen propiedades variables y constantes, por lo cual el estado de un objeto será igual al valor de sus diferentes propiedades en un momento específico, lo cual hace que el estado de un objeto sea una condición que varía dinámicamente con el tiempo.

- El *comportamiento* de un objeto describe las acciones y reacciones del mismo, es decir, el conjunto de tareas o acciones realizadas por el objeto o sobre el objeto. Las operaciones se desencadenan por estímulos externos (mensajes) enviados por otros objetos.

- La *identidad* de un objeto representa una característica intrínseca del objeto que permite identificarlo sin ambigüedades, es decir, permite diferenciar dos objetos con estados idénticos.

2.3.2. ENCAPSULAMIENTO

El encapsulamiento se conoce como el empaque de un conjunto de datos y métodos, para que así el objeto pueda ocultar sus datos de los demás objetos.

El encapsulamiento evita la corrupción de los datos de un objeto. Si todos los programas pudieran tener acceso a los datos de cualquier forma que quisieran los usuarios, los datos se podrían corromper o utilizar de mala manera. El encapsulamiento protege los datos del uso arbitrario y no pretendido.

El encapsulamiento oculta los detalles de su implementación interna a los usuarios de un objeto. Los usuarios se dan cuenta de las operaciones que puede solicitar del objeto, pero desconocen los detalles de cómo se lleva a cabo la operación. Todos los detalles específicos de los datos del objeto y la codificación de sus operaciones están fuera del alcance del usuario.

2.3.3. MENSAJES

Un mensaje es la especificación de la transmisión de información entre objetos y representa la unidad fundamental de comunicación entre ellos. Un mensaje debe contener algunos de los siguientes datos:

- El nombre del servicio requerido por algún objeto que llama.
- Copias de la información requerida para ejecutar el servicio y el nombre de quien tiene esta información.

Existen varios tipos de mensajes entre los cuales se encuentran los siguientes:

Llamado: Invoca un método en el objeto receptor.

Retorno: Regresa un valor al emisor.

Envío: Envía un valor al receptor.

Creación: Crea un objeto.

Destrucción: Destruye un objeto.

2.3.4. CLASES

El término de clase se refiere a la implementación de un tipo de objeto. El tipo de objeto es una noción de concepto. Especifica una familia de objetos sin estipular la forma en que se implementen. Así, una clase es una implementación de un tipo de objeto. Especifica una estructura de datos y los métodos operativos permisibles que se aplican a cada uno de sus objetos.

2.3.5. HERENCIA

Existen diversas maneras de clasificar elementos. Dentro del enfoque orientado a objetos la forma más común de representar la clasificación es por medio de una técnica conocida como herencia, la cual permite construir una clase a partir de otras ya existentes.

El producto de este proceso (subclase) contiene tanto atributos generales heredados de sus clases antecesoras (superclases) como características propias específicas, siendo un mecanismo sencillo para compartir atributos y métodos entre clases.

Por ejemplo, el tipo de objeto persona puede tener subclases estudiante y empleado que contienen todas las propiedades de una persona, pero además tienen datos muy particulares de cada uno de ellos.

2.3.6. POLIMORFISMO

El polimorfismo consiste en dar el mismo nombre a servicios implementados en diferentes objetos, estos servicios pueden implementarse de forma diferente pero producirán el mismo tipo de resultados. Se utiliza el polimorfismo cuando se quiere enviar el mismo mensaje a diferentes objetos sin saber el objeto específico al que se le está enviando el mensaje. Existen diversas formas de polimorfismo:

- *Ad-Hoc*: Es la sobrecarga de operadores, donde una función es llamada basándose en su firma definida como la lista de tipos de argumentos en su lista de parámetros.

- *Puro*: Una clase es subclase de otra. Las funciones disponibles en la superclase pueden usarse en la subclase. Se puede tener distinta implementación de la función en la subclase. La función a utilizar se determina dinámicamente en tiempo de ejecución. Esto se conoce como ligadura dinámica.

2.4. DIAGRAMAS 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 software debe ofrecer un conjunto de modelos que permitan expresar el producto desde cada una de las perspectivas de interés. El código fuente del sistema es el modelo más detallado del sistema (y además ejecutable).

Un diagrama es una representación gráfica de una colección de elementos de modelado, no es un elemento semántico, un diagrama muestra representaciones de elementos semánticos del modelo, pero su significado no se ve afectado por la forma en que son representados.

Estos diagramas proporcionan múltiples perspectivas del sistema bajo análisis. El modelo subyacente integra estas perspectivas de forma que se puede construir un sistema autoconsistente. Estos diagramas, junto con la documentación de soporte, es lo primero que ve el diseñador.

Por otro lado podemos ver el modelo de una forma estática o de una forma dinámica. Estas perspectivas nos dan la siguiente clasificación:

- Modelo estático (estructural):
  - Diagrama de clases.
  - Diagrama de paquetes.
  - Diagrama de objetos.
  - Diagrama de componentes.
  - Diagrama de instalación.
- Modelo dinámico (comportamiento):
  - Diagrama de casos de uso.
  - Diagrama de colaboración.
  - Diagrama de estados.
  - Diagrama de actividades.
  - Diagrama de secuencia.

2.4.1. DIAGRAMA DE CLASES

Los *diagramas de clases* describen la estructura y el contenido estático de un sistema usando elementos tales como las clases, los paquetes y los objetos para ilustrar las relaciones. Estos diagramas representan una abstracción de entidades que componen al sistema que incluye definiciones para atributos y operaciones. Entre los mecanismos de abstracción que encontramos para representar las relaciones entre los objetos están:

1. Clasificación o Instanciación.
2. Composición o Descomposición.
3. Agrupación o Individualización.
4. Especialización o Generalización.

Para comprender mejor la notación de UML, a continuación se expondrán cada uno de los elementos que componen a los diagramas de clases:

Clase: Cada clase es representada con un rectángulo dividido en tres compartimentos (ver Figura 2.1):

- Nombre de la clase.
- Atributos de la clase.
- Operaciones de la clase.

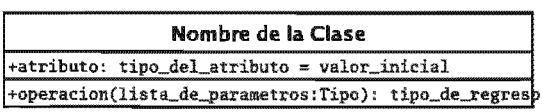

Figura 2.1: Diagrama de una clase

Visibilidad: Para restringir los accesos a los atributos y las operaciones de una clase, se usan los marcadores de *visibilidad*. Existen tres tipos de visibilidad (ver Figura 2.2):

- *Private*: Los atributos y operaciones marcados con este tipo de visibilidad estarán ocultos para las otras clases.
- *Public*: Los atributos y operaciones marcados con este tipo de visibilidad serán accesibles para las otras clases.

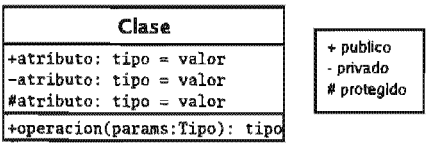

Figura 2.2: Visibilidad de una clase

- *Protected*: Los atributos y operaciones marcados con este tipo de visibilidad serán visibles solo para las clases derivadas de ésta.

Asociación: Representa las relaciones estáticas entre clases. Una *asociación* es una abstracción de la relación existente en los enlaces entre los objetos (ver Figura 2.3).

La notación para representar la asociación es la siguiente:

- Se traza una línea que una a las dos clases a asociar.
- El nombre de la asociación deberá escribirse arriba (o abajo) de la línea.
- Usa una flecha (de color negro) para indicar la dirección de la relación.
- Indica el rol cerca del final de una asociación. Un rol representa la manera en que cada clase verá a la otra.

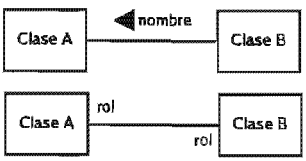

Figura 2.3: Asociación de las clases

Multiplicidad: Representa la cardinalidad de la relación. Para representarla, se usa la siguiente notación (ver Figura 2.4):

- 1 (*no más de uno*).
- 0..1 (*cero o uno*).
- * (*muchos*).

- 0..* (*cero o muchos*).
- 1..* (*uno o muchos*).

Estos símbolos indican el número de instancias de una clase ligada a otra instancia de otra clase. Por ejemplo, una compañía tendrá uno o más empleados, pero cada empleado trabaja solo para una compañía.

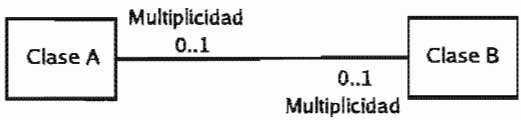

Figura 2.4: Multiplicidad

Agregación: Representa un tipo especial de la asociación, en la cual dos clases se encuentran relacionadas. Este tipo de relación se representa visualmente mediante una línea de asociación con un pequeño *diamante blanco* junto a la clase más importante e implica dependencia de objetos, atributos o métodos de una clase con respecto a otra (ver Figura 2.5).

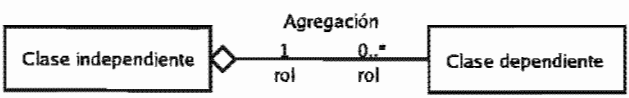

Figura 2.5: Agregación

Composición: Es un tipo especial de agregación, en el cual ciertas clases están contenidas físicamente en una clase. Este caso especial de relación se representa con un *diamante negro* junto a la clase compuesta (ver Figura 2.6).

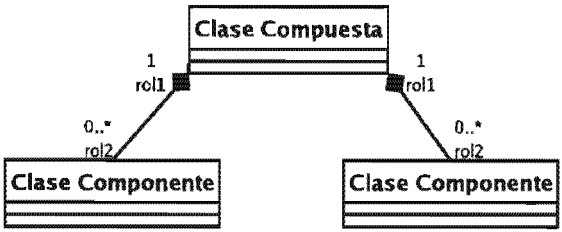

Figura 2.6: Composición de las clases

Restricción: Se indica colocándola entre los símbolos { y } (ver Figura 2.7).

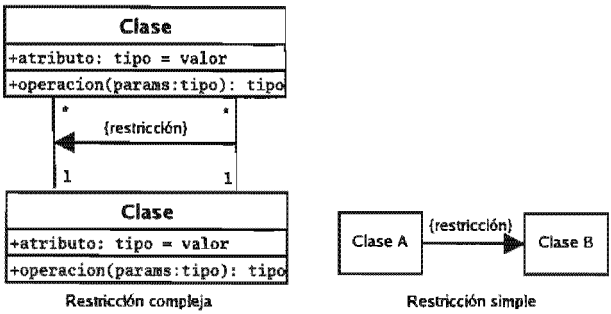

Figura 2.7: Restricción

Generalización: Representa la herencia (o también llamada relación *is a*). Ésta se refiere a la relación entre dos clases, donde una clase es una especialización de la otra. La generalización se representa como una flecha que va de la clase especializada a la clase general (ver Figura 2.8).

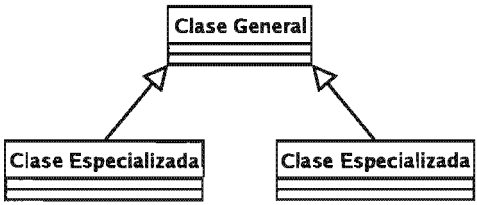

Figura 2.8: Generalización

2.4.2. DIAGRAMA DE PAQUETES

Los *diagramas de paquetes* organizan los elementos de un sistema en grupos relacionados para minimizar las dependencias.

Los elementos básicos de un diagrama de paquetes son:

Paquetes: Los símbolos con forma de *folders con pestañas* sirven para representar un paquete, y al igual que en las clases, se pueden listar los atributos de un paquete. Dentro de este *folder* (o en su *pestaña*) se escribe el nombre del paquete (ver Figura 2.9).

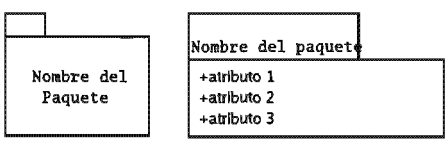

Figura 2.9: Diagrama de un paquete

Visibilidad: Los *marcadores de visibilidad* permiten restringir los accesos al paquete. Existen tres tipos de visibilidad (ver Figura 2.10):

- *Private.* Los atributos u operaciones privadas no pueden ser accedidas por alguien fuera del paquete.
- *Public.* Los atributos u operaciones públicas pueden ser accedidas por otros paquetes.
- *Protected.* Los atributos u operaciones privadas solo pueden ser accedidas por aquellos paquetes que hereden de éste.

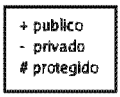

Figura 2.10: Visibilidad de un paquete

Dependencia: Define la relación en la cual los cambios en un paquete afectarán a otro paquete. La importación es un tipo de dependencia donde se concede el acceso al contenido de un paquete a otro paquete (ver Figura 2.11).

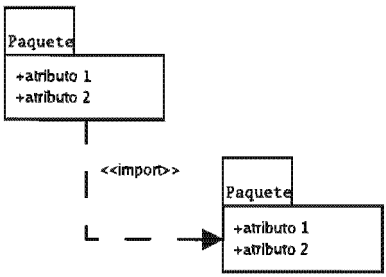

Figura 2.11: Dependencia de un paquete

2.4.3. DIAGRAMA DE OBJETO

Los diagramas de objeto estan muy ligados a los diagramas de clases. Un objeto es una instancia de una clase, mientras que un diagrama de objeto puede verse como una instancia de un diagrama de clase. El diagrama de objeto describe la estructura estática de un sistema en un tiempo en particular y son usados para probar la exactitud de los diagramas de clase.

Los elementos básicos de un diagrama de objeto son:

Nombres de los objetos: Cada objeto se representa con un rectángulo, que contiene el nombre del objeto y el de su clase subrayados y separados por dos puntos (ver Figura 2.12).

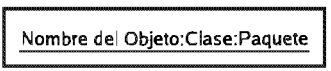

Figura 2.12: Nombres de los objetos

Atributos de los objetos: Como en las clases, se pueden listar los atributos de un objeto en un compartimento por separado. Sin embargo, al contrario de las clases, los atributos de los objetos deben tener valores asignados (ver Figura 2.13).

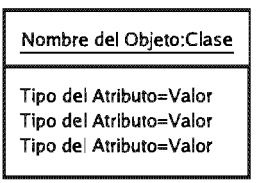

Figura 2.13: Atributos de los objetos

Objetos activos: Los objetos que tienen un flujo de acción son llamados objetos activos. Estos objetos se representan igual que los objetos solo que con un contorno más grueso (ver Figura 2.14).

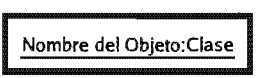

Figura 2.14: Objetos Activos

Multiplicidad: Se pueden representar múltiples objetos como un solo símbolo si los atributos de los objetos no son importantes (ver Figura 2.15).

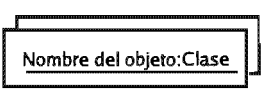

Figura 2.15: Multiplicidad

Ligas: Las ligas son instancias de las asociaciones. Estas ligas se representan usando las mismas líneas que se usan en los diagramas de clases (ver Figura 2.16).

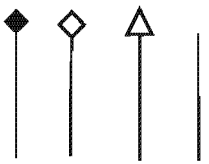

Figura 2.16: Ligas

Autoligado: Son los objetos que juegan mas de un rol (ver Figura 2.17).

Figura 2.17: Objetos Autoligados

2.4.4. DIAGRAMA DE COMPONENTES

Los diagramas de componentes describen los elementos físicos del sistema y sus relaciones, además de que muestran las opciones de realización incluyendo código fuente, binario y ejecutable. Los componentes representan todos los tipos de elementos software que entran en la fabricación de aplicaciones computacionales, por ejemplo, simples archivos, paquetes, bibliotecas cargadas dinámicamente, etc.

Los elementos básicos de un diagrama de componentes son:

Componente: Es un bloque físico de construcción de un sistema. Se representa con un rectángulo con pestañas (ver Figura 2.18).

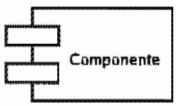

Figura 2.18: Componente

Interfaz: Describe un grupo de operaciones usadas o creadas por los componentes. La forma usual con que se representa es con una línea con un círculo en un extremo (ver Figura 2.19).

Figura 2.19: Interfaz

Dependencia: Se representa con lineas punteadas entre los componentes (ver Figura 2.20).

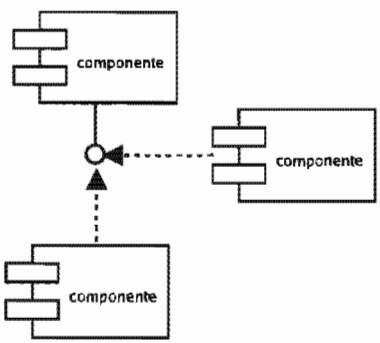

Figura 2.20: Dependencia

2.4.5. DIAGRAMA DE INSTALACIÓN

Los diagramas de instalación muestran la disposición física de los distintos nodos que componen un sistema y el reparto de los componentes sobre dichos nodos. La vista de instalación 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 tal como una computadora, un dispositivo o memoria, etc.

Los elementos básicos de un diagrama de instalación son:

Nodo: Es un recurso físico que ejecuta el código de los componentes, el cual se representa mediante un cubo con una etiqueta que contiene el nombre del nodo subrayado (ver Figura 2.21).

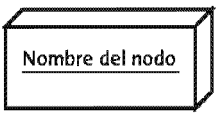

Figura 2.21: Nodo

Asociación: Se refiere a una conexión física entre dos nodos, tales como una red. Ésta se representa mediante una línea gruesa (ver Figura 2.22).

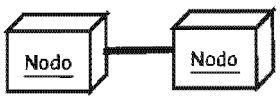

Figura 2.22: Asociación

Componentes y nodos: Los componentes se colocan dentro de un nodo (ver Figura 2.23).

2.4.6. DIAGRAMA DE CASOS DE USO

Los diagramas de casos de uso modelan las funcionalidades de un sistema usando los actores y los casos de uso. Los casos de uso son servicios o funciones provistos por el sistema a sus usuarios.

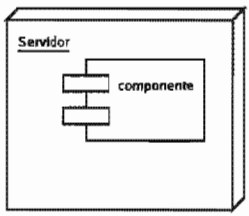

Figura 2.23: Componentes y nodos

Los elementos básicos de un diagrama de casos de uso son:

Sistema: Para representar los alcances del sistema se debe dibujar un rectángulo que contenga una etiqueta con el nombre del sistema y los casos de uso, a los actores se les debe colocar fuera del rectangulo (ver Figura 2.24).

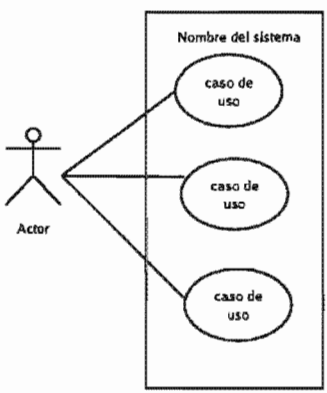

Figura 2.24: Sistema

Caso de Uso: Un caso de uso representa las acciones del sistema y se representa usando un círculo ovalado que contiene una etiqueta interna (ver Figura 2.25).

Un caso de uso debe ser simple, inteligible, claro y conciso. Generalmente hay pocos actores asociados a cada caso de uso.

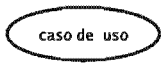

Figura 2.25: Caso de Uso

Para construir los casos de uso se deben considerar algunas de las siguientes preguntas:

1. ¿Cuáles son las tareas del actor?
2. ¿Qué información crea, guarda, modifica, destruye o lee el actor?
3. ¿Debe el actor notificar al sistema los cambios externos?
4. ¿Debe el sistema informar al actor de los cambios internos?

Actores: Son los usuarios del sistema y se representan mediante el dibujo de un ser humano, debajo del cual se coloca una etiqueta, que contiene el nombre del actor (ver Figura 2.26). Los tipos de actores que existen son:

- *Principales*: Personas que usan el sistema.
- *Secundarios*: Personas que mantienen o administran el sistema.
- *Material externo*: Dispositivos materiales imprescindibles que forman parte del ámbito de la aplicación y deben ser utilizados.
- *Otros sistemas*: Sistemas con los que el sistema interactúa.

Figura 2.26: Actor

Relaciones: Las relaciones entre los casos de uso y el actor se representan mediante una línea simple. Para las relaciones entre casos de uso, se usan flechas etiquetadas, ya sea con «include» o «extends» (ver Figura 2.27).

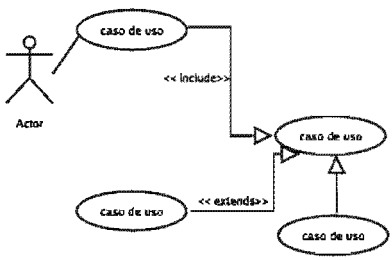

Figura 2.27: Relaciones

UML define tres tipos de relaciones:

- *Inclusión («include»):* Una instancia del caso de uso origen incluye también el comportamiento descrito por el caso de uso destino. Reemplazó al denominado «uses».
- *Extensión («extend»):* El caso de uso origen extiende el comportamiento del caso de uso destino.
- *Herencia:* El caso de uso origen hereda la especificación del caso de uso destino y posiblemente la modifica y/o amplía.

2.4.7. DIAGRAMA DE COLABORACIÓN

Un diagrama de colaboración describe las interacciones entre los objetos en términos de mensajes secuenciales. Los diagramas de colaboración representan una combinación de información tomada de los diagramas de clases, de secuencia y de los diagramas de casos de uso describiendo tanto la estructura estática como el comportamiento dinámico de un sistema. A diferencia de los diagramas de secuencia, los diagramas de colaboración muestran explícitamente las relaciones entre roles. Por otra parte, un diagrama de colaboración no muestra el tiempo como una dimensión aparte, por lo que resulta necesario etiquetar con números de secuencia tanto la secuencia de mensajes como los hilos concurrentes.

Los elementos de un diagrama de colaboración son:

Roles de las Clases: Los roles de las clases describen como los objetos se comportan. Se usa el símbolo de objeto de UML para ilustrar los roles de las clases, pero sin mostrar los atributos de los objetos (ver Figura 2.28).

Figura 2.28: Roles de las Clases

Roles de Asociación: Los roles de asociación describen como una asociación se va a comportar dada una situación particular. Los roles de asociación se pueden dibujar usando simples lineas etiquetadas con estereotipos[1](ver Figura 2.29).

Figura 2.29: Roles de Asociación

Mensajes: Al contrario de los diagramas de secuencia, los diagramas de colaboración no tienen una forma explícita para denotar el tiempo, en lugar de eso se numera a los mensajes en el orden de ejecución. La numeración puede ser anidada usando el sistema decimal. Por ejemplo, unos mensajes anidados dentro de un primer mensaje pueden ser etiquetados por 1.1, 1.2 y así sucesivamente. La condición para un mensaje se ubica entre paréntesis cuadrados inmediatamente después del número de secuencia. Se usa un * después del número de secuencia para indicar un ciclo (ver Figura 2.30).

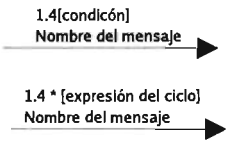

Figura 2.30: Mensajes

2.4.8. DIAGRAMA DE ESTADOS

Los diagramas de estado muestran el comportamiento de las clases en respuesta de estímulos externos. Este diagrama modela el flujo de control dinámico de un

[1]Un estereotipo es un concepto que nos permite definir un nuevo significado de la semántica para el elemento a modelar.

estado a otro dentro de un sistema.

El estado está caracterizado parcialmente por los valores de algunos 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 elementos de un diagrama de estado son:

Estados: Representan situaciones durante el ciclo de vida de un objeto, los cuales se ilustran mediante un rectángulo con las esquinas redondeadas (ver Figura 2.31).

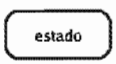

Figura 2.31: Estado

Transiciones: Indican el camino entre los diferentes estados de un objeto y se representan mediante una flecha de color negra. Una transición se etiqueta con el evento lanzado y la acción que resulta de éste (ver Figura 2.32).

Figura 2.32: Transición

Eventos: Un evento es una ocurrencia que puede causar la transición de un estado a otro de un objeto. Esta ocurrencia puede ser:

- Condición que toma el valor de verdadero o falso.
- Recepción de una señal de otro objeto en el modelo.
- Recepción de un mensaje.

- Paso de cierto periodo de tiempo, después de entrar al estado o de cierta hora y fecha particular.

El nombre del evento tiene alcance dentro del paquete en el cual está definido, no es local a la clase que lo nombra.

Estado Inicial: Es el estado en que siempre se encuentra cualquier objeto de la clase al comenzar. Este estado se representa mediante un círculo relleno seguido por una flecha (ver Figura 2.33).

Figura 2.33: Estado inicial

Estado Final: Es el estado en el que los objetos de una clase pueden finalizar. El estado final se representa mediante un círculo relleno situado dentro de otro círculo (ver Figura 2.34).

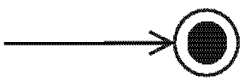

Figura 2.34: Estado final

Sincronización y Partición: La sincronización se representa mediante una barra con dos transiciones entrantes. La partición se representa mediante una barra con dos transiciones salientes (ver Figura 2.35).

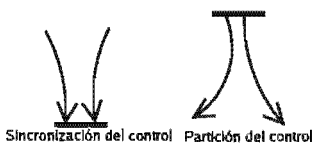

Figura 2.35: Sincronización y Partición

2.4.9. DIAGRAMAS DE ACTIVIDADES

Los *diagramas de actividades* representan la naturaleza dinámica de un sistema al modelar el flujo de control de una actividad a otra. Una actividad representa una operación sobre alguna clase en el sistema que resulta de un cambio en el estado del sistema. Típicamente, los diagramas de actividad son usados para modelar el flujo de trabajo o el proceso del negocio y las operaciones internas. Dado que un diagrama de actividad es un tipo especial del diagrama de estado, éste usa algunas de las mismas convenciones en la notación.

Los símbolos y la notación para un diagrama básico de actividad son los siguientes:

Estado de la acción: Representa las acciones ininterrumpibles de los objetos. Los estados de la acción se representan mediante un rectángulo con las *esquinas redondeadas* (ver Figura 2.36).

Figura 2.36: Estado de la acción

Flujo de la acción: EL flujo de la acción representa las relaciones entre los estados de la acción (ver Figura 2.37).

Figura 2.37: Flujo de la acción

Flujo de los objetos: EL flujo de los objetos se refiere a la creación y modificación de los objetos por las actividades. Cuando la flecha del flujo de un objeto va de una acción a un objeto significa que la acción crea o influencia al objeto. Cuando la flecha del flujo de un objeto va de un objeto a una acción significa que el estado de la acción usa al objeto (ver Figura 2.38).

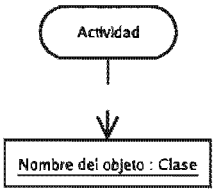

Figura 2.38: Flujo de un objeto

Estado inicial: Para representar el estado inicial de la acción se utiliza un *círculo negro* seguido de una *flecha* (ver Figura 2.39).

Figura 2.39: Estado inicial

Estado final: Para representar el estado final de la acción se utiliza una flecha que apunte a un *círculo* que tiene dentro otro *círculo*, el cual es *negro* (ver Figura 2.40).

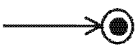

Figura 2.40: Estado final

Ramificación: Un *diamante* representa una decisión con caminos alternativos. Las alternativas que salen deben estar etiquetadas con una condición. También se puede etiquetar alguno de los caminos con *else* (ver Figura 2.41).

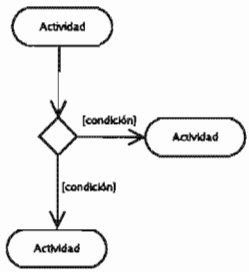

Figura 2.41: Ramificación

Sincronización: Sirve para representar las transiciones paralelas, y se representa con una *barra de sincronización* (ver Figura 2.42).

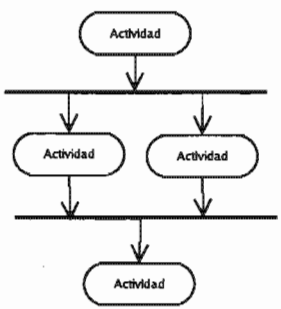

Figura 2.42: Sincronización

Carriles: Los *carriles* sirven para agrupar las actividades relacionadas en una columna (ver Figura 2.43).

2.4.10. DIAGRAMAS DE SECUENCIA

Los *diagramas de secuencia* describen las interacciones entre las clases en términos de un intercambio de mensajes.

Los símbolos y la notación para un diagrama básico de secuencia son los siguientes:

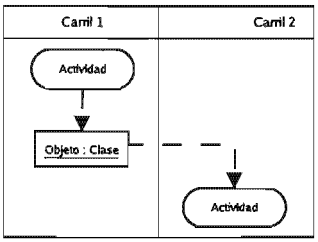

Figura 2.43: Carriles

Roles de las clases: Describen la manera en que un objeto se comportará dentro de un contexto. Para ilustrar los roles de las clases se usa el símbolo de objeto, pero sin listar los atributos de un objeto (ver Figura 2.44):

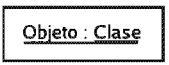

Figura 2.44: Diagrama de un objeto

Activación: Las cajas de activación representan el tiempo que un objeto necesita para completar una tarea (ver Figura 2.45).

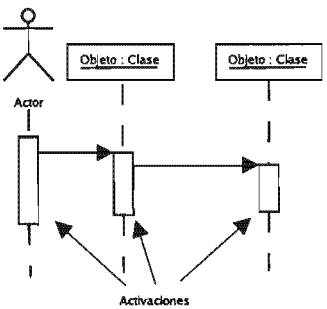

Figura 2.45: Activación

Mensajes: Los mensajes representan la comunicación entre objetos y son simbolizados con *flechas*. Los mensajes asíncronos son enviados desde un objeto que no esperará una respuesta del receptor antes de continuar sus tareas. Los mensajes asíncronos son representados por líneas con solo la *mitad de una flecha*. Para representar los mensajes, UML proporciona una lista de símbolos (ver Figura 2.46):

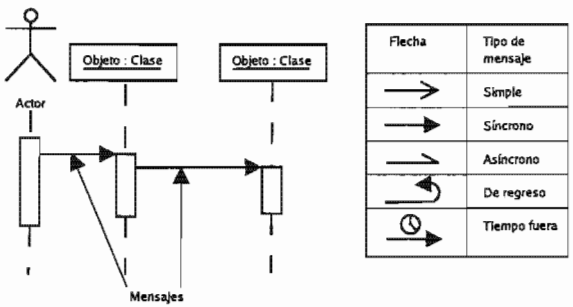

Figura 2.46: Mensajes

Líneas de vida: Representa la vida del objeto durante la interacción. Éstas son representadas por *líneas punteadas* que indican la presencia del objeto en un tiempo determinado (ver Figura 2.47).

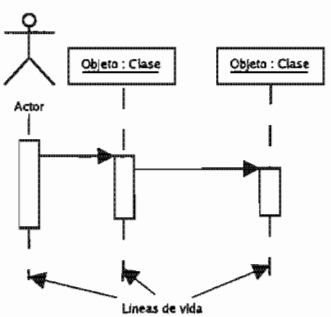

Figura 2.47: Líneas de vida

Destrucción de objetos: Para indicar que el tiempo de vida de un objeto se ha terminado se usa una *flecha* etiquetada con *«destroy»* que apunta a una *X* (ver Figura 2.48).

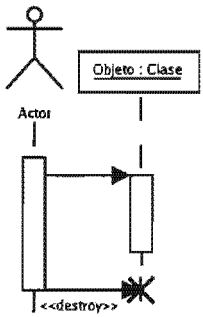

Figura 2.48: Destrucción de un objeto

Ciclos: Un ciclo dentro un diagrama de secuencia es representado por un rectángulo. La condición de fin de un ciclo debe indicarse en la esquina inferior izquierda entre los símbolos *[* y *]* (ver Figura 2.49).

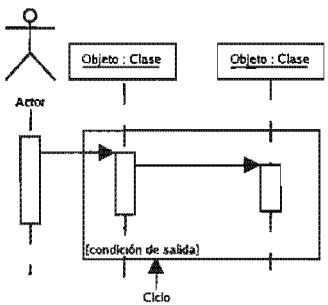

Figura 2.49: Ciclos

CAPÍTULO 3

LENGUAJE EXTENDIBLE DE MARCADO

3.1. INTRODUCCIÓN

El Lenguaje Extendible de Marcado (Extensible Markup Language - XML) es un formato para representar información de manera independiente a cualquier formato. XML define una sintaxis genérica que, mediante marcas, describe los datos que se representan.

El objetivo de XML es que la información de un documento XML pueda ser usada por cualquier aplicación, bien sea una base de datos, un procesador de textos o cualquier otra, que incluso puede no existir aún.

XML se define por las siguientes características:

- XML es un metalenguaje con el que podemos definir otros lenguajes de etiquetas, para su uso particular.
- Con XML lo que hacemos es definir semánticamente la información.
- XML es un lenguaje interpretable por los humanos y por las máquinas.
- Con XML lo que hacemos es separar el contenido de su presentación.

3.2. ESTRUCTURA DE UN DOCUMENTO XML

Un documento XML contiene exclusivamente datos que se autodefinen. En XML se separa el contenido de la presentación de forma total.

3.2.1. DOCUMENTOS XML BIEN-FORMADOS

Para que un documento XML se considere bien formado, debe cumplir con las siguientes características:

- Estructura jerárquica de elementos.
- No debe contener etiquetas vacías.
- Contener un solo elemento raíz.
- Se deben diferenciar las letras mayúsculas de las minúsculas.
- No se pueden crear nombres que empiecen con la cadena *xml*, *xML*, *XML* o cualquier otra variante. Las letras y rayas se pueden usar en cualquier parte del nombre, mientras que con los dígitos, guiones y caracteres de punto no puede empezar un nombre. El resto de los caracteres, como algunos símbolos y espacios en blanco, no se pueden usar.

3.3. TECNOLOGÍAS ASOCIADAS

3.3.1. DTD

Los DTDs son los que definen la estructura de los documentos XML es decir crear una definición de documentos o DTD es como crear nuestro propio lenguaje de marcado, para una aplicación específica.

EL DTD define los tipos de elementos, atributos y entidades permitidas, y puede expresar algunas limitaciones para combinarlos.

Los documentos XML que se ajustan a un DTD, se denominan *válidos*. El concepto de validez no tiene nada que ver con el de estar bien formado. Un documento bien formado simplemente respeta la estructura y sintáxis definidas por la especificación de XML. Un documento bien formado puede además ser válido si cumple las reglas de un DTD determinado.

3.3.2. ESQUEMAS

Los Esquemas (*Schemas*) vienen a sustituir a los DTDs, dado que sirven para lo mismo pero éstos son mucho más potentes. Algunas características de los esquemas son:

- Define los elementos y atributos que pueden aparecer en un documento.
- Define cuales elementos son elementos hijos.
- Define si un elemento es vacio o puede contener texto.
- Define los tipos de dato para los elementos y sus atributos.

3.3.3. PROCESADORES DE DOCUMENTOS XML

Los procesadores de documentos XML más representativos son DOM y SAX, por lo cual es necesario retroceder un poco a lo que es la manipulación de datos en un sistema de información.

El manipular datos para nuestro uso propio resulta sencillo pues nosotros conocemos su estructura interna, pero cuando deseamos intercambiar información a otros sistemas de cómputo o empresas, les resulta difícil o imposible interpretar esta información. La solución es generar un documento descriptivo, por ejemplo, podemos representar la información en un documento XML y ya teniéndolo en este formato surge la pregunta ¿cómo se traslada esta información a un programa en Java, Perl o una aplicación de servidor?. Esto se hace mediante un *procesador* (parser) DOM o SAX, así como la transformación de los datos a otros formatos, tales como HTML o PDF con XSLT.

DOM (Document Object Model)

DOM es una especificación definida por el World Wide Web Consortium (W3C). Algunas de sus características son:

- Da una representación interna estándar de la estructura de un documento. Esta representación es un árbol jerárquico en memoria del documento, donde algunos nodos pueden tener nodos hijos o bien ser nodos terminales u hojas.
- Proporciona una interfaz (API) al programador para acceder de forma fácil, consistente y homogénea a los elementos y atributos de un documento.
- Es un modelo independiente de la plataforma y del lenguaje de programación.
- Un DOM completo debe permitir :
  - Reconstruir el documento completo a partir del modelo.
  - Acceder a cualquiera de las partes del documento.

- Manipular, adicionar y eliminar en el documento.

- La especificación de DOM consta de tres niveles :
  - Nivel 1: Contiene los modelos para HTML y XML, además de las funcionalidades para la navegación y manipulación de documentos. Este nivel consta de dos partes: el core referida a XML y la parte de HTML.
  - Nivel 2: Contiene el modelo de objetos y la interfaz de acceso a las partes de un documento.
  - Nivel 3: Permite el acceso a los DTD, hojas de estilo y espacios de nombres. Este nivel se encuentra actualmente en desarrollo.
- La especificación DOM del W3C utiliza el lenguaje OMG IDL (Object Management Group Interface Definition Language).
- En IDL:
  - Los objetos tienen interfaces con el mundo exterior.
  - Cada interfaz tiene atributos que describen las propiedades del objeto.
  - El objeto se manipula a través de métodos.
  - Los métodos devuelven un resultado a la aplicación solicitante.
- Su especificación más reciente es la 2.0.

Simple API for XML (SAX)

SAX es un API para XML que procesa el documento o información en XML de una manera muy diferente a DOM, puesto que SAX procesa la información por eventos, a diferencia de DOM que genera un árbol jerárquico en memoria. SAX procesa la información en XML conforme ésta es presentada (evento por evento), manipulando cada elemento a un determinado tiempo sin incurrir en uso excesivo de memoria. Las caracteríticas principales de SAX son:

- El analizador de SAX:
  - No crea ninguna estructura de datos para representar al documento.
  - Va analizando el documento y generando eventos (comienzo de un elemento, final de un elemento, etc.).
  - SAX es una interfaz de un analizador más que una API para una estructura de datos en árbol (DOM).

- Está en un nivel más bajo que DOM.
- Será mejor que DOM cuando:
  - El documento no quepa en memoria.
  - Las tareas sean irrelevantes para la estructura del documento (contar todos los elementos, extraer el contenido de un elemento específico, etc.).
- La aplicación debe registrar manipuladores de eventos a un objeto analizador que implementa la clase org.sax.Parser.
- SAX tiene tres interfaces de manipuladores: DocumentHandler, DTDHandler y ErrorHandler.
- El analizador empaqueta los datos XML en eventos que la aplicación puede manipular.
- Su especificación más reciente es la 2.0.

3.3.4. EXTENSIBLE STYLESHEET LANGUAGE (XSL)

XSL es un lenguaje que nos permite definir una presentación o formato para un documento XML. Un mismo documento XML puede tener varias hojas de estilo XSL que lo muestren en diferentes formatos (HTML, PDF, RTF, VRML, etc.).

La aplicación de una hoja de estilo XSL a un documento XML puede ocurrir tanto en el origen (por ejemplo, un servlet que convierta de XML a HTML para que sea mostrado a un navegador conectado a un servidor de WEB), o en el mismo navegador.

Básicamente, XSL es un lenguaje que define una transformación entre un documento XML de entrada y otro documento XML de salida.

Una hoja de estilo XSL es una serie de reglas que determinan como va a ocurrir la transformación. Cada regla se compone de un patrón (pattern) y una acción o plantilla (template). De este modo, cada regla afecta a uno o varios elementos del documento XML. El efecto de las reglas es recursivo para que un elemento situado dentro de otro elemento pueda ser también transformado. La hoja de estilo tiene una regla raíz que, además de ser procesada, llama a las reglas adecuadas para los elementos hijos.

CAPÍTULO 4

JAVA 2 ENTERPRISE EDITION

4.1. INTRODUCCIÓN

Java 2 Enterprise Edition (J2EE) es un conjunto de especificaciones para poder desarrollar aplicaciones multicapas empresariales. La tecnología J2EE provee una aproximación basada en componentes para diseñar, desarrollar y distribuir aplicaciones empresariales para la Web.

4.2. COMPONENTS DE LA APLICACIÓN

Existen cuatro componentes de la aplicación dentro de la plataforma J2EE:

- Aplicaciones cliente (clientes independientes de Java).
- Applets (código Java que se ejecuta dentro del navegador).
- Componentes Web (JSPs, Servlets).
- Componentes del servidor (EJBs, implementaciones de API J2EE).

Un producto no necesita contener todos los tipos de componentes, la norma es proveer una implementación para soportar algún componente en particular. Sin embargo, todos los componentes tienen en común que corren dentro de un contenedor, el cual es responsable de proveer un ambiente en tiempo de ejecución, el mecanismo para identificar y entender los formatos de los archivos usados para la entrega y los servicios estándares para el uso de los componentes de la aplicación.

A continuación describiremos cada uno de estos componentes:

Aplicaciones cliente Son aplicaciones independientes escritas en Java. Éstas se ejecutan dentro de la máquina virtual y pueden usar los servicios estándar de J2EE para acceder a componentes localizados en otra capa.

Applets Son similares a las aplicaciones cliente, pero se ejecutan dentro de un navegador Web. Inicialmente los applets generaron mucha atención pues le dieron más dinamismo a las páginas Web. Casi todos los navegadores cuentan con una Máquina Virtual de Java (JVM).

Componentes Web Estos componentes estan del lado del servidor, y son usados para proveer la capa de presentación que será regresada a un cliente. Existen dos tipos de componentes Web: *JavaServer Pages* (ver Capítulo 6) y los *Java Servlets* (ver Capítulo 5).

Componentes en el Servidor Estos componentes son conocidos como *Enterprise JavaBeans* (EJBs). Los EJBs se ejecutan dentro de un contenedor que maneja el comportamiento del EJB en tiempo de ejecución. Los EJBs es el lugar donde se encuentra usualmente la lógica del negocio.

4.3. APIs DE J2EE

La especificación de J2EE establece un número de APIs diferentes, los cuales no necesariamente deben de encontrarse en todos los componentes de la aplicación.

J2EE establece un conjunto de servicios estándares, de los cuales algunos son provistos por *Java 2 Standard Edition* (J2SE), mientras que otros estan contenidos en paquetes opcionales dentro de J2SE, pero obligatorios dentro de J2EE. Estos servicios son:

HyperText Transfer Protocol/HyperText Transfer Protocol Secure sockets (HTTP/HTTPS) Ambos protocolos deben ser soportados por los servidores J2EE.

Java Transaction API (JTA) Provee una interfaz para delimitar transacciones. Ésto permite al desarrollador agregar el procesamiento de transacciones a los sistemas.

Remote Method Invocation to Internet Inter-ORB Protocol (RMI-IIOP) Los componentes EJB usan este servicio de comunicación. El protocolo IIOP puede ser usado para acceder a objetos *CORBA* que se localizan en sistemas remotos.

Java Database Connectivity (JDBC) Provee una interfaz Java para ejecutar declaraciones *SQL* (ver Capítulo 7).

Java Message Service (JMS) Es un servicio de mensajería asíncrono que permite al usuario enviar y recibir mensajes.

JavaMail Habilita la entrega y recepción de correo electrónico.

Java Naming and Directory Interface (JNDI) Es usado para acceder a directorios, tales como *Lightweight Directory Access Protocol* (LDAP). Comunmente, los componentes usan este API para obtener referencias a otros componentes.

JavaBeans Activation Framework (JAF) JavaMail usa JAF para manejar diferentes tipos de *Multipurpose Internet Mail Extensions* (MIME) que pueden ser incluidos dentro de un mensaje de correo electrónico. Éste convierte los flujos de bytes MIME en objetos Java que puedan ser manejados por los JavaBeans correspondientes.

Java API for XML Parsing (JAXP) Incluye los APIs *Simple API for XML* (SAX) y *Document Object Model* (DOM) para manipular documentos XML. El API de JAXP también habilita al *Extensible Stylesheet Language* como un agregado.

J2EE Connector Architecture Especifica el mecanismo a través del cual se pueden agregar nuevos recursos al servidor de J2EE.

Servicios de Seguridad Son provistos vía *Java Authentication and Authorization Service* (JAAS), el cual permite a los servidores de J2EE controlar el acceso a los servicios.

Servicios Web El soporte a los servicios Web es provisto vía *Simple object Access Protocol* (SOAP).

Management Los APIs *Java 2 Plataform Enterprise Edition Management* y *Java Management Extensions* (JMX) son usados para proveer el manejo del soporte para la consulta a un servidor durante el tiempo de ejecución.

Java Authorization Service Provider Contract for Containers (JACC) Es la interfaz entre los servidores de la aplicación y los proveedores de las políticas de autorización.

4.4. ENTERPRISE JAVABEANS

Enterprise JavaBeans es una especificación para crear aplicaciones escalables del lado del servidor, que manejen transacciones, que sean multiusuario y seguras a un nivel empresarial. Un EJB encapsula la lógica del negocio de una aplicación.

4.4.1. ARQUITECTURA DE LOS EJBS

Una arquitectura típica de un EJB consiste en:

- Un servidor EJB.
- Contenedores EJB que corren dentro de los servidores.
- EJBs que corran dentro de los contenedores.
- Clientes EJB.
- Otros sistemas auxiliares como *Java Naming and Directory Interface* (JNDI), entre otros.

A continuación describiremos brevemente estos elementos:

Servidores EJB Proveen los servicios del sistema, tales como el ambiente de ejecución, el multiprocesamiento, acceso a los dispositivos, etc.

Contenedores EJB Actúan como una interfaz entre un EJB y el mundo exterior. Un cliente EJB nunca accede a un bean directamente. Cualquier acceso a los EJB se realiza a través de métodos generados por el contenedor. Existen dos tipos de contenedores: contenedores de sesiones y contenedores de entidades.

Clientes EJB Hacen uso de los EJB mediante sus métodos. Localizan al contenedor EJB a través de la interfaz JNDI, tras lo cual, los clientes usan al contenedor EJB para invocar sus métodos.

La Figura 4.1 muestra la arquitectura de los EJBs.

4.4.2. TIPOS DE EJBS

Existen tres tipos de EJBs: *Session Beans*, *Entity Beans* y *Message-Driven Beans*.

Session Beans Representan el trabajo que está siendo desarrollado por el código del cliente que lo está llamando. Los *Session Beans* son objetos para el proceso del negocio. Éstos implementan la lógica del negocio, las reglas del negocio y el flujo de trabajo.

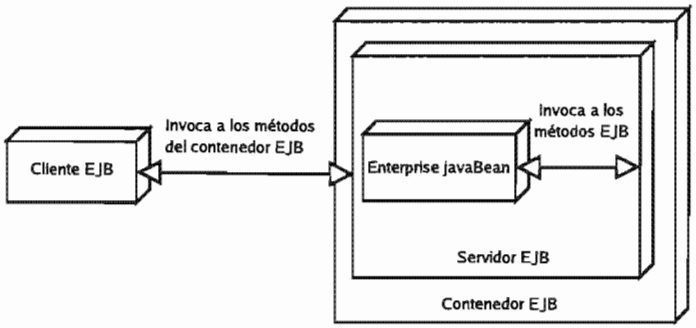

Figura 4.1: Arquitectura de los EJBs

Los *Session Beans* son llamados así porque ellos viven tanto como una sesión (o tiempo de vida) del código del cliente que esta llamando al *Session Bean*. Existen dos subtipos de *Session Beans*:

- *Stateful Session Beans*. Mantienen el estado de un cliente en particular. Si el estado de un *Stateful Session Bean* es cambiado durante la invocación de un método, ese mismo estado se encontrará disponible para el mismo cliente en la siguiente invocación.
- *Stateless Session Beans* No mantienen un estado conversacional para un cliente en particular. Cuando un cliente invoca un método de un *Stateless Session Bean* las variables de instancia de éste pueden contener un estado, pero solo durante la duración de la invocación. Cuando el método se termina, el estado no se mantiene. Excepto durante la invocación del método, todas las instancias de un *Stateless Session Bean* son equivalentes, permitiendo al contenedor EJB asignar una instancia a cualquier cliente.

Entity Beans Representan un objeto del negocio en un mecanismo de almacenamiento persistente. Existen dos tipos de *Entity Beans*:

- *Container-Managed Persistence Entity Beans*, aquí la persistencia del bean es manejada por el contenedor EJB.
- *Bean-Managed Persistence Entity Beans*, aquí la persistencia del bean tiene que ser manejada por el desarrollador, esto le da un mayor control al desarrollador del EJB para guardar y obtener datos, pero este tipo de EJBs son más dificiles de construir.

Message-Driven Beans Permiten a las aplicaciones J2EE manejar mensajes de manera asíncrona. Estos mensajes pueden ser enviados por cualquier componente J2EE, ya sea una aplicación cliente, otro EJB o un componente Web, o por una aplicación *Java Message Service* (JMS) o un sistema que no use la tecnología J2EE.

4.4.3. ELEMENTOS FUNDAMENTALES DE UN EJB

Los elementos principales para cualquier tipo de EJB son:

Instancia de un *enterprise bean* es una instancia de un objeto Java de una clase *enterprise bean*, la cual contiene las implementaciones de los métodos del negocio definidos en la interfaz remota. La instancia del *enterpise bean* no se ocupa de la lógica de la red.

Interfaz Remota es una interfaz Java que enumera los métodos del negocio expuestos en la clase *enterprise bean*. El código del cliente siempre va a través de la interfaz remota y nunca interactúa con la instancia del *enterprise bean*. La interfaz remota sigue las reglas de Java RMI.

Objeto EJB es la implementacion de la interfaz remota generada por el contenedor. El objeto EJB es un intermediario que tiene conocimiento de la red entre el cliente y la instancia del *bean*, manejando todos los asuntos necesarios en la capa intermedia. Todas las invocaciones del cliente son a través del objeto EJB. El objeto EJB delega todas las llamadas a las instancias del *enterprise bean*.

Interfaz local es una interfaz Java que sirve como una fábrica de objetos EJB. El código del cliente que quiere trabajar con objetos EJB debe usar la interfaz local para generarlos. La interfaz local tiene conocimiento sobre la red porque es usada por los clientes a través de la red.

Objeto Local es la implementación de la interfaz local generada por el contenedor. El objeto local tiene conocimiento sobre la red y sigue las reglas de Java RMI.

Descriptor de Distribución (deployment descriptor) especifica los requerimientos de la capa intermedia del *enterprise bean*. Éste se usa para informar al contenedor como manejar al *bean*, cuantos ciclos de vida de un bean se necesitan, y las necesidades de transacción, persistencia y seguridad.

Propiedades del *bean* son los atributos que el *bean* usa en tiempo de ejecución.

Archivo Ejb-jar Es un componente completo y terminado que contiene a la clase *enterprise bean*, la interfaz remota, la interfaz local, las propiedades del bean y al descriptor de distribución.

4.4.4. INTERACCIÓN CLIENTE-EJB

A grandes rasgos la interacción entre un cliente con un EJB consta de los siguientes pasos que también se ilustran en la Figura 4.2.

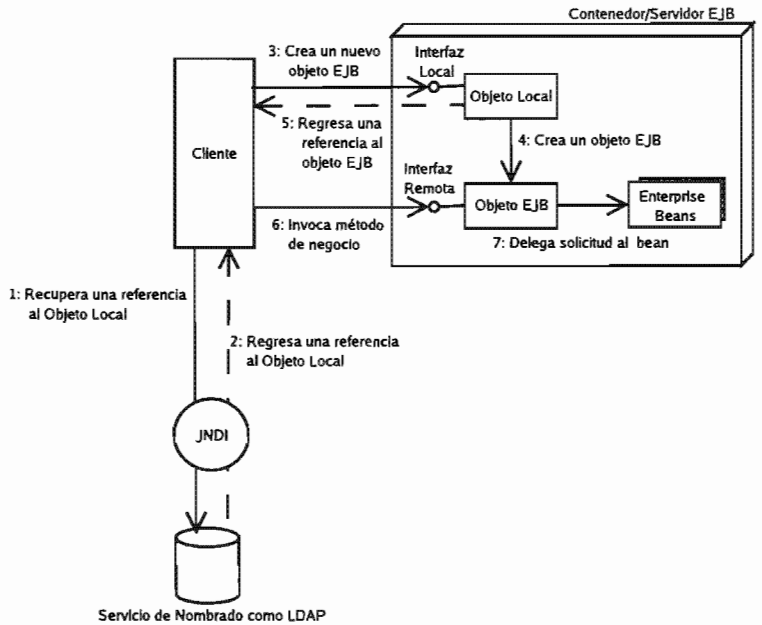

Figura 4.2: Interacción Cliente-EJB

1. **Obtiene un objeto local**. Para obtener una referencia a un objeto local el cliente debe relizar dos pasos previos:

   a) *Instalar su ambiente*. El cliente debe especifiar qué servicio de directorio esta usando, dónde se localiza en la red el servicio que uno desea, y

especificar cualquier nombre o contraseñas que pueden ser requeridos para la autenticación.

b) *Especificar el contexto inicial.* Como vimos anteriormente el contexto inical es un punto inicial para las estructuras de directorio. El cliente necesita dar el contexto inicial a las propiedades del ambiente.

A continuación, el cliente debe ejecutar el método *lookup()* de JNDI, el cual regresa un objeto RMI remoto, al cual el cliente debe hacer un *cast* a un objeto local.

La Figura 4.3 ilustra estos pasos:

```
/*Obtén propiedades del sistema para la inicialización JNDI*/
Properties props=System.getProperties();

/*Construye un contexto inicial*/
Context ctx=new InitialContext(props);

/*Obtén una referencia al Objeto Local - la fábrica de objetos EJBs*/
MiHome casa=(MiHome) ctx.lookup("MiHome");
```

Figura 4.3: Especificar un contexto inicial

2. **Crea un Objeto EJB.** Una vez que el código del cliente ha hecho referencia a un objeto local, el cliente puede usar este objeto como una fábrica de objetos EJB. Para crear un objeto EJB, hay que llamar a algún método *create()*. La Figura 4.4 ilustra esto.

```
MIRemoteInterface objetoEJB=home.create();
```

Figura 4.4: Creación de un objeto EJB

3. **Llama a un método.** Ahora que el cliente tiene un objeto EJB, puede empezar a llamar uno o más métodos del *bean* a través de objeto EJB. Cuando el cliente llama a un método en el objeto EJB, el objeto EJB debe elegir una instancia del *bean* para atender la solicitud. El objeto EJB puede crear una nueva instancia o reutilizar una instancia existente. La Figura 4.5 ilustra la llamada al método del negocio *add()* a través del objeto EJB.

```
objetoEJB.add();
```

Figura 4.5: Llamada a un método

4. **Destruye al objeto EJB**. Finalmente cuando el cliente quiera destruir al objeto EJB, llama al método *remove()* del objeto EJB o del objeto local. Esto habilita al contenedor para destruir al objeto EJB. La Figura 4.6 ilustra esto.

```
objetoEJB.remove();
```

Figura 4.6: Destrucción del objeto EJB

CAPÍTULO 5

SERVLETS

5.1. INTRODUCCIÓN

El API de Java para Servlets es un estándar abierto creado por la compañía Sun, para crear aplicaciones web usando el lenguaje de programación Java. Un *servlet* es un objeto responsable de recibir una petición de un cliente de la red y de regresar la respuesta -una página HTML que será desplegada en el navegador-. El API para Servlets define una interfaz y una clase base para los servlets, esta clase base también sirve para otras clases de soporte, tales como *HttpServletRequest*, la cual representa una solicitud y le permite al servlet acceder a los parámetros de ésta.

5.2. CONTENEDOR

Un servlet opera dentro de un *contenedor de servlets*, el cual sirve como un puente entre el *Servidor de Web* y el código del servlet que el programador escribe. El contenedor de servlets es el responsable de la instanciación e inicialización que un servlet necesita. Éste puede ser stand-alone[1], tales como Apache Tomcat, o puede ser parte de un servidor de aplicación, entre las cuales encontramos a BEALogic, IBM WebSphere, o el código abierto de la aplicación JBoss.

El contenedor de servlets es responsable de seleccionar el servlet correcto que es invocado, basándose en el URL de la solicitud; una simple aplicación Web contendrá varios servlets. EL descriptor de distribución[2] (deployment descriptor) proporciona el nombre y la clase en Java para cada uno de los servlets e identifica qué URLs están mapeados a qué servlets.

[1]Un sistema stand-alone es aquel que es capaz de operar sin la ayuda de otros programas, bibliotecas, computadoras, hardware, redes, etc, es decir, es independiente del contexto.

[2]Un descriptor de distribución es un archivo XML empaquetado con la aplicación.

Dentro de un contenedor de servlets cada servlet debe ser instanciado una sola vez; todas las solicitudes son mapeadas al URL del servlet, el cual será pasado a través del método *service()* de un objeto servlet.

5.3. ESTRUCTURA DE UN SERVLET

La idea en general del API para Servlets se basa en la interfaz *Servlet*. Todos los servlets implementan esta interfaz, ya sea directamente o usualmente extendiendo una clase que la implementa, tal como *HttpServlet*. La interfaz *Servlet* provee los métodos que manejan al servlet y a sus comunicaciones con los clientes. Los programadores sobreescriben algunos o todos estos métodos cuando desarrollan un servlet.

Cuando un servlet acepta la llamada de un cliente, éste recibe dos objetos: un *ServletRequest* y un *ServletResponse*. La clase *ServletRequest* encapsula la comunicación del cliente al servidor, mientras que la clase *ServletResponse* encapsula la comunicación del servidor al cliente.

La interfaz *ServletRequest* permite al servlet acceder a la información proveniente del cliente, como por ejemplo, los nombres y valores de los parámetros pasados o el protocolo que éste usa. Ésta también le provee al servlet acceso a los flujos de entrada por medio de la clase, *ServletInputStream*, a través del cual el servlet obtiene datos del cliente que están usando los protocolos de la aplicación, por ejemplo, *POST* y *GET*. Las subclases de *ServletRequest* permiten al servlet alcanzar datos de protocolos más específicos.

La interfaz *ServletResponse* proporciona al servlet los métodos para responder al cliente. Esta interfaz permite al servlet cambiar el tamaño del contenido y el formato *MIME* de la respuesta, y provee el flujo de salida, *ServletOutputStream*, y un objeto de la clase *Writer* mediante el cual el servlet enviará los datos de la respuesta. Las subclases de *ServletResponse* proporcionan al servlet capacidades de protocolos más específicos.

Las clases e interfaces descritas arriba componen la estructura de un servlet básico. Los servlets *HTTP* tienen algunos objetos adicionales que proveen la capacidad de seguimiento de sesiones. El objeto *Writer* del servlet puede usar estas APIs para mantener el estado entre el servlet y el cliente que persiste a través de múltiples conexiones durante algún periodo de tiempo.

```
import javax.servlet.*;
import javax.servlet.http.*;
public class miServlet extends HttpServlet {

   public void init(ServletConfig config) throws ServletException {
       super.init(config);
   }

   public void destroy() { ... }

   protected void doGet(HttpServletRequest request,
                        HttpServletResponse response)
   throws ServletException, IOException { ... }

   protected void doPost(HttpServletRequest request,
                         HttpServletResponse response)
   throws ServletException, IOException { ... }
}
```

Figura 5.1: Estructura de un Servlet

5.4. CICLO DE VIDA DE UN SERVLET

Los servidores cargan y ejecutan servlets, aceptando cero o más peticiones de los clientes y responderles con algunos datos (ver Figura 5.2). Los servidores pueden también eliminar servlets.

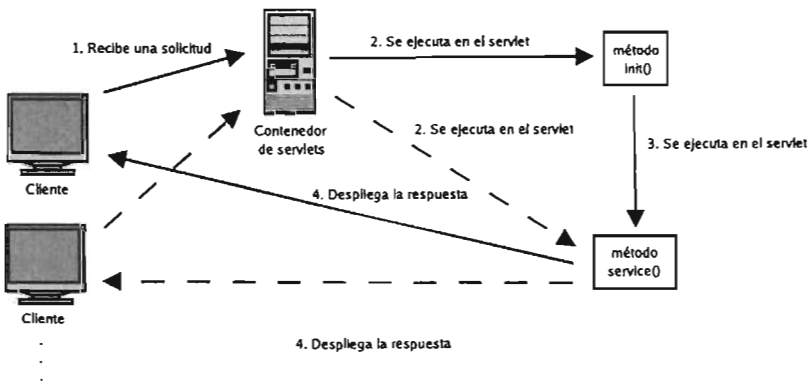

Figura 5.2: Ciclo de solicitudes

A continuación se explicará en detalle los pasos del ciclo de vida de un servlet:

- Cuando un servidor carga un servlet, éste ejecuta el método *init()* del servlet. Algunos servlets son ejecutados en servidores multi tarea, por lo tanto no se presentan problemas con la concurrencia durante la inicialización del servlet. Esto es porque el servidor llama al método *init()* solo una vez, cuando carga al servlet , y no vuelve a llamar a este método , a menos que éste sea cargado nuevamente por el servlet. El servidor puede eliminarlo llamando al método *destroy()*. La inicialización es llevada a cabo antes de que las peticiones de los clientes sean manejadas (esto es, antes de que el método *service()* sea llamado) o el servlet sea eliminado.

- Los servlets pueden ejecutar múltiples métodos *service()* hasta que son eliminados. Dado que esto es importante, los métodos *service()* son escritos de una forma segura para cuando haya varias ejecuciones simultáneas. Por ejemplo, si un método *service()* puede actualizar un campo en el objeto servlet, ese acceso debe ser sincronizado. Si por alguna razón, un servidor no debe ejecutar múltiples métodos *service()* simultáneamente, el servlet debe implementar a la interfaz *SingleThreadModel*. Esta interfaz garantiza que no habrá dos ejecuciones simultáneas del método *service()*.

- Los servlets se ejecutarán hasta que sean eliminados desde el método *service()*, por ejemplo, en una petición del sistema administrador. Cuando un servidor elimina un servlet, éste ejecutará al método *destroy()*. Este método es ejecutado solo una vez; el servidor no lo ejecutará de nuevo hasta después de que éste sea vuelto a cargar y reinicialice el servlet. Cuando el método *destroy()* se ejecuta, otros hilos de ejecución pueden estar corriendo otras peticiones. Si es necesario acceder a recursos compartidos (tales como cerrar las conexiones a una red), esos accesos deben ser sincronizados.

CAPÍTULO 6

JAVASERVER PAGES

6.1. INTRODUCCIÓN

JavaServer Pages (JSP) es una tecnología para desarollar páginas Web que incluyen contenido dinámico. A diferencia de una página HTML, en la cual el contenido es estático y siempre permanece sin cambios, una página JSP puede cambiar su contenido con base en las entradas del cliente.

Una página JSP contiene elementos del lenguaje de marcado estándar, tales como etiquetas HTML. Sin embargo, una página JSP también contiene elementos especiales del JSP para llevar a cabo una amplia variedad de objetivos, tales como la obtención de información de una base de datos o registrar las preferencias del cliente.

Las principales características de la tecnología JSP son:

- Tiene un lenguaje para desarrollar páginas JSP, las cuales son documentos basados en texto que describen cómo se procesa una solicitud y se construye la respuesta.
- Construye los objetos que estarán del lado del servidor.
- Tiene los mecanismos para definir las extensiones al lenguaje JSP.
- Permite separar la parte dinámica de nuestras páginas Web del HTML estático.
- Es más conveniente escribir un JSP que un Servlet puesto que es más fácil escribir o modificar código HTML para el contenido estático y luego añadirle el código para el contenido dinámico, que escribir un servlet donde se tiene que estar agregando líneas que generen ese contenido estático y dinámico.

6.2. PROCESAMIENTO DE UN JSP

Una página JSP no puede enviarse tal cual al navegador. Todos los elementos de un JSP primero deben ser procesados por el servidor. Esto es hecho mediante la conversión de la página JSP a un servlet, y luego se lleva a cabo la ejecución de este servlet.

Asi como un servidor Web necesita un contenedor de servlets para proveer de una interfaz a un servlet, el servidor necesita un contenedor de JSP para procesar las páginas JSP.

Un contenedor JSP es el responsable de convertir la página JSP a un servlet (conocido como *la clase que implementa al JSP*) y luego compilarlo. Estos dos pasos constituyen *la fase de traducción*. El contenedor de JSP inicia automáticamente la fase de traducción de una página cuando la primera solicitud es recibida. El contenedor de JSP también es responsable de invocar a la clase que implementa al JSP para procesar cada solicitud y generar la respuesta, conocida como *la fase de procesamiento de solicitudes*. Estas dos fases se ilustran en la figura 6.1

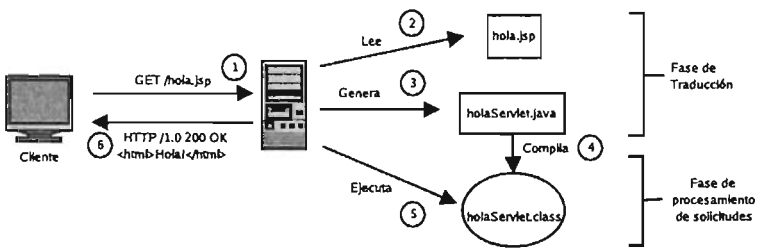

Figura 6.1: Procesamiento de un jsp

En tanto la página JSP permanezca sin cambios, cualquier procesamiento subsecuente irá directamente a la fase de procesamiento de solicitudes. Cuando una página JSP es modificada, ésta pasa otra vez a la fase de traducción antes de entrar a la fase de procesamiento de solicitudes.

A excepción de la fase de traducción, una página JSP es manejada exactamente como un servlet tradicional: éste es cargado una sola vez y es llamado en repetidas ocasiones, hasta que se apague el servidor.

6.3. ELEMENTOS DE UN JSP

Existen tres tipos de elementos de los JavaServer Pages: *directivas, elementos de script y acciones.*

Directivas son usadas para especificar información con respecto a la página misma que se utilizará para todas las solicitudes, por ejemplo, el lenguaje de script que es usado por la página o el nombre de la página que debe ser usado para reportar los errores. Algunas directivas las podemos ver en la Tabla 6.1:

Elemento	Descripción
< %@*page*...% >	Define los atributos de la página, tales como el lenguaje de scripting, la página de error y los requerimientos de almacenamiento.
< %@*include*...% >	Incluye un archivo durante la fase de traducción.
< %@*taglib*...% >	Declara una biblioteca de etiquetas que contiene opciones personalizadas usadas en la página.

Cuadro 6.1: Directivas

Elementos de script permiten adicionar pequeñas partes de código a una página JSP, tal como una declaración *if* para generar diferente HTML dependiendo de una cierta condición. Algunos elementos de script se muestran en la Tabla 6.2.

Elemento	Descripción
< %...% >	Scriptlet, usado para incluir código script.
< % = ...% >	Expresión, usada para incluir expresiones Java cuando el resultado debe de ser adicionado a la respuesta.
< %!...% >	Declaración, usada para declarar variables de instancia y métodos en la implementación de la clase de la página del JSP.

Cuadro 6.2: Elementos de script

Acciones realizadas dinámicamente al momento que la página JSP es solicitada por un cliente. Un elemento de acción puede acceder a los parámetros enviados con la solicitud. Las acciones también pueden generar HTML dinámico, tal como una tabla que contenga información obtenida de un sistema externo.

La especificación de JSP define algunos elementos de acción predefinidos que se pueden ver en la Tabla 6.3.

Elemento	Descripción
$< jsp : useBean >$	Habilita a un componente JavaBeans en una página.
$< jsp : getProperty >$	Toma el valor de una propiedad de un componente JavaBean y lo adiciona a la respuesta.
$< jsp : setProperty >$	Cambia el valor de la propiedad de un JavaBean.
$< jsp : include >$	Incluye la respuesta de un servlet o una página JSP durante la fase de procesamiento de solicitudes.
$< jsp : forward >$	Reenvia el procesamiento de la solicitud de un servlet o una página JSP.
$< jsp : param >$	Adiciona el valor de un parámetro a una solicitud liberada a otro servlet o página JSP.
$< jsp : plugin >$	Genera HTML que contiene los elementos apropiados dependientes del navegador del cliente que se necesitan para ejecutar un applet con el plugin del software.

Cuadro 6.3: Acciones

6.4. ESTRUCTURA DE UN JSP

En la Figura 6.2 se muestra la estructura general de una página JSP.

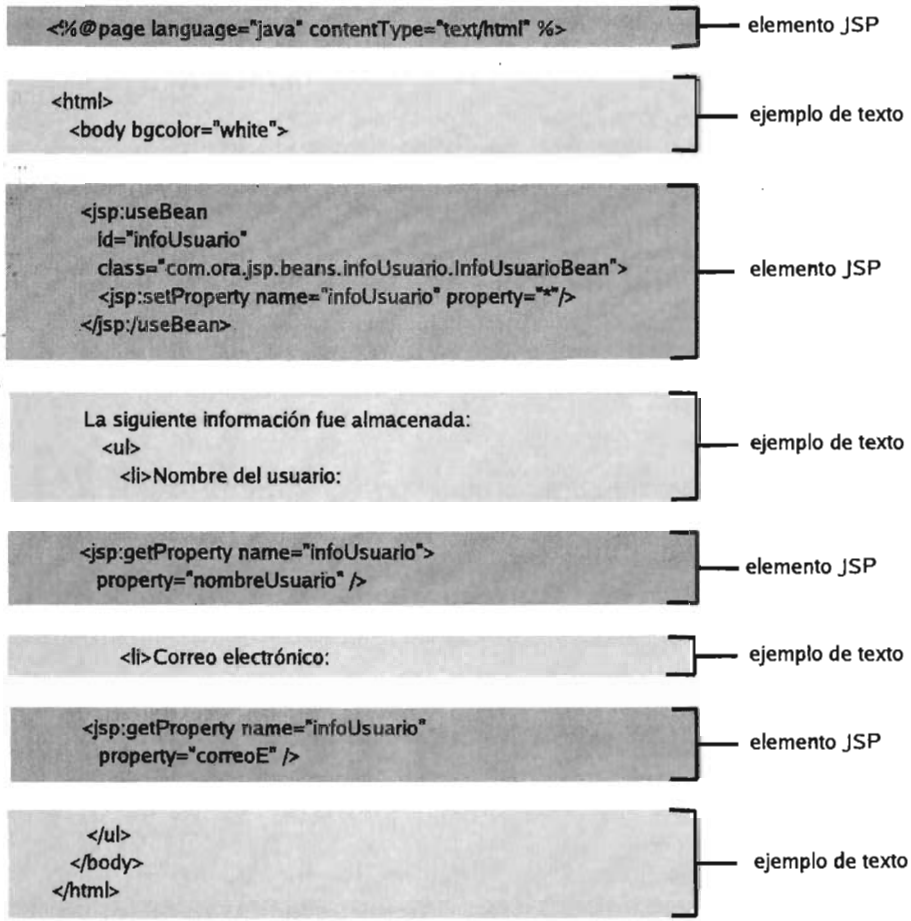

Figura 6.2: Estuctura general de un jsp

CAPÍTULO 7

JAVA DATABASE CONNECTIVITY

7.1. INTRODUCCIÓN

JDBC[1] es un API de Java que usa los manejadores de JDBC en un nivel bajo para habilitar los accesos a una base de datos a través de programas escritos en el lenguaje de programación Java.

JDBC consiste en un conjunto de clases e interfaces escritas en Java que permiten al programador enviar declaraciones SQL a un servidor de una base de datos para que se ejecuten.

Usando JDBC, es fácil trabajar, virtualmente, con cualquier Sistema Manejador de Bases de Datos (SMBD), es decir, no es necesario escribir un programa para una SMBD en particular.

El objetivo de JDBC es ser una interfaz independiente al Sistema Manejador de Bases de Datos (SMBD), un marco de trabajo para acceder a una base de datos usando SQL genérico y una interfaz uniforme para distintas fuentes de datos.

7.2. BASES DE DATOS RELACIONALES

Una *base de datos* es un conjunto de datos estructurados, almacenados en algún dispositivo de almacenamiento de datos y se puede acceder a ella desde uno o varios programas. En términos más simples una *base de datos relacional* es aquella que presenta la información en tablas[2] con renglones y columnas. Un SMBD

[1] JDBC es el nombre de una marca, no una abreviación; sin embargo, JDBC puede pensarse como *Java Database Connectivity*.

[2] Una tabla es conocida como una relación, lo cual explica el nombre de base de datos relacional.

maneja la manera en que los datos serán almacenados, mantenidos y consultados. En el caso de las bases de datos relacionales es el Sistema Manejador de Bases de Datos Relacionales (SMBDR) el que lleva a cabo estas tareas.

7.2.1. COMANDOS COMUNES DE SQL

Los comandos de SQL están divididos en categorías, las dos principales son: los comandos del *Lenguaje de Manipulación de Datos* (DML) y los comandos del *Lenguaje de Definición de Datos* (DDL). Los comandos en DML manejan los datos, ya sea para consultarlos o modificarlos y mantenerlos actualizados. Los comandos en DDL manejan los datos para crear o cambiar las tablas u otros objetos de la base de datos, tales como vistas e índices. A continuación se muestra una lista con los comandos DML más comunes:

SELECT es usado para consultar y desplegar los datos de una base de datos. La declaración SELECT especifica que columnas se incluirán en el resultado. La mayoría de los comandos SQL usados en las aplicaciones son declaraciones SELECT.

INSERT agrega nuevos renglones a una tabla. INSERT es usado para poblar una tabla recién creada.

DELETE elimina un renglón específico o un conjunto de renglones en una tabla.

UPDATE cambia un valor existente en una columna de la tabla.

Los comandos DDL más comunes son:

CREATE TABLE crea una tabla con los nombres especificados por el usuario para las columnas. También es necesario que el usuario especifique el tipo de datos de cada columna. Los tipos de datos varían, según el SMBDR que se emplee, por lo cual el usuario puede necesitar usar metadatos[3] para establecer los tipos de datos que se utilizarán en la base de datos particular.

DROP TABLE elimina todos los renglones de una tabla.

ALTER TABLE agrega o elimina una columna de una tabla.

[3]Con metadatos nos referimos a aquella información que nos dice las características propias de una base de datos.

7.2.2. CURSORES Y RESULT SETS

Los renglones que satisfacen las condiciones en una consulta son llamados *conjunto de resultados* (result set). El número de renglones en un conjunto de resultados pueden ser cero, uno o más. Una vez que se ha accedido a los datos en un conjunto de resultados, el cursor provee el significado para hacer esto. Un *cursor* puede ser pensado como un apuntador en un archivo que contiene los renglones de un conjunto de resultados, el cual tiene la habilidad de saber el renglón actual que está siendo accedido. Un cursor permite que cada renglón del conjunto de resultados pueda ser utilizado en un proceso iterativo.

Muchos de los SMBD crean un cursor automáticamente cuando un conjunto de resultados ha sido generado.

7.2.3. TRANSACCIONES

Cuando alguien está accediendo a los datos, puede suceder que otro usuario también este accediendo a los mismos datos. De ser así, si uno de los usuarios está actualizando algunas columnas en una tabla, mientras que otro está seleccionando las columnas de esa tabla, puede suceder que algunos datos tengan los anteriores valores y otros datos tengan los valores actualizados. Por ésta y otras razones, los SMBD usan *transacciones* para mantener los datos en un estado consistente y permitir que más de un usuario acceda a la base de datos al mismo tiempo.

Una *transacción* es un conjunto de una o más declaraciones SQL que forman una unidad lógica de trabajo. Una transacción termina ya sea con una confirmación o con una restauración de los datos, dependiendo de si hay problemas con la consistencia de los datos o con el acceso de varias usuarios a los datos al mismo tiempo. La *declaración de confirmación* (commit statement) hace permanentes los cambios que resultaron de varias declaraciones SQL en una transacción, y la *declaración de restauración* (rollback statement) deshace todos los cambios que resultaron de las declaraciones SQL en una transacción.

Una cerradura es un mecanismo que prohibe que dos transacciones manipulen los datos al mismo tiempo. Por ejemplo, una tabla cerrada previene que ésta sea eliminada si existe una transacción que no ha sido confirmada. Un renglón cerrado previene que dos transacciones modifiquen el mismo renglón, o que una de las transacciones seleccione un renglón mientras que otra transacción aún está modificándolo.

7.3. MODELOS DE ACCESO A UNA BASE DE DATOS EN JDBC

El API JDBC soporta los modelos de dos o tres capas para acceder a la base de datos.

Modelo de dos capas En este modelo una aplicación escrita en Java se comunica directamente con la base de datos, la cual requiere un manejador de JDBC que pueda comunicarse con un sistema manejador de la base de datos en particular. Las declaraciones de un usuario de SQL son enviadas a la base

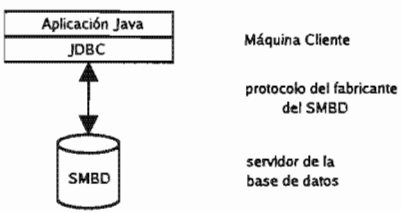

Figura 7.1: Modelo de dos capas

de datos, y los resultados de estas declaraciones son enviadas de regreso al usuario. La base de datos puede estar localizada en otra máquina a la cual el usuario esta conectado mediante una red. Esto se conoce como una configuración cliente/servidor, donde la máquina del usuario es el cliente, y la máquina que contiene a la base de datos es el servidor. Este modelo lo podemos observar en la Figura 7.1.

Modelo de tres capas En este modelo los comandos son enviados a una capa intermedia de servicios, el cual entonces envía declaraciones en SQL a la base de datos. La base de datos procesa las declaraciones SQL y envia los resultados de regreso a la capa intermedia, la cual entonces los envía al usuario. Este modelo lo podemos observar en la Figura 7.2.

7.4. ARQUITECTURA DE JDBC

Como podemos observar en la Figura 7.3, la arquitectura de JDBC está compuesta entre otras cosas por tres componentes:

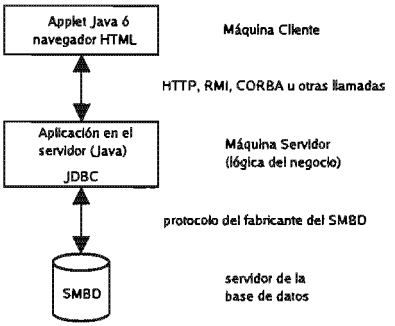

Figura 7.2: Modelo de tres capas

- API de JDBC
- Administrador de los manejadores
- Manejadores

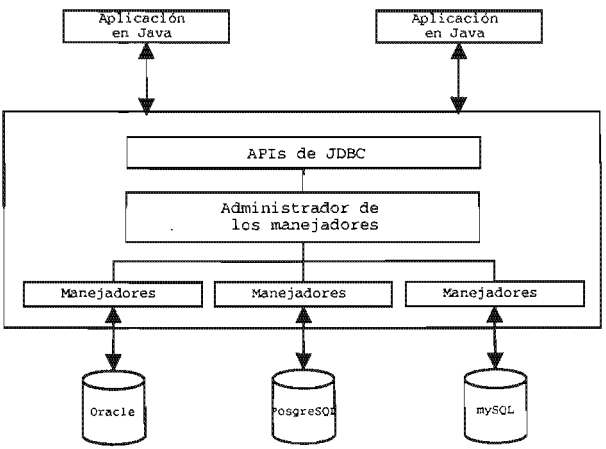

Figura 7.3: Arquitectura de JDBC

A continuación se expondran cada uno estos componentes con un mayor detalle:

API de JDBC está disponible en los paquetes java.sql y javax.sql. En la Tabla 7.1 mostramos las clases más importantes, interfaces y excepciones que se encuentran en el paquete java.sql.

Nombre de la clase	Descripción
DriverManager	Esta clase se encarga de mantener un registro de los manejadores que están disponibles y de manejar el establecimiento de una conexión entre una base de datos y el manejador apropiado. Carga a los manejadores de JDBC en memoria. Incluso puede ser usada para abrir conexiones a una base de datos.
Connection	Representa una conexion con una base de datos. También puede crear objetos del tipo Statement, PreparedStatement o CallableStatement.
Statement	Esta clase representa una declaración estática de SQL. Puede ser usada para obtener objetos de tipo ResultSet.
PreparedStatement	Representa una declaración precompilada de SQL, es una alternativa de la clase Statement para mejorar el desempeño.
CallableStatement	Provee una forma para llamar procedimientos almacenados de una manera estándar para todos los sistemas manejadores de bases de datos.
ResulSet	Representa el conjunto de resultados generados al ejecutar la declaración de SQL SELECT.
SQLExcepcion	Encapsula los errores del acceso a la base de datos.

Cuadro 7.1: API de JDBC

Administrador de los manejadores es el puente que existe entre los manejadores y los usuarios. Este componente se encarga de mantener un registro de los manejadores que están disponibles y de manejar el establecimiento de una conexión entre una base de datos y el manejador apropiado.

Manejadores de JDBC son un conjunto de clases que permiten que una aplicación Java pueda comunicarse con una base de datos en particular. Los manejadores son usados por la máquina virtual de Java para traducir las

invocaciones JDBC genéricas en invocaciones que la base de datos pueda entender. Los manejadores se cargan en ejecución.

Los manejadores JDBC se clasifican en cuatro categorías:

Tipo 1 JDBC-ODBC Este manejador debe traducir invocaciones JDBC a invocaciones ODBC a través de bibliotecas ODBC del sistema operativo o del SMBD. Éste es parte de la plataforma Java pero no es un manejador completamente en Java (ver Figura 7.4).

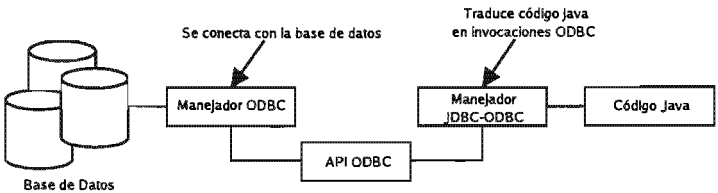

Figura 7.4: Manejador Tipo 1

Tipo 2 Conceptualmente es similar al manejador de tipo 1, excepto que se usa una capa menos (no está la capa de traducción ODBC). Cuando se realiza una invocación a la base de datos a través de JDBC, el manejador traduce el requerimiento en algo que el API del fabricante de la base de datos entiende. La base de datos procesa el requerimiento y devuelve el resultado a través de la API que lo reevía al manejador. El manejador le da formato al resultado al estándar JDBC y lo devuelve al programa (ver Figura 7.5).

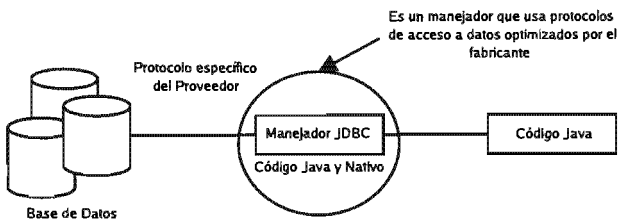

Figura 7.5: Manejador Tipo 2

Tipo 3 No es un driver, es un front-end para acceder a servidores de bases de datos. El programa envía una invocación a través del manejador de aplicaciones (*proxy*) quien lo envía a la capa intermedia (middleware), ésta completa el requerimiento usando otro manejador JDBC y habla con la base de datos a través de un manejador tipo 1 o 2 (ver Figura 7.6).

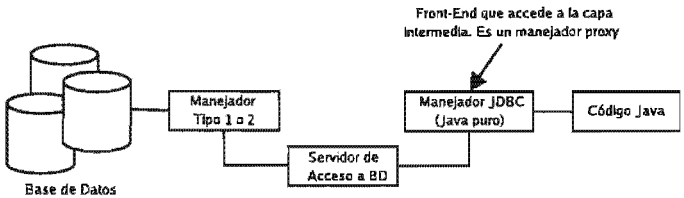

Figura 7.6: Manejador Tipo 3

Tipo 4 Es un manejador escrito en Java puro, el cual habla directamente con la base de datos, se considera que es el método más eficiente de acceso a la base de datos (ver Figura 7.7).

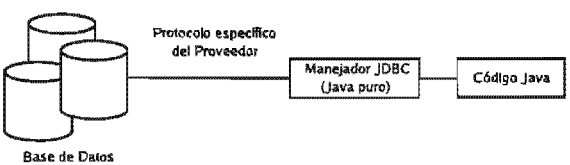

Figura 7.7: Manejador Tipo 4

7.5. USANDO JDBC

Los 7 pasos básicos para poder consultar una base de datos:

1. **Cargar el manejador de JDBC**. Simplemente se debe invocar al método *Class.forName*, pasándole como parámetro el nombre de la clase del manejador como una cadena, para cargar y registrar el manejador. Por ejemplo, en el siguiente código se carga y registra la clase *Driver*:

```
Class.forName("org.postgresql.Driver");
```

2. **Definir el URL de la conexión**. Un URL JDBC da una manera para identificar a una base de datos, tal que el manejador pueda reconocerla y establecer una conexión con ella.

   La sintáxis estándar para el URL está compuesta por tres elementos, los cuales son separados por dos puntos:

   ```
   jdbc:<subprotocolo>:<subnombre>
   ```

   A continuación describiremos cada una de estas partes:

   a) jdbc - el protocolo. El protocolo en un URL JDBC siempre es jdbc.

   b) <subprotocolo> - el nombre del manejador o el nombre del mecanismo de conexión a la base de datos, el cual debe de ser soportado por uno o varios manejadores. Un ejemplo de subprotocolo es *odbc*.

   ```
   jdbc:odbc:MiBase
   ```

   c) <subnombre>- una forma de identificar a la base de datos. El objetivo del subnombre es dar suficiente información para localizar la base de datos.

   ```
   //nombre_del_host:puerto/subsubnombre
   ```

3. **Establecer la conexión**. Para establecer una conexión con una base de datos se debe llamar al método *DriverManager.getConnection()*, y así obtener un objeto de tipo *Connection*. Este método toma una cadena que contiene el URL de la base de datos. La clase *DriverManager* se ocupa de localizar un manejador que pueda conectarse con la base de datos que se encuentra en tal URL.

   ```
   DriverManager.getConnection(URL, USER, PASSWD);
   ```

4. **Crear un objeto *Statement***. Una vez que se ha establecido una conexión a una base de datos, esta conexión puede ser usada para enviar declaraciones SQL. Un objeto *Statement* es creado con el método *createStatement* de la clase *Connection*, como se puede ver en el siguiente fragmento de código:

```
Connection con=DriverManager.getConnection(URL, USER, PASSWD);
Statement stmt=con.createStatement();
```

5. **Ejecutar una consulta**. Una vez creado el objeto *Statement*, éste se puede usar para enviar declaraciones SQL usando el método *executeQuery*, el cual regresa un objeto del tipo *ResultSet*, como se puede ver en el siguiente código:

```
String q = "SELECT col1, col2, col3 FROM miTabla";
ResultSet resultSet = stmt.executeQuery(q);
```

   Para modificar la base de datos, se usa el método *executeUpdate* en lugar del método *executeQuery*.

6. **Procesar los resultados**. La forma más simple de manejar los resultados es procesando un renglón a la vez, usando el método *next* de la clase *ResultSet*. Dentro de un renglón, la clase *ResultSet* provee varios métodos *getXxx* que toman el índice de una columna o el nombre de la columna como argumento y que regresan el elemento o valor de la columna como tipo Xxx de Java.

7. **Cerrar la conexión**. Para cerrar la conexion explícitamente, un objeto de tipo *Connection* debe llamar al método *close*.

CAPÍTULO 8

TAPESTRY

8.1. INTRODUCCIÓN

Tapestry es un marco de trabajo para el desarrollo de aplicaciones Web orientado a componentes y escrito en Java. Tapestry define un modelo de componentes que pueden ser desplegados para constituir una interfaz visual de una aplicación Web. En Tapestry todo es componente.

La filosofía de Tapestry está basada en *objetos, métodos y propiedades*. Tapestry se ocupa de las tareas relacionadas con el API de Servlets: peticiones, respuestas, sesiones, atributos, parámetros, URL's, etc.

Tapestry es un marco de trabajo que nos permite desarrollar aplicaciones Web de una forma muy similar a como se realiza con *Java Swing*, basándose en un modelo para representar a las páginas de una aplicación Web por medio de objetos. El marco de trabajo de Tapestry es una capa que se encuentra entre un contenedor de servlets (como Tomcat) y una aplicación de Tapestry.

Por lo regular las aplicaciones Web se implementan en términos de tres capas:

- La *capa de presentación* es responsable de recibir las solicitudes HTTP y crear las respuestas HTML.
- La *capa de aplicación* es responsable de toda la lógica del negocio.
- La *capa de almacenamiento* es responsable de mantener la persistencia de los datos mediante una base de datos.

Tapestry se enfoca solamente a la capa de presentación de una aplicación, es decir, Tapestry se encarga de la presentación de los datos para el usuario final y de manejar las entradas del usuario mediante el uso de ligas y formas HTML.

8.2. PÁGINAS Y COMPONENTES DE TAPESTRY

Una aplicación Tapestry está compuesta por un conjunto de páginas donde cada página está formada por componentes y cada componente, a su vez, puede estar formado por otros componentes, sin que exista ningún límite en la profundidad de anidamiento. Las páginas en Tapestry son componentes especiales en sí mismas, pero con algunas responsabilidades especiales.

Todos los componentes de Tapestry, así como las páginas (que no dejan de ser unos componentes *especiales*) están formados en términos generales por:

- Una *especificación* es un documento XML en el que se indica el nombre del componente, qué parámetros (excepto en las páginas) y propiedades posee y de qué tipo son, por qué componentes está formado y cómo se conectan los parámetros de dichos componentes, entre otras cosas.
- Una *plantilla* está formada en su mayoría por etiquetas HTML y un atributo *jwcid* (Java Web Component Id) que añade Tapestry, el cual es único e indica que la etiqueta en la que aparace representa a un componente. Por lo tanto en la plantilla se define la apariencia visual del componente, situando los componentes contenidos en él en la posición deseada.
- Una *implementación* Cada componente tiene una clase (por ejemplo un JavaBean) que implementa su comportamiento, conectando propiedades a esta clase y ejecutando los métodos adecuados cuando se produzcan los eventos.

Podemos clasificar a los componentes en: *Implícitos* y *Declarados*.

Implícitos son componentes que provee el marco de trabajo de Tapestry. Estos componentes se pueden definir anteponeniendo el símbolo @, seguido del tipo de componente del que se trata, los cuales se pueden considerar como anónimos y entre los cuales encontramos:

- @Insert
- @Conditional
- @Form
- @Body
- etc.

Declarados Son aquellos que son definidos por el programador, los cuales están compuestos por:

- Una plantilla (HTML, WML, etc.).
- Una especificación (XML).
- Una clase JavaBean.

Para entender mejor en que consiste una página y un componente, además de las diferencias entre éstos, en las siguientes subsecciones los explicaremos más a fondo .

8.2.1. PÁGINAS

Las páginas en Tapestry son un tipo *especial* de componentes, éstas son los componentes más externos, ya que no pueden estar contenidas dentro de otros componentes y por lo tanto no pueden tener parámetros. Dado que las páginas tambien son componentes éstas tienen su especificación, su plantilla y su implementación:

Especificación La especificación de una página se realiza en un archivo XML con extensión *.page*. Un ejemplo de este archivo lo podemos ver en la Figura 8.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE page-specification PUBLIC
    "-//Apache Software Foundation//Tapestry Specification 3.0//EN"
    "http://jakarta.apache.org/tapestry/dtd/Tapestry_3_0.dtd">

<page/specification class="hola.Home">
    <component id="insertaAlgunTexto" type="Insert" />
        <binding name="value" expression="algunTexto" />
    </component>
</page-specification>
```

Figura 8.1: Especificacion de una página

Plantilla La plantilla de una página está contenida en un archivo HTML en la cual se describe la apariencia visual de la página. La inserción del atributo *jwcid* en una etiqueta determinada provoca que Tapestry sustituya toda la etiqueta por el codigo HTML producido por el despliegue del componente al que

representa. Un ejemplo de una plantilla para una página lo podemos ver en la Figura 8.2.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
  <head>
    <title>Hola Mundo</title>
  </head>
  <body>
    Bienvenido a tu primer <b>aplicación Tapestry</b>
    <p>
    <h1>
      <span jwcid="insertaAlgunTexto">Este texto será reemplazado por Tapestry</span>
    </h1>
  </body>
</html>
```

Figura 8.2: Plantilla de una página

Implementación Cada página (por ser un componente) tiene una clase que la implementa. El nombre de esta clase viene dado en la especificación, como ya hemos visto. Un ejemplo de implementación lo podemos ver en la Figura 8.3.

```
package hola;
import org.apache.tapestry.html.Basepage;
public class Home extends BasePage
{
    public String getAlgunTexto()
    {
        return "Es un mundo maravilloso!";
    }
}
```

Figura 8.3: Implementación de una página

Estado de las páginas

Las páginas y los componentes que hay en ella tienen estado. Cuando hablamos de estado nos referimos a un conjunto de valores para las propiedades de las páginas.

En Tapestry la duración de la vida de cada propiedad es muy importante. Podemos distinguir tres tipo de propiedades según su duración:

Persistentes Los cambios en la propiedad son almacenados y hechos persistentes entre ciclos de petición. Las propiedades persistentes se restauran cuando la página se vuelve a cargar. Las propiedades persistentes son específicas para cada usuario.

Transitorias La propiedad se establece antes de que la página sea desligada y será reiniciada (a su valor por defecto) al final del ciclo de la petición actual.

Dinámicas La propiedad cambia incluso mientras la página se despliega, pero como ocurre en las transitorias, es reestablecida al final del ciclo de la petición actual.

Algunos ejemplos de las *propiedades persistentes* son el nombre de usuario, el producto que está siendo mostrado por una aplicación de comercio electrónico, etc. Las *propiedades transitorias* son comúnmente datos usados sólo una vez, como puede ser un mensaje de error. Las *propiedades dinámicas* están intimamente ligadas al proceso de despliegue. Por ejemplo, para mostrar una lista de elementos en un pedido, puede ser necesario tener una propiedad dinámica que tome el valor de cada elemento, como parte de un ciclo.

Páginas requeridas

Cada aplicación requiere tener al menos cinco páginas definidas con nombres específicos. Tapestry proporciona una implementación por defecto para cuatro de ellas, pero una aplicación puede sobreescribir cualquiera de ellas para proporcionar una apariencia consistente. En la Tabla 8.1 daremos una breve descripción de cada una de estas páginas:

8.2.2. COMPONENTES

El desarrollo de componentes es muy similar al visto para las páginas, con la excepción de que los componentes pueden tener parámetros y las páginas no.

Nombre de la página	Requerida	Descripción
Exception	Proporcionada por defecto, puede ser sobreescrita	Página usada para presentar al usuario las excepciones no capturadas.
Home	Debe ser proporcionada por el desarrollador	Página inicial mostrada cuado se inicia la aplicación
Inspector	Proporcionada, nunca sobreescrita	Inspector que permite a las aplicaciones Tapestry ser interrogadas sobre su estructura.
StaleLink	Proporcionada	Página mostrada cuando se lanza una excepción *StaleLinkException* durante el procesamiento de la petición. Esta excepción es lanzada cuando se detecta una URL incorrecta que no corresponde con el estado de la aplicación en el servidor.
StaleSession	Proporcionada	Página mostrada cuando se lanza una excepción StaleSession durante el procesamiento de la petición. Esta excepción es lanzada cuando se detecta que la sesión ha caducado.

Cuadro 8.1: Páginas de Tapestry

Como vimos anteriormente, los componentes estan formados por su *plantilla*, su *especificación* y su *implementación*.

Especificación En la especificación del componente se indica qué parámetros posee y de qué tipo son, además de qué componentes esta formado y como se conectan los parámetros del componente padre con los hijos. Esta especificación esta contenida en un archivo con extensión *.jwc*. Un ejemplo de un archivo de especificación de un componente lo podemos ver en la Figura 8.4.

Plantilla En la plantilla se indica en qué posición hay que desplegar cada uno de los componentes que forman el componente y las etiquetas HTML adicionales que se necesiten. Un ejemplo de una plantilla para un componente la podemos ver en la Figura 8.5.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE page-specification PUBLIC
    "-//Apache Software Foundation//Tapestry Specification 3.0//EN"
    "http://jakarta.apache.org/tapestry/dtd/Tapestry_3_0.dtd">

<component-specification allow-informal-parameters="no" allow-body="no"
   class="components.Escritor">
   <parameter name="texto" type="java.lang.String" required="yes" />
   <component id=""insertaTexto" type="Insert">
      <inherited-binding name="value" parameter-name="texto" />
   </component>
   <component id="insertaOtroTexto" type="Insert">
      <binding name="value" expression="otroTexto" />
   </component>
</component-specification>
```

Figura 8.4: Especificación de un componente

```
<span jwcid="insertaOtroTexto"/><b><span jwcid="insertaTexto"/></b>
```

Figura 8.5: Plantilla de un componente

Implementación En la implementación del componente creamos los métodos necesarios para resolver las expresiones indicadas usando la sintaxis de *Object Graph Navigation Language* (OGNL) en la especificación. Un ejemplo de una implementación para un componente la podemos ver en la Figura 8.6.

```
package componentes;
import org.apache.tapestry.html.Basepage;
public class Escritor extends BaseComponent
 {
       public String getOtroTexto()
        {
             return "Este es el texto: ";
        }
 }
```

Figura 8.6: Implementación de un componente

Parámetros y ligaduras

Los componentes de Tapestry están diseñados para trabajar unos con otros, dentro del contexto de una página y una aplicación. Cada componente tiene un conjunto específico de parámetros, los cuales tienen un nombre, un tipo y pueden ser obligatorios u opcionales.

Los parámetros definen el tipo de valor necesitado, pero no el valor actual. El valor lo proporciona un objeto especial llamado *ligadura* (binding). La ligadura es un puente entre el componente y el valor del parámetro, exponiendo ese valor al componente cuando lo necesita.

Cuando un componente necesita el valor de uno de sus parámetros, debe obtener la ligadura corecta, que es una instancia de la interfaz *IBinding*, e invocar métodos sobre ella para obtener el valor. Se usan métodos adicionales con los parámetros de salida para actualizar la propiedad de la ligadura.

En la mayoría de los casos, Tapestry puede ocultar las ligaduras al desarrollador, automatizando el proceso de obtención de la ligadura, obtención del valor a partir de ella y asignación de ésta a una propiedad JavaBean del componente.

Hay dos tipos de ligaduras: *estática* y *dinámica*. Las ligaduras estáticas son sólo de lectura y su valor es dado en la especificación del componente. Las ligaduras dinámicas son más usuales y útiles. Una ligadura dinámica usa el nombre de una propiedad JavaBean para obtener el valor cuando el componente lo necesita. La fuente de estos datos es una propiedad de algún componente.

De hecho, las propiedades dinámicas usan rutas de propiedades, permitiendo a una ligadura navegar profundamente a través de una gráfica de objetos para acceder al valor que necesita. Para este cometido se usan las expresiones proporcionadas por la librería *OGNL*.

8.3. CLASES DE TAPESTRY

Aunque el marco de trabajo de Tapestry está compuesto por más de 400 clases, por lo regular es necesario enfocarse solo a unas cuantas, las cuales se muestran en la Figura 8.7:

A continuación daremos una breve descripción de estas clases e interfaces:

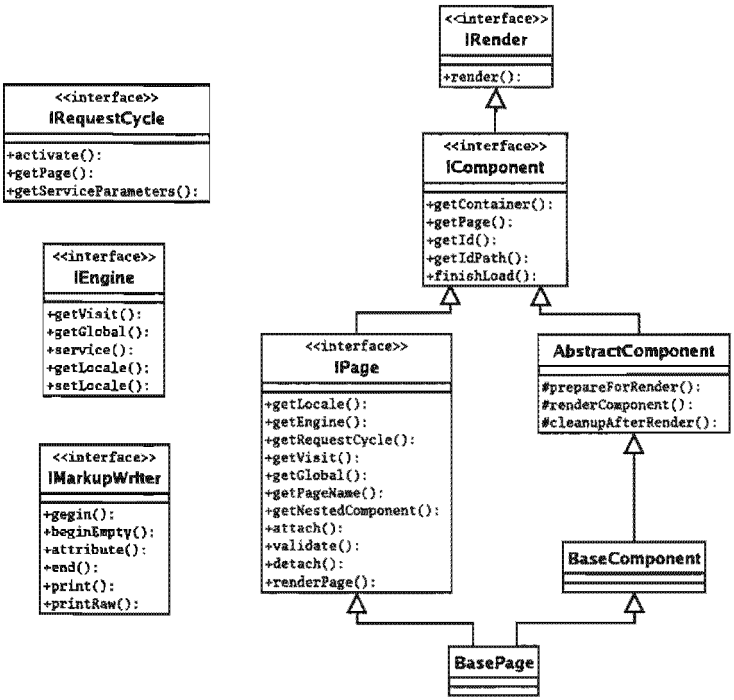

Figura 8.7: Clases de Tapestry

IRender Esta interfaz es implementada por todos los objetos que quieren desplegar HTML.

IComponent Esta interfaz se encarga de representar a los componentes de Tapestry, la cual es usada consistentemente a través del marco de trabajo como un tipo de parámetro o el valor de regreso.

IPage Esta interfaz se encarga de representar a las páginas de Tapestry.

AbstractComponent Esta clase es la base para implementar componentes. Como su nombre lo dice es abstracta, por lo cual define pero no implementa al método *renderComponent()*, solo sus subclases implentan este método, produciendo todas las salidas HTML en código Java.

BaseComponent Esta clase extiende a la clase *AbstractComponent* añadiéndole la inicialización lógica que sabe como localizar y leer una plantilla.

BasePage Esta clase es subclase de *BaseComponent*. Cuando se crean nuevas páginas siempre se deberá crear subclases de *BasePage*.

IRequestCycle Esta interfaz almacena información con respecto a la solicitud actual, manteniendo un registro de las páginas activas relacionadas en la solicitud y es usada para organizar el despliegue de una respuesta. Esta clase también puede ser usada para acceder a los objetos del API de Servlet (*HttpServletRequest, HttpSession y HttpServletResponse*) cuando sea necesario.

IMarkupWriter Esta interfaz es usada para producir salida HTML cuando una página está desplegando una respuesta. Actúa como *java.io.PrintWriter*, pero ésta incluye métodos adicionales útiles para crear salidas en un lenguaje de marcado (XML).

IEngine Esta interfaz es responsable de mantener el estado del lado del servidor, pero también actúa como un puente para un número de subsistemas usados internamente por Tapestry.

Engine Esta clase implementa la interfaz *IEngine*. Es el objeto principal encargado de mantener el estado del servidor. El *Engine* se almacena como un atributo de la clase HttpSession.

Una aplicación Tapestry tiene un único servlet: *ApplicationServlet*. Cuando llega una petición de un cliente, éste localizará el Engine asociado a la sesión del usuario que realiza la petición. Una vez localizado el Engine, el *ApplicationServlet* delegará en él la responsabilidad de responder a la petición. Para realizar su trabajo, el Engine localizará el servicio (entidad encargada de generar y responder a las *URL*'s) que generó la URL y le solicitará que responda a la petición.

8.4. OGNL

Object Graph Navigation Language (OGNL) es un lenguaje que nos permite navegar a través de objetos Java mediante una sintáxis concisa que permite especificar, donde es posible, los valores a obtener o modificar. El lenguaje especifica una sintáxis para proveer una abstracción de alto nivel para navegar las gráficas de objetos.

En Tapestry OGNL sirve para ligar los componentes Web y los objetos que se encuentran en la capa del modelo. Algunas características de OGNL son:

Proyección Esto es cuando se le aplica el mismo método a cada elemento de una colección y se construye una nueva colección con los resultados.

Selecciones Esto es cuando se le aplica un filtro a una colección para obtener un subconjunto.

Expresiones Lambda En pocas palabras, son funciones que pueden ser reusadas.

Accesos a métodos y atributos estáticos, y constructores de clases.

8.4.1. SINTÁXIS

Una propiedad en OGNL es lo mismo que una propiedad en un *JavaBean*, lo cual significa que un par de métodos para tomar y cambiar (get/set), o alternativamente un atributo, definen una propiedad.

La unidad fundamental de una expresión OGNL es una cadena para navegar. La cadena más simple consiste de las siguientes partes:

- Nombres de las propiedades.
- Llamadas a métodos.
- Arreglo de índices.

Cualquier expresión en OGNL puede ser parte de una cadena de navegación, separada por un punto (.).

Todas las expresiones en OGNL son evaluadas en el contexto del objeto actual, y una cadena simplemente usa el resultado de la liga anterior en la cadena del objeto actual para el siguiente. Un ejemplo de cadena se muestra en la Figura 8.8:

```
nombre.toCharArray()[0].numericValue().toString();
```

Figura 8.8: Una cadena en OGNL

Muchos de los operadores de OGNL son los mismos de Java, aunque OGNL se encarga de añadir algunos nuevos. Estos operadores se comportan igual que en Java, excepto por el detalle de que OGNL es un lenguaje sin tipos, es decir, OGNL procura darle a cada valor su significado de acuerdo a la situación en que se esté utilizando.

La siguiente lista muestra algunos de los operadores más importantes de OGNL:

- Secuencia (,)
- Asignación (=)
- Condicional (?:)
- Lógicos (|| o, && y, ! no)
- Lógico excluyentes (| bor, & band, ^xor,)
- Igualdad (== eq, != neq)
- Comparación (< lt, <= lte, > gt, >= gte, in, not in)
- Corrimiento de bits (<< shl, >> shr)
- Aritméticos (+, -, *, /, %)
- Pertenencia a una clase (instanceof)
- Proyección (e1.{e2})
- Selección (e1.{?e2})

CAPÍTULO 9

HIBERNATE

9.1. INTRODUCCIÓN

Hibernate es un entorno de trabajo que tiene como objetivo facilitar la persistencia de objetos Java en bases de datos relacionales y al mismo tiempo la consulta de estas bases de datos para obtener objetos. Dado esto el desarrollador solo tendrá que encargarse de la lógica del negocio.

9.2. PROBLEMÁTICA ENTRE EL MODELO RELACIONAL Y EL MODELO ORIENTADO A OBJETOS

Cualquier persona que se dedique al desarrollo orientado a objetos se encontrará con la problemática de implementar la persistencia de objetos en un soporte relacional.

Por ejemplo supongamos que tenemos la base de datos de una escuela, con las siguientes características:

- Tenemos la lista de personas que pertenecen a la escuela.
- Cada persona pertenece a una sola categoría (profesor, alumno, personal administrativo).
- En la escuela existen tres niveles: primero, segundo y tercero.
- Cada nivel tiene tres clases: A, B, C.
- Cada clase tiene una lista de personas.

En el diagrama de la Figura 9.1 se muestra la relación que existe entre las entidades identificadas.

El diseño orientado a objetos con el que nos gustaría trabajar no es igual al diseño relacional. La Figura 9.2 muestra nuestro diseño con un diagrama UML.

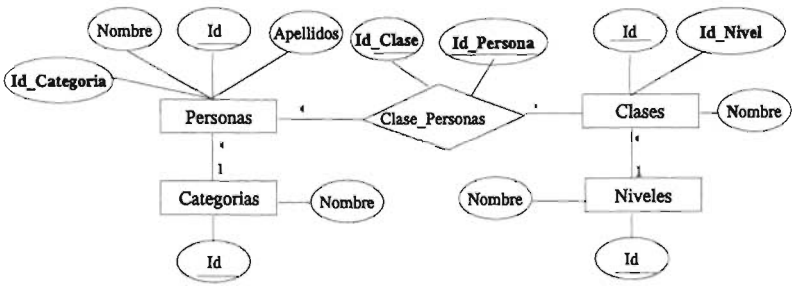

Figura 9.1: Diagrama entidades

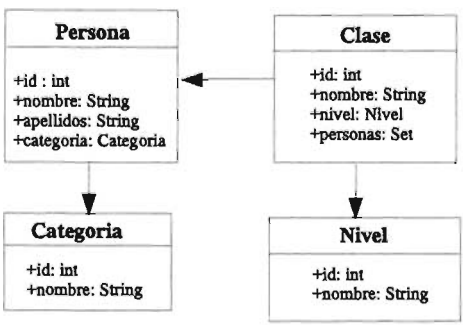

Figura 9.2: Diagrama clases

Algunas de las diferencias que podemos ver entre los diagramas anteriores son:

- Que una instancia de un objeto corresponde con un renglón de una tabla, es decir en el ejemplo la tabla *Personas* es un conjunto de personas mientras la clase *Persona* representa una sola persona.
- Que los tipos de las propiedades de un objeto no siempre corresponden con los tipos de las columnas de las tablas por ejemplo la clase *Persona* no tiene un entero que apunte a la tabla *Categorias*, mas bien tiene un objeto de tipo *Categoria*.
- Que en el modelo orientado a objetos no es necesario crear un objeto puente

que relacione los objetos. Como podemos ver en el diagrama de clases no existe un objeto *ClasePersonas*.

9.3. MAPEO OBJETO/RELACIONAL

Hibernate surge como una solución a la problemática planteada en la sección anterior, por lo cual se encarga de casar los dos modelos de manera que nosotros trabajemos desde Java con el modelo orientado a objetos. La información necesaria para que Hibernate mapee estos dos modelos es:

- Qué está detrás del modelo relacional (el sistema manejador de bases de datos –SMBD–, la base de datos, etc.).
- Cómo se mapean las propiedades y campos de las tablas (las propiedades que corresponden con las llaves primarias, los tipos de las propiedades y las columnas, el manejo de las relaciones entre las tablas, etc.).

Para proporcionarle esta información a Hibernate se utilizan dos archivos distintos:

1. El archivo de configuración de Hibernate (hibernate.cfg.xml) es el encargado de determinar los aspectos relacionados con el SMBD y las conexiones con él.
2. Los archivos que definen el mapeo (mapping) de propiedades con tablas y columnas (*.hbm.xml).

9.3.1. ARCHIVO DE CONFIGURACIÓN DE HIBERNATE

La información mínima que debe contener este archivo (ver ejemplo en la Figura 9.3):

- El dialecto SQL con el que Hibernate se comunicará con el SMBD.
- La información necesaria para poder establecer una conexión.
- Los archivos que hacen el mapeo.

```
...
<property name="connection.datasource">java:comp/env/jdbc/MiBase</property>
<property name="dialect">net.sf.hibernate.dialect.PostgreSQLDialect</property>
<mapping resource="miPaquete/Clase.hbm.xml"/>
...
```

Figura 9.3: hibernate.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<DOCTYPE hibernate-mapping PUBLIC '-//Hibernate/Hibernate Mapping DTD 2.0//EN'
   'http://hibernate.sourceforce.net/hibernate-mapping-2.0.dtd'>
<hibernate-mapping>
   <class name="Categoria" table="categorias">
      <id name="id" type="integer" column="ID">
         <generator class="identity"/>
      </id>
      <property name="nombre" column="NOMBRE" type="string"/>
   </class>
</hibernate-mapping>
```

Figura 9.4: Archivo de mapeo (*.hbm.xml)

9.3.2. ARCHIVO DE MAPEO

La información que debe contener este archivo se muestra en la Figura 9.4, la cual se describe posteriormente.

Cabecera XML Es similar a la que contienen todos los documentos xml (ver cabecera de la Figura 9.4).

El elemento <hibernate-mapping> Después de la cabecera sigue la descripción del mapeo, que está delimitada por el elemento *<hibernate-mapping>*. Este elemento permite especificar mediante atributos diversas características tal como el esquema de la base datos.

El elemento <class> Dado que se quiere mapear una clase con una tabla resulta necesario especificar cuales son. Esto lo hacemos utilizando al elemento *<class>* (ver Figura 9.5).

```
<class name="Categoria" table="categorias">
```

Figura 9.5: Elemento <class>

El elemento <id> Una vez mapeados el objeto y la tabla hay que mapear las propiedades del objeto con las columnas de la tabla. Para describir este mapeo se utilizan dos elementos XML distintos: <id> y <property>.

El elemento <id> mapea una de las propiedades del objeto con la llave primaria de la tabla (ver Figura 9.6).

```
<id name="id" type="integer" column="ID">
```

Figura 9.6: Elemento <id>

Donde el atributo *name* nos permite especificar el nombre de la propiedad con la que se mapeará la llave primaria especificada con el atributo *column*. Si el nombre de la propiedad y el nombre de la columna coinciden el atributo *column* puede omitirse. El atributo *type* sirve para especificar el tipo de datos de la propiedad con la que estamos trabajando.

El subelemento <generator> Nos permite definir como se generan los valores de las llaves primarias (ver Figura 9.7).

```
<generator class="identity"/>
```

Figura 9.7: Subelemento <generator>

El elemento <property> Sirve para mapear aquellas propiedades del objeto que no forman parte de la llave primaria, con las correspondientes columnas de una o más tablas (ver Figura 9.8).

```
<property name="nombre" column="NOMBRE" type="string"/>
```

Figura 9.8: Elemento <property>

Como en el caso del elemento *id*, el atributo *name* indica el nombre de la propiedad, el atributo *column* el nombre de la columna en la tabla y el atributo *type* nos indica el tipo de Hibernate.

9.4. USO DE HIBERNATE

Una vez que se especificó como se mapean los objetos y las tablas de la base de datos, ahora veremos como utilizar Hibernate en una aplicación Java.

Antes que nada para que una clase pueda beneficiarse de las ventajas de Hibernate, es necesario que cumpla con una condición de los Java Beans: las propiedades del objeto persistente deben ser accesibles mediante métodos *get* y *set*.

En cualquier aplicación que use Hibernate aparecen cuatro objetos básicos:

Configuration: Es el objeto que contiene la información necesaria para conectarse a la base de datos. Es el encargado de leer el archivo *hibernate.cfg.xml*.

SessionFactory: Es una fábrica de *Sessions*. Un objeto *Configuration* es capaz de crear una *SessioFactory* ya que tiene toda la información necesaria.

Session: Es la principal interfaz entre la aplicación Java e Hibernate. Es la que mantiene las conversaciones entre la aplicación y la base de datos. Permite añadir, modificar y borrar objetos en la base de datos.

Transaction: Permite definir unidades de trabajo.

Como sabemos las cuatro operaciones básicas que podemos querer realizar con los objetos persistentes son: Altas, Bajas, Actualizaciones y Consultas, por lo tanto a continuación explicaremos como se pueden llevar a cabo estas operaciones en Hibernate, excepto las consultas dado que éstas están implícitas en las otras operaciones.

9.4.1. ALTAS

Para dar de alta a un objeto en la base de datos hay que seguir los siguientes pasos (ver ejemplo de la Figura 9.9):

1. Crear una instancia del objeto persistente.
2. Crear una instancia del objeto *Session*.
3. Iniciar una transacción.
4. Salvar al objeto persistente mediante el método *save* del objeto *Session*.
5. Confirmar la transacción.
6. Si ha ocurrido alguna excepción, cancelar la transacción y cerrar la transacción.
7. Cerrar la sesión.

```
public void alta(String nombre){
    Categoria categoria=new Categoria();
    categoria.setNombre(nombre);
    Session sesion=getSession();
    if(sesion==null){
        System.out.println("Error al abrir la sesion");
        return;
    }
    Transaction tx=null;
    try{
        tx=sesion.beginTransaction();
        sesion.save(categoria);
        tx.commit();
    }catch(hibernateException e){
        if(tx!=null){
            try{
                tx.rollback();
            }catch(HibernateException e){}
        }
        finally{
            try{
                sesion.close();
            }catch(HibernateException e){}
        }
    }
}
```

Figura 9.9: Dar de alta a un objeto persistente

9.4.2. BAJAS

El esquema general para dar de baja un objeto persistente es básicamente el mismo que para darlo de alta (ver Figura 9.10).

```
public void baja(int idCategoria){
    [...]
    Categoria categoria=new Categoria();
    categoria.setId(idCategoria);
    sesion.beginTransaction();
    sesion.delete(categoria);
    sesion.commit();
    [...]
}
```

Figura 9.10: Dar de baja a un objeto persistente

Existen tres estrategias para borrar los registros de un objeto persistente. Uno de ellos puede ser usando una consulta basada en el identificador de la llave primaria (ver Figura 9.11).

```
public void baja(int idCategoria){
    [...]
    sesion.beginTransaction();
    Categoria categoria=(Categoria)sesion.load(Categoria.class,
                                               new Integer(idCategoria));
    sesion.delete(categoria);
    sesion.commit();
    [...]
}
```

Figura 9.11: Dar de baja a un objeto persistente utilizando load

Donde el método *load()* de la clase *Session* nos devuelve un objeto a partir de un identificador (llave primaria) y la clase que está mapeada a la tabla en la que debe buscar.

Y la tercer estrategia (ver Figura 9.12) consiste en que el método *delete()* reciba como parámetro una consulta en HQL (ver sección 9.5).

```
[...]
String sel="FROM Categoria AS C WHERE C.idCategoria="+idCategoria;
sesion.delete(sel);
[...]
```

Figura 9.12: Dar de baja a un objeto persistente usando una consulta HQL

9.4.3. ACTUALIZACIONES

Como es de esperarse las modificaciones siguen el mismo esquema que las altas y las bajas (ver Figura 9.13).

9.5. HQL

En el modelo relacional disponemos del SQL (*Standard Query Language*) que nos permite obtener información haciendo consultas basadas en las tablas y columnas de la base de datos. El equivalente en modelo orientado a objetos es HQL

```
public void actualizar(int idCategoria){
    [...]
    sesion.beginTransaction();
    Categoria categoria=(Categoria)sesion.load(Categoria.class,
                                               new Integer(idCategoria));
    categoria.setNombre("Una nueva categoria");
    sesion.update(categoria);
    sesion.commit();
    [...]
}
```

Figura 9.13: Actualizar a un objeto persistente

(*Hibernate Query Language*), el cual nos permite hacer consultas basadas en los objetos y sus propiedades.

Hibernate traduce las consultas que hacemos desde el modelo orientado a objetos en HQL al lenguaje de consulta del modelo relacional SQL y transforma los resultados obtenidos en el modelo relacional (renglones y columnas) en aquello que tiene sentido en el modelo orientado a objetos.

En una aplicación JDBC tradicional, si cambiamos el SMBD y no hemos tenido cuidado con las instrucciones SQL, nos podemos ver obligados a tener que adaptar las instrucciones SQL a las peculiaridades del dialecto del nuevo sistema manejador.

En una aplicación con Hibernate, el problema desaparece. Sólo hay que cambiar el *dialecto* en el archivo *hibernate.cfg.xml* e Hibernate se encargará de traducir el HQL al dialecto SQL correspondiente al SMBD.

9.5.1. Consultas en HQL

El principal uso de HQL es el de consultar los objetos mapeados a las tablas de la base de datos. Para definir estas consultas, HQL nos proporciona, entre otras cosas, las *Funciones escalares*, los *Listados de colecciones*, la cláusula *Group by*, etc.

9.5.2. Ejecución de consultas

Existen diferentes maneras para ejecutar consultas elaboradas con HQL:

- Pasando como parámetro la consulta HQL al método find() del objeto *Session*.

Funciones escalares	
Función	**Ejemplo**
COUNT	SELECT COUNT(*) FROM Persona P
AVG	SELECT AVG(P.id) FROM Persona P
SUM	SELECT SUM(P.id) FROM Persona P
MAX	SELECT MAX(P.id) FROM Persona P
MIN	SELECT MIN(P.id) FROM Persona P

Listado de colecciones	
Función	**Ejemplo**
SELECT	SELECT C.nivel FROM Clase C
elements	SELECT elements(C.personas) FROM Clase C

Cláusula Group By	
Función	**Ejemplo**
GROUP BY	SELECT C.nivel, C.nombre, count(elements(C.personas)) FROM Clase C GROUP BY C

- Pasando como parámetro la consulta HQL al método iterate() del objeto *Session*.
- Obteniendo un objeto *Query* utilizando el método *createQuery()* del objeto *Session*.

Parte II

Sistema de Inventario Personal

CAPÍTULO 10

LANZAMIENTO

10.1. INTRODUCCIÓN

Como sabemos, los profesores de la Facultad prestan sus libros, sus tesis, sus discos compactos, etc. a los alumnos o a otros profesores pero estas pertenencias no siempre regresan a ellos dado que no llevan un registro de qué y cuántas pertenencias tienen y a quiénes se las prestaron. Es por esto que se busca proveer a los profesores de una aplicación Web[1] en la que ellos puedan llevar un inventario de sus pertenencias, un registro de la personas que le han solicitado una pertenencia, de los préstamos que se le realicen al solicitante y de las devoluciones que el solicitante haga de las pertenencias que se le han prestado, así como del envío de mensajes para solicitar la devolución de alguna pertenencia prestada.

Dado que cada profesor posee distintas pertenencias tales como libros, revistas, tésis, etc, la aplicación le permitirá definir la categoría en que caen sus pertenencias y los atributos que desea almacenar de éstas, es decir un profesor puede definir la categoría artículos publicados con atributos fecha de publicación, título y editorial, mientras que otro profesor a la mejor quisiera almacenar sus discos compactos con los atributos intérprete, disquera y ubicación.

En esta segunda parte nos enfocaremos a documentar y explicar como se llevó a cabo el desarrollo del Inventario Personal para los profesores de la Facultad de Ciencias siguiendo la *Programación Extrema*.

[1] El término aplicación Web se refiere aquellas aplicaciones accedidas a través de la Internet usando un navegador.

10.2. ETAPA INICIAL

El objetivo del sistema es el de proporcionar una aplicación Web para los profesores de la Facultad de Ciencias, el cual les permita mantener un inventario de sus pertenencias. Las principales funcionalidades con las que contará el sistema son:

1. Configuración y control de Categorías.
2. Configuración y control de Pertenencias.
3. Configuración y control de Solicitantes.
4. Configuración y control de Usuarios.
5. Control y consulta de los préstamos y devoluciones de las pertenencias.
6. Envío de mensajes para la solicitud de devolución de las pertenencias.

10.3. MODELO DEL SISTEMA

Como se ha visto el patrón de diseño más recomendado para el desarrollo de aplicaciones gráficas es el *Modelo-Vista-Controlador* (MVC). El patrón MVC define la interacción del usuario dentro de la aplicación en tres categorías de objetos:

Modelo El objeto modelo es usado para almacenar la información que será presentada al usuario (y posiblemente editada). Estos objetos se denominan como los objetos del dominio (objetos que representan el dominio específico de la aplicación, como Categoria, Usuario, etc.). El Modelo debe ser completamente independiente de la interfaz gráfica.

Vista El objeto vista es responsable de presentar la información obtenida del modelo en un formato apropiado para la aplicación.

Controlador El objeto controlador tiene dos funciones: primero, es el puente entre el Modelo y la Vista, leyendo los datos del modelo y proporcionándoselos a la vista. Segundo, es el responsable de interpretar lo que ingresa el usuario y actualizar el Modelo.

Como se puede ver en la Figura 10.1, el Modelo y la Vista no tienen una conexión directa, en lugar de eso el Controlador media entre los dos, además de tener la responsabilidad de procesar la entrada del usuario.

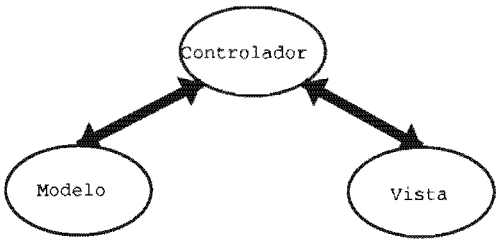

Figura 10.1: Modelo-Vista-Controlador

10.4. ESQUEMA DE TRABAJO

Para llevar a cabo la elaboración de la aplicación se seguirá la metodológia de desarrollo de software *Programación eXtrema (XP)*. Como lo indica XP el esquema de trabajo está basado en dividir el desarrollo de las historias en iteraciones. Cada iteración estará conformada por un subconjunto de historias las cuales se determinarán en lo que XP se conoce como *Juego de planeación*. A su vez el desarrollo de cada una de las historias estará compuesto por cuatro etapas *Análisis, Pruebas, Diseño y Desarrollo*.

El flujo de trabajo a seguir para el desarrollo de una iteración se explica a continuación:

1. Determinar qué historias se elaborarán en la iteración.
2. Llenar por cada historia la forma docHnIm (ver sección 10.5) que contendrá toda la documentación del desarrollo de la historia.
    a) Elegir y analizar una historia llenando la sección de *Analisis de la Historia* de la forma (ver Figura 10.2).
    b) Por cada historia se deberán diseñar pruebas, las cuales se documentan en la sección *Pruebas* de la forma (ver Figura 10.3).
    c) Por cada historia se identificarán las entidades y las relaciones entre ellas a nivel base de datos, se registrará su documentación en la sección *Entidades* (ver Figura 10.4).
    d) Elaborar el diseño de la historia para cada una de las tres capas, por lo cual se deberá llenar la sección de *Diseño* de la forma (ver Figura 10.5).

e) Identificar y documentar las tareas necesarias para resolver la historia llenando la sección de *Tareas* de la forma (ver Figura 10.6).

f) Implementar la historia correspondiente siguiendo el diseño establecido.

g) Realizar la integración de las capas.

h) Llevar a cabo la prueba documentada en la sección de *Pruebas* de la forma.

i) En caso de ser necesario se corregirán los errores encontrados.

10.5. COMPOSICIÓN DE LA FORMA DOCHNIM

La forma *docHnIm* de la historia n iteración m está compuesta por las secciones: *Análisis de la historia*, *Pruebas*, *Entidades*, *Diseño* y *Tareas*.

10.5.1. ANÁLISIS DE LA HISTORIA

Esta sección deberá contener la información que se muestra en la Figura 10.2.

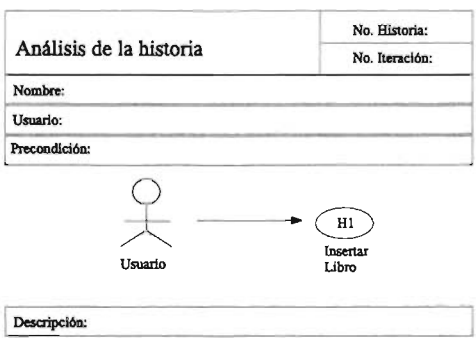

Figura 10.2: Registro de historia

10.5.2. PRUEBAS

Esta sección deberá contener la información que se muestra en la Figura 10.3.

Prueba	No. Prueba:
	No. Historia:
	No. Iteración:
Nombre:	
Precondición:	

Prueba		
Paso	Descripción	Excepción

Excepciones de la Prueba		
Id	Descripción	Acción

Postcondición:
Evaluación:
Defectos:

Figura 10.3: Registro de Pruebas

10.5.3. ENTIDADES

Esta sección deberá contener la información que se muestra en la Figura 10.4.

Entidades	No. Historia:
	No. Iteración:
Nombre:	

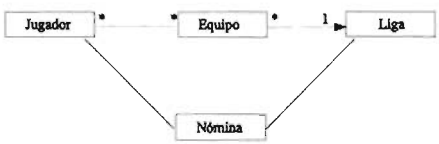

Figura 10.4: Registro de entidades y sus relaciones

10.5.4. DISEÑO

Esta sección deberá contener la información que se muestra en la Figura 10.5.

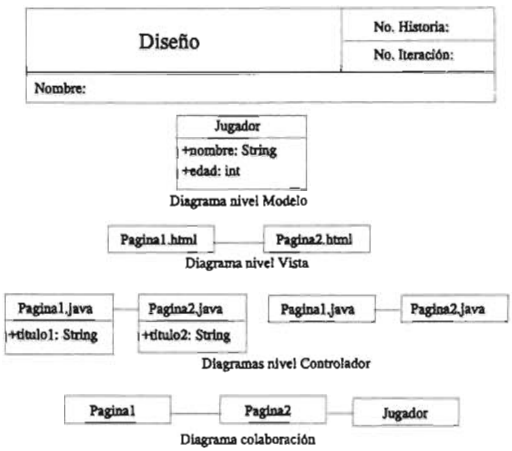

Figura 10.5: Registro de Diseño

10.5.5. TAREAS

Esta sección deberá contener la información que se muestra en la Figura 10.6. El Modelo, la Vista y el Controlador se refieren a las tareas necesarias para resolver el diseño en los niveles respectivos.

Tareas	No. Tarea:
	No. Historia:
	No. Iteración:
Nombre:	
Descripción: Modelo: Tareas para resolver nivel Modelo Vista: Tareas para resolver nivel Vista Controlador: Tareas para resolver nivel Controlador	
Observaciones:	

Figura 10.6: Registro de Tareas

10.6. TECNOLOGÍAS UTILIZADAS

Para llevar a cabo el proceso de implementación, se utilizarán las siguientes tecnologías:

- **Lenguaje**: Java (j2sdk1.4.0).
- **Contenedor de servlet**: Tomcat (jakarta-tomcat-5.0.19).
- **Marco de trabajo de aplicaciones Web**: Tapestry (Tapestry-3.0-rc-3).
- **Marco de trabajo para el manejo de la persistencia**: Hibernate (Hibernate2).
- **Sistema Manejador de Base de Datos**: Postgresql (Postgresql 7.4.3).
- **Herramienta de construcción y configuración**: Ant (Ant 1.6.1).
- **APIs**: JavaMail.
- **Herramienta de implementación**: NetBeans (NetBeans IDE 3.5.1).

CAPÍTULO 11

ITERACIÓN 1: USUARIOS

11.1. INTRODUCCIÓN

En esta primera iteración nos enfocaremos a la resolución de todas aquellas historias relacionadas con el usuario.

11.2. PLANEACIÓN DE LA ITERACIÓN

Después de varias discusiones identificamos seis historias principales: *Registrar usuario, Ingresar al sistema, Encontrar usuario, Editar usuario, Eliminar usuario y Cambiar Contraseña*, las cuales podemos ver en la Figura 11.1.

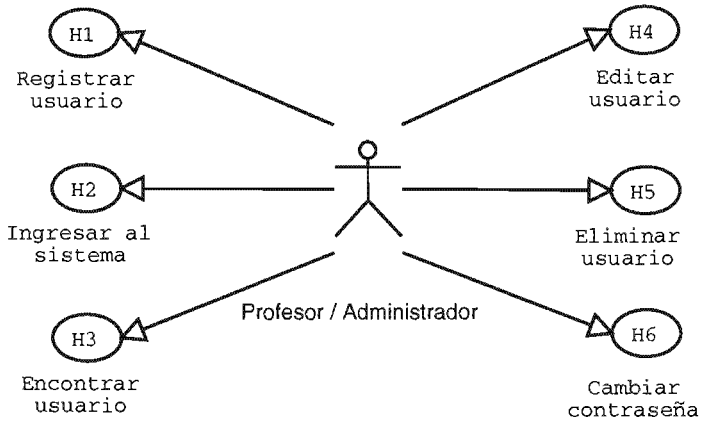

Figura 11.1: Historias de la iteración

Como podemos observar en la Figura 11.1, los actores encontrados son dos: *Profesor y Administrador.*

Cabe destacar que el usuario administrador solo se encargará de desempeñar las funciones: encontrar a los usuarios registrados en el sistema, editar y eliminar algún usuario, mientras que el usuario profesor podrá registrarse, ingresar al sistema, editar su información, entre otras (las cuales se desarrollarán en otras iteraciones).

11.3. RESOLVIENDO HISTORIA 1: *Registrar un nuevo usuario*

11.3.1. ANÁLISIS

Como sabemos este sistema está dirigido a los profesores de la facultad de ciencias, por lo cual, antes que nada, para que un usuario se pueda registrar, se requiere verificar que efectivamente el usuario que se registra es un profesor de la facultad, es por eso que primero se le solicitará su RFC que se comparará con el que se tiene registrado en las bases de datos de la facultad.

Una vez proporcionado éste, consideramos que la información necesaria para que un profesor pueda usar el sistema es su nombre, su correo electrónico, un identificador de usuario y una contraseña. Estos dos últimos serán necesarios para que el profesor pueda ingresar al sistema posteriormente.

Dado el análisis anterior llegamos a que, para resolver esta historia tenemos las subhistorias que se muestran en la Figura 11.2.

11.3.2. PRUEBAS

Esta prueba consistirá en que el usuario ingresará su nombre, su correo electrónico, un identificador de usuario y una contraseña. Para poder desarrollar esta prueba se pedirá como precondición que el usuario haya ingresado al sitio usando un navegador.

Para comprobar el buen funcionamiento de esta historia, se probarán algunos puntos claves como son:

- Que no se pueda registrar un usuario que no sea profesor de la facultad.

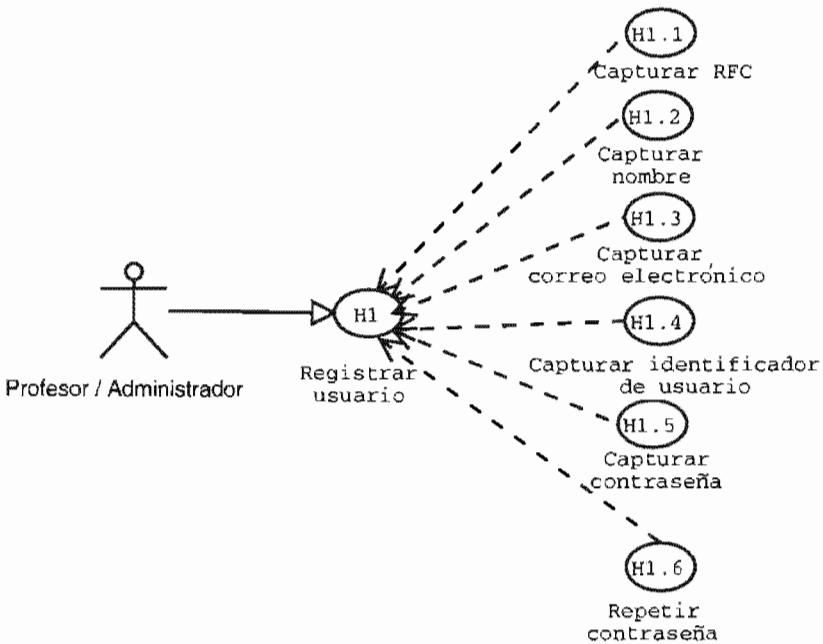

Figura 11.2: Subhistorias encontradas

- Que ingresen toda la información solicitada (nombre, correo electrónico, identificador de usuario, contraseña y repetición de la contraseña).
- Que la contraseña coincida con la repetición de la misma.

11.3.3. DISEÑO

Dado el analisis anterior, se llegó a las pantallas de las Figuras 11.3 y 11.4.

El diseño de las historias que seguiremos es el que está basado en el Modelo-Vista-Controlador, y por otro lado, las principales tecnologías utilizadas en el desarrollo del sistema son Tapestry (ver Capítulo 8) y Hibernate (ver Capítulo 9), por lo cual todo lo que se refiere al Modelo lo realizaremos con Hibernate y SMBD, lo que se refiere a la Vista lo realizaremos en lo que en Tapestry se conoce como las

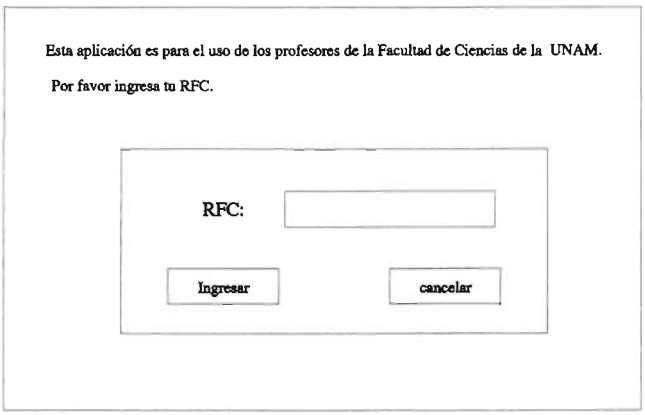

Figura 11.3: Validación de usuario.

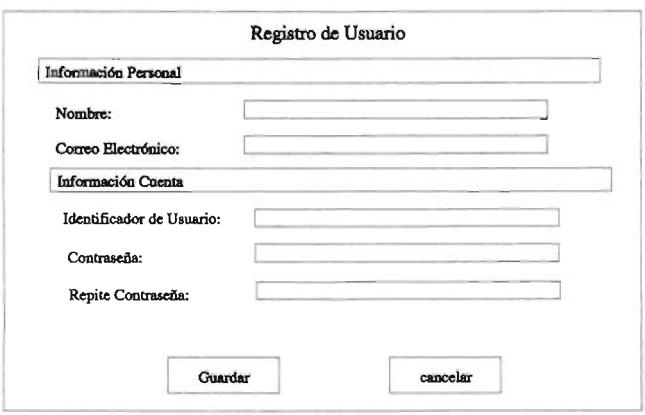

Figura 11.4: Registro de usuario.

plantillas y el Controlador lo realizaremos con lo que en Tapestry se conoce como la especificación y la implementación, y también hacemos uso de algunas clases auxiliares. A continuación mostraremos que entidades, páginas y componentes se identificaron para resolver esta historia.

Modelo: Aquí identificamos a la entidad *Usuario*, la cual podemos ver en la Figura 11.5 y la clase que modela esta entidad se muestra en la Figura 11.6. Esta clase se definirá en el documento entidad.hbm.xml [1], la cual se mapeará con la entidad *Usuario* definida en la base de datos.

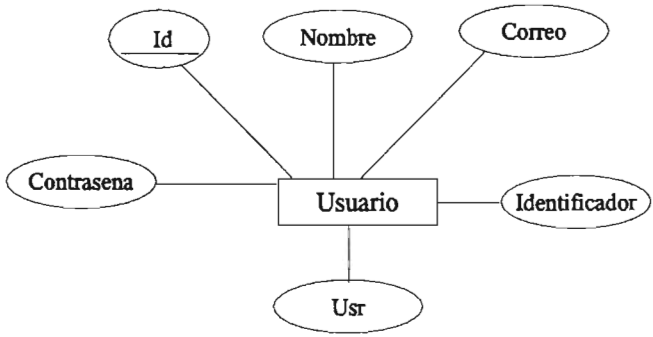

Figura 11.5: Diagrama entidades a nivel Modelo

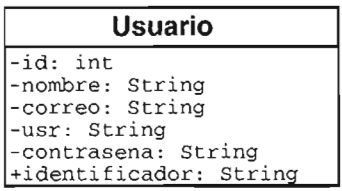

Figura 11.6: Diagrama clases a nivel Modelo

Como podemos ver, la entidad *Usuario* posee los atributos id e identificador. El atributo id es un número que se genera secuencialmente, el cual es la llave primaria de la entidad, mientras que el identificador es el atributo con el cual se verificará que el usuario realmente se encuentre en las bases de datos de la facultad.

[1] Este documento es un archivo de configuración de Hibernate, donde uno debe definir las entidades y las relaciones entre éstas, siguiendo la nomenclatura especificada en la dtd de Hibernate. Estas entidades deben mapearse a las entidades y relaciones que se definen en la base de datos.

La elección de que el atributo id sea la llave primaria, se dió porque así nos aseguramos de que sea única, además de que es más eficiente comparar enteros que cadenas completas y porque Hibernate se encarga de generarlos y manipularlos.

Como sabemos en Tapestry una página o un componente están compuestos por la plantilla, la especificación y la implementación, por lo cual a nivel *Vista* tendremos las plantillas y a nivel *Controlador* tendremos a las especificaciones e implementaciones.

Las páginas y componentes que identificamos son *EncabezadoInicial, Home, ReconoceUsuario, CapturaUsuario y CCapturaUsuario.*.

Vista: Las plantillas son *EncabezadoInicial.html, Home.html, ReconoceUsuario.html, CapturaUsuario.html y CCapturaUsuario.html* (ver Figura 11.7). Las plantillas *Home* y *EncabezadoInicial* nos sirven para

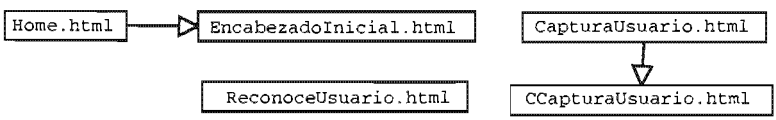

Figura 11.7: Diagrama plantilla a nivel Vista

desplegar la opción de registro. La plantilla *ReconoceUsuario* se encarga de capturar el RFC del usuario para saber si el usuario es un profesor de la facultad. La plantilla *CapturaUsuario* sirve para capturar la información del usuario.

Controlador: Tenemos las especificaciones *EncabezadoInicial.jwc, Home.page, ReconoceUsuario.page, CapturaUsuario.page y CCapturaUsuario.jwc* (ver Figura 11.8) y las implementaciones *EncabezadoInicial.java, Home.java, ReconoceUsuario.java, CapturaUsuario.java y CCapturaUsuario.java* (ver Figura 11.9).

La implementación *ReconoceUsuario* se encarga de verificar en la base de datos de la facultad que efectivamente el usuario que se va a registrar sea un profesor de la facultad.

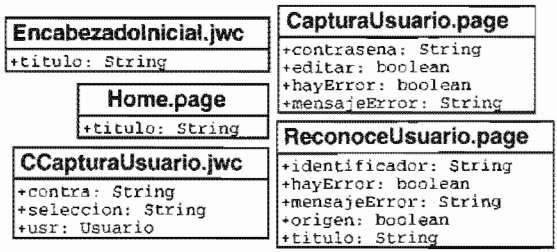

Figura 11.8: Diagrama especificación a nivel Controlador

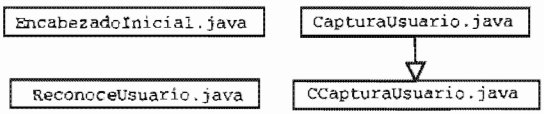

Figura 11.9: Diagrama implementación a nivel Controlador

Cabe mencionar que para llevar a cabo cualquier captura de información relacionada con la creación o edición de una entidad, se crearán el componente CCapturaEntidad (Usuario en este caso), el cual es el encargado de capturar la información, y la página CapturaEntidad (Usuario en este caso), recibe la información que el componente capturó para actualizar o crear a la entidad en la base de datos.

11.3.4. DESARROLLO

Dadas las entidades, páginas y componentes identificados anteriormente, podemos ver en la Figura 11.10 como interaccionan entre sí.

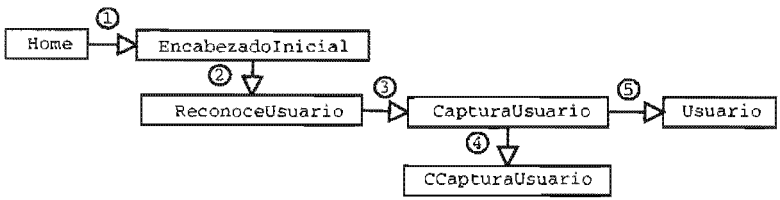

Figura 11.10: Diagrama de colaboración: Registrar un usuario

11.4. RESOLVIENDO HISTORIA 2: *Ingresar al sistema*

11.4.1. ANÁLISIS

Como se estableció en la historia anterior la información que se requiere para que el usuario ingrese al sistema son el identificador de usuario y la contraseña que proporcionó al registrarse.

Dado que tenemos dos tipos de usuario (Profesor y Administrador) cada uno de ellos tendrá acceso para poder realizar sus actividades o funciones establecidas, por ejemplo cuando el usuario administrador proporcione su identificador de usuario y su contraseña, él podrá tener acceso a la edición o eliminación de alguno de los usuarios registrados, mientras que el usuario profesor podrá tener acceso solo a su información.

Dado el análisis anterior llegamos a que, para resolver esta historia tenemos las subhistorias que se muestran en la Figura 11.11.

11.4.2. PRUEBAS

Para llevar a cabo esta prueba es necesario que el usuario ingrese su identificador de usuario y su contraseña, proporcionadas cuando se registró.

Para desarrollar esta prueba es necesario que el usuario ya se haya registrado.

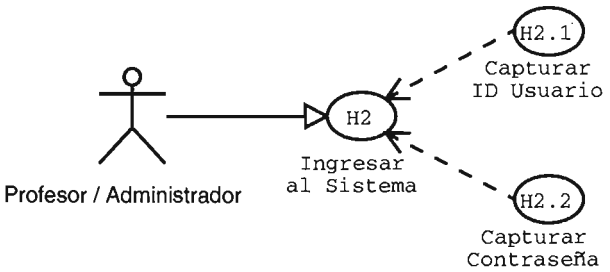

Figura 11.11: Subhistorias encontradas

Algunos puntos importantes a evaluar son:

- Que el usuario ingrese tanto su identificador de usuario como su contraseña.
- Que la contraseña corresponda con el identificador de usuario.
- Que no ingrese un usuario con un identificador de usuario y contraseña que no estén almacenadas.

11.4.3. DISEÑO

Para realizar el ingreso al sistema se estableció la pantalla de la Figura 11.12.

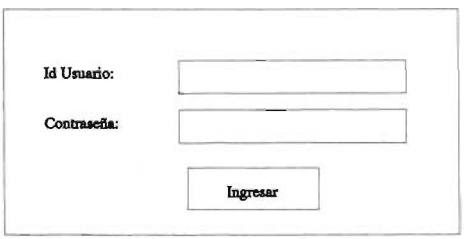

Figura 11.12: Ingresar al sistema

Modelo: La entidad identificada es *Usuario*.

Las páginas y componentes que identificamos son: *Home, EncabezadoInicial, Encabezado, Inicial, EncabezadoAdmin e InicialAdmin.*

Vista: Las plantillas son: *Home.html, EncabezadoInicial.html, Encabezado.html, Inicial.html, EncabezadoAdmin.html e InicialAdmin.html* (ver Figura 11.13).

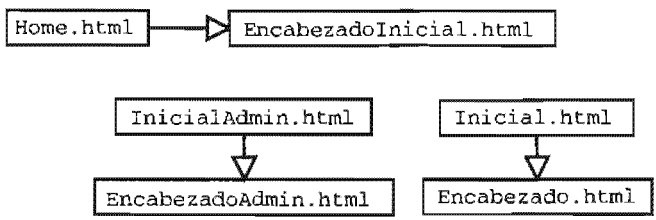

Figura 11.13: Diagrama plantilla a nivel Vista

Las plantillas *Home* y *EncabezadoInicial* estan involucradas ya que éstas son las que recibirán el identificador de usuario y la contraseña para que el usuario pueda ingresar al sistema.

Las plantillas *Inicial* y *Encabezado* nos sirven para el despliegue del menú de opciones para un usuario profesor, mientras que las plantillas *InicialAdmin* y *EncabezadoAdmin* son para el despliegue del menú de opciones del administrador.

Controlador: Tenemos las especificaciones *Home.page, EncabezadoInicial.jwc, Encabezado.jwc, Inicial.page, EncabezadoAdmin.jwc e InicialAdmin.page* (ver Figura 11.14) y las implementaciones *Home.java, EncabezadoInicial.java, Encabezado.java, Inicial.java, EncabezadoAdmin.java e InicialAdmin.java* y se creó la clase auxiliar *Visit* (ver Figura 11.15).

Donde la implementación *EncabezadoInicial* se encarga de buscar en la base de datos al usuario que tenga el identificador de usuario y la contraseña, para así redireccionar a la página *Inicial* si se trata del usuario profesor o si se trata del usuario administrador redireccionarlo a *InicialAdmin.*

Se creó el objeto *Visit* que tiene como atributo a un usuario, el cual es accesible a todas las páginas dentro de la aplicación y contiene información

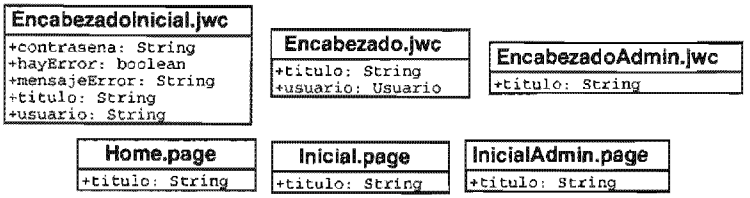

Figura 11.14: Diagrama especificación a nivel Controlador

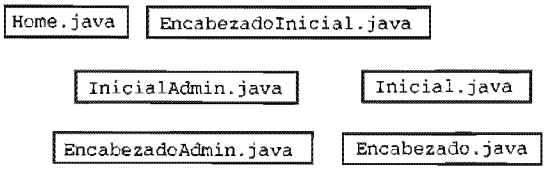

Figura 11.15: Diagrama implementación a nivel Controlador

específica de un usuario de la aplicación, es decir todas las páginas de la aplicación podrán tener acceso al usuario, apartir de que el usuario ingrese al sistema y hasta que el usuario salga de la aplicación.

11.4.4. DESARROLLO

La relación que existe entre las entidades, páginas y componentes se muestran en la Figura 11.16.

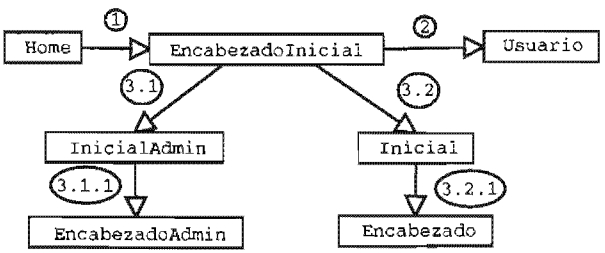

Figura 11.16: Diagrama de colaboración: Ingresar al sistema

11.5. RESOLVIENDO HISTORIA 3: *Encontrar un usuario*

11.5.1. ANÁLISIS

Lo que se pretende realizar en esta historia es que el usuario administrador pueda encontrar, seleccionar y consultar la información de un usuario.

Por encontrar entendemos que el administrador pueda:

- Ingresar información referente al usuario para realizar una búsqueda y navegar entre los resultados.
- Navegar entre todos los usuario existentes.

Si existen bastantes usuarios registrados resultará necesario mostrarle al usuario un subconjunto de usuarios a la vez, por lo cual por navegar se entiende a la acción de moverse entre las páginas que contienen a estos subconjuntos. Este comportamiento se mantendrá en toda lista que el sistema despliegue.

Las subhistorias identificadas para esta historia son las que se muestran en la Figura 11.17.

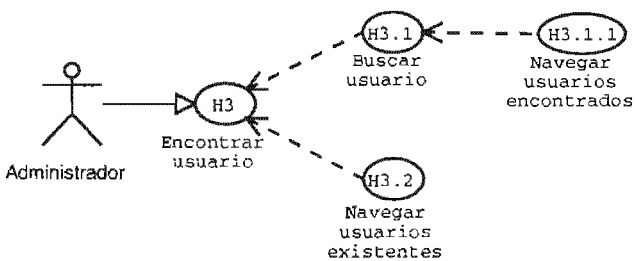

Figura 11.17: Subhistorias encontradas

11.5.2. PRUEBAS

Para poder llevar a cabo esta prueba el usuario debe de haber ingresado al sistema como usuario administrador.

El usuario administrador navegará entre todos los usuarios registrados hasta que encuentre al usuario buscado y lo seleccione.

El usuario administrador buscará a un usuario profesor en particular proporcionando información referente a éste, navegará entre los resultados de la búsqueda y seleccionará al usuario profesor.

Algunos puntos importantes a evaluar son:

- Que existan usuarios registrados.
- Que la información proporcionada corresponda a un usuario registrado.

11.5.3. DISEÑO

Para poder desarrollar esta historia, se definieron las pantallas de las Figuras 11.18 y 11.19.

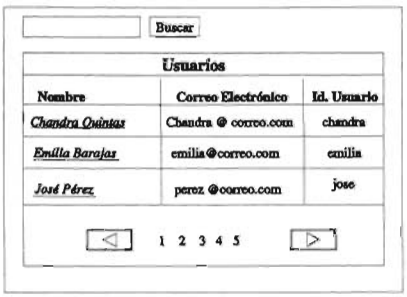

Figura 11.18: Encontrar un usuario

Modelo: La entidad identificada es *Usuario*.

Las páginas y componentes identificados son *EncabezadoAdmin, PrincipalUsuario, CBusquedaUsuario, Paginador y ConsultaUsuario*.

Vista: Las plantillas identificadas son *EncabezadoAdmin.html, PrincipalUsuario.html, CBusquedaUsuario.html, Paginador.html y ConsultaUsuario.html* (ver Figura 11.20).

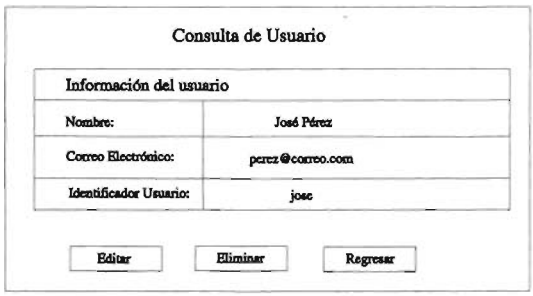

Figura 11.19: Consultar un usuario

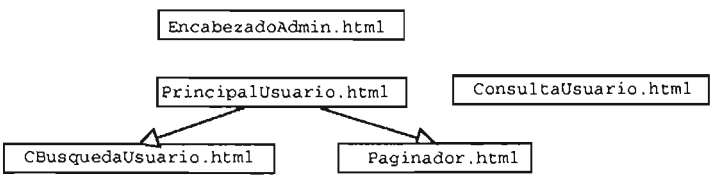

Figura 11.20: Diagrama plantilla a nivel Vista

Donde la plantilla *EncabezadoAdmin* es el que despliega la opción para ingresar a la sección de usuarios. La plantilla *PrincipalUsuario* es la encargada de desplegar a los usuarios y a los componentes *CBusquedaUsuario* y *Paginador* y la plantilla *ConsultaUsuario* es la encargada de desplegar la información del usuario seleccionado.

Controlador: Tenemos las especificaciones *EncabezadoAdmin.jwc, PrincipalUsuario.page, CBusquedaUsuario.jwc, Paginador.jwc y ConsultaUsuario.page* (ver Figura 11.21) y las implementaciones *EncabezadoAdmin.java, PrincipalUsuario.java, CBusquedaUsuario.java, Paginador.java y ConsultaUsuario.java* (ver Figura 11.22).

Donde la implementación *CBusquedaUsuario* se encarga de obtener la lista de usuarios que cumplan con la búsqueda, la implementación *Paginador* se encarga de dividir los elementos de la lista en sublistas para que la página *PrincipalUsuario* despliegue los elementos de las sublistas.

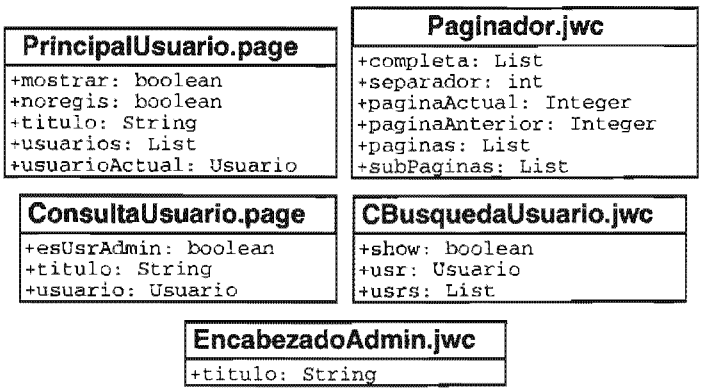

Figura 11.21: Diagrama especificación a nivel Controlador

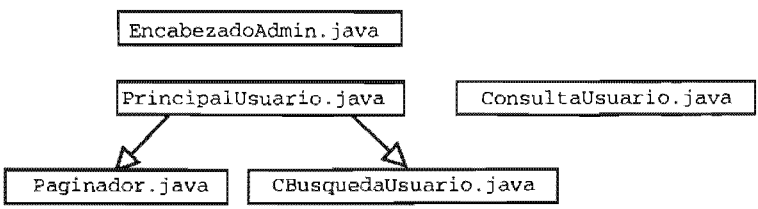

Figura 11.22: Diagrama implementación a nivel Controlador

11.5.4. DESARROLLO

Dadas las entidades, páginas y componentes identificados anteriormente en la Figura 11.23 se pueden ver las relaciones entre éstos.

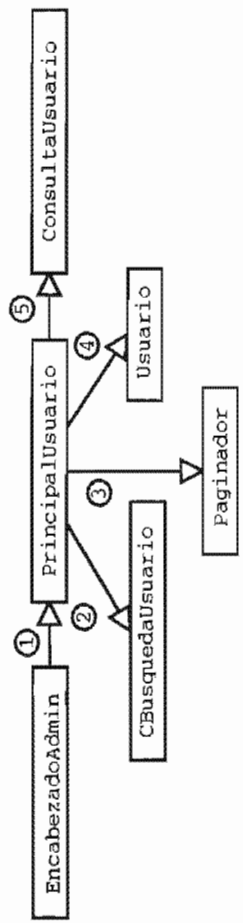

Figura 11.23: Diagrama de colaboración: Encontrar un usuario

11.6. RESOLVIENDO HISTORIA 4: *Editar la información de un usuario*

11.6.1. ANÁLISIS

La edición de la información referente a un usuario la puede realizar tanto el usuario profesor como el usuario administrador. Cuando se trata del usuario administrador éste debe encontrar al usuario que desea editar, es decir puede modificar a cualquiera de los usuarios registrados, mientras que el usuario profesor solo puede editar su información desde su menú de opciones.

La información que se puede editar de un usuario es nombre, identificador de usuario, correo electrónico (ver Figura 11.24).

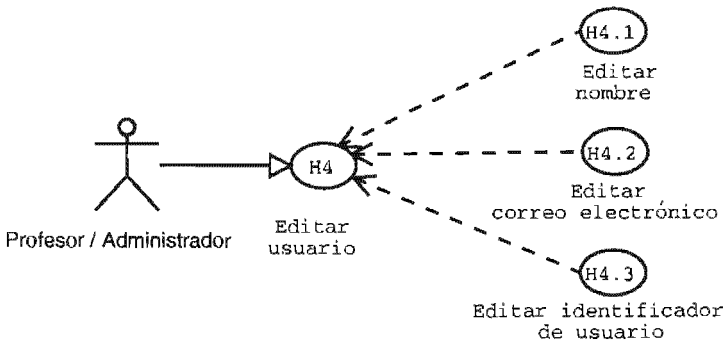

Figura 11.24: Subhistorias encontradas

11.6.2. PRUEBAS

Para llevar a cabo esta historia es necesario que exista al menos un usuario registrado.

En esta prueba el usuario debe ingresar al sistema como administrador, encontrar, seleccionar y consultar (ver Figura 11.19) a un usuario profesor para poder editar el nombre, el correo electrónico y el identificador de usuario.

Otra prueba es aquella en la que el usuario profesor ingresa al sistema y edita su nombre, su correo electrónico o su identificador de usuario.

Algunos puntos importantes a evaluar son:

- Que ingresen toda la información solicitada (nombre, correo electrónico e identificador de usuario).

11.6.3. DISEÑO

Al realizar esta historia llegamos a la pantalla de la Figura 11.25.

Figura 11.25: Editar la información del usuario

Modelo: La entidad identificada es *Usuario*.

Las páginas y componentes encontrados son *Encabezado, CapturaUsuario, CCapturaUsuario y ConsultaUsuario*

Vista: Las plantillas son *Encabezado.html, CapturaUsuario.html, CCapturaUsuario.html y ConsultaUsuario.html* (ver Figura 11.26).

La plantilla *Encabezado* le da la opción al usuario profesor de editar su información, mientras que al usuario administrador la plantilla *ConsultalUsuario* le despliega la información referente al usuario profesor seleccionado y le da la opción para editar esta información.

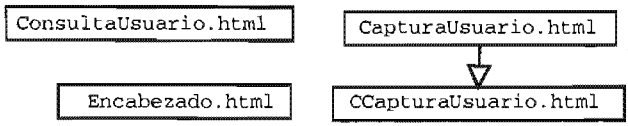

Figura 11.26: Diagrama plantilla a nivel Vista

Ambos caminos llevan a la página *CapturaUsuario* que despliega al componente *CCapturaUsuario* con la información del usuario a editar.

Controlador: Tenemos las especificaciones *Encabezado.jwc, CapturaUsuario.page, CCapturaUsuario.jwc y ConsultaUsuario.page* (ver Figura 11.27) y las implementaciones *Encabezado.java, CapturaUsuario.java, CCapturaUsuario.java y ConsultaUsuario.java* (ver Figura 11.28).

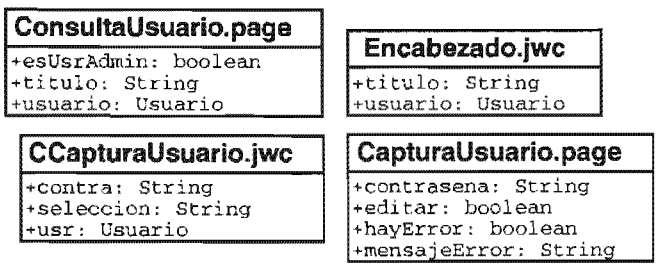

Figura 11.27: Diagrama especificación a nivel Controlador

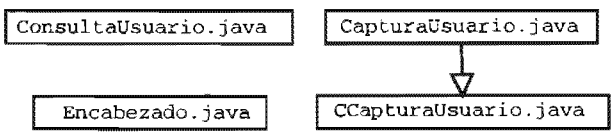

Figura 11.28: Diagrama implementación a nivel Controlador

La implementación *ConsultaUsuario* solamente se encarga de redireccionar a la página *CapturaUsuario* pasándole el usuario que el administrador selec-

cionó, mientras que *Encabezado* redirecciona a la página *CapturaUsuario* con la información del usuario que ingresó al sistema.

11.6.4. DESARROLLO

La relación que existe entre las entidades, páginas y componentes se muestra en la Figura 11.29.

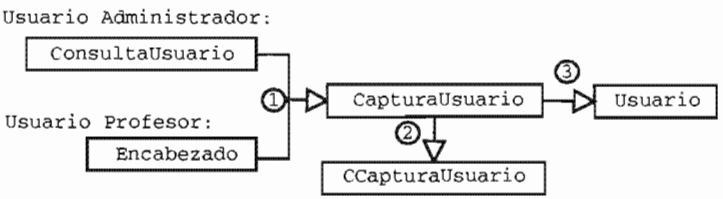

Figura 11.29: Diagrama de colaboración: Editar un usuario

11.7. RESOLVIENDO HISTORIA 5: *Eliminar un usuario*

11.7.1. ANÁLISIS

La eliminación de un usuario solo podrá ser realizada por el usuario administrador, para esto el administrador debe encontrar, seleccionar y consultar (ver Figura 11.19) al usuario profesor que desee eliminar, además de que debe confirmar que realmente quiere eliminar a ese usuario.

Las subhistorias identificadas para esta historia son las que se muestran en la Figura 11.30.

11.7.2. PRUEBAS

Para llevar a cabo esta prueba es necesario que el usuario administrador ingrese al sistema, encuentre, seleccione y consulte al usuario profesor que desee eliminar y confirme su eliminación.

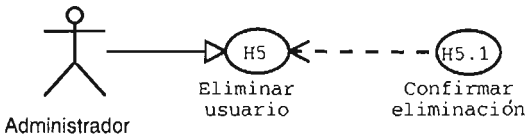

Figura 11.30: Subhistorias encontradas

Los puntos importantes a evaluar son:

- Que existan usuarios registrados.

11.7.3. DISEÑO

Al realizar esta historia llegamos a la pantalla de la Figura 11.31.

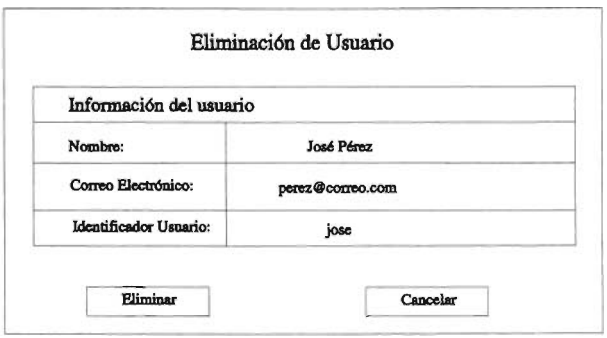

Figura 11.31: Eliminar un usuario

Modelo: La entidad identificada es *Usuario*.

Las páginas y componentes encontrados son *ConsultaUsuario* y *BorraUsuario*.

Vista: Las plantillas son *ConsultaUsuario.html* y *BorraUsuario.html* (ver Figura 11.32).

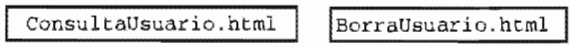

Figura 11.32: Diagrama plantilla a nivel Vista

Donde la plantilla *ConsultaUsuario* despliega la información referente al usuario profesor seleccionado y da la opción para eliminar al usuario. La plantilla *BorraUsuario* es el encargado de desplegar la información referente al usuario que el administrador eligió para eliminar.

Controlador: Tenemos las especificaciones *ConsultaUsuario.page* y *BorraUsuario.page* (ver Figura 11.33) y las implementaciones *ConsultaUsuario.java* y *BorraUsuario.java*(ver Figura 11.34).

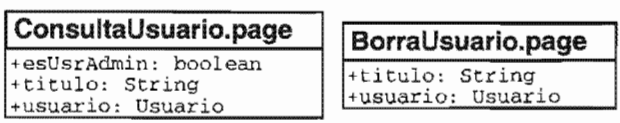

Figura 11.33: Diagrama especificación a nivel Controlador

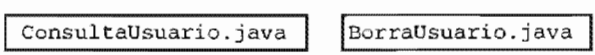

Figura 11.34: Diagrama implementación a nivel Controlador

Donde la implementación *ConsultaUsuario* redirecciona a la página *BorraUsuario*. La implementación *BorraUsuario* se encarga de actualizar la base de datos, eliminando al usuario que el administrador eligió.

11.7.4. DESARROLLO

La relación que existe entre las entidades, páginas y componentes se muestra en la Figura 11.35.

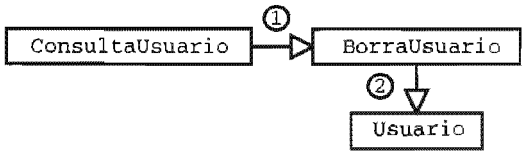

Figura 11.35: Diagrama de colaboración: Eliminar un usuario

11.8. RESOLVIENDO HISTORIA 6: *Cambiar contraseña*

11.8.1. ANÁLISIS

El cambio de contraseña se da para aquellos usuarios que olvidaron su contraseña, por lo tanto como este sistema está enfocado para los profesores de la facultad, podemos identificarlos al solicitarles el RFC que se encuentra en las bases de datos de la facultad.

Además del RFC se le solicitará la nueva contraseña con la que podrá ingresar posteriormente.

Las subhistorias identificadas para esta historia se muestran en la Figura 11.36.

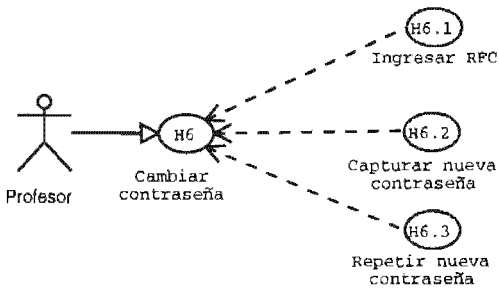

Figura 11.36: Subhistorias encontradas

11.8.2. PRUEBAS

Para realizar esta prueba es necesario que el usuario profesor se haya registrado anteriormente.

El usuario debe proporcionar su RFC, la nueva contraseña y repetir esta nueva contraseña.

Algunos puntos importantes a evaluar:

- Que sea un profesor de la Facultad de Ciencias.
- Que se encuentre registrado en el sistema.
- Que la nueva contraseña coincida con la repetición de la misma.

11.8.3. DISEÑO

Para realizar el cambio de contraseña se estableció la pantalla de la Figura 11.37.

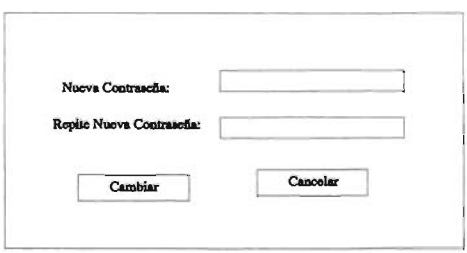

Figura 11.37: Cambiar contraseña

Modelo: La entidad identificada es *Usuario*.

Las páginas y componentes que se identificaron para resolver esta historia son: *EncabezadoInicial, ReconoceUsuario y CambiaContrasena*.

Vista: Las plantillas son: *EncabezadoInicial.html, ReconoceUsuario.html y CambiaContrasena.html* (ver Figura 11.38).

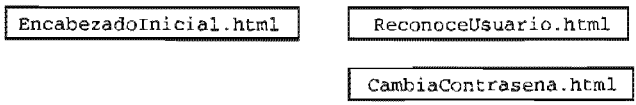

Figura 11.38: Diagrama plantilla a nivel Vista

Donde la plantilla *EncabezadoInicial* da la opción para que el usuario pueda cambiar su contraseña, la plantilla *ReconoceUsuario* captura el RFC del usuario y la plantilla *CambiaContrasena* se encarga de capturar la nueva contraseña y su repetición.

Controlador: Tenemos las especificaciones *EncabezadoInicial.jwc, ReconoceUsuario.page y CambiaContrasena.page* (ver Figura 11.39) y las implementaciones *EncabezadoInicial.java, ReconoceUsuario.java y CambiaContrasena.java* (ver Figura 11.40).

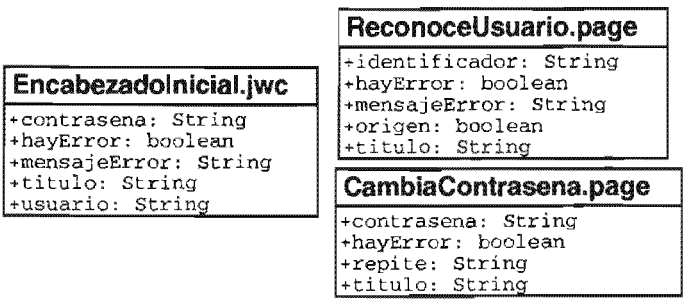

Figura 11.39: Diagrama especificación a nivel Controlador

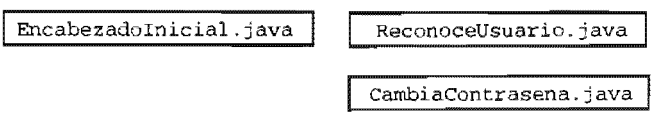

Figura 11.40: Diagrama implementación a nivel Controlador

Donde la implementación *ReconoceUsuario* busca el RFC capturado en la base de datos del sistema y la implementación *CambiaContrasena* valida la nueva contraseña y actualiza la base de datos.

11.8.4. DESARROLLO

La relación que existe entre las entidades, páginas y componentes se muestra en la Figura 11.41.

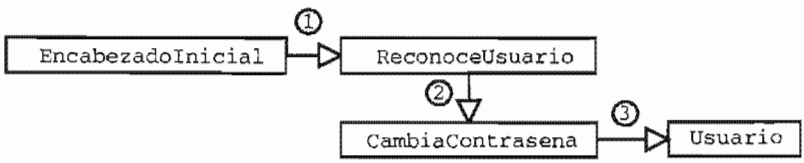

Figura 11.41: Diagrama de colaboración: Cambiar contraseña

CAPÍTULO 12

ITERACIÓN 2: CATEGORÍAS

12.1. INTRODUCCIÓN

El enfoque del sistema es que el usuario profesor pueda tener su inventario personal en línea, es decir se quiere que éste pueda definir que es lo que quiere almacenar. Para realizar esto es necesario que el usuario defina que categorías de pertenencia desea almacenar, por ejemplo *Libros, Artículos Publicados, Tesis, etc* y que atributos de las pertenencias le interesan almacenar, por ejemplo si la categoría es *Libros* al usuario le puede interesar los atributos *Título, Autor, ISBN, Colocación, etc.*.

En esta iteración nos enfocaremos a resolver todas las historias relacionadas con las categorías de pertenencias que poseerán cada uno de los usuarios profesor en el sistema de inventarios.

12.2. PLANEACIÓN DE LA ITERACIÓN

Después de varias discusiones identificamos cuatro historias principales *Crear nueva categoría, Encontrar categoría, Editar una categoría y Eliminar una categoría*, las cuales podemos ver en la Figura 12.1.

12.3. RESOLVIENDO HISTORIA 1: *Crear nueva categoría*

12.3.1. ANÁLISIS

Como se dijo en la sección 12.1, el usuario deberá definir al menos una categoría de pertenencia. Esta definición consiste en que el usuario proporcione el

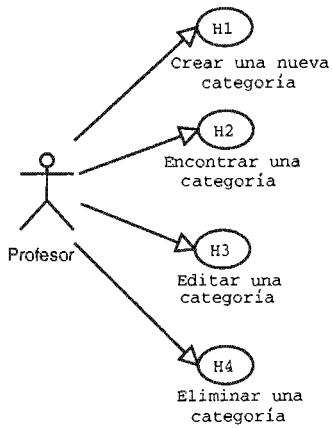

Figura 12.1: Historias de la Iteración

nombre de la categoría y los nombres de los atributos que le interesan de las pertenencias.

Estos nombres de los atributos deben agregarse uno a uno, los cuales se irán listando, teniendo así que en cualquier momento se puede eligir algún atributo de la lista para editar su nombre o incluso eliminarlo.

Dado el análisis anterior, llegamos a que para resolver esta historia tenemos las subhistorias que se muestran en la Figura 12.2.

12.3.2. PRUEBAS

Para llevar a cabo esta prueba es necesario que el usuario profesor haya ingresado al sistema.

El usuario debe ingresar el nombre de la categoría y debe agregar uno a uno los nombres de los atributos que desee que posean sus pertenencias, así como editar o eliminar alguno de los atributos que agregó.

Algunos puntos importantes a evaluar en la prueba son:

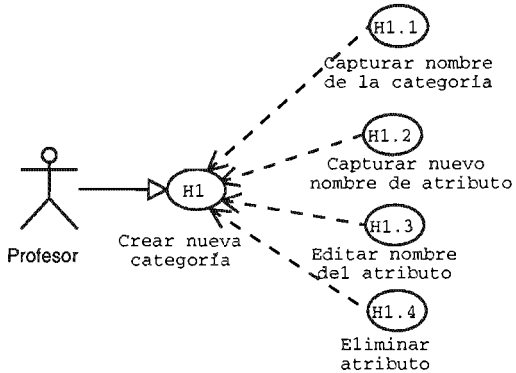

Figura 12.2: Subhistorias encontradas

- Que el usuario ingrese el nombre de la categoría.
- Que el nombre de la categoría no sea un nombre definido anteriormente para otra categoría.
- Que ingrese al menos un nombre de atributo.
- Que el nombre del atributo a agregar no se encuentre en la lista de atributos de esta categoría.

12.3.3. DISEÑO

Después del análisis anterior se llegó a la pantalla de la Figura 12.3.

Modelo: Un usuario va a poseer una lista categorías, es por eso que se identificaron nuevas entidades[1] *UsuarioCategoria, Categoria, AtributoCategoria, TipoAtributo, Atributo, AtributoDato y NumTuplas*, además de *Usuario*, las cuales las podemos ver en la Figura 12.4 y las clases que modelan a éstas se muestran en la Figura 12.5.

Donde la entidad *UsuarioCategoria* relaciona al conjunto de categorías con el usuario que las definió.

[1] Los nombres de los atributos que se encuentran en negritas representan a las llaves foráneas, mientras que los nombres de los atributos subrayados representan a las llaves primarias.

Figura 12.3: Crear nueva categoría

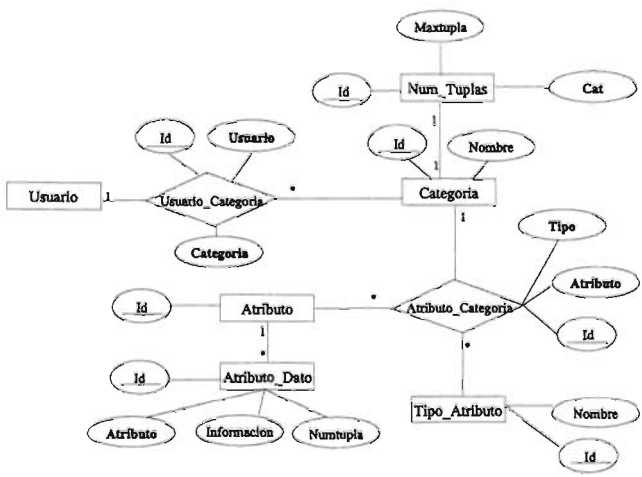

Figura 12.4: Diagrama entidades a nivel Modelo

Por otro lado, la entidad *Categoria* posee una lista de *AtributoCategoria* que es la entidad que relaciona a la categoría el conjunto de entidades que definen a una pertenencia, es decir la relaciona con *TipoAtributo, Atributo, AtributoDato y NumTuplas*.

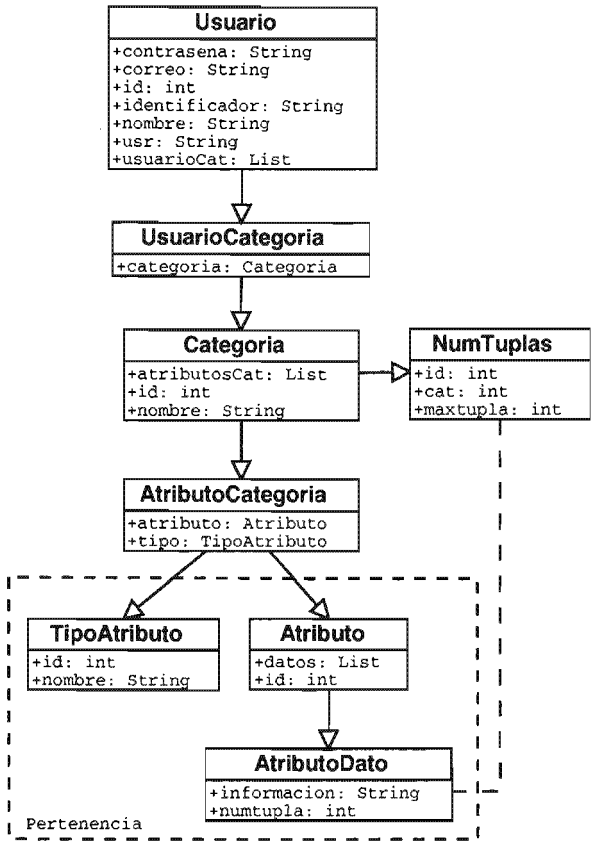

Figura 12.5: Diagrama de clases a nivel Modelo

La definición de la pertenencia consiste en que la entidad *TipoAtributo* almacenará uno de los nombres de atributos que posee la categoría, mientras que *Atributo* tendrá una lista de *AtributoDato* que almacenarán la información referente a cada una de las pertenencias que el usuario quiere almacenar. Por ejemplo si tenemos la categoría *Libros* un TipoAtributo puede ser *Título* y la lista de *AtributoDato* puede contener *Como programar en java, Extreme Programming Explained*, otro *TipoAtributo* puede ser *Autor* y la

lista de *AtributoDato* puede contener *Deitel, Kent Beck*. Una representación gráfica de este ejemplo se muestra en la Figura 12.6.

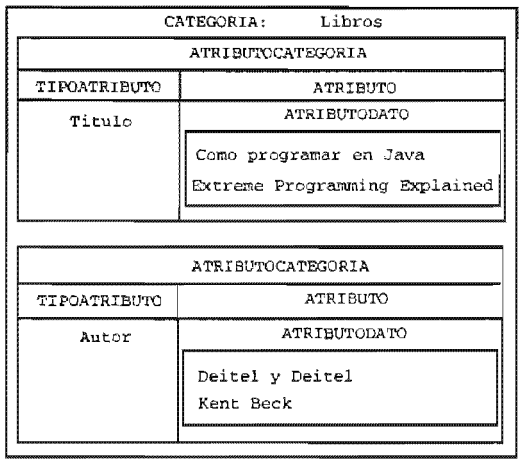

Figura 12.6: Ejemplo

La entidad *NumTuplas* se encargará de almacenar el máximo número de pertenencias que ha definido el usuario para una categoría.

Cabe mencionar que la definición de las pertenencias se hizo así dado que no se sabe que atributos, para una categoría dada, definirá un usuario.

Las páginas y componentes que identificamos son *Encabezado, PrincipalCategoria, CapturaCategoria y CCapturaCategoria.*

Vista: Las plantillas son *Encabezado.html, PrincipalCategoria.html, CapturaCategoria.html y CCapturaCategoria.html* (ver Figura 12.7).

Donde la plantilla *Encabezado* da la opción para que el usuario pueda ingresar a la sección de las categorías. La plantilla *PrincipalCategoria* nos da la opción para que el usuario pueda crear la nueva categoría, mientras que la

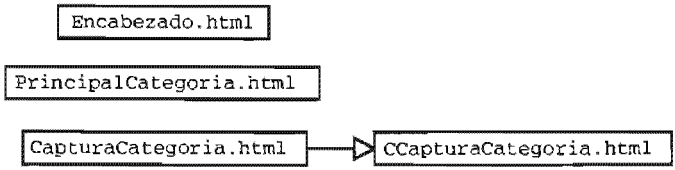

Figura 12.7: Diagrama plantilla a nivel Vista

plantilla *CapturaCategoria* y la plantilla *CCapturaCategoria* son los encargados de capturar el nombre de la categoría y los nombres de los atributos, así como de dar la opción de editar o eliminar a los atributos que se hayan agregado.

Controlador: Tenemos las especificaciones *Encabezado.jwc, PrincipalCategoria.page, CapturaCategoria.page y CCapturaCategoria.jwc* (ver Figura 12.8), las implementaciones *Encabezado.java, PrincipalCategoria.java, CapturaCategoria.java y CCapturaCategoria.java* (ver Figura 12.9), y se creó la clase auxiliar *Tipo.java* la cual es utilizada por la implementación *CCapturaCategoria.java* creando una lista de éstos para que a su vez la implementación *CapturaCategoria* se encargue de asociar esta lista a los atributos de la categoría y hacer la actualización en la base de datos.

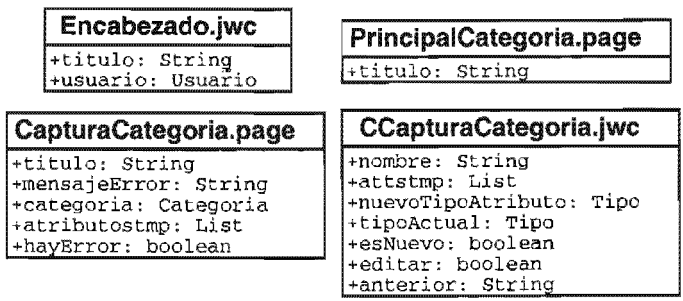

Figura 12.8: Diagrama especificación a nivel Controlador

donde la implementación *CCapturaCategoria* y *CapturaCategoria* como se vió anteriormente uno se encarga de capturar la información y el otro se encarga de actualizar la base de datos, respectivamente.

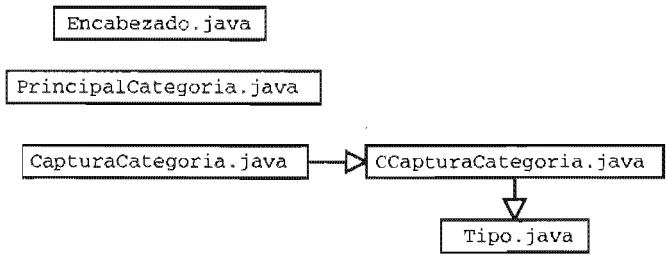

Figura 12.9: Diagrama implementación a nivel Controlador

12.3.4. DESARROLLO

La relación que existe entre las entidades, páginas y componentes se muestra en la Figura 12.10.

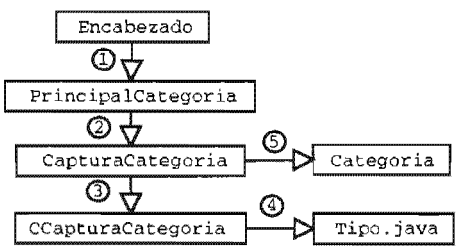

Figura 12.10: Diagrama de colaboración: Crear nueva categoría

12.4. RESOLVIENDO HISTORIA 2: *Encontrar una categoría*

12.4.1. ANÁLISIS

Al igual que se planteó en los usuarios existen dos formas de encontrar una categoría:

- Ingresar el nombre de la categoría para realizar una búsqueda entre las categorías que ha definido el usuario, para así navegar entre los resultados.

- Navegar entre todas las categorías que haya definido el usuario.

Además el usuario deberá seleccionar y consultar la categoría deseada. Las subhistorias identificadas para esta historia se muestran en la Figura 12.11.

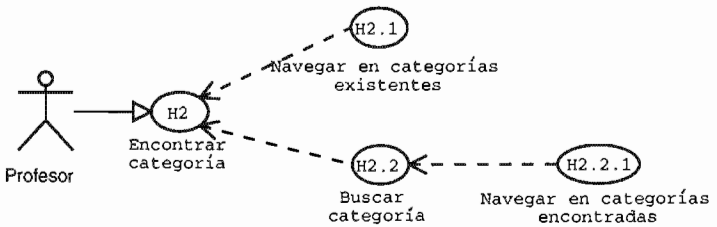

Figura 12.11: Subhistorias encontradas.

12.4.2. PRUEBAS

Para poder llevar a cabo esta prueba el usuario profesor debera haber ingresado al sistema.

El usuario deberá ingresar el nombre de la categoría o parte de ella para realizar la búsqueda, tras lo cual deberá navegar entre los resultados o el usuario buscará entre todas las categorías que se encuentren definidas hasta encontrar a la categoría deseada, a continuación seleccionarla y consultarla.

Algunos puntos importantes a evaluar en esta prueba son:

- Que existan categorías.
- Que la información proporcionada corresponda al nombre de alguna categoría definida.

12.4.3. DISEÑO

Al realizar esta historia llegamos a las pantallas de las Figuras 12.12 y 12.13.

Modelo: Se identificaron las entidades *Usuario, UsuarioCategoria y Categoria.*

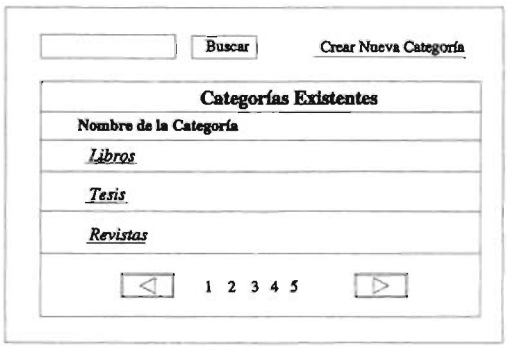

Figura 12.12: Encontrar una categoría

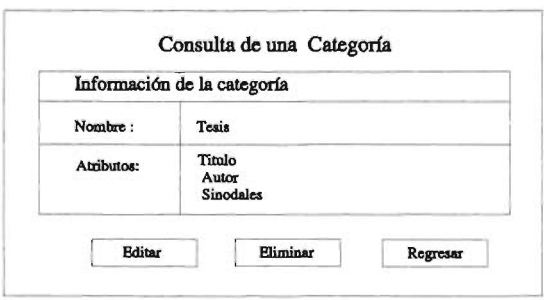

Figura 12.13: Consultar una categoría

Tenemos las páginas y componentes *Encabezado, PrincipalCategoria, CBusquedaCategoria, Paginador y ConsultaCategoria.*

Vista: Tenemos las páginas y componentes *Encabezado.html, PrincipalCategoria.html, CBusquedaCategoria.html, Paginador.html y ConsultaCategoria.html* (ver Figura 12.14).

Donde la plantilla *PrincipalCategoria* se encarga de desplegar los nombres de las categorías definidas por el usuario. La plantilla *CBusquedaCategoria* se encarga de capturar la información que proporciona el usuario para

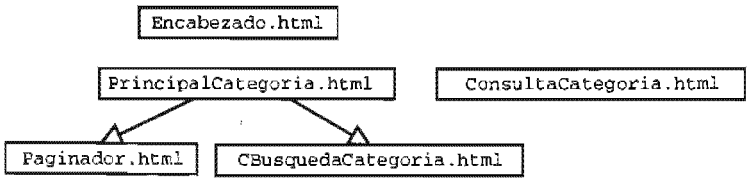

Figura 12.14: Diagrama plantillas a nivel Vista

realizar la búsqueda. La plantilla *ConsultaCategoria* se encarga de desplegar la información de la categoría.

Controlador: Tenemos las especificaciones *Encabezado.jwc, PrincipalCategoria.page, CBusquedaCategoria.jwc, Paginador.jwc y ConsultaCategoria.page* (ver Figura 12.15) y las implementaciones *Encabezado.java, PrincipalCategoria.java, CBusquedaCategoria.java, Paginador.java y ConsultaCategoria.java* (ver Figura 12.16).

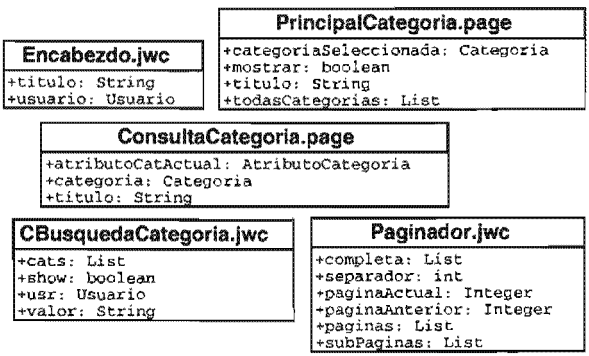

Figura 12.15: Diagrama especificación a nivel Controlador

La implementación de *CBusquedaCategoria* es la encargada de buscar las categorías definidas por el usuario, la implementación *Paginador* se encarga de dividir la lista de categorías obtenidas por la búsqueda en sublistas y la página *PrincipalCategoria* una vez seleccionada la categoría redirecciona a la página *ConsultaCategoria*.

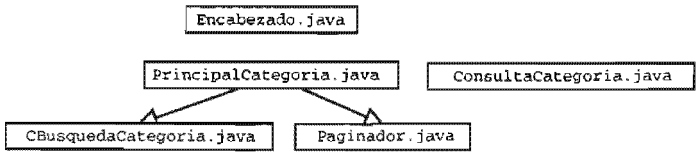

Figura 12.16: Diagrama implementación a nivel Controlador

12.4.4. DESARROLLO

La relación que existe entre las entidades, páginas y componentes se muestra en la Figura 12.17.

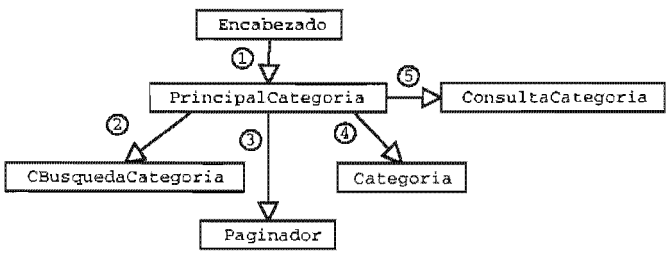

Figura 12.17: Diagrama de colaboración: Encontrar una categoría

12.5. RESOLVIENDO HISTORIA 3: *Editar una categoría*

12.5.1. ANÁLISIS

Cuando se edite una categoría se permitirá editar el nombre de la categoría, agregar el nombre de un atributo, editar o eliminar alguno de los atributos de la categoría.

Las subhistorias identificadas en esta historia se muestran en la Figura 12.18.

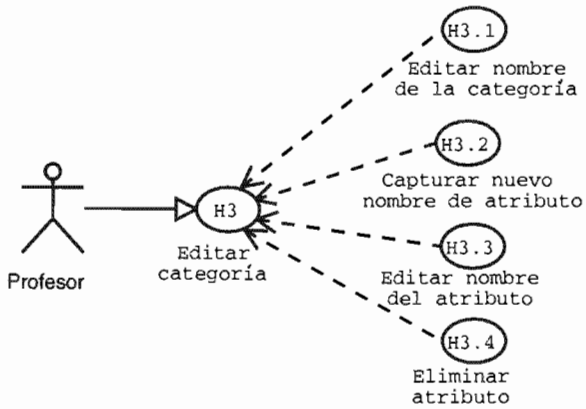

Figura 12.18: Subhistorias encontradas

12.5.2. PRUEBAS

Para llevar a cabo esta prueba es necesario que el usuario haya definido anteriormente al menos una categoría.

El usuario profesor debe encontrar, seleccionar y consultar a una categoría tras lo cual debe elegir la opción de editar para asi entonces poder editar el nombre de la categoría, agregar, editar o eliminar algún atributo.

Los puntos importantes a evaluar en esta prueba son:

- Que el usuario ingrese el nombre de la categoría.
- Que el nombre de la categoría no sea un nombre definido anteriormente para otra categoría.
- Que ingrese al menos un nombre de atributo.
- Que el nombre del atributo a agregar no se encuentre en la lista de atributos de esta categoría.

12.5.3. DISEÑO

Dado el análisis anterior se llegó a la pantalla de la Figura 12.19.

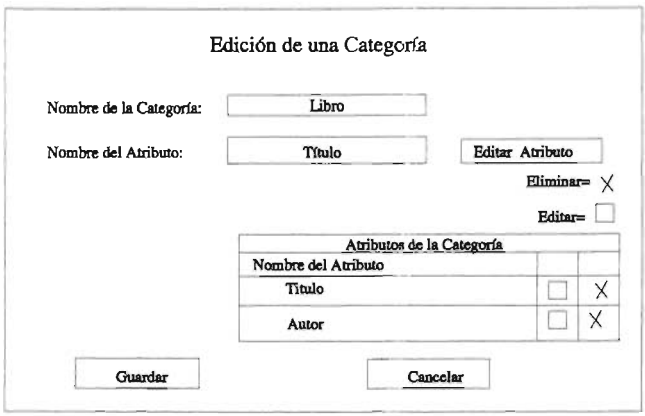

Figura 12.19: Editar una categoría

Modelo: Las entidades identificadas son *Usuario, UsuarioCategoria, Categoria, AtributoCategoria, TipoAtributo.*

No fue necesario definir más páginas o compor entes para la edición sencillamente se reutilizaron las páginas y componentes *ConsultaCategoria, CapturaCategoria, CCapturaCategoria.*

Vista: Las plantillas son *ConsultaCategoria.html, CapturaCategoria.html, CCapturaCategoria.html* (ver Figura 12.20).

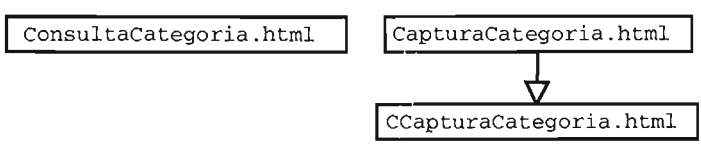

Figura 12.20: Diagrama plantilla a nivel Vista

Donde la plantilla *ConsultaCategoria* despliega la información de la categoría y da la opción de editarla, mientras que la plantilla *CapturaCategoria* despliega al componente *CCapturaCategoria* con la información de la Categoría.

Controlador: Tenemos las especificaciones *ConsultaCategoria.page, CapturaCategoria.page, CCapturaCategoria.jwc* (ver Figura 12.21), las implementaciones *ConsultaCategoria.java, CapturaCategoria.java, CCapturaCategoria.java* (ver Figura 12.22) y la clase auxiliar *Tipo.java.*

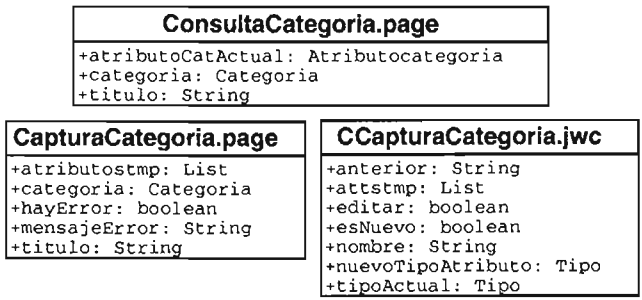

Figura 12.21: Diagrama especificación a nivel Controlador

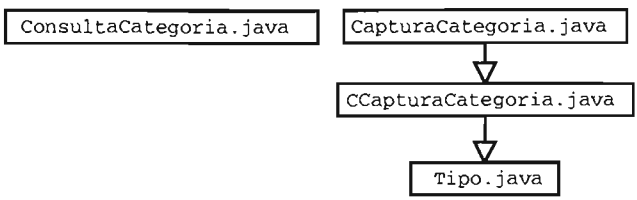

Figura 12.22: Diagrama implementación a nivel Controlador

Donde la implementación *ConsultaCategoria* redirecciona a la página *CapturaCategoria*, la cual actualiza la información de la categoría en la base de datos que capturó el componente *CCapturaCategoria.*

12.5.4. DESARROLLO

La relación que existe entre las entidades, páginas y componentes se muestra en la Figura 12.23.

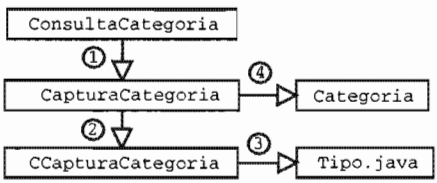

Figura 12.23: Diagrama de colaboración: Editar una categoría

12.6. RESOLVIENDO HISTORIA 4: *Eliminar una categoría*

12.6.1. ANÁLISIS

Para llevar a cabo la eliminación de una de las categorías, solo es necesario que el usuario encuentre, seleccione y consulte la categoría a eliminar y confirme su eliminación.

Las subhistorias identificadas para esta historia se muestran en la Figura 12.24.

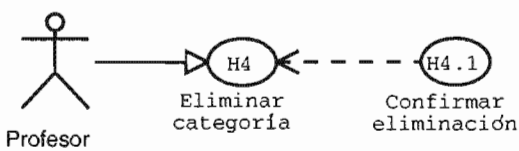

Figura 12.24: Subhistorias encontradas

12.6.2. PRUEBAS

Esta prueba requiere que el usuario haya definido al menos una categoría anteriormente, dado esto el usuario debe encontrar, seleccionar y consultar (ver Figu-

ra 12.13) la categoría que desea eliminar y confirmar su eliminación.

12.6.3. DISEÑO

Dado el análisis anterior se llegó a la pantalla de la Figura 12.25.

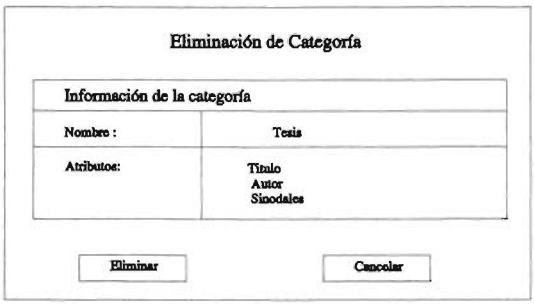

Figura 12.25: Eliminar una categoría

Modelo: Las entidades involucradas son *Usuario, UsuarioCategoria, Categoria.*

Las páginas identificadas son *ConsultaCategoria y BorraCategoria.*

Vista: Las plantillas son *ConsultaCategoria.html y BorraCategoria.html* (ver Figura 12.26).

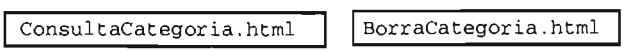

Figura 12.26: Diagrama de plantilla a nivel Vista

Donde la plantilla *ConsultaCategoria* se encarga de desplegar la información de la categoría y dar la opción de eliminar esa categoría, mientras que la plantilla *BorraCategoria* se encarga de mostrar el nombre de la categoría y los atributos que tiene definidos.

Controlador: Tenemos las especificaciones *ConsultaCategoria.page y BorraCategoria.page* (ver Figura 12.27) y las implementaciones *ConsultaCategoria.java y BorraCategoria.java* (ver Figura 12.28).

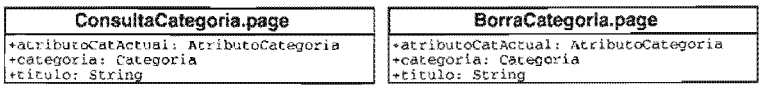

Figura 12.27: Diagrama especificación a nivel Controlador

Figura 12.28: Diagrama implementación a nivel Controlador

Donde la implementación *ConsultaCategoria* se encarga de redireccionar a la página *BorraCategoria* y la implementación *BorraCategoria* se encarga de actualizar la base de datos.

12.6.4. DESARROLLO

La relación que existe entre las entidades, páginas y componentes se muestra en la Figura 12.29.

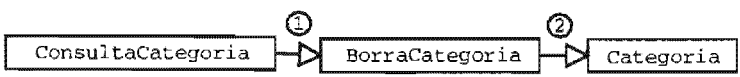

Figura 12.29: Diagrama de colaboración: Eliminar una categoría

CAPÍTULO 13

ITERACIÓN 3: PERTENENCIAS

13.1. INTRODUCCIÓN

Como vimos la iteración anterior se enfocó a todo lo relacionado a las categorías y a la definición de los atributos que van a tener las pertenencias, por lo cual esta iteración se enfocará a que el usuario profesor pueda manipular sus pertenencias, es decir pueda crear, encontrar, editar o eliminar la información de sus pertenencias. Por ejemplo si se definió la categoría *Libros* con los atributos *Título y Autor*, el usuario podrá tener la pertenencia con el título *Como Programar en Java* y el autor *Deitel y Deitel*.

13.2. PLANEACIÓN DE LA ITERACIÓN

Para resolver esta iteración se identificaron las historias *Crear una nueva Pertenencia, Encontrar una pertenencia, Editar una pertenencia y Eliminar una pertenencia* que se muestran en la Figura 13.1.

13.3. RESOLVIENDO HISTORIA 1: *Crear nueva pertenencia*

13.3.1. ANÁLISIS

Dado que el usuario profesor puede definir muchas categorías, lo primero que el usuario debe determinar es la categoría de la pertenencia que desea crear, hecho esto el usuario proporcionará alguna información de la pertenencia, es decir bastará con que proporcione la información de al menos uno de los atributos que definió para la categoría.

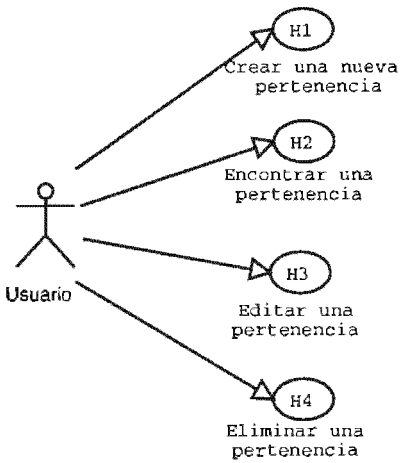

Figura 13.1: Historias de la Iteración

Las subhistorias identificadas para realizar esta historia se muestran en la Figura 13.2.

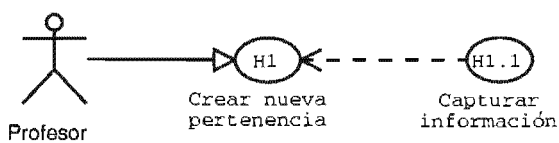

Figura 13.2: Subhistorias encontradas

13.3.2. PRUEBAS

Para llevar a cabo esta prueba es necesario que el usuario haya creado la categoría de la pertenencia. El usuario debe elegir la categoría de la pertenencia y proporcionar la información referente a la pertenencia.

Para comprobar el buen funcionamiento de la historia se probarán:

- Que exista la categoría de la pertenencia a crear.
- Que el usuario proporcione la información de al menos uno de los atributos.

13.3.3. DISEÑO

Dado el análisis anterior se llegó a la pantalla de la Figura 13.3.

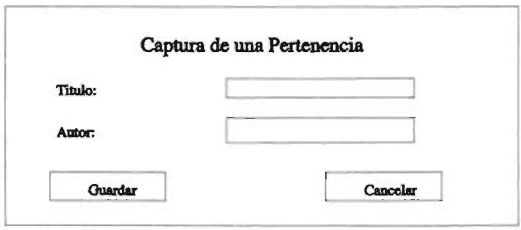

Figura 13.3: Crear nueva pertenencia

Modelo: La entidades involucradas en la resolución de la historia son *Usuario, UsuarioCategoria, Categoria, AtributoCategoria, TipoAtributo, Atributo, AtributoDato y NumTuplas*

Las páginas y componentes identificados son *Encabezado, CategoriaSeleccionada, CapturaPertenencia, CCapturaPertenencia.*

Vista: Las plantillas son *Encabezado.html, CategoriaSeleccionada.html, CapturaPertenencia.html, CCapturaPertenencia.html* (ver Figura 13.4).

Donde la plantilla *Encabezado.html* es el encargado de dar la opción para ingresar a la sección de las pertenencias. La plantilla *CategoriaSeleccionada* nos permite elegir la categoría de la pertenencia y da la opción para crear la nueva pertenencia. El componente *CCapturaPertenencia* se encarga de capturar la información referente a la pertenencia.

Controlador: Tenemos las especificaciones *Encabezado.jwc, CategoriaSeleccionada.page, CapturaPertenencia.page, CCapturaPertenencia.jwc* (ver Figura 13.5) y las implementaciones *Encabezado.java, CategoriaSeleccionada.java, CapturaPertenencia.java, CCapturaPertenencia.java* (ver Figura 13.6) y la clase auxiliar *TipoValor.java.*

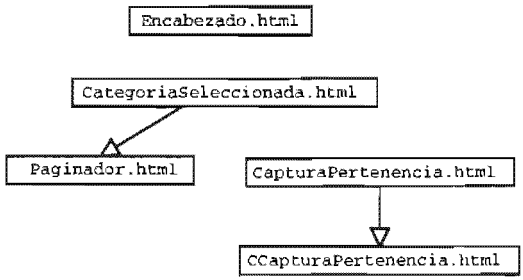

Figura 13.4: Diagrama plantilla a nivel Vista

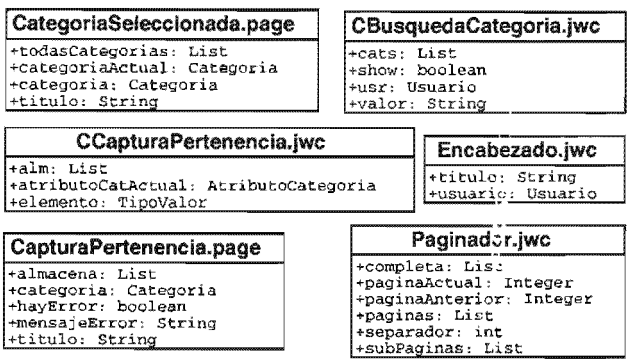

Figura 13.5: Diagrama especificación a nivel Controlador

Como se puede ver en la Figura 13.6 el componente *CCapturaPertenencia* posee una lista de objetos *TipoValor* para capturar la información de la pertenencia, tras lo cual la implementacion *CapturaPertenencia* asocia cada elemento de esta lista con las entidades *AtributoDato* de cada entidad *Atributo* de una categoría. Por ejemplo si tuvieramos la categoría *Libros* con los atributos *Título y Autor*, además de las pertenencias *Como programar en java, Deitel y Deitel* y *Extreme Programming Explained, Kent Beck*, ésta información quedaría almacenada como se muestra en la Figura 13.7.

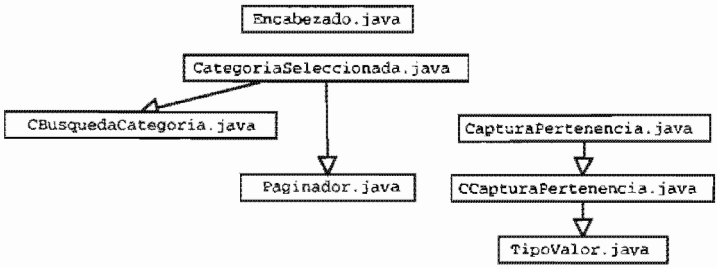

Figura 13.6: Diagrama implementación a nivel Controlador

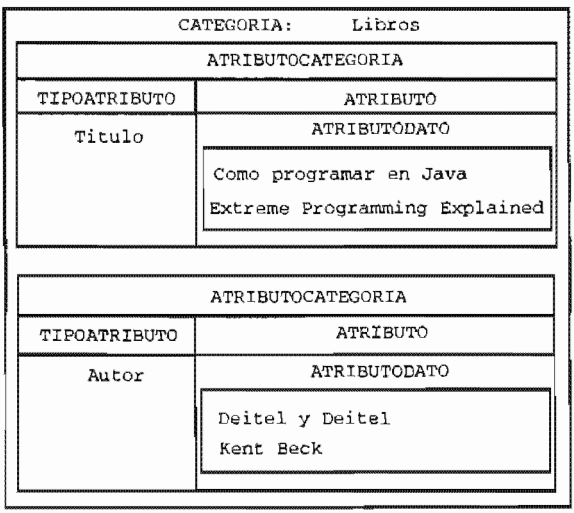

Figura 13.7: Ejemplo

13.3.4. DESARROLLO

La relación entre las entidades, páginas, componentes y la clase auxiliar se muestra en la Figura 13.8.

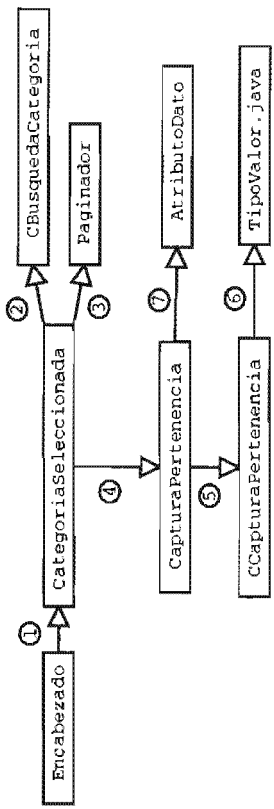

Figura 13.8: Diagrama de colaboración: Crear nueva pertenencia

13.4. RESOLVIENDO HISTORIA 2: *Encontrar una pertenencia*

13.4.1. ANÁLISIS

Para que el usuario pueda encontrar una pertenencia debe establecer a que categoría pertenece ésta y a continuación podrá encontrar la pertenencia por medio de:

- Navegar entre todas las pertenencias de la categoría hasta encontrar la deseada.
- Podrá ingresar cualquier información de la pertenencia y realizar una búsqueda, para así navegar entre los resultados de la búsqueda.

La búsqueda se llevará a cabo sobre todos los valores de los atributos de las categorías de las pertenencias. A continuación deberá seleccionar y consultar la pertenencia.

Las subhistorias identificadas para realizar esta historia se muestran en la Figura 13.9.

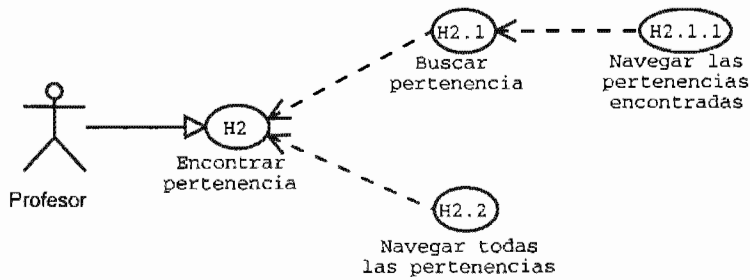

Figura 13.9: Subhistorias encontradas

13.4.2. PRUEBAS

Para realizar esta prueba es necesario que el usuario haya creado la categoría de la pertenencia que busca.

El usuario debe elegir la categoría de la pertenencia que está buscado para a continuación encontrar, seleccionar y consultar la pertenencia.

Algunos puntos importantes a evaluar de la prueba son:

- Que exista la categoría de la pertenencia.
- Que existan pertenencias en esa categoría.
- Que la información proporcionada corresponda a alguno de los valores de los atributos de la categoría de alguna pertenencia.

13.4.3. DISEÑO

Dado el análisis anterior se llegaron a las pantallas de las Figuras 13.10 y 13.11.

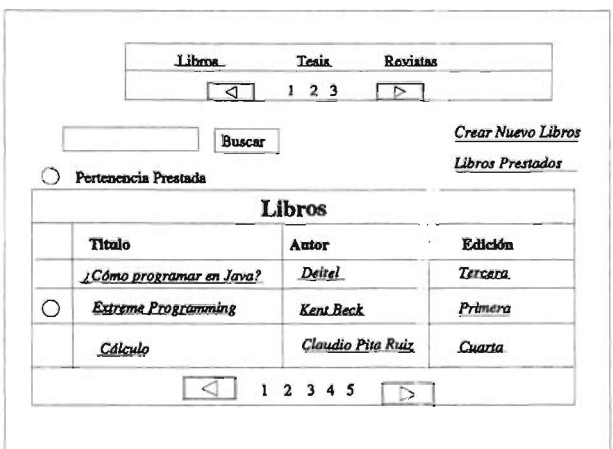

Figura 13.10: Encontrar una pertenencia

Modelo: Las entidades identificadas son *Usuario, UsuarioCategoria, Categoria, AtributoCategoria, TipoAtributo, Atributo, AtributoDato y NumTuplas.*

Las páginas y componentes identificadas son *CategoriaSeleccionada, CBusquedaPertenencia, Paginador y ConsultaPertenencia.*

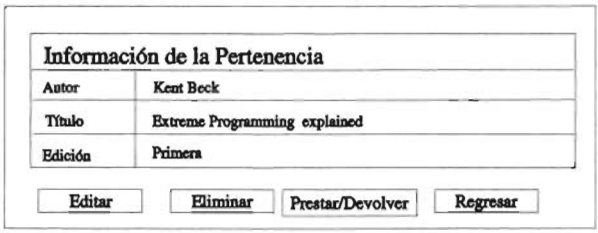

Figura 13.11: Consultar una pertenencia

Vista: Las plantillas son *CategoriaSeleccionada.html, CBusquedaPertenencia.html, Paginador.html y ConsultaPertenencia.html* (ver Figura 13.12).

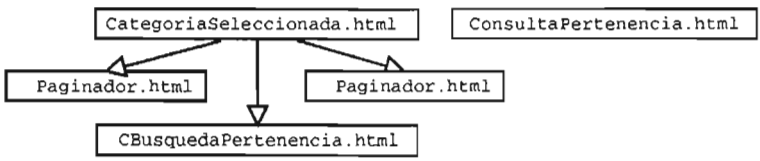

Figura 13.12: Diagrama plantilla a nivel Vista

Donde la plantilla *CategoriaSeleccionada* se encarga de desplegar los valores de los atributos de la categoría para las pertenencias, mientras que la plantilla *CBusquedaPertenencia* se encarga de capturar la información proporcionada por el usuario para realizar la búsqueda y la plantilla *ConsultaPertenencia* despliega la información de la pertenencia.

Controlador: Tenemos las especificaciones *CategoriaSeleccionada.page, CBusquedaPertenencia.jwc, Paginador.jwc y ConsultaPertenencia.page* (ver Figura 13.13), las implementaciones *CategoriaSeleccionada.java, CBusquedaPertenencia.java, Paginador.java y ConsultaPertenencia.java* y las clases auxiliares *Pertenencia.java y TipoValor.java* (ver Figura 13.14).

Donde la implementación *CBusquedaPertenencia* es la encargada de construir las pertenencias mediante las clases auxiliares *Pertenencia* y *Tipo-*

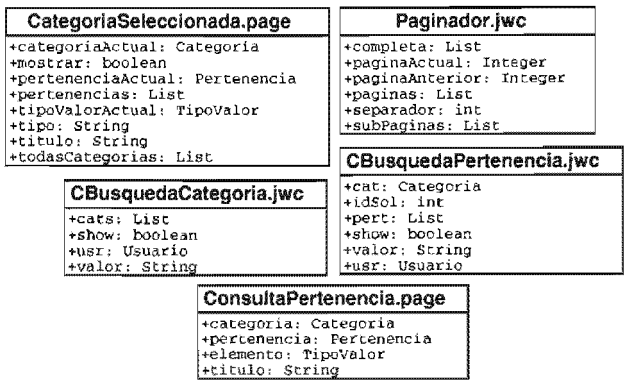

Figura 13.13: Diagrama especificación a nivel Controlador

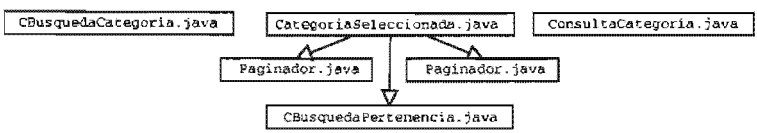

Figura 13.14: Diagrama implementación a nivel Controlador

Valor asociadas a la búsqueda. La implementación *Paginador* se encarga de dividir en sublistas los resultados de la búsqueda.

13.4.4. DESARROLLO

La relación que existe entre las entidades, páginas, componentes y clases auxiliares se muestran en la Figura 13.15.

13.5. RESOLVIENDO HISTORIA 3: *Editar una pertenencia*

13.5.1. ANÁLISIS

Se le permitirá al usuario que pueda editar cualquier valor de los atributos de las pertenencias que haya creado anteriormente.

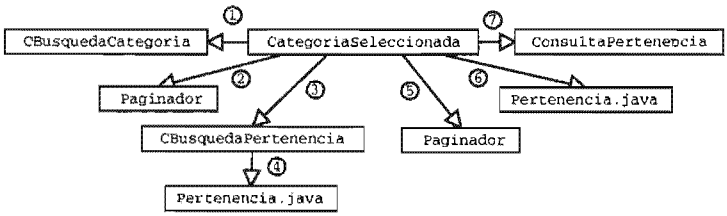

Figura 13.15: Diagrama de colaboración: Encontrar una pertenencia

Las subhistorias identificadas para realizar esta historia se muestran en la Figura 13.16.

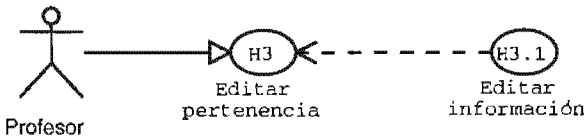

Figura 13.16: Subhistorias encontradas

13.5.2. PRUEBAS

Para llevar a cabo esta prueba es necesario que el usuario profesor encuentre, seleccione y consulte la pertenencia a editar (ver sección 13.4), tras lo cual deberá editar cualquiera de los valores de los atributos de la pertenencia.

Los puntos importantes a evaluar son:

- Que el usuario proporcione la información de al menos uno de los atributos.

13.5.3. DISEÑO

Al realizar esta historia llegamos a la pantalla de la Figura 13.17.

Modelo: Las entidades identificadas son *Usuario, UsuarioCategoria, Categoría, AtributoCategoria, TipoAtributo, Atributo, AtributoDato y NumTuplas.*

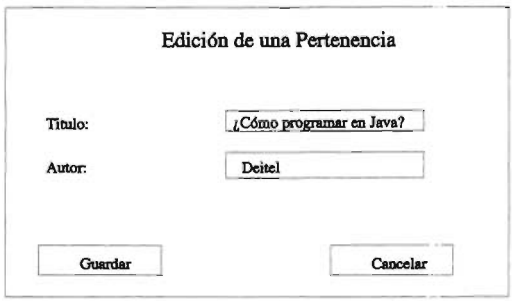

Figura 13.17: Editar una pertenencia

Para realizar esta historia no se crearon más páginas o componentes, únicamente se reutilizaron *ConsultaPertenencia, CapturaPertenencia, CCapturaPertenencia.*

Vista: Las plantillas identificadas son *ConsultaPertenencia.html, CapturaPertenencia.html, CCapturaPertenencia.html* (ver Figura 13.18).

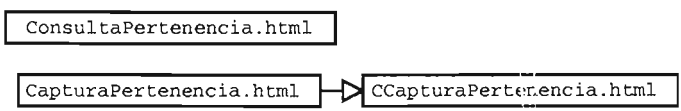

Figura 13.18: Diagrama plantilla a nivel Vista

La plantilla *ConsultaPertenencia* da la opción de editar los atributos de la pertenencia seleccionada. La plantilla *CapturaPertenencia* despliega al componente *CCapturaPertenencia* con la información de la pertenencia a editar.

Controlador: Las especificaciones son *ConsultaPertenencia.page, CapturaPertenencia.page, CCapturaPertenencia.jwc* (ver Figura 13.19) y las implementaciones son *ConsultaPertenencia.java, CapturaPertenencia.java, CCapturaPertenencia.java* y la clase auxiliar *TipoValor.java* (ver Figura 13.20).

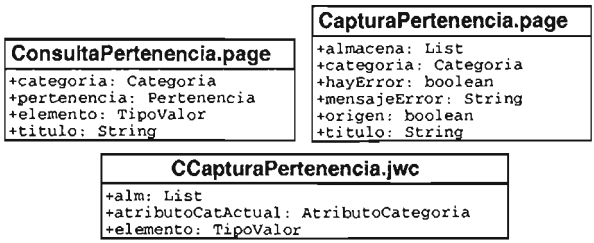

Figura 13.19: Diagrama especificación a nivel Controlador

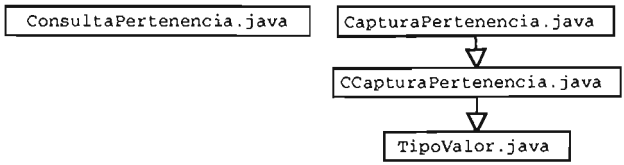

Figura 13.20: Diagrama implementación a nivel Controlador

Donde la implementación *ConsultaPertenencia* redirecciona a la página *CapturaPertenencia* la cual actualiza la información de la pertenencia en la base de datos que capturó el componente *CCapturaPertenencia*.

13.5.4. DESARROLLO

La relación que existe entre las entidades, páginas, componentes y la clase auxiliar se muestra en la Figura 13.21.

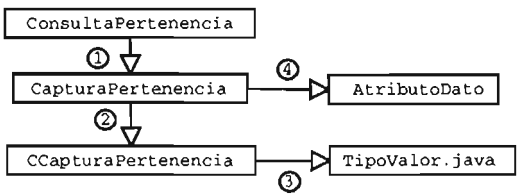

Figura 13.21: Diagrama de colaboración: Editar una pertenencia

13.6. RESOLVIENDO HISTORIA 4: *Eliminar una pertenencia*

13.6.1. ANÁLISIS

Para llevar a cabo la eliminación de una pertenencia, es necesario que el usuario profesor encuentre, seleccione y consulte la pertenencia a eliminar (ver sección 13.4) y confirme su eliminación.

Las subhistorias identificadas para realizar esta historia se muestran en la Figura 13.22.

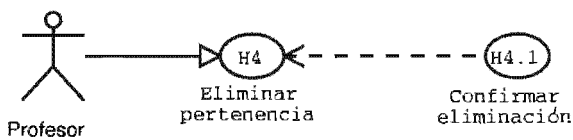

Figura 13.22: Subhistorias encontradas

13.6.2. PRUEBAS

Esta prueba requiere que el usuario encuentre, seleccione y consulte la pertenencia a eliminar (ver Figura 13.11) y confirme su eliminación.

13.6.3. DISEÑO

Dado el análisis anterior se llegó a la pantalla de la Figura 13.23.

Modelo: Las entidades involucradas son *Usuario, UsuarioCategoria, Categoría, AtributoCategoria, TipoAtributo, Atributo, AtributoDato y NumTuplas.*

Las páginas y componentes involucrados son *ConsultaPertenencia y BorraPertenencia.*

Vista: Las plantillas son *ConsultaPertenencia.html y BorraPertenencia.html* (ver Figura 13.24).

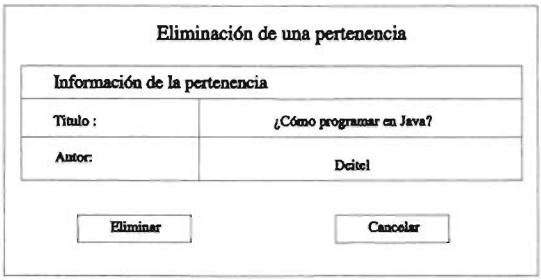

Figura 13.23: Eliminar una pertenencia

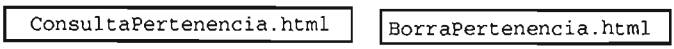

Figura 13.24: Diagrama plantilla a nivel Vista

Donde la plantilla *ConsultaPertenencia* da la opción de eliminar la pertenencia, mientras que la plantilla *BorraPertenencia* despliega la información de la pertenencia a eliminar.

Controlador: Tenemos las especificaciones *ConsultaPertenencia.page y BorraPertenencia.page* (ver Figura 13.25) y las implementaciones *ConsultaPertenencia.java y BorraPertenencia.java* (ver Figura 13.26) y la clase auxiliar *Pertenencia.java*.

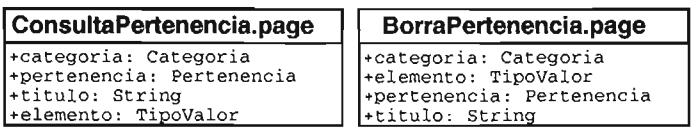

Figura 13.25: Diagrama especificación a nivel Controlador

Donde la implementación *ConsultaPertenencia* redirecciona a la página *BorraPertenencia* y la implementación *BorraPertenencia* actualiza la base de datos.

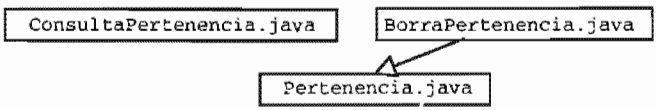

Figura 13.26: Diagrama implementación a nivel Controlador

13.6.4. DESARROLLO

La relación entre las entidades, páginas, componentes y la clase auxiliar se muestra en la Figura 13.27.

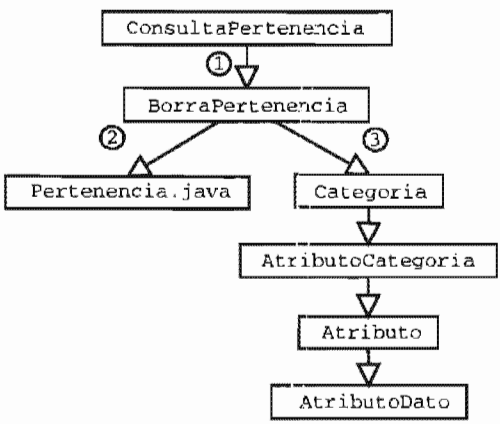

Figura 13.27: Diagrama de colaboración: Eliminar una pertenencia

CAPÍTULO 14

ITERACIÓN 4: SOLICITANTES

14.1. INTRODUCCIÓN

Lo que se pretende en esta iteración es que el usuario profesor pueda tener una agenda de todos sus solicitantes. Por solicitante entendemos a todas aquellas personas que le pidan una o varias de sus pertenencias.

Es por eso que esta iteración se enfocará a la resolución de las historias relacionadas a la manipulación de los solicitantes.

14.2. PLANEACIÓN DE LA ITERACIÓN

Las historias identificadas para llevarse a cabo en esta iteración son *Crear un nuevo solicitante, Encontrar un solicitante, Editar un solicitante y Eliminar un solicitante.* Estas historias se muestran en la Figura 14.1.

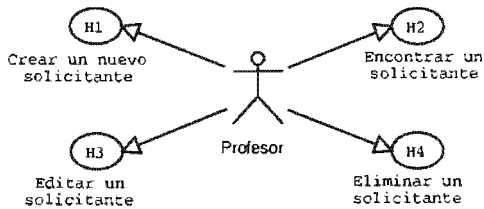

Figura 14.1: Historias de la iteración

14.3. RESOLVIENDO HISTORIA 1: *Crear un nuevo solicitante*

14.3.1. ANÁLISIS

La información que se le solicitará al usuario acerca del solicitante es *nombre, dirección, teléfono y correo electrónico*, siendo el nombre la única información obligatoria.

Las subhistorias identificadas para esta historia se muestran en la Figura 14.2.

14.3.2. PRUEBAS

Para llevar a cabo esta historia es necesario que el usuario profesor ingrese al sistema.

El usuario debe proporcionar el nombre, la dirección, el teléfono y el correo electrónico del solicitante.

Algunos puntos importantes a evaluar son:

- Que ingrese al menos el nombre del solicitante.

14.3.3. DISEÑO

Dado el análisis anterior se llegó a la pantalla de la Figura 14.3.

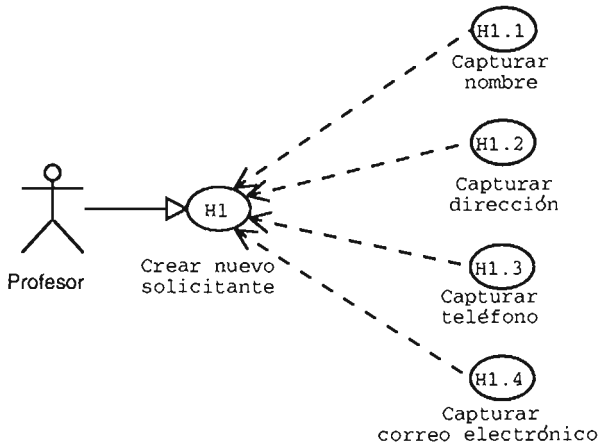

Figura 14.2: Subhistorias encontradas

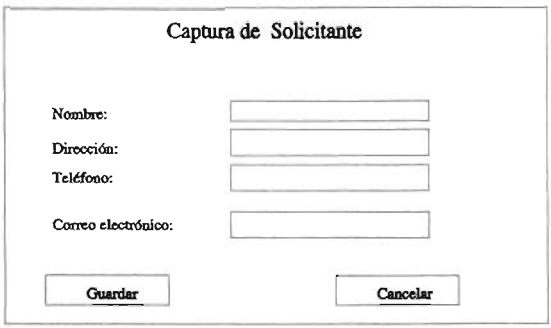

Figura 14.3: Crear nuevo solicitante

Modelo : Las entidades involucradas son *Usuario, UsuarioSolicitante, Solicitante, PrestamoDato.* Las relaciones que existen entre estas entidades [1] se muestran en la Figura 14.4 y las clases que modelan estas entidades se muestran en la Figura 14.5.

[1] Los nombres de los atributos que se encuentran en negritas representan a las llaves foráneas, mientras que los nombres de los atributos subrayados representan a las llaves primarias.

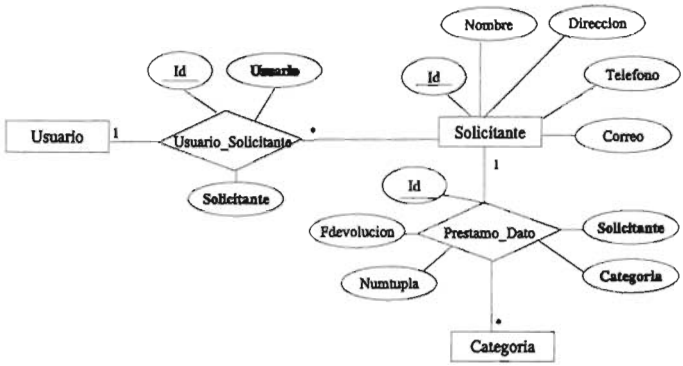

Figura 14.4: Diagrama de entidades a nivel Modelo

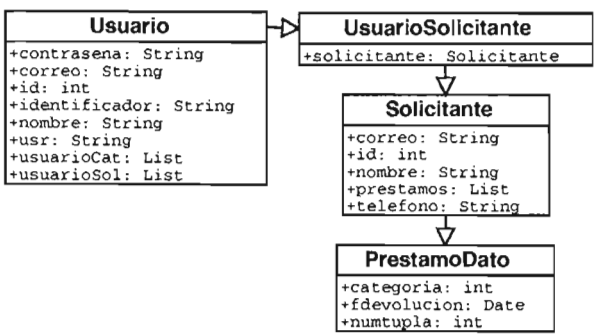

Figura 14.5: Diagrama clases a nivel Modelo

Como se puede observar en las Figura 14.4, estas entidades poseen algunos atributos que no se encuentran definidos en las clases que las modelan, ya que Hibernate al generar las clases mediante el documento *entidad.hbm.xml* omite esos atributos para realizar una manipulación interna de ellos.

La entidad *UsuarioSolicitante* relaciona al conjunto de Solicitantes con el usuario que los definió. Por otro lado, la entidad *Solicitante* posee una lista

de *PrestamoDato*, la cual explicaremos posteriormente.

Las páginas y componentes identificados para resolver esta historia son *Encabezado, PrincipalSolicitante, CapturaSolicitante, CCapturaSolicitante.*

Vista: Las plantillas son: *Encabezado.html, PrincipalSolicitante.html, CapturaSolicitante.html, CCapturaSolicitante.html* (ver Figura 14.6).

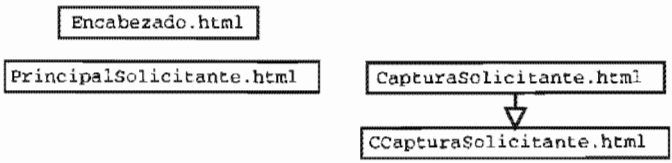

Figura 14.6: Diagrama plantilla a nivel Vista

Donde la plantilla *Encabezado* da la opción para que el usuario pueda ingresar a la sección de solicitantes. La plantilla *PrincipalSolicitante* da la opción para que el usuario pueda crear un nuevo solicitante. La página *CapturaSolicitante* despliega al componente *CCapturaSolicitante* que es el encargado de capturar la información referente al solicitante.

Controlador: Tenemos las especificaciones *Encabezado.jwc, PrincipalSolicitante.page, CapturaSolicitante.page, CCapturaSolicitante.jwc* (ver Figura 14.7) y las implementaciones *Encabezado.java, PrincipalSolicitante.java, CapturaSolicitante.java, CCapturaSolicitante.java* (ver Figura 14.8).

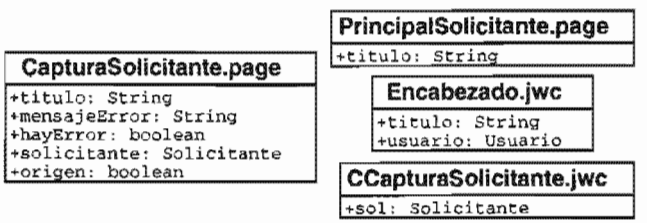

Figura 14.7: Diagrama especificación a nivel Controlador

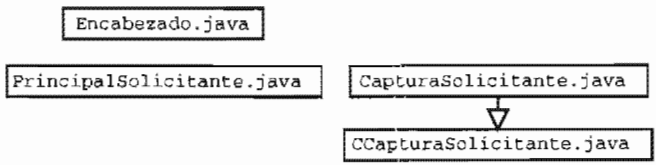

Figura 14.8: Diagrama implementación a nivel Controlador

Donde las implementaciones *CCapturaSolicitante* y *CapturaSolicitante* se encargan de capturar la información del solicitante y de actualizar la base de datos respectivamente.

14.3.4. DESARROLLO

La relación entre las entidades, páginas y componentes se muestran en la Figura 14.9.

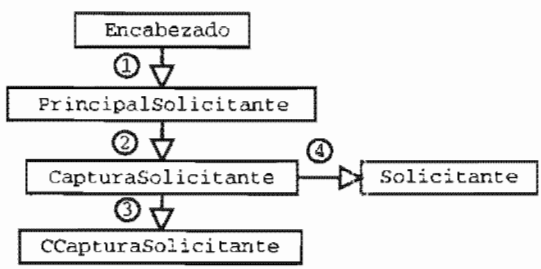

Figura 14.9: Diagrama de colaboración: Crear nuevo solicitante

14.4. RESOLVIENDO HISTORIA 2: *Encontrar un solicitante*

14.4.1. ANÁLISIS

Siguiendo el comportamiento que se ha planteado en las búsquedas anteriores, un usuario podrá encontrar un solicitante por medio de:

- Ingresar cualquier información o parte de la información referente al solicitante (nombre, direccion o teléfono).
- Navegar entre todos los solicitantes registrados por el usuario.

Además el usuario deberá seleccionar y consultar al solicitante deseado. Las subhistorias identificadas para realizar esta historia se muestran en la Figura 14.10.

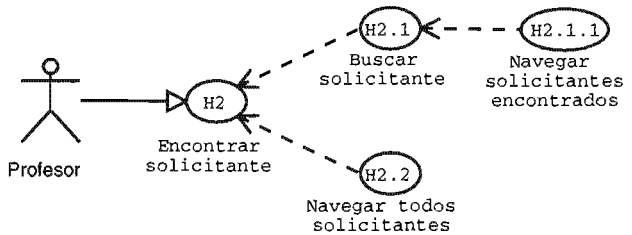

Figura 14.10: Subhistorias encontradas

14.4.2. PRUEBAS

Para realizar esta prueba es necesario que el usuario haya ingresado al sistema. El usuario encontrará un solicitante, ya sea navegando la lista de solicitantes registrados o realizando una búsqueda proporcionado el nombre o la dirección o el teléfono, dado esto debe seleccionar y consultar al solicitante deseado.

Algunos puntos importantes a evaluar son:

- Que existan solicitantes registrados por el usuario.
- Que la información proporcionada por el usuario corresponda con la información de algún solicitante registrado.

14.4.3. DISEÑO

Dado el anális anterior llegamos a las pantallas de las Figuras 14.11 y 14.12.

Modelo: La entidades identificadas son *Usuario, UsuarioSolicitante, Solicitante.*

Las paginas y componentes identificados para la resolución de esta historia son *PrincipalSolicitante, CBusquedaSolicitante, Paginador y ConsultaSolicitante.*

Figura 14.11: Encontrar un solicitante

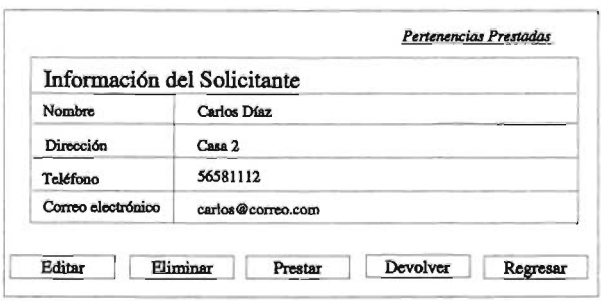

Figura 14.12: Consultar un solicitante

Vista: Las plantillas son *PrincipalSolicitante.html, CBusquedaSolicitante.html, Paginador.html y ConsultaSolicitante.html* (ver Figura 14.13).

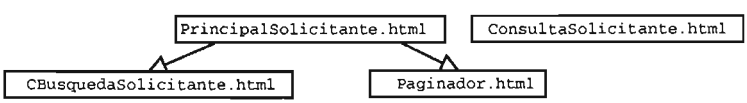

Figura 14.13: Diagrama plantilla a nivel Vista

Donde la plantilla *PrincipalSolicitante* es la encargada de desplegar la información de los solicitantes registrados por el usuario. La plantilla *CBusquedaSolicitante* se encarga de capturar la información que proporciona el usuario para realizar la búsqueda y la plantilla *ConsultaSolicitante* despliega la información del solicitante.

Controlador: Tenemos las especificaciones *PrincipalSolicitante.page, CBusquedaSolicitante.jwc, Paginador.jwc y ConsultaSolicitante.page* (ver Figura 14.14) y las implementaciones *PrincipalSolicitante.java, CBusquedaSolicitante.java, Paginador.java y ConsultaSolicitante.java* (ver Figura 14.15).

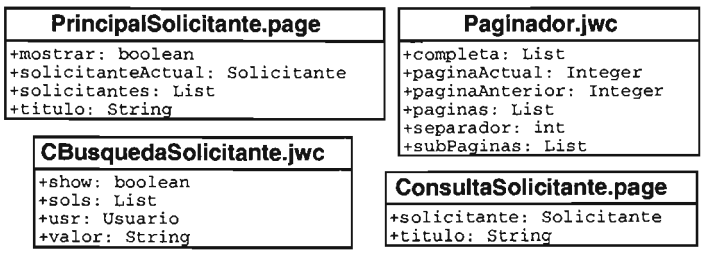

Figura 14.14: Diagrama especificación a nivel Controlador

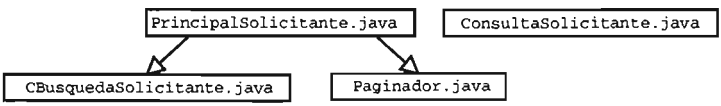

Figura 14.15: Diagrama implementación a nivel Controlador

Donde la implementación *CBusquedaSolicitante* es la encargada de buscar los solicitantes registrados por un usuario. La implementación *Paginador* se encarga de dividir la lista de resultados de la búsqueda en sublistas.

14.4.4. DESARROLLO

La relación que existe entre las entidades, páginas y componentes se muestra en la Figura 14.16.

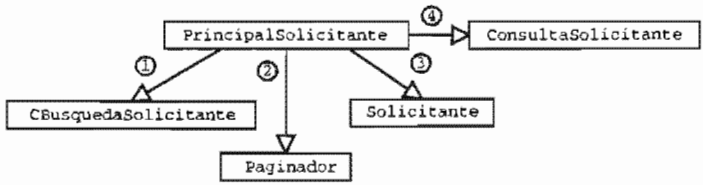

Figura 14.16: Diagrama de colaboración: Encontrar un solicitante

14.5. RESOLVIENDO HISTORIA 3: *Editar un solicitante*

14.5.1. ANÁLISIS

El usuario podrá editar el nombre, la dirección, el teléfono y el correo electrónico de un solicitante.

Las subhistorias identificadas en esta historia se muestran en la Figura 14.17.

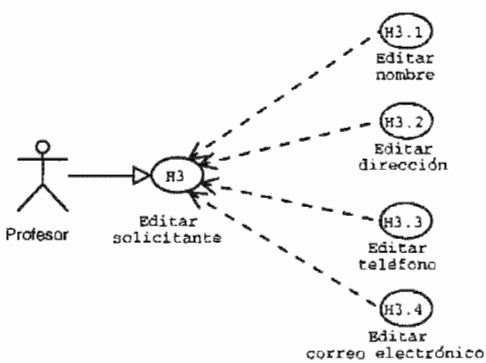

Figura 14.17: Subhistorias encontradas

14.5.2. PRUEBAS

Para llevar a cabo esta prueba es necesario que el usuario haya encontrado al solicitante a editar (ver sección 14.4).

El usuario deberá editar el nombre, la dirección, el teléfono o el correo electrónico del solicitante.

Algunos puntos importantes a evaluar son:

- Que el usuario ingrese al menos el nombre del solicitante.

14.5.3. DISEÑO

Dado el análisis anterior se llegó a la pantalla de la Figura 14.18.

Figura 14.18: Editar un solicitante

Modelo: Las entidades identificadas son *Usuario, UsuarioSolicitante y Solicitante.*

No se fue necesario definir más páginas o componentes para realizar esta historia, únicamente se reutilizaron las páginas y componentes *ConsultaSolicitante, CapturaSolicitante y CCapturaSolicitante*

Vista: Las plantillas son *ConsultaSolicitante.html, CapturaSolicitante.html y CCapturaSolicitante.html* (ver Figura 14.19).

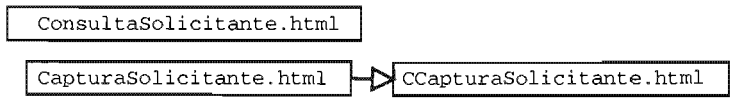

Figura 14.19: Diagrama plantilla a nivel Vista

Donde la plantilla *ConsultaSolicitante* da la opción de editar. La plantilla *CapturaSolicitante* despliega al componente *CCapturaSolicitante* que es el encargado de capturar la información del solicitante a editar.

Controlador: Tenemos las especificaciones *ConsultaSolicitante.page, CapturaSolicitante.page y CCapturaSolicitante.jwc* (ver Figura 14.20) y las implementaciones *ConsultaSolicitante.java, CapturaSolicitante.java y CCapturaSolicitante.java* (ver Figura 14.20).

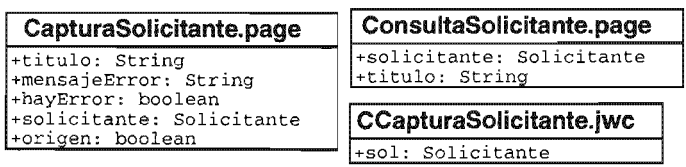

Figura 14.20: Diagrama especificación a nivel Controlador

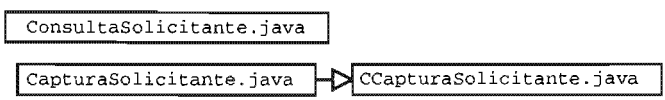

Figura 14.21: Diagrama implementación a nivel Controlador

Donde la implementación *ConsultaSolicitante* redirecciona a la página *CapturaSolicitante*. La implementación *CapturaSolicitante* actualiza la base de datos con la información que capturó el componente *CCapturaSolicitante*.

14.5.4. DESARROLLO

La relación que existe entre las entidades, páginas y componentes se muestra en la Figura 14.22.

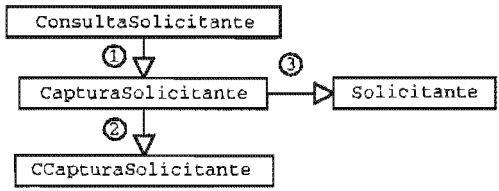

Figura 14.22: Diagrama de colaboración: Editar un solicitante

14.6. RESOLVIENDO HISTORIA 4: *Eliminar un solicitante*

14.6.1. ANÁLISIS

Para llevar a cabo la eliminación de un solicitante solo es necesario que el usuario encuentre, seleccione y consulte al solicitante a eliminar y confirme su eliminación.

Las subhistorias identificadas para esta historia se muestran en la Figura 14.23.

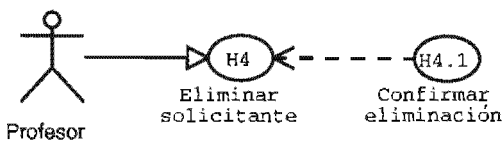

Figura 14.23: Subhistorias encontradas

14.6.2. PRUEBAS

Esta prueba requiere que el usuario encuentre, seleccione y consulte (ver sección 14.4) al solicitante que desea eliminar y confirme su eliminación.

14.6.3. DISEÑO

Dado el análisis anterior se llegó a la pantalla de la Figura 14.24.

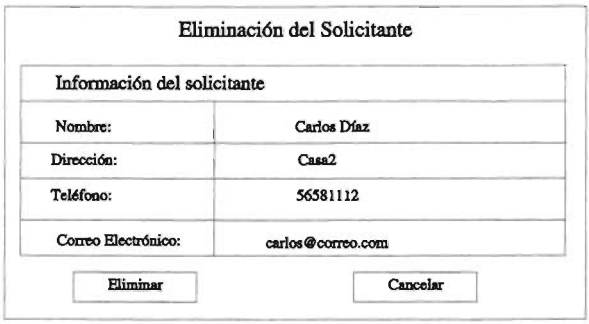

Figura 14.24: Eliminar un solicitante

Modelo: Las entidades identificadas son *Usuario, UsuarioSolicitante y Solicitante.*

Las páginas y componentes identificados para realizar esta historia son *ConsultaSolicitante y BorraSolicitante.*

Vista: Las plantillas son *ConsultaSolicitante.html y BorraSolicitante.html* (ver Figura 14.25).

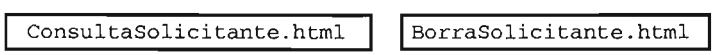

Figura 14.25: Diagrama plantilla a nivel Controlador

Donde la plantilla *ConsultaSolicitante* da la opción para eliminar al solicitante. La plantilla *BorraSolicitante* se encarga de mostrar la información del solicitante a eliminar.

Controlador: Tenemos las especificaciones *ConsultaSolicitante.page y BorraSolicitante.page* (ver Figura 14.26) y las implementaciones *ConsultaSolicitante.java y BorraSolicitante.java* (ver Figura 14.27).

Donde la implementación *ConsultaSolicitante* redirecciona a la página *BorraSolicitante.* La implementación *BorraSolicitante* se encarga de actualizar la base de datos.

Figura 14.26: Diagrama especificación a nivel Controlador

Figura 14.27: Diagrama implememtación a nivel Controlador

14.6.4. DESARROLLO

La relación que existe entre las entidades, páginas y componentes se muestra en la Figura 14.28.

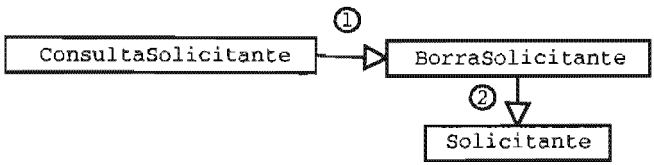

Figura 14.28: Diagrama de colaboración: Eliminar un solicitante

CAPÍTULO 15

ITERACIÓN 5: PRÉSTAMOS

15.1. INTRODUCCIÓN

Como sabemos muchos profesores comparten sus pertenencias con otros profesores, alumnos e incluso personas en general, por lo tanto en esta iteración se pretende realizar todas aquellas historias que permitan al profesor llevar un control de los préstamos que realice a los solicitantes, así como el registro de la devolución y envío de mensajes para solicitar la devolución de su pertenencia.

15.2. PLANEACIÓN DE LA ITERACIÓN

Las principales historias que identificamos para esta iteración son: *Prestar una pertenencia a un solicitante dada la información de la pertenencia*, *Devolver una pertenencia dada la información de la pertenencia*, *Prestar una pertenencia dada la información del solicitante*, *Devolver una pertenencia dada la información del solicitante*, *Consultar todas las pertenencias prestadas de una categoría*, *Consultar todas las pertenencias prestadas a un solicitante*, *Enviar aviso de solicitud de devolución* (ver Figura 15.1).

15.3. RESOLVIENDO HISTORIA 1: *Prestar una pertenencia a un solicitante dada la información de la pertenencia*

15.3.1. ANÁLISIS

Lo que se quiere es que el usuario pueda registrar el préstamo de la pertenencia que él seleccione de entre las pertenencias, tras lo cual él deberá establecer al solicitante al que desea prestarle la pertenencia. Por establecer al solicitante nos

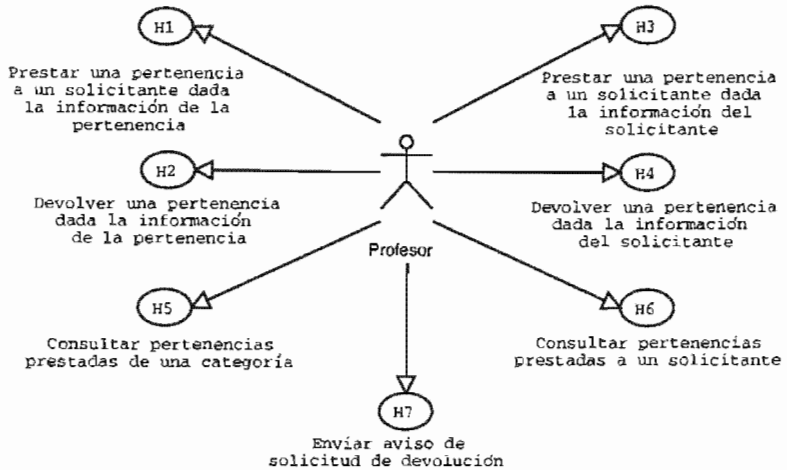

Figura 15.1: Historias de la Iteración

referimos a que el usuario deberá encontrar a un solicitante ya registrado o bien registrar la información del solicitante.

Las subhistorias identificadas para esta historia se muestran en la Figura 15.2.

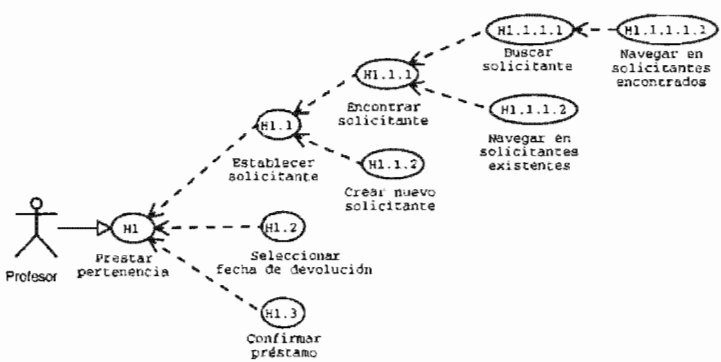

Figura 15.2: Subhistorias encontradas

15.3.2. PRUEBAS

Para llevar llevar a cabo esta historia es necesario que el usuario haya creado una pertenencia anteriormente.

El usuario deberá seleccionar la categoría de la pertenencia, seleccionar y consultar la pertenencia que desea prestar, establecer al solicitante (navegar entre todos los solicitantes registrados, realizar una búsqueda, o crear un nuevo solicitante), seleccionar la fecha de devolución y confirmar que efectivamente quiere registrar el préstamo de esa pertenencia a ese solicitante.

Algunos puntos importantes a evaluar son:

- Que establezca una fecha de devolución.
- Que la fecha de devolución sea posterior al día que se está efectuando el préstamo.

15.3.3. DISEÑO

Al realizar esta historia llegamos a las pantallas de la Figura 15.3, 15.4 y 15.5.

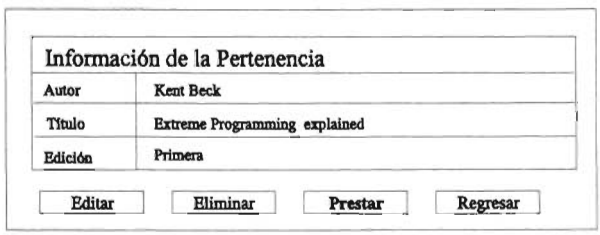

Figura 15.3: Consultar la pertenencia a prestar

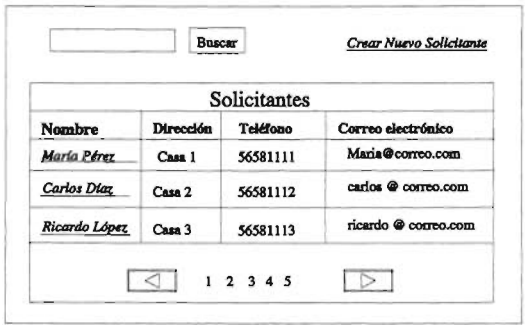

Figura 15.4: Establecer al solicitante

Figura 15.5: Confirmar préstamo

Modelo: Las entidades identificadas son *Usuario, UsuarioCategoria, Categoria, AtributoCategoria, TipoAtributo, AtributoDato, UsuarioSolicitante, Solicitante, PrestamoDato* (ver Figura 15.6).

Las páginas y componentes identificados son *ConsultaPertenencia, Prestamo, CBusquedaSolicitante, Paginador, ConfirmacionPrestamo*.

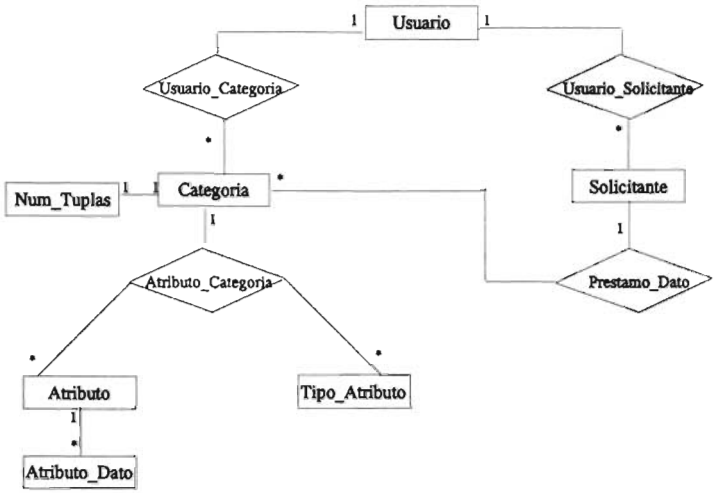

Figura 15.6: Diagrama entidades a nivel modelo

Vista: Las plantillas son *ConsultaPertenencia.html, Prestamo.html, CBusqueda-Solicitante.html, Paginador.html, ConfirmacionPrestamo.html* (ver Figura 15.7).

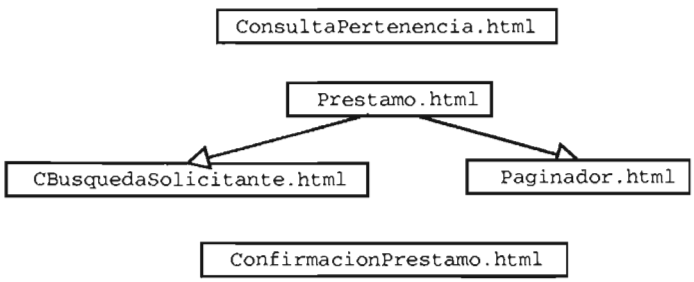

Figura 15.7: Diagrama plantilla a nivel Vista

Donde la pantilla *ConsultaPertenencia* se encarga de desplegar los datos de la pertenencia seleccionada y de dar la opción de préstamo. La plantilla *Prestamo* se encarga de desplegar la lista de solicitantes dando la opción de que el usuario seleccione al solicitante o cree uno nuevo. La plantilla *ConfirmacionPrestamo* despliega la información de la pertenencia, la información del solicitante y el componente que permite seleccionar o capturar la fecha de devolución.

Controlador: Tenemos las especificaciones *ConsultaPertenencia.page, Prestamo.page, CBusquedaSolicitante.jwc, Paginador.jwc, ConfirmaciónPrestamo.page* (ver Figura 15.8), las implementaciones *ConsultaPertenencia.java, Prestamo.java, CBusquedaSolicitante.java, Paginador.java, ConfirmaciónPrestamo.java* (ver Figura 15.9) y las clases auxilares *Pertenencia.java y SolicitanteReducido.java.*

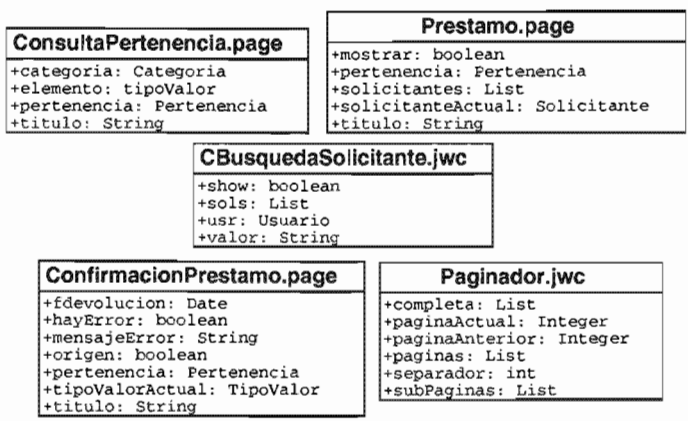

Figura 15.8: Diagrama especificación a nivel Controlador

Donde la implementación *ConsultaPertenencia* redirecciona a la página *Prestamo*, la implementación *Prestamo* se encarga ya sea de redireccionar a la página *ConfirmacionPrestamo* pasándole el solicitante establecido, o redirecciona a la página *CapturaSolicitante* en el caso de que el usuario quiera registrar la información de un nuevo solicitante. La implementación *ConfirmacionPrestamo* actualizará la lista de entidades *PrestamoDato* del solicitante en la base de datos.

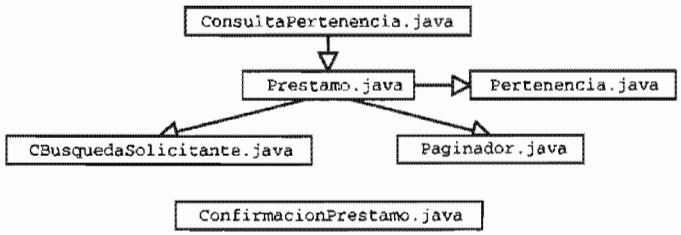

Figura 15.9: Diagrama implementación a nivel Controlador

La clase *Pertenencia* tendrá un *SolicitanteReducido* para que cada *Pertenencia* tenga determinado si está prestado y a que solicitante.

15.3.4. DESARROLLO

La relación que existe entre las entidades, páginas, componentes y clases auxiliares se muestra en la Figura 15.10.

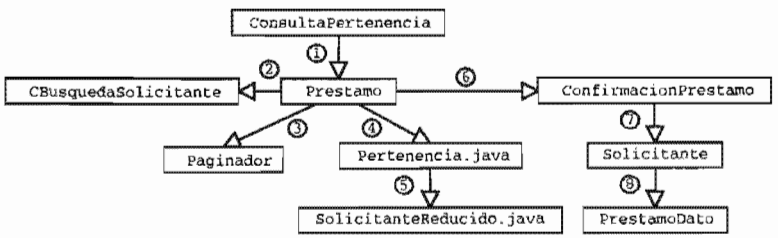

Figura 15.10: Diagrama de colaboración: Prestar una pertenencia

15.4. RESOLVIENDO HISTORIA 2: *Devolver una pertenencia dada la información de la pertenencia*

15.4.1. ANÁLISIS

Dado que ya se dió la posibilidad de registrar los préstamos el siguiente paso será permitir que el usuario registre la devolución de una pertenencia. Para llevar a cabo esto bastará con que el usuario localice a la pertenencia (identificándola por un símbolo que represente que la pertenencia se encuentra prestada), la consulte y confirme la devolución.

Las subhistorias encontradas se muestran en la Figura 15.11.

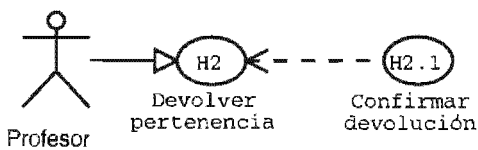

Figura 15.11: Subhistorias encontradas

15.4.2. PRUEBAS

Para llevar a cabo esta prueba es necesario que anteriormente se haya registrado el préstamo de la pertenencia.

El usuario debe seleccionar la categoría de pertenencia, establecer a la pertenencia a devolver, consultar la pertenencia y confirmar la devolución de la pertenencia.

15.4.3. DISEÑO

Dado el análisis anterior se llegaron a las pantallas de las Figuras 15.12 y 15.13.

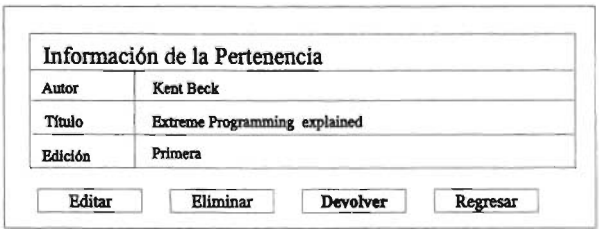

Figura 15.12: Consultar la pertenencia a devolver

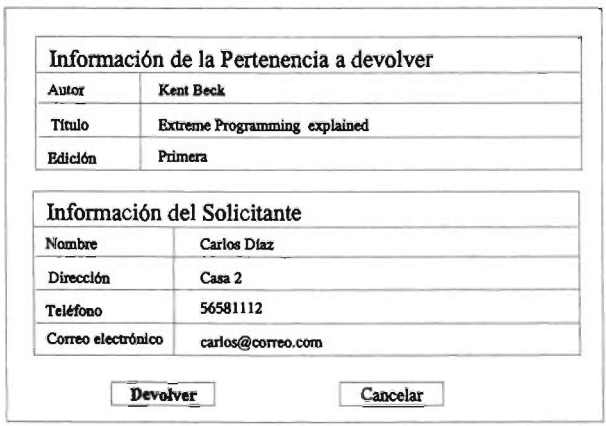

Figura 15.13: Confirmar devolución

Modelo: Las entidades identificadas son *Usuario, UsuarioSolicitante, Solicitante, PrestamoDato.*

Las páginas y componentes identificados son *ConsultaPertenencia, ConfirmacionDevolucion.*

Vista: Las plantillas son *ConsultaPertenencia.html, ConfirmacionDevolucion.html* (ver Figura 15.14).

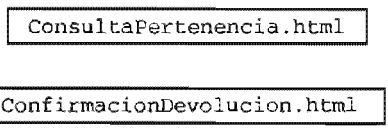

Figura 15.14: Diagrama plantilla a nivel Vista

Donde la plantilla *ConsultaPertenencia* da la opción para devolver una pertenencia y la plantilla *ConfirmacionDevolucion* despliega la información de la pertenencia que se está devolviendo y la información del solicitante.

Controlador: Tenemos las especificaciones *ConsultaPertenencia.page, ConfirmacionDevolucion.page* (ver Figura 15.15) y las implementaciones *ConsultaPertenencia.java, ConfirmacionDevolucion.java* (ver Figura 15.16).

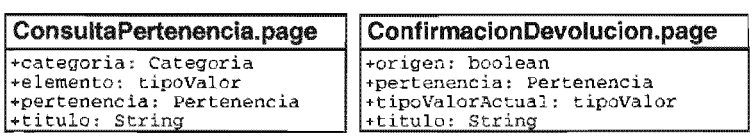

Figura 15.15: Diagrama especificación a nivel Controlador

Donde la implementación *ConsultaPertenencia* redirecciona a la página *ConfirmacionDevolucion* y la implementación *ConfirmacionDevolucion* remueve el préstamo de la lista de préstamos asociados al solicitante, haciendo la actualización en la base de datos.

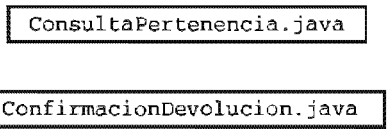

Figura 15.16: Diagrama implementación a nivel Controlador

15.4.4. DESARROLLO

La relación que existe entre las entidades y las páginas se muestra en la Figura 15.17.

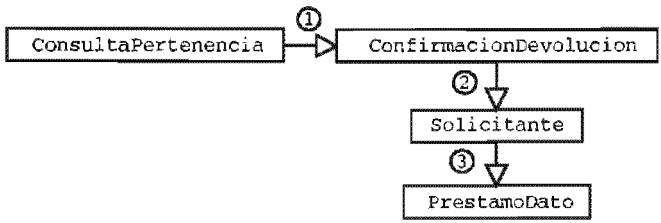

Figura 15.17: Diagrama de colaboración: Devolver pertenencia prestada

15.5. RESOLVIENDO HISTORIA 3: *Prestar una pertenencia dada la información del solicitante*

15.5.1. ANÁLISIS

Se le permitirá al usuario realizar un préstamo a partir de consultar a un solicitante, para así entonces seleccionar la categoría de pertenencia y establecer la pertenencia que desea prestar. Por establecer la pertenencia nos referimos a que el usuario deberá de encontrar la pertenencia registrada anteriormente (entre todas las pertenencias o en una búsqueda) o registrar en ese momento la información de la pertenencia a prestar.

Las subhistorias identificadas para esta historia son las que se muestran en la Figura 15.18.

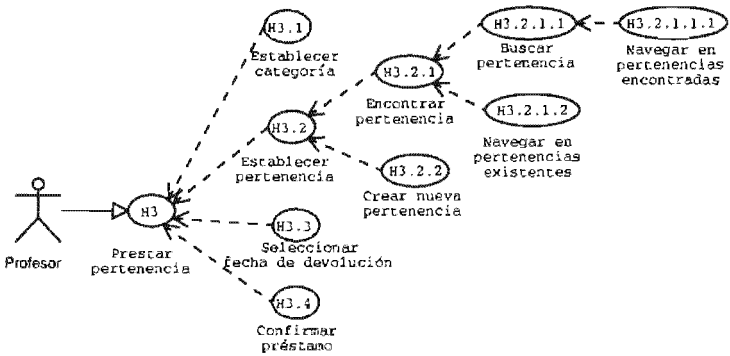

Figura 15.18: Subhistorias encontradas

15.5.2. PRUEBAS

Para llevar a cabo esta prueba es necesario que el usuario haya registrado anteriormente al solicitante.

El usuario deberá encontrar, seleccionar y consultar al solicitante de la pertenencia, seleccionar la categoría de pertenencia, establecer la pertenencia (ya sea navegando entre todas las pertenencias de la categoría, realizando una búsqueda o creando una nueva pertenencia), seleccionar la fecha de devolución y confirmar que efectivamente quiere registrar el préstamo de esa pertenencia a ese solicitante.

Algunos puntos importantes a evaluar son:

- Que establezca una fecha de devolución.
- Que la fecha de devolución sea posterior al día que se está efectuando el préstamo.

15.5.3. DISEÑO

Al realizar esta historia llegamos a las pantallas de las Figuras 15.19, 15.20 y 15.21.

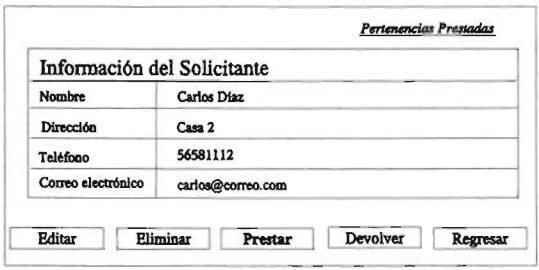

Figura 15.19: Consultar al solicitante

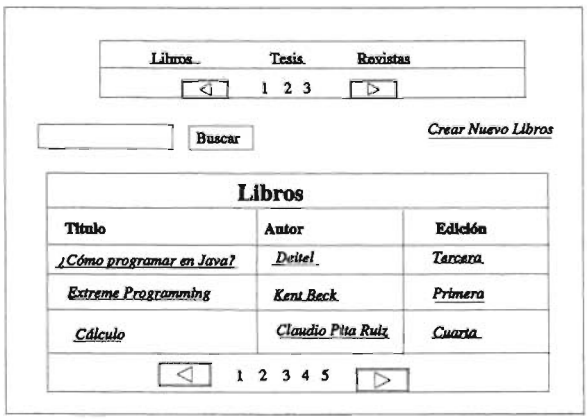

Figura 15.20: Establecer la pertenencia a prestar

Modelo: Las entidades identificadas son *Usuario, UsuarioCategoria, Categoria, AtributoCategoria, TipoAtributo, Atributo, AtributoDato, NumTuplas, UsuarioSolicitante, Solicitante, PrestamoDato.*

Figura 15.21: Confirmar el préstamo

Las páginas y componentes que identificamos son *ConsultaSolicitante, PrestamoSolP, CBusquedaCategoria, CBusquedaPertenencia, Paginador, ConfirmacionPrestamo*.

Vista: Las plantillas son *ConsultaSolicitante.html, PrestamoSolP.html, CBusquedaPertenencia.html, Paginador.html, ConfirmacionPrestamo.html* (ver Figura 15.22).

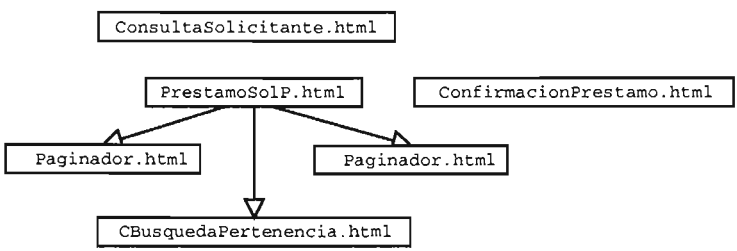

Figura 15.22: Diagrama plantilla a nivel Vista

Donde la plantilla *ConsultaSolicitante* despliega los datos del solicitante y da la opción para prestar. La plantilla *PrestamoSolP* despliega la lista de categorías de pertenencia y la lista de pertenencias que se encuentran disponibles para para prestárselas al solicitante, y la plantilla *ConfirmacionPrestamo* despliega la información de la pertenencia, la información del solicitante y el componente que permite seleccionar o capturar la fecha de devolución.

Controlador: Tenemos las especificaciones *ConsultaSolicitante.page, PrestamoSolP.page, CBusquedaCategoria.jwc, CBusquedaPertenencia.jwc, Paginador.jwc, ConfirmacionPrestamo.page* (ver Figura 15.23), las implementaciones *ConsultaSolicitante.java, PrestamoSolP.java, CBusquedaCategoria.java, CBusquedaPertenencia.java, Paginador.java, ConfirmacionPrestamo.java* (ver Figura 15.24) y las clases auxiliares *Pertenencia.java y SolicitanteReducido.java.*

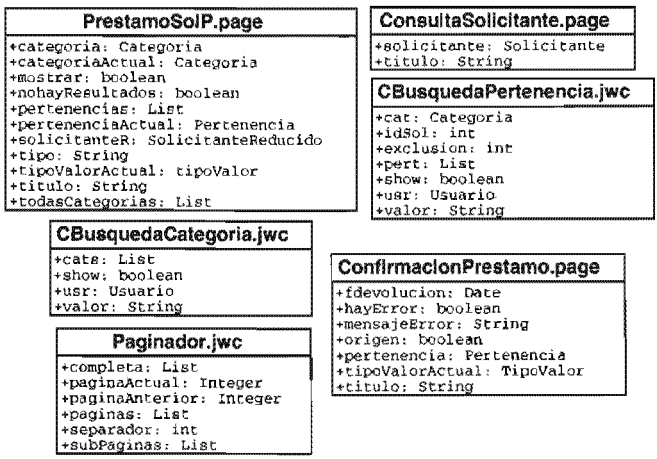

Figura 15.23: Diagrama especificación a nivel Controlador

Donde la implementación *ConsultaSolicitante* se encarga de redireccionar a la página *PrestamoSolP* pasándole el solicitante. La implementación *PrestamoSolP* se encarga de redireccionar ya sea a la página *ConfirmacionPrestamo* pasándole la pertenencia establecida o a la página *CapturaPertenencia* para registrar la información de una nueva pertenencia. La implementación

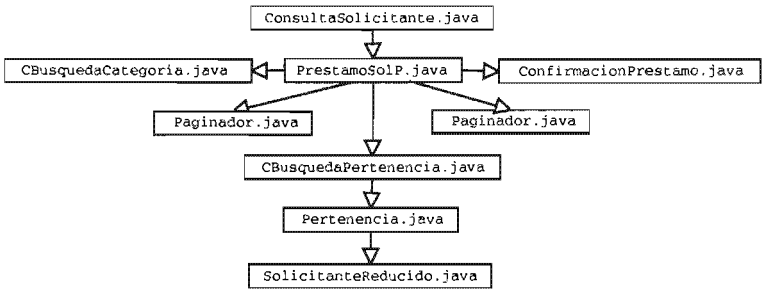

Figura 15.24: Diagrama implementación a nivel Controlador

ConfirmacionPrestamo actualizará la lista de entidades *PrestamoDato* del solicitante en la base de datos.

15.5.4. DESARROLLO

La relación que existe entre las entidades, páginas, componentes y clases auxiliares se muestra en la Figura 15.25.

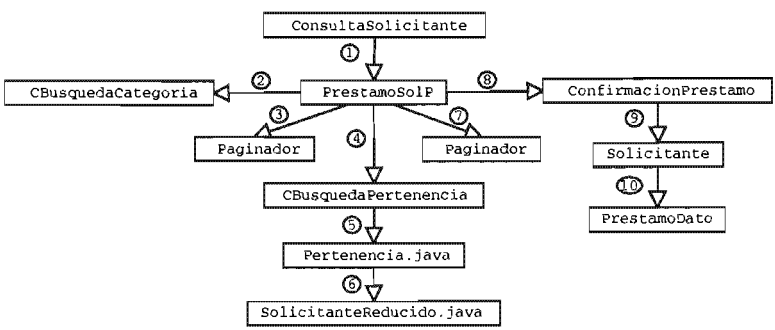

Figura 15.25: Diagrama de colaboración: Prestar una pertenencia

15.6. RESOLVIENDO HISTORIA 4: *Devolver una pertenencia dada la información del solicitante*

15.6.1. ANÁLISIS

Dado que la historia anterior permite realizar el préstamo de una pertenencia dada la información de un solicitante, esta historia se enfocará a que el usuario pueda devolver la pertenencia teniendo la información del solicitante.

Una vez que el usuario ya localizó al solicitante es necesario que lo consulte, seleccione la categoría de pertenencia y enseguida seleccione y confirme la pertenencia que desea devolver.

Las subhistorias encontradas en esta historia se muestran en la Figura 15.26.

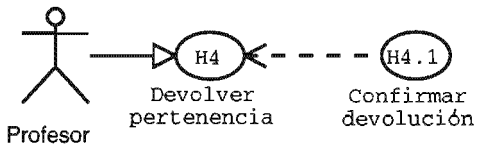

Figura 15.26: Subhistorias encontradas

15.6.2. PRUEBAS

Para realizar esta prueba es necesario que anteriormente se haya registrado el préstamo de la pertenencia al solicitante. El usuario deberá seleccionar al solicitante, seleccionar la categoría de pertenencia, establecer y seleccionar la pertenencia y confirmar la devolución.

15.6.3. DISEÑO

Dado el análisis anterior se llegaron a las pantallas de las Figuras 15.27, 15.28 y 15.29.

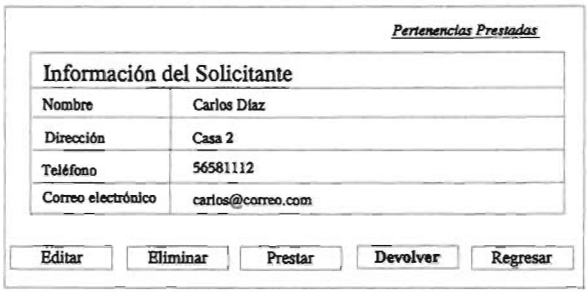

Figura 15.27: Consultar Solicitante

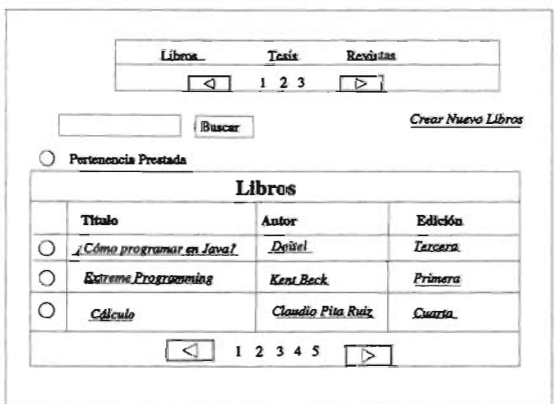

Figura 15.28: Establecer pertenencia

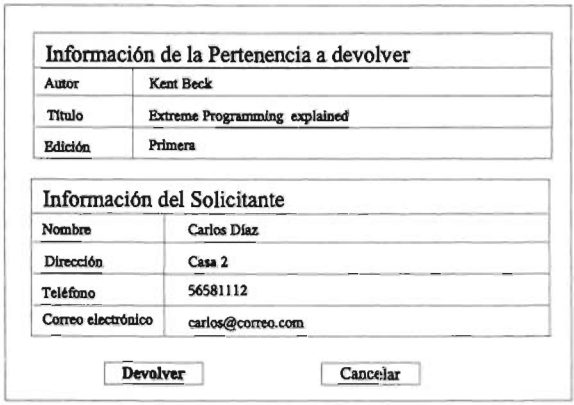

Figura 15.29: Confirmar devolución

Modelo: Las entidades identificadas son *Usuario, UsuarioSolicitante, Solicitante, PrestamoDato.*

Las páginas y componentes identificados para esta historia son *ConsultaSolicitante, PrestamoSolP, CBusquedaCategoria, CBusquedaPertenencia, Paginador y ConfirmacionDevolucion.*

Vista: Las plantillas son *ConsultaSolicitante.html, PrestamoSolP.html, CBusquedaPertenencia.html, Paginador.html y ConfirmacionDevolucion.html* (ver Figura 15.30).

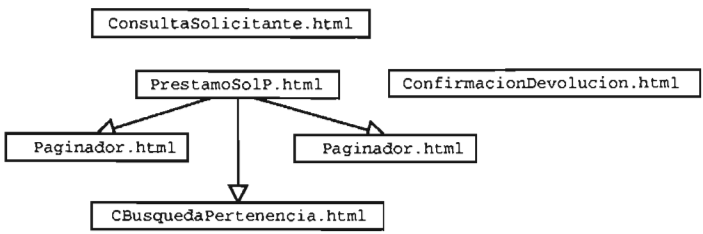

Figura 15.30: Diagrama plantilla a nivel Vista

Donde la plantilla *ConsultaSolicitante* despliega los datos del solicitante y da la opción para devolver. La plantilla *PrestamoSolP* despliega la lista de categorias de pertenencia y la lista de pertenencias prestadas anteriormente al solicitante y la plantilla *ConfirmacionDevolucion* despliega la información de la pertenencia que se esta devolviendo y la información del solicitante.

Controlador: Tenemos las especificaciones *ConsultaSolicitante.page, PrestamoSolP.page, CBusquedaCategoria.jwc, CBusquedaPertenencia.jwc, Paginador.jwc y ConfirmacionDevolucion.page* (ver Figura 15.31) y las implementaciones *ConsultaSolicitante.java, PrestamoSolP.java, CBusquedaCategoria.java, CBusquedaPertenencia.java, Paginador.java y ConfirmacionDevolucion.java* (ver Figura 15.32) y las clases auxiliares *Pertenencia.java y SolicitanteReducido.java.*

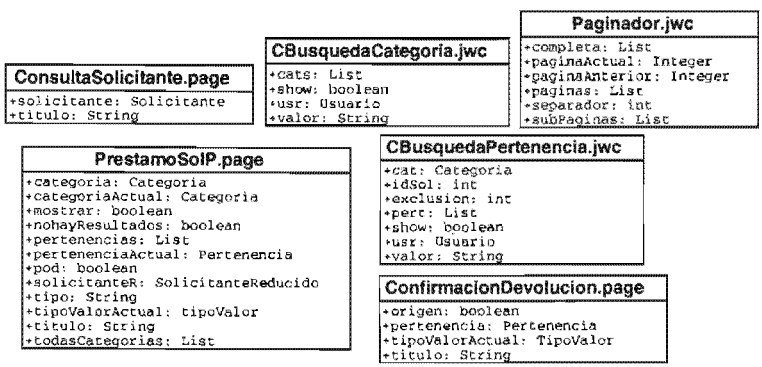

Figura 15.31: Diagrama especificación a nivel Controlador

Donde la implementación *ConsultaSolicitante* se encarga de redireccionar a la página *PrestamoSolP* pasándole el solicitante. La implementación *PrestamoSolP* se encarga de redireccionar a la página *ConfirmacionDevolucion* pasándole la pertenencia establecida. La implementación *ConfirmacionDevolucion* remueve el préstamo de la lista de préstamos asociados al solicitante, haciendo la actualización en la base de datos.

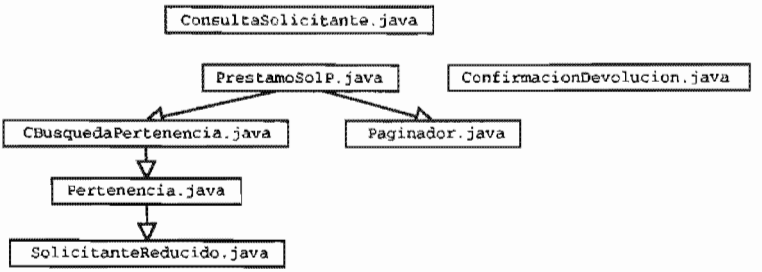

Figura 15.32: Diagrama implementación a nivel Controlador

15.6.4. DESARROLLO

La relación que existe entre las entidades, páginas, componentes y clases auxiliares se muestra en la Figura 15.33.

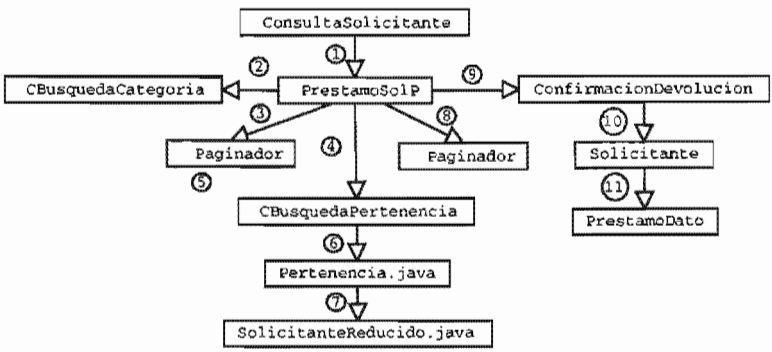

Figura 15.33: Diagrama de colaboración: Devolver una pertenencia prestada

15.7. RESOLVIENDO HISTORIA 5:*Consultar todas las pertenencias prestadas de una categoría*

15.7.1. ANÁLISIS

Esta historia se encargará de proporcionarle al usuario un forma para poder consultar todas las pertenencias prestadas de una categoría así como el nombre del solicitante al que está prestada cada una.

Las subhistorias encontradas para realizar esta historia se muestran en la Figura 15.34.

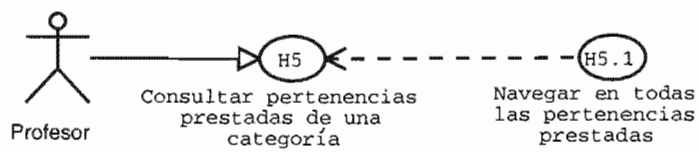

Figura 15.34: Subhistorias encontradas

15.7.2. PRUEBAS

Para llevar a cabo esta historia es necesario que el usuario haya ingresado a la sección de las pertenencias.

El usuario deberá seleccionar la categoría de pertenencia y elegir la opción que te permite consultar las pertenencias prestadas de esa categoría.

Algunos puntos importantes a evaluar en esta prueba son:

- Que exista al menos una pertenencia prestada de esa categoría.

15.7.3. DISEÑO

Dado el análisis anterior se llegó a la pantalla de la Figura 15.35.

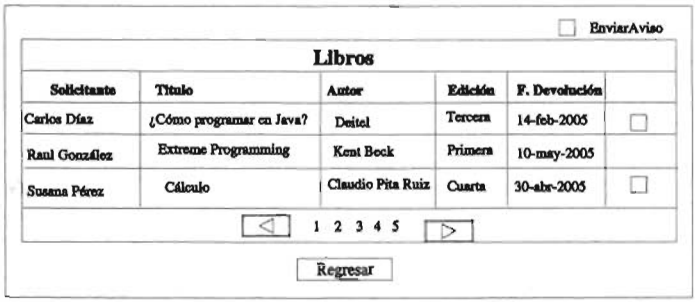

Figura 15.35: Consultar todas las pertenencias prestadas de una categoría

Modelo: Las entidades identificadas son *Usuario, UsuarioCategoria, Categoria, AtributoCategoria, TipoAtributo, Atributo, AtributoDato, UsuarioSolicitante, Solicitante, PrestamoDato.*

Las páginas y componentes identificados son *CategoriaSeleccionada, CBusquedaPertenencia, Paginador y PertenenciasPrestadas.*

Vista: Las plantillas son *CategoriaSeleccionada.html, Paginador.html y PertenenciasPrestadas.html* (ver Figura 15.36).

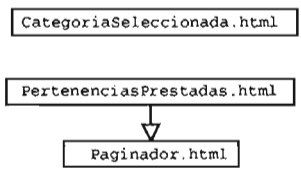

Figura 15.36: Diagrama plantilla a nivel Vista

Donde la plantilla *CategoriaSeleccionada* da la opción de consultar las pertenencias prestadas de la categoría seleccionada. La plantilla *PertenenciasPrestadas* se encarga de desplegar la lista de las pertenencias prestadas y el nombre del solicitante de cada una de ellas.

Controlador: Las especificaciones son *CategoriaSeleccionada.page, CBusquedaPertenencia.jwc, Paginador.jwc y PertenenciasPrestadas.page*

(ver Figura 15.37), las implementaciones *CategoriaSeleccionada.java, CBusquedaPertenencia.java, Paginador.java y PertenenciasPrestadas.java* (ver Figura 15.38) y las clases auxiliares *Pertenencia.java y SolicitanteReducido.java.*

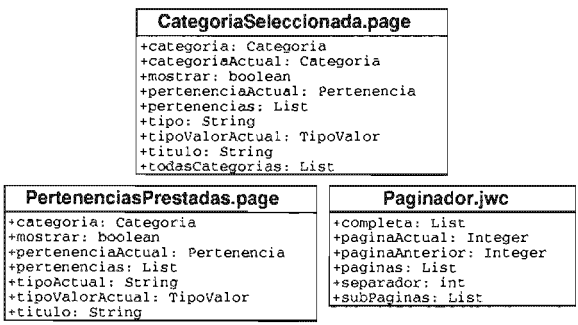

Figura 15.37: Diagrama especificación a nivel Controlador

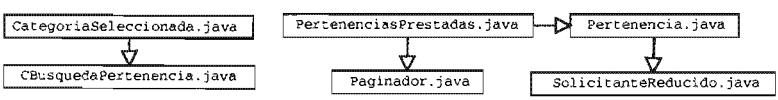

Figura 15.38: Diagrama implementación a nivel Controlador

Donde la implementación *CategoriaSeleccionada* redirecciona a la página *PertenenciasPrestadas.*

15.7.4. DESARROLLO

La relación entre las páginas, componentes y clases auxiliares se muestra en la Figura 15.39.

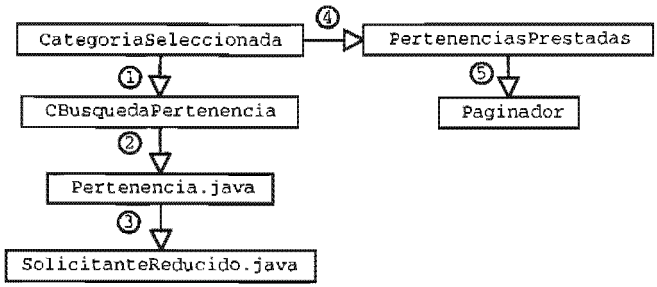

Figura 15.39: Diagrama de colaboración: Pertenencias prestadas de una categoría

15.8. RESOLVIENDO HISTORIA 6:*Consultar todas las pertenencias prestadas a un solicitante*

15.8.1. ANÁLISIS

Esta historia búsca proveer al usuario la opción de poder consultar todas las pertenencias prestadas a un solicitante.

Las subhistorias encontradas en esta historia se muestran en la Figura 15.40.

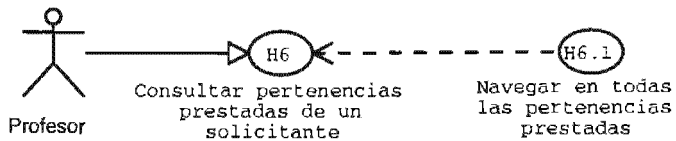

Figura 15.40: Subhistorias encontradas

15.8.2. PRUEBAS

Para llevar esta prueba es necesario que se le haya prestado al menos una pertenencia al solicitante.

El usuario deberá encontrar y consultar al solicitante, elegir la opción que permite consultar las pertenencias prestadas de ese solicitante.

15.8.3. DISEÑO

Dado el análisis anterior se llegó a la pantalla de la Figura 15.41.

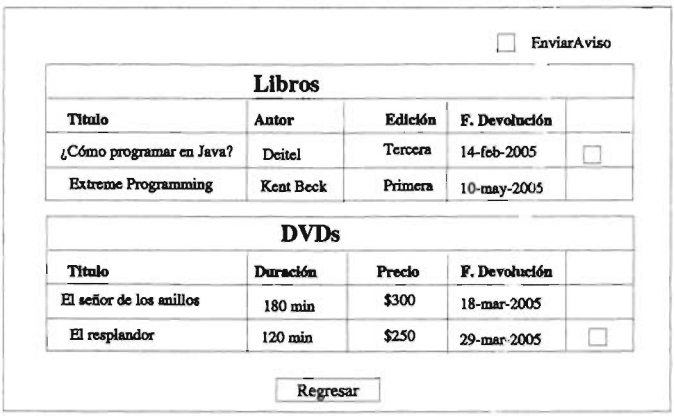

Figura 15.41: Consultar pertenencias prestadas a un solicitante

Modelo: Las entidades identificadas son *Usuario, UsuarioCategoria, Categoria, AtributoCategoria, TipoAtributo, Atributo, AtributoDato, UsuarioSolicitante, Solicitante, PrestamoDato.*

Las páginas y componentes identificados son *ConsultaSolicitante, CBusquedaPertenencia y PertenenciasSolicitante.*

Vista: Las plantillas son *ConsultaSolicitante.html, y PertenenciasSolicitante.html* (ver Figura 15.42).

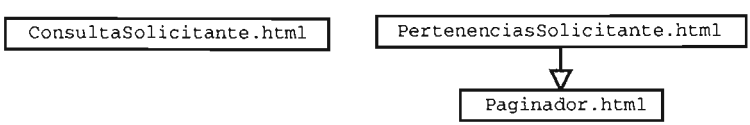

Figura 15.42: Diagrama plantilla a nivel Vista

Donde la plantilla *ConsultaSolicitante* despliega la información del solicitante y da la opción para consultar las pertenencias prestadas a ese solicitante. La plantilla *PertenenciasSolicitante* despliega las listas de las pertenencias prestadas a ese solicitante de cada categoría de pertenencia.

Controlador: Tenemos las especificaciones *ConsultaSolicitante.page, CBusquedaPertenencia.jwc y PertenenciasSolicitante.page* (ver Figura 15.43), las implementaciones *ConsultaSolicitante.java, CBusquedaPertenencia.java y PertenenciasSolicitante.java* (ver Figura 15.44) y las clases auxiliares *Pertenencia.java y SolicitanteReducido.java.*

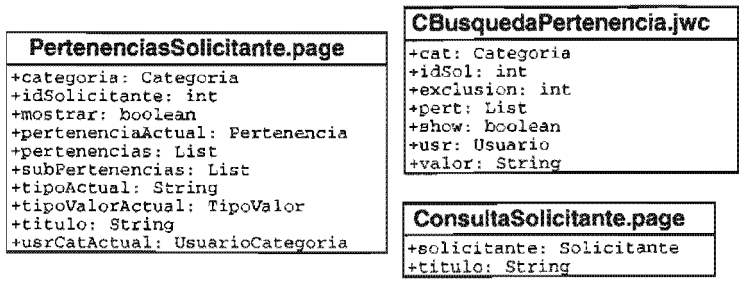

Figura 15.43: Diagrama especificación a nivel Controlador

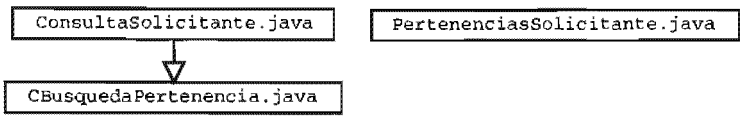

Figura 15.44: Diagrama implementación a nivel Controlador

Donde la implementacion *ConsultaSolicitante* redirecciona a la página *PertenenciaSolicitante.*

15.8.4. DESARROLLO

La relación que existe entre las páginas, componentes y clases auxiliares se muestra en la Figura 15.45.

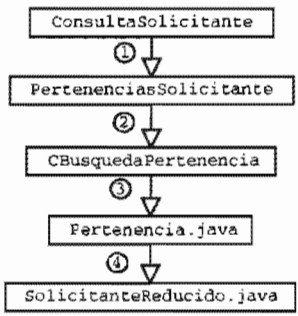

Figura 15.45: Diagrama de colaboración: Pertenencias prestadas a un solicitante

15.9. RESOLVIENDO HISTORIA 7: *Enviar aviso de solicitud de devolución*

15.9.1. ANÁLISIS

Puede suceder que los solicitantes se retrasen en la devolución de alguna pertenencia, por lo cual el usuario querrá enviar un aviso de solicitud de devolución de ésta, es por esto que esta historia se enfocará en proveer al usuario la posibilidad de enviar este aviso a aquellos solicitantes que tengan registrado su correo electrónico.

Esta funcionalidad la podrá llevar a cabo el usuario ya sea por la consulta de todas las pertenencias prestadas de una categoría (ver sección 15.7) o por consultar las pertenencias prestadas a un solicitante (ver sección 15.8).

Las subhistorias encontradas para esta historia se muestran en la Figura 15.46.

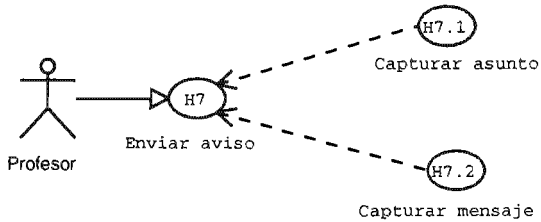

Figura 15.46: Subhistorias encontradas

15.9.2. PRUEBAS

Para llevar a cabo esta prueba es necesario que el usuario haya consultado todas las pertenencias prestadas de una categoría o las pertenencias prestadas a un solicitante.

El usuario deberá elegir la opción para enviar el aviso de una pertenencia en particular, deberá capturar el asunto y editar el mensaje que desea enviar.

Algunos puntos importantes a evaluar son:

- Que haya provisto la información necesaria (destinatario, remitente y asunto).

15.9.3. DISEÑO

Dado el análisis anterior se llegó a la pantalla de la Figura 15.47.

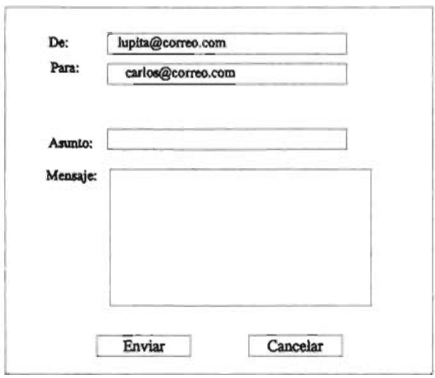

Figura 15.47: Enviar aviso de solicitud de devolución

Modelo: No se identificaron entidades.

Las páginas y componentes identificados son *PertenenciasPrestadas, PertenenciasSolicitante y EnviaCorreo.*

Vista: Las plantillas son *PertenenciasPrestadas.html, PertenenciasSolicitante.html y EnviaCorreo.html* (ver Figura 15.48).

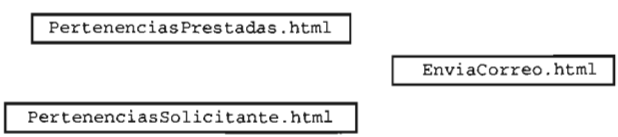

Figura 15.48: Diagrama plantilla a nivel Vista

Donde las plantillas *PertenenciasPrestadas y PertenenciasSolicitante* dan la opción de enviar un aviso de solicitud de devolución de la pertenencia. La plantilla *EnviaCorreo* se encarga de capturar la información necesaria para enviar el aviso.

Controlador: Tenemos las especificaciones *PertenenciasPrestadas.html, PertenenciasSolicitante.html y EnviaCorreo.html* (ver Figura 15.49), las implementaciones *PertenenciasPrestadas.java, PertenenciasSolicitante.java y EnviaCorreo.java* (ver Figura 15.50) y la clase auxiliar *MailUtils.java.*

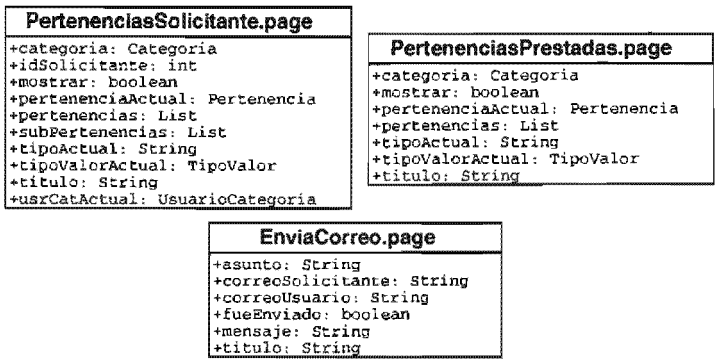

Figura 15.49: Diagrama especificación a nivel Controlador

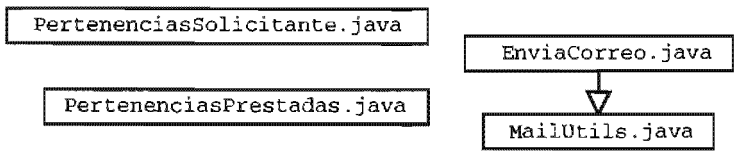

Figura 15.50: Diagrama implementación a nivel Controlador

Donde las implementaciones *PertenenciasPrestadas y PertenenciasSolicitante* redireccionan a la página *EnviaCorreo.* La implementación *EnviaCorreo* se encarga de enviar el correo utilizando la clase auxiliar *MailUtils.*

15.9.4. DESARROLLO

La relación que existe entre las páginas, componentes y la clase auxiliar se muestra en la Figura 15.51.

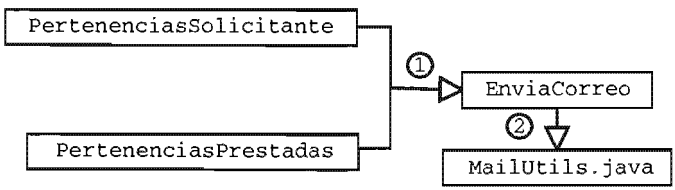

Figura 15.51: Diagrama de colaboración: Enviar Aviso

CONCLUSIONES

15.10. PROGRAMACIÓN EXTREMA

Lo que podemos concluir sobre ésta metodología es que es fácil de seguir, es práctica dado que el proceso de documentación se reduce y promueve buenas prácticas de programación, de entre las cuales la que consideramos más útil fue la *programación por pares*, la cual nos permitió desarrollar un producto de calidad con el apoyo, asesoramiento y corrección mútua. En general estamos convencidas de que es una buena metodología dado que se enfoca en la producción y pruebas constantes del código más que en la generación masiva de documentos.

Aunque XP promueve la poca generación de documentación nosotras buscamos llegar a un equilibrio en el cual no perdieramos las prácticas aprendidas de las metodologías monumentales (TSPi, RUP, etc.), es decir establecimos algunos documentos que nos permitieran representar las ideas fundamentales del sistema.

15.11. TECNOLOGÍAS

En cuanto a las tecnologías utilizadas podemos concluir que cumplen con su propósito: que el programador pueda producir una aplicación más sofisticada y robusta de manera rápida y eficiente, aunque también nos enfrentamos con el problema de que son muy recientes, por lo que no se cuenta con mucha información o ejemplos para aprenderlas, teniendo así que la curva de aprendizaje fue mucho más grande de lo que esperabamos.

Por estas razones creemos que el presente trabajo será de utilidad para todas aquellas personas que quieran incursionar en estas tecnologías, pues en la primera

parte ofrecemos un panorama general de ellas a fin de que cuenten con la información básica para empezar a utilizarlas. Además, proveemos el código del sistema, que es un ejemplo concreto del uso de dichas tecnologías y su interacción, y cuyo diseño fue ampliamente explicado en la segunda parte del trabajo.

15.12. SISTEMA

Sobre la aplicación Web desarrollada podemos decir que será de gran utilidad para los profesores de la facultad ya que, como se ha mencionado a lo largo de la tesis, ésta les permitirá llevar un control de sus pertenencias a fin de no perderlas y poder seguir proporcionando sus materiales de apoyo a futuras generaciones.

Algunas mejoras a la aplicación que proponemos para el futuro son:

- Aumentar el nivel de seguridad.
- Que no sólo los profesores de la facultad tengan acceso a la aplicación.
- Que las consultas sobre las pertenencias prestadas se puedan imprimir.
- Que el sistema tenga funcionalidades para usuarios avanzados.
- Que el sistema permita a los estudiantes de la facultad consultar algunas de las pertenencias de los profesores a fin de saber si ellos poseen alguna pertenencia en particular para solicitárselas posteriormente.

15.13. PERSONALES

Pocos trabajos durante nuestra vida académica en la Facultad de Ciencias retaron nuestros conocimientos, nuestra entereza y nuestra imaginación como la realización de esta tesis.

A lo largo de cuatro años de estudio en la Facultad de Ciencias adquirimos la técnica, las bases, el conocimiento y la motivación para emprender la aventura que significó este trabajo. Gracias a nuestro recorrido tecnológico y a la ejecución de un equipo profesional de desarrollo, tuvimos la oportunidad de ensayar y poner en práctica una serie de habilidades que difícilmente hubiésemos entendido hace unos cuantos meses.

Investigamos, nos pusimos retos donde existían sólo ideas, nos ceñimos a una metodología seria y mundialmente reconocida de desarrollo de software. Entender las tecnologías e integrarlas para construir un sistema nos deja un grato sabor de boca ya que nos ayudará para continuar nuestro camino.

Por supuesto, más de una vez nos encontramos con obstáculos que parecían infranqueables, pero empujamos, insistimos y logramos concluir un proyecto del cual estamos orgullosas.

BIBLIOGRAFÍA

[1] Christian Bauer. *Hibernate in action.* Manning, 2005.

[2] Dan Brown. *A Beginners Guide to UML Part I.* Dustan Thomas Consulting, 2003. `http://consulting.dthomas.co.uk`.

[3] Dan Brown. *A Beginners Guide to UML Part II.* Dustan Thomas Consulting, 2004. `http://consulting.dthomas.co.uk`.

[4] Debbie Bode Carson. *J2EE Tutorial.* Sun Microsystems Press, Abril 2002. `http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/JSPIntro7.html`.

[5] Hibernate community. *Hibernate.* 2005. `http://www.hibernate.org`.

[6] Drew Davidson. *The OGNL Expresion and Binding Language.* Tucson Java User's Group, Marzo 2004.

[7] Drew Davidson. *The OGNL Language Guide.* OGNL Technology, Inc, 2004.

[8] Javier Garcia de Jalon y Jose Ignacio Rodriguez. *Aprenda Servlets de Java como si estuviera en segundo.* Universidad de Navarra, Abril 1999.

[9] National Centre for Software Technology. *Database Programming with Java (JDBC).* Centre for Development of Advanced Computing, 2004.

[10] Apache Software Foundation. *Tapestry.* Apache Software Foundation, 2005. `http://jakarta.apache.org/tapestry/`.

[11] David M. Geary. *Advanced JavaServer Pages.* Sun Microsystems Press y Prentice Hall, Mayo 2004.

[12] Marty Hall. *Core Servlets and JavaServer Pages (JSP).* Sun Microsystems Press y Prentice Hall, segunda edicion edition, 2003. `http://www.apl.jhu.edu/~hall/java/Servlet-Tutorial/`.

[13] MageLang Institute. *JavaServer Pages Fundamentals*. Sun Microsystems Press, Septiembre 2000. .

[14] Alexandra Rey Jimenez. *Curso de UML*. Creangel, Consultoria de Seguridad, 2002. `http://www.creangel.com/uml/intro.php`.

[15] Faisal Khan. *A Brief Introduction to JDBC*. Stardeveloper, Agosto 2003. `http://www.stardeveloper.com/articles/`.

[16] Luis Neves. *Tapestry Sample Applications*. The Apache Software Foundation, 2003. `http://pwp.netcabo.pt/lneves/tapestryapps/`.

[17] Community Development of Java Technology Specifications. *Understanding Java and the J2EE Platform*. Java Community Process, 2004.

[18] Tim O'Reilly. *Extreme Programming, Pocket Guide*. O'Reilly Media, 2004. Curso de UML `http://www.creangel.com/uml/intro.php`.

[19] Gopalan Suresh Raj. *Enterprise JavaBeans*. Gopalan Suresh Raj's Web Cornucopia, 2001. `http://my.execpc.com/~gopalan/java/ejb.html`.

[20] Ed Roman. *Mastering Enterprise JavaBeans and the Java 2 Platform, Enterprise Edition*. John Wiley and Sons, Inc, 1999.

[21] Jose Luis Fernandez Sanchez. *Reusabilidad y Desarrollo Orientado a Objetos*. Asociacion de Tecnicos de Informatica, 1998. `http://www.ati.es/gt/LATIGOO/OOp96/Ponen7/atio6p07.html`.

[22] Howard Lewis Ship. *Tapestry User's Guide*. The Apache Software Foundation, 2004. `http://pwp.netcabo.pt/lneves/tapestryapps/`.

[23] Howard M. Lewis Ship. *Beginning Tapestry: Java Web Components*. The Apache Software Foundation, Octubre 2003. `http://www.java201.com/resources/browse/60-all-10.html`.

[24] Howard M. Lewis Ship. *Tapestry in Action*. Manning, Marzo 2003.

[25] SmartDraw. *How to draw UML diagrams*. SmartDraw, 2004. `http://smartdraw.com/`.

[26] Eric M. Burke y Brian M. Coyner. *Java Extreme Programming Cookbook*. O'Reilly Media, 2004.

[27] Ina Schieferdecker y Clay Williams. *UML Testing Profile*. University of Lubeck, Diciembre 2003.

[28] Howard Lewis Ship y Harish Krishnaswamy. *Tapestry Developer's Guide*. The Apache Software Foundation, 2004. `http://pwp.netcabo.pt/lneves/tapestryapps/`.

[29] Osmar R. Zaiane. *JDBC: Databases the Java Way*. University of Alberta, 2004.