UNIVERSIDAD NACIONAL AUTÓNOMA

DE MÉXICO

FACULTAD DE INGENIERÍA

MODELADO Y CONSTRUCCIÓN DE UN SISTEMA DE CONTROL Y ESTADÍSTICA DE TARJETAS PARA EL LABORATORIO DE DISPOSITIVOS LÓGICOS PROGRAMABLES

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

INGENIERO EN COMPUTACIÓN

P R E S E N T A N :

FRANCISCO JAVIER JAIMES BERISTAIN

JORGE CÉSAR PLASENCIA SÁNCHEZ

DIRECTOR DE TESIS:

M.I NORMA ELVA CHÁVEZ RODRÍGUEZ

CD. UNIVERSITARIA MEXICO D.F 2012

Índice temático

Introducción

El responsable del Laboratorio de Dispositivos Lógicos Programables de la Facultad de Ingeniería de la UNAM, tiene a su resguardo varias tarjetas de desarrollo lógico programables de la firma de Xilinx y Altera, se manejan tres tipos de tarjetas (SPARTAN-3, SPARTAN-6 Y ALTERA UP2), para procesamiento de señales, programación VHDL, diseñar sistemas digital, etc. Estas tarjetas son prestadas a los alumnos que utilizan los recursos del laboratorio para la realización de prácticas y proyectos.

Los préstamos que se realizan en el laboratorio no cuentan con un control de las tarjetas prestadas; tal como un registro de los alumnos que se le presta alguna tarjeta, así como el total de las tarjetas que se clasifican en reparación, el total de las tarjetas prestadas en un determinado instante y la estadística del tipo de tarjetas prestadas por semestre. Por lo que surge la necesidad de desarrollar un sistema que cubra todos estos requerimientos.

Todo la información referente a las tarjetas estaba desorganiza por lo que el objetivo general de este trabajo de tesis consistió en modelar y construir un sistema de control de préstamos de tarjetas para el laboratorio de Dispositivos Lógicos Programables. Esta información se organizó, es decir, se realizó un modelado de la base de datos del sistema.

Para la solución de las necesidades planteadas se consideró realizar una aplicación WEB la cual tenga acceso a la base de datos donde se almacenan y consultan los datos relativos al control de las tarjetas.

Este trabajo de tesis cuenta con seis apartados que a continuación se describen. En los tres primeros capitulos se presentan los fundamentos teoricos requeridos para realizar el sistema y en los tres últimos se describen la metodología usada, el cómo se modelo y construyo el sistema de control de tarjetas, respectivamente.

El capítulo **I** se detalla la metodología genérica para proyectos de software, describiendo el modo sistemático de realizar, gestionar y administrar un proyecto para llevarlo a cabo con altas posibilidades de éxito.

Para el capítulo **II** se describen los fundamentos del modelado orientado a objetos utilizando el lenguaje de modelado unificado UML: se explican los modelos de casos de uso, modelos de clases, modelos de secuencia y modelos de estado y en el modelado de la base de datos se expone el modelo entidad-relación y el modelo relacional.

En el capítulo **III** se revisan los fundamentos técnicos de la plataforma Java EE. Se presentan los detalles de los marcos de trabajo Hibernate y Struts 2. Además, se revisa el funcionamiento de AJAX y finalmente se expone el marco de trabajo Prototype Windows Class.

El capítulo **IV** se reseña cómo se desarrolló el sistema de control de tarjetas implementando la metodología genérica de software la cual comprende un conjunto de métodos (Comunicación, Planeación, Modelado, Construcción y Despliegue) que fueron adaptados para la construcción del sistema.

El capítulo **V** se presentan los modelos UML realizados (casos de uso, diagrama de clases, diagramas de secuencia). También se presentan los modelos entidad-relación y el relacional de la base de datos. Todos estos modelos presentados tienen la finalidad que sirvan como ayuda en la etapa de construcción.

Por último, en el capítulo **VI** se especifica la construcción del sistema utilizando la plataforma Java EE que ofrece reutilización, como el uso del patrón MVC. Para construir el modelo se utilizó Hibernate, para el controlador se usó Struts 2 y para la realización de la vista se emplearon Struts 2, AJAX y Prototype Windows Class.

Objetivos

Objetivo general:

El presente trabajo tiene como objetivo primordial el modelar y construir un sistema de control de préstamos de tarjetas de desarrollo lógico programables, con estadística de uso para el laboratorio de dispositivos lógicos programables.

Objetivos Particulares:

1. Desarrollar el sistema de control de tarjetas adaptándolo a la metodología genérica de software.

2. Modelar el sistema de control de tarjetas con lenguaje de modelado unificado UML y con modelos de entidad-relación para diseñar la base de datos.

3. Hacer uso de la automatización para facilitar la construcción.

4. Utilizar el concepto de reutilización para elegir el software con el que se decidió construir el sistema

5. Construir un prototipo como auxilio para obtener requerimientos faltantes del sistema de control de préstamos de tarjetas y llegar a un producto final.

Capítulo I

Fundamentos de metodología

1.1 Metodología genérica para proyectos de software

Los métodos de la ingeniería de software proporcionan los "cómo" técnicos para construir software. Los métodos abarcan un amplio espectro de tareas que incluyen la comunicación, el análisis de requisitos, el modelo del diseño, la construcción del programa, la realización de pruebas y el soporte.

El siguiente marco de trabajo[1] genérico de proceso (véase la figura 1.1), se puede aplicar en la inmensa mayoría de los proyectos de software:

Comunicación. Esta actividad del marco de trabajo implica una intensa colaboración y comunicación con los clientes; además, abarca la investigación de requisitos y otras actividades relacionadas.

Planeación. Esta actividad establece un plan para el trabajo de la ingeniería del software. Describe las tareas técnicas que deben realizarse, los riesgos probables, los recursos que serán requeridos, los productos del trabajo que han de producirse y un programa de trabajo.

Modelado. Esta actividad abarca la creación de modelos que permiten al desarrollador y al cliente entender mejor los requisitos del software y el diseño que logrará satisfacerlos.

Construcción. Esta actividad combina la generación del código y la realización de pruebas necesarias para descubrir errores en el código.

[1] Conjunto de actividades, una colección de tareas para realizar cada actividad, productos de trabajos generados como consecuencia de las tareas que acompañan al proceso de software.

Despliegue. El software se entrega al cliente, quien evalúa el producto recibido y proporciona información basada en su valoración.

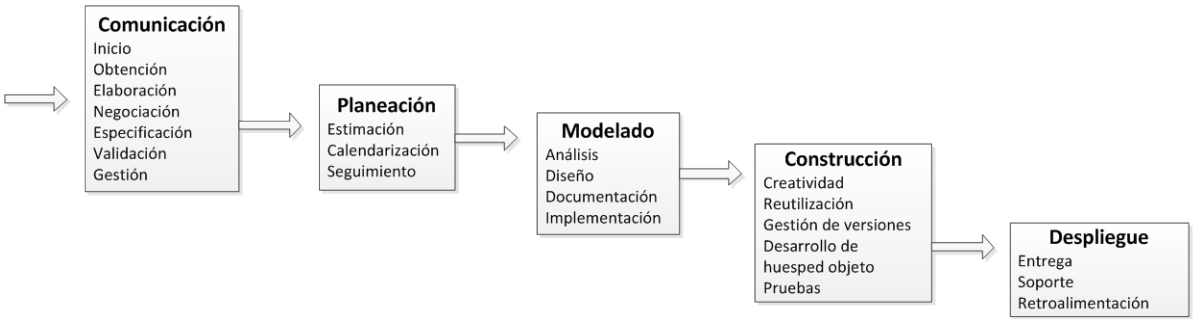


Figura 1.1 Marco de trabajo genérico

1.1.1 La actividad de comunicación

Antes de que los requisitos del cliente puedan analizarse, modelarse o especificarse, estos deben recompilarse por medio de una actividad de comunicación (también llamada obtención de requisitos).

La ingeniería de requisitos proporciona un mecanismo para entender lo que el cliente quiere, analizar las necesidades, evaluar la factibilidad, negociar una solución razonable, especificar la solución sin ambigüedades, validar la especificación, y administrar los requisitos conforme estos se transforman en un sistema operacional. El proceso de la ingeniería de requisitos[2] se lleva a cabo a través de siete distintas funciones: inicio, obtención, elaboración, negociación, especificación, validación y gestión.

[2] Los requerimientos para un sistema son descripciones de lo que el sistema debe hacer: el servicio que ofrece y restricciones en su operación. Tales requerimientos reflejan las necesidades de los clientes por un sistema que atienda cierto propósito. Al proceso de descubrir, analizar, documentar y verificar estos servicios y restricciones se le llama ingeniería de requisitos.

Resulta importante declarar que algunas de estas funciones de la ingeniería de requisitos ocurren en paralelo y que todas deben adaptarse a las necesidades del proyecto.

Inicio

Al inicio del proyecto los ingenieros de software hacen una serie de preguntas libres de contexto enfocadas en el cliente y otros interesados, metas generales y en los beneficios. El objetivo es establecer una compresión básica del problema, las personas que quieren una solución, la naturaleza de la solución que se desea.

Obtención

Éste es el proceso de interactuar con los participantes del sistema para descubrir sus requerimientos. También los requerimientos de dominio de los participantes y la documentación se descubren durante esta actividad. La interacción con los participantes es a través de entrevistas[3] y observaciones[4] de cómo los participantes pueden interactuar con el nuevo sistema.

[3] Las entrevistas son una parte de la mayoría de los procesos de ingeniería de requerimientos. En las entrevistas, el equipo de ingeniería de requerimientos formula preguntas a las personas interesadas sobre el sistema que se va a desarrollar. Los requerimientos se derivan de las respuestas a dichas preguntas.

[4] Los sistemas de software no existen aislados. Se usan en un contexto social y organizacional, y dicho escenario podría derivar o restringir los requerimientos del sistema de software. A menudo satisfacer dichos requerimientos sociales y organizacionales es crítico para el éxito del sistema.

La técnica de observación que ayuda a entender los requerimientos operacionales y ayuda a derivar requerimientos de apoyo para dichos procesos, se le conoce como etnografía. Un analista se adentra en el ambiente laboral donde se usara el sistema. Observa el trabajo diario y toma nota acerca de las tareas existentes en que intervienen los participantes. El valor de la etnografía es que ayuda a descubrir requerimientos implícitos del sistema que reflejan las formas actuales en que trabaja la gente, en vez de los procesos formales definidos por la organización.

Elaboración

Esta actividad toma la compilación no estructurada de requerimientos, agrupa requerimientos relacionados y los organiza en grupos coherentes.

Esta actividad se enfoca en el desarrollo de un modelo de análisis que define el dominio de información, las funciones y el comportamiento del problema.

Negociación

Inevitablemente, cuando intervienen diversos participantes, los requerimientos entraran en conflicto. Esta actividad se preocupa por priorizar los requerimientos. Por lo general, los participantes tienen que reunirse para resolver las diferencias. El objetivo de esta negociación es desarrollar un plan de proyecto que satisfaga las necesidades al mismo tiempo que refleja las restricciones del mundo real a las que está sometido el equipo de software.

Especificación

En una primera etapa en una especificación de un sistema, debe decidir sobre las fronteras del sistema.

Los requerimientos adquiridos hasta el momento se documentan de tal forma que puedan usarse para ayudar al hallazgo de requerimientos. En esta etapa, podría generarse una primera versión del documento de requerimientos del sistema, con secciones faltantes y requerimientos incompletos.

En este documento se incluyen dos clases de requerimientos:

1 Los requerimientos del usuario son informes abstractos de requerimientos del sistema para el cliente y el usuario final del sistema.

2 Los requerimientos de sistema son una descripción detallada de la funcionalidad a ofrecer.

Los requerimientos del usuario se escriben casi siempre en lenguaje natural, complementado con diagramas y tablas adecuados en el documento de requerimientos. Los requerimientos del sistema se escriben también en lenguaje natural, pero de igual modo se utilizan otras notaciones basadas en formas, modelos gráficos del sistema o modelos matemáticos del sistema.

Para minimizar la interpretación errónea al escribir los requerimientos en lenguaje natural, se recomienda seguir algunos lineamientos sencillos:

1. Se elabora un formato estándar y se asegura de que todas las definiciones de requerimientos se adhieran a dicho formato. Al estandarizar el formato es menos probable cometer omisiones y más sencillo comprobar los requerimientos.

2. Utilizar el lenguaje de manera clara para distinguir entre requerimientos obligatorios y deseables. Los primeros son requerimientos que el sistema debe soportar y por lo general, se escriben en futuro “debe ser”. En tanto que los requerimientos deseables no son necesarios y se escriben en tiempo pospretérito o como condicional “debería ser”.

3. Un texto resaltado (negrita, cursiva o color) para seleccionar partes clave del requerimiento.

4. No deducir que los lectores entienden el lenguaje técnico de la ingeniería de software.

5. Siempre que sea posible, se asocia una razón con cada requerimiento de usuario. La razón debe explicar por qué se incluyó el requerimiento. Es particularmente útil cuando los requisitos cambian, pues ayuda a decidir cuales cambios serán indeseables.

Validación

La validación es el proceso de verificar que los requerimientos definan realmente el sistema que en verdad quiere el cliente. Durante este proceso es inevitable descubrir errores en el documento de requerimientos. En consecuencia, deberán modificarse con la finalidad de corregir dichos problemas.

Por lo tanto, se tiene que realizar diferentes tipos de comprobaciones que a continuación se enlistan:

1. Comprobaciones de consistencia. Los requerimientos en el documento no deben estar en conflicto. Esto es, no debe haber restricciones contradictorias o descripciones diferentes de la misma función del sistema.

2. Comprobaciones de totalidad. El documento de requerimientos debe incluir requerimientos que definan todas las funciones y las restricciones pretendidas por el usuario del sistema.

3. Comprobaciones de realismo. Al usar el conocimiento de la tecnología existente, los requerimientos deben comprobarse para garantizar que en realidad pueden implementarse.

4. Verificabilidad. Para reducir el potencial de disputas entre cliente y contratista, los requerimientos del sistema deben escribirse siempre de manera que sean verificables. Esto significa que se debe ser capaz de escribir un conjunto de pruebas que demuestren que el sistema entregado cumpla cada requerimiento especificado.

Hay algunas técnicas de validación de requerimientos que se usan individualmente o en conjunto con otras:

1. Revisiones de requerimientos. Los requerimientos se analizan sistemáticamente usando un equipo de revisores que verifican errores y consistencias.

2. Creación de prototipos[5]. En esta aproximación a la validación, se muestra un modelo ejecutable del sistema en cuestión a los usuarios finales y clientes. Así, ellos podrán experimentar con este modelo para constatar si cubre sus necesidades reales.

3. Generación de casos de prueba. Los requerimientos deben de ser comprobables. Si las pruebas para los requerimientos se diseñan como parte del proceso de

[5] Un prototipo de software es una versión inicial de un sistema de software que se usa para demostrar conceptos, tratar opciones de diseño y encontrar más sobre el problema y sus posibles soluciones.

Puede desarrollarse un sistema prototipo para demostrar a los clientes algunas características del sistema. Los prototipos no tienen que ser ejecutables para ser útiles. Los modelos en papel de la interfaz gráfica de usuario del sistema pueden ser efectivos para ayudar a las partes interesadas a refinar un diseño de interfaz gráfica.

validación, esto revela con frecuencia problemas en los requerimientos. Si una prueba es difícil o imposible de diseñar, esto generalmente significa que los requerimientos serán difíciles de implementar, por lo que deberían reconsiderarse.

Gestión de requisitos

La gestión de requisitos es un conjunto de actividades que ayudan al equipo de proyecto a identificar, controlar y rastrear los requisitos y los cambios a éstos en cualquier momento mientras se desarrolla el producto.

Los proyectos necesitan administrarse porque la ingeniería de software profesional está sujeta siempre a restricciones organizacionales de presupuesto y fecha.

Para la mayoría de los proyectos, las metas son:

1. Entregar el software al cliente en el tiempo acordado.
2. Mantener costos dentro del presupuesto general.
3. Entregar software que cumpla con las expectativas del cliente.
4. Mantener un equipo de desarrollo óptimo y con buen funcionamiento.

1.1.2 La actividad de planeación

Esta actividad establece el nivel de detalle que se requiere en la administración de requerimientos.

Durante la etapa de administración de requerimientos, se tiene que decidir sobre:

1. Identificación de requerimientos. Cada requerimiento debe identificarse de manera exclusiva, de forma que pueda tener referencia cruzada con otros requerimientos y usarse en las evaluaciones de seguimiento.

2. Un proceso de administración de cambio. Éste es el conjunto de actividades que valoran el efecto y costo de los cambios.

3. Políticas de seguimiento. Dichas políticas definen las relaciones entre cada requerimiento, así como entre los requerimientos y el diseño del sistema que debe registrarse. La política de seguimiento también tiene que definir cómo mantener dichos registros.

La mayoría de los administradores en esta etapa, tomaran la responsabilidad de varias o todas las siguientes actividades:

1. Planeación del proyecto. Los administradores de proyecto son responsables de la planeación, estimación[6] y calendarización del proyecto[7].

2. Informes. Los administradores de proyectos deben ser capaces de comunicarse en varios niveles, desde codificar información técnica detallada hasta elaborar resúmenes administrativos.

[6] La estimación se utiliza para definir el presupuesto del proyecto.

[7] La calendarización de proyectos es el proceso de decidir cómo se organizará el trabajo en un proyecto como tareas separadas, y cuándo y cómo se ejecutarán dichas tareas.

3. Gestión de riesgos[8]. Los administradores de proyecto tienen que valorar los riesgos que pueden afectar un proyecto, monitorear dichos riesgos y emprender acciones cuando surjan dichos riesgos.

4. Gestión de personal. Los administradores de proyecto son responsables de administrar un equipo de personas. Deben elegir a los integrantes de sus equipos y establecer formas de trabajar que conduzcan a desempeño efectivo de equipo.

5. Redactar propuestas. La primera etapa en un proyecto de software puede implicar escribir una propuesta para obtener un contrato de trabajo.

La planeación a través del ciclo de vida de un proyecto

La planeación se presenta durante tres etapas en un ciclo de vida del proyecto:

1. En la etapa de propuestas, cuando se presenta una licitación con vistas a obtener un contrato para desarrollar o proporcionar un sistema de software.

2. Durante la fase de inicio, cuando debe determinar quién trabajará en el proyecto, cómo se dividirá el proyecto en incrementos, cómo se asignarán los recursos a través de su compañía, etcétera.

3. Periódicamente a lo largo del proyecto, cuando el plan se modifica a la luz de la experiencia obtenida y la información del monitoreo del avance del trabajo.

[8] Un riesgo es algo preferible que no ocurra. Los riesgos pueden amenazar el proyecto, el software que se desarrolla o la organización.

La planeación en la etapa de la propuesta, inevitablemente, se estima[9], pues muchas veces no se cuenta con un conjunto completo de requerimientos para el software a desarrollar. La estimación incluye calcular cuánto esfuerzo se requiere para terminar cada actividad.

El plan del proyecto siempre evoluciona durante el proceso de desarrollo. Por lo tanto, el calendario[10] y la estimación de costos se deben revisar a medida que se desarrolla el software.

1.1.3 La actividad de modelado

El modelado de sistemas es el proceso para desarrollar modelos abstractos de un sistema, donde cada modelo presenta una visión o perspectiva diferente de dicho sistema. En general, el modelado de sistemas se ha convertido un medio para representar el sistema usando un tipo de notación gráfica.

Es posible desarrollar modelos tanto del sistema existente como del sistema a diseñar:

1. Los modelos para sistema existentes ayudan a aclarar lo que hace el sistema existente y pueden utilizarse como base para discutir sus fortalezas y debilidades.

[9] Existen dos tipos de técnicas de estimación:

1. Técnicas basadas en la experiencia. La estimación de los requerimientos de esfuerzo futuro se basan en la experiencia del administrador con proyectos anteriores y el dominio de la aplicación.

2. Modelado algorítmico de costo. En este caso se usa un enfoque formulista para calcular el esfuerzo del proyecto con base en estimaciones de atributos del producto (por ejemplo, el tamaño), así como las características del proceso (por ejemplo, experiencia del personal implicado).

[10] El calendario de un proyecto indica las dependencias entre las actividades, el tiempo estimado requerido para alcanzar cada plazo y la asignación de personal a las actividades.

Posteriormente, conducen a los requerimientos para el nuevo sistema.

2. Los modelos para un sistema nuevo ayudan a explicar los requerimientos propuestos a otros participantes del sistema.

El aspecto más importante de un modelo del sistema es que deja fuera los detalles. Un modelo es una abstracción del sistema a estudiar, y no una representación alternativa de dicho sistema.

Modelos de análisis y modelos de diseño

En el trabajo de la ingeniería de software se crean dos clases de modelos: modelos de análisis y modelos de diseño. Los modelos de análisis representan los requisitos del cliente al presentar el software en tres dominios diferentes: el dominio de la información, el dominio funcional y el dominio del comportamiento. Los modelos de diseño representan características del software que ayudan a los profesionales a construirlo de manera efectiva: la arquitectura[11], la interfaz gráfica de usuario, el detalle a nivel de componentes[12] y el modelo de la base de datos[13].

Cuando se crean modelos de un sistema, se puede ser flexible en la forma que se usa la notación gráfica. No siempre

[11] Un modelo arquitectónico que describe la forma en que se organiza el sistema como un conjunto de componentes en comunicación. Además, se identifica la estructura global del sistema, los principales componentes (llamados en ocasiones subsistemas o módulos), sus relaciones y como se distribuyen.

Los componentes individuales implementan los requerimientos funcionales del sistema. Los requerimientos no funcionales dependen de la arquitectura del sistema, es decir, la forma en que dichos componentes se organizan y comunican.

[12] Se toma cada componente del sistema y se diseña cómo funcionará.
[13] Se diseñan las estructuras del sistema de datos y cómo se representarán en una base de datos.

necesitará apegarse rigurosamente a los detalles de una notación. El detalle y el rigor de un modelo dependen de cómo lo use. Hay tres formas en que los modelos gráficos se emplean con frecuencia:

1. Como medio para facilitar la discusión sobre un sistema existente o propuesto.

2. Como una forma de documentar un sistema existente.

3. Como una descripción detallada del sistema que sirve para generar una implementación del sistema.

Uso de modelos como documentación

Cuando los modelos se usan como documentación, no tiene que estar completos, pues quizás solo se desee desarrollar modelos para algunas partes del sistema. Sin embargo, estos modelos deben ser correctos: tienen que usar adecuadamente la notación y ser una descripción precisa del sistema.

Uso de modelos para la implementación

En el tercer caso, en que los modelos se usan como parte de un proceso de desarrollo basado en modelo, los modelos de sistema deben ser completos y correctos. La razón para esto es que se usan como base para generar el código fuente del sistema. Por lo tanto, debe ser muy cuidadoso de no confundir símbolos equivalentes, que tienen significados diferentes.

1.1.4 La actividad de construcción

La actividad de construcción o implementación abarca una serie de tareas de codificación y realización de pruebas que

conducen al software operativo que está listo para entregarlo al cliente o usuario final.

Las actividades de diseño e implementación de software se encuentran invariablemente entrelazadas. El diseño de software es una actividad creativa donde se identifican los componentes del software y sus relaciones, con base en los requerimientos de un cliente. La implementación es el proceso de realizar el diseño como un programa. El diseño trata de cómo resolver un problema, de modo que siempre existe un proceso de diseño. Sin embargo, en ocasiones no es necesario o adecuado describir con detalle el diseño utilizando el lenguaje de modelado unificado (UML[14]) u otro lenguaje de descripción de diseño.

La implementación quizá requiera el desarrollo de programas en lenguajes de programación de alto nivel o bajos niveles, o bien, la personalización y adaptación de sistemas comerciales genéricos para cubrir los requerimientos específicos de una organización.

Aspectos de implementación que son muy importantes para la ingeniería de software:

1. Reutilización. La mayoría del software moderno se construye por la reutilización de componentes o sistemas existentes. Cuando se desarrolla software, debe usarse el código existente tanto como sea posible.

2. Administración de la configuración. Durante el proceso de desarrollo se crean versiones diferentes de cada componente de software.

[14] El lenguaje unificado de modelado (UML) es un lenguaje de modelado visual que se usa para especificar, visualizar, construir y documentar componentes de un sistema orientado a objetos.

3. Desarrollo de huésped-objeto. La producción de software no se ejecuta por lo general en la misma computadora que el entorno de desarrollo de software.

Reutilización

La reutilización de software es posible en algunos niveles diferentes:

1. El nivel de abstracción. En este nivel no se utiliza el software directamente, sino más bien se utiliza el conocimiento de abstracciones exitosas en el diseño de su software. Los patrones de diseño[15] y arquitectónicos son vías de representación del conocimiento abstracto para la reutilización.

2. El nivel objeto. En este nivel se reutilizan directamente los objetos de una librería en vez de escribir uno mismo en código. Para implementar este tipo de reutilización, se deben encontrar librerías adecuadas y describir si los objetos y métodos ofrecen la funcionalidad que se necesita.

Al reutilizar el software existente, es factible desarrollar nuevos sistemas más rápidamente, con menos riesgos de desarrollo y también costos menores. Puesto que el software reutilizado se probó en otras aplicaciones, debe ser más confiable que el software nuevo.

[15] Un patrón de diseño es una descripción del problema y la esencia de su solución, de modo que la solución puede reutilizarse en diferentes configuraciones. El patrón no es una especificación detallada. Más bien, puede considerarla como una descripción de sabiduría y experiencia acumuladas, una solución bien probada a un problema común.

Administración de la configuración

La administración de la configuración es el nombre dado al proceso general de gestionar un sistema de software cambiante. La meta de la administración de la configuración es apoyar el proceso de integración del sistema, de modo que todos los desarrolladores tengan acceso controlada al código del proyecto y a los documentos, así como descubrir qué cambios se realizaron. Por lo tanto, hay tres actividades fundamentales en la administración de la configuración:

1. Gestión de versiones, donde se da soporte para hacer un seguimiento de las diferentes versiones de los componentes de software.

2. Integración del sistema, donde se da soporte para ayudar a los desarrolladores a definir qué versiones de componentes se usan para crear cada versión de un sistema.

3. Rastreo de problemas, donde se da soporte para que los usuarios reporten errores y otros problemas, y también para que todos los desarrolladores sepan quién trabaja en dichos problemas y cuándo se corrigen.

Desarrollo de huésped-objeto

La mayoría de software se basa en un modelo huésped-objetivo. En un sentido más amplio, puede hablarse de una plataforma de desarrollo y de una plataforma de ejecución.

Pruebas de software

Las pruebas intentan demostrar que un programa hace lo que se intenta que haga, así como descubrir defectos en el programa

antes de usarlo[16]. Al probar el software, se ejecutara un programa con datos artificiales.

El proceso de prueba tiene dos metas distintas:

1. Demostrar al desarrollador y al cliente que el software cumple con los requerimientos.

2. Encontrar situaciones donde el comportamiento del software sea incorrecto, indeseable o no esté de acuerdo con su especificación. La prueba de defectos tiene la finalidad de erradicar el comportamiento indeseable del sistema, como caídas del sistema, interacciones indeseadas con otros sistemas, cálculos incorrectos y corrupción de datos.

La finalidad de la verificación es comprobar que el software cumpla con su funcionalidad y con los requerimientos no funcionales establecidos. Sin embargo, la validación es un proceso más general. La meta de la validación es garantizar que el software cumpla con las expectativas del cliente. Va más allá del simple hecho de comprobar la conformidad con la especificación, para demostrar que el software hace lo que el cliente espera que haga.

Por lo general, un sistema de software puede pasar por tres etapas de pruebas:

1. Pruebas de desarrollo, donde el sistema se pone a prueba durante el proceso para descubrir errores y defectos. Es

[16] Un principio general de buena práctica en la ingeniería de requerimientos es que éstos deben ser comprobables; esto es, los requerimientos tienen que escribirse de forma que pueda diseñarse una prueba para dicho requerimiento. Luego, un examinador comprueba que el requerimiento se cumpla.

posible que en el desarrollo de pruebas intervengan diseñadores y programadores del sistema.

2. Pruebas de versión, donde un equipo de prueba por separado experimenta una versión completa del sistema, antes de presentarlo a los usuarios.

3. Pruebas de usuario[17], donde los usuarios reales o potenciales de un sistema prueban el sistema en su propio entorno. Las pruebas de aceptación[18] se efectúan cuando el cliente prueba de manera formal un sistema para decidir si debe aceptarse del proveedor del sistema, o si se requiere más desarrollo.

Pruebas de desarrollo

Durante el desarrollo, las pruebas[19] se realizan en tres niveles de granulación:

1. Pruebas de unidad, donde se ponen a prueba unidades de programa o clases de objetos individuales.

2. Pruebas de componente, donde muchas unidades individuales se integran para crear componentes compuestos.

[17] Las pruebas de usuario son esenciales, aun cuando se hayan realizado pruebas abarcadoras del sistema y de versión. La razón de esto es que la influencia del entorno de trabajo del usuario tiene un gran efecto sobre la fiabilidad, el rendimiento, el uso y la robustez de un sistema.

[18] Las pruebas de aceptación tienen lugar después de las pruebas de versión. Implican a un cliente que prueba de manera formal un sistema, para decidir si debe o no aceptarlo del desarrollador del sistema. La aceptación implica que debe realizarse el pago por el sistema.

[19] El desarrollo dirigido por pruebas es un enfoque de diseño de programas donde se entrelazan el desarrollo de pruebas y el de código. En esencia, el código se desarrolla incrementalmente, junto con una prueba de ese incremento. No se avanza hacia el siguiente incremento sino hasta que el código diseñado pasa la prueba.

3. Pruebas del sistema, donde algunos o todos los componentes en un sistema se integran y el sistema se prueba como un todo. Las pruebas del sistema deben enfocarse en poner a prueba las interacciones de los componentes.

Las pruebas de desarrollo son, ante todo, un proceso de prueba de defecto, en los cuales la meta consiste en descubrir errores en el software. Por lo tanto, a menudo están entrelazadas con la depuración: el proceso de localizar problemas con el código y cambiar el programa para corregirlos.

Pruebas de unidad

Cuando pone a prueba las clases de objetos, tiene que diseñar las pruebas para brindar cobertura a todas las características del objeto. Esto significa que debe:

1 Probar todas las operaciones asociadas con el objeto.

2 Establecer y verificar el valor de todos los atributos relacionados con el objeto.

3 poner el objeto en todos los estados posibles. Esto quiere decir que tiene que simular todos los eventos que causen un cambio de estado.

Siempre que sea posible, se deben automatizar las pruebas de unidad. En estas pruebas de unidad automatizadas, podría usarse un marco de automatización de pruebas (como JUnit, Maven o Ant) para escribir y correr sus pruebas de programa.

Pruebas de componentes

En general, los componentes de software son componentes compuestos constituidos por varios objetos en interacción. Por consiguiente, la prueba de componentes compuestos tiene que enfocarse en mostrar que la interfaz de componente se comporta según su especificación.

Pruebas de sistema

Las pruebas de sistema demuestran que los componentes son compatibles, que interactúan correctamente y que transfieren los datos correctos en el momento adecuado a través de sus interfaces. Evidentemente, se traslapan con las pruebas de componentes, pero existen dos importantes diferencias:

1. Durante las pruebas de sistema, los componentes reutilizables desarrollados por separado y los sistemas comerciales pueden integrarse con componentes desarrollados recientemente. Entonces se prueba el sistema completo.

2. Los componentes desarrollados por diferentes miembros del equipo o de grupos pueden integrarse en esta etapa. La prueba de sistema es un proceso colectivo más que individual.

Por lo tanto, las pruebas de sistema deben enfocarse en poner a prueba las interacciones entre los componentes y los objetos que constituyen el sistema. Las pruebas de interacción también ayudan a encontrar interpretaciones erróneas, cometidas por desarrolladores de componentes, acerca de otros componentes del sistema.

1.1.5 La actividad de despliegue

La actividad de despliegue abarca tres acciones: entrega, soporte y retroalimentación. Como el software moderno es evolutivo por naturaleza, el despliegue no se presenta una sola vez, sino varias veces conforme el software avanza hacia su terminación. Cada ciclo de entrega les proporciona al cliente y a los usuarios finales un incremento de software operativo que provee funciones y características útiles. Cada ciclo de soporte proporciona documentación y asistencia humana para todas las funciones y características introducidas durante todos los ciclos de despliegue que se presenten.

Capítulo II

Fundamentos de modelado

2.1 Modelado orientado a objetos con UML

Al usar el UML, por lo general se desarrollan dos tipos de modelo de diseño:

1. Modelos estructurales, los cuales muestran la organización de un sistema en términos de los componentes que constituyen dicho sistema y sus relaciones. Además, éstos describen la estructura estática del sistema. Los modelos estructurales de un sistema se crean cuando se discute y diseña la arquitectura del sistema.

2. Modelos dinámicos, los cuales explican la estructura dinámica del sistema.

Todos los sistemas incluyen interacciones de algún tipo. Éstas pueden ser interacciones del usuario, que implican entradas y salidas del usuario; interacciones entre el sistema a desarrollar y otros sistemas; o interacciones entre los componentes del sistema. El modelo de interacción del usuario es importante, pues ayuda a identificar los requerimientos del usuario.

Dos enfoques relacionados con el modelado de interacción son:

1. Modelado de *caso de uso*.

2. Diagramas de *secuencia*.

En las primeras fases del proceso de diseño, se considera que existen tres modelos que son útiles particularmente para agregar detalle a los modelos de casos de uso y arquitectónico:

1. Modelos de subsistemas, se representan mediante una forma de diagrama de clase en que cada subsistema se muestra como un paquete con objetos encerrados. Los modelos de subsistema son modelos estáticos (estructurales).

2. Modelos de secuencia, que ilustran la secuencia de interacciones de objetos. Se representa mediante una secuencia UML o un diagrama de colaboración. Los modelos de secuencia son modelos dinámicos.

3. Modelos de máquina de estado, que muestran cómo los objetos individuales cambian su estado en respuesta a eventos. Se representan el UML a través de diagramas de estado. Los modelos de máquina de estado son modelos dinámicos.

2.1.1 Casos de uso

Los casos de uso son una técnica de descubrimiento de requerimientos, los cuales pueden tomarse como simples escenarios[1] que describen lo que espera el usuario de un sistema. Además, los casos de uso representan una tarea discreta que implica interacción externa con el sistema. En su forma más simple, la figura 2.1 muestra un caso de uso representado como una elipse y muestra un actor que interviene con el caso de uso.

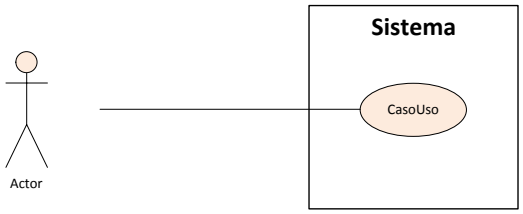


Figura 2.1 Modelado de un caso de uso

[1] Los escenarios son particularmente útiles para detallar un bosquejo de descripción de requerimientos.

Al brindar los casos de uso un panorama bastante sencillo, se tiene que ofrecer más detalle para entender lo que está implicado. Este detalle puede ser una simple descripción textual, o descripción estructurada en una tabla o un diagrama de secuencia. Es posible elegir el formato más adecuado, dependiendo del caso de uso y del nivel de detalle que se considere que se requiere en el modelo.

En un modelo de casos de uso se pueden tener varias relaciones, además de poder asociarse con actores. La tabla 2.1 muestra este tipo de relaciones: asociación, extensión, generalización y herencia.

Relación	**Función**	**Notación**
asociación	La línea de comunicación entre un actor y un caso de uso en el que participa.	_____
extensión	La inserción de comportamiento adicional en un caso de uso base que no tiene conocimiento sobre de él.	<<extend>> -- -- -- →
generalización	Una relación entre un caso de uso general y un caso de uso más específico, que hereda y añade propiedades a aquel.	----------▶
inclusión	Inserción de comportamiento adicional en un caso de uso base, que describe explícitamente la inserción.	<<include>> -- -- -- →

Tabla 2.1 Tipo de relaciones en los casos de uso

2.1.2 Modelos de subsistemas

Un modelo de subsistema puede diseñarse como modelos de clases detallados, que representan todos los objetos en los sistemas y sus asociaciones (herencia, generalización, agregación, etcétera).

Los diagramas de clase en el UML pueden expresarse con diferentes niveles de detalle. Cuando se desarrolla un modelo, la primera etapa con frecuencia implica, identificar los objetos esenciales y representarlos como clases.

A continuación la figura 2.2 muestra la forma más sencilla de hacer esto, se tiene que escribir el nombre de la clase en un recuadro.

Figura 2.2 Diagrama de clases de forma sencilla

Para definirlas las clases con más detalle, se agrega información sobre sus atributos y métodos. En la figura 2.3 se muestran los atributos y operaciones al extender el rectángulo simple que representa una clase, en donde:

1. El nombre de clase de objeto está en la sección superior.

2. Los atributos de clase están en la sección media. Esto debe incluir los nombres del atributo y opcionalmente, sus tipos.

3. Las operaciones asociadas con la clase de objeto están en la sección inferior del rectángulo.

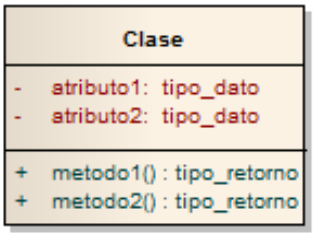


Figura 2.3 Diagrama de clases con atributos y métodos

También puede anotar la existencia de una relación dibujando simplemente una línea entre las clases. Hay dos categorías de relaciones: asociaciones y relaciones todo/parte.

Por lo general, los atributos (o propiedades) se designan como privados, o disponibles sólo para el objeto. Esto se representa en un diagrama de clases mediante un signo de resta antes del nombre del atributo. Los atributos también pueden designarse como protegidos, lo cual se indica con el símbolo de número (#), estos atributos están ocultos para todas las clases, excepto para las subclases inmediatas. En circunstancias poco comunes, un atributo es público, lo cual significa que es visible para otros objetos fuera de su clase. Al hacer privados a los atributos sólo están disponibles para los objetos externos a través de los métodos de la clase, una técnica llamada encapsulamiento, u ocultamiento de información.

Asociaciones

El tipo más simple de relación es una asociación, o una conexión estructural entre clases u objetos. Las asociaciones se muestran como una línea simple en un diagrama de clases. Los puntos finales de la línea se etiquetan con un símbolo que indica la multiplicidad[2], un cero representa ninguno, un uno representa uno y sólo uno, y un asterisco representa muchos. La notación 0..1 representa de cero a uno, y la

[2] Los diagramas de clases no restringen el límite inferior de asociación, ni tampoco, los limites superiores.

notación 1..* representa de uno a muchos. Las asociaciones se ilustran en la figura 2.4.

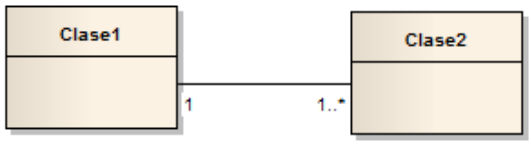


Figura 2.4 Asociación con cardinalidad

Relaciones todo/parte

Una relación todo/parte podría ser un objeto que tiene partes distintas, además estas relaciones tienen varias categorías: agregación y composición.

Agregación. A menudo, una agregación se describe como una relación "tiene un". La agregación proporciona un medio para mostrar que el objeto total se compone de la suma de sus partes (otros objetos), además si el objeto total se elimina los otros objetos no serán eliminados. La figura 2.5 muestra una clase *Automovil* que se compone de las clases *Carroceria* y *Motor*, se observa un trazo en forma de diamante, junto con la clase *Automovil* que representa el todo.

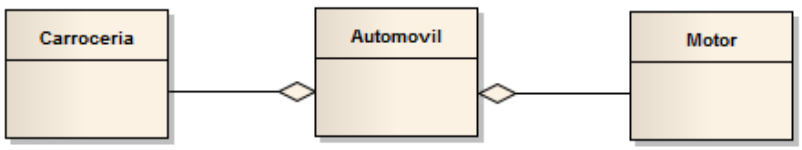


Figura 2.5 Relación de agregación

Composición. La composición se describe como una relación "siempre contiene". Un objeto se compone de otros objetos, pero si el objeto total se elimina los otros objetos serán eliminados. La figura 2.6 muestra una clase Curso que se compone de otra clase Tarea, se observa junto la clase Curso un diamante sólido.

Figura 2.6 Relación de composición

Generalización

En el modelado de sistemas, con frecuencia es útil examinar las clases en un sistema, con la finalidad de haber si hay ámbito para la generalización. La generalización se muestra en la figura 2.7, una flecha apunta hacia la clase más general. La clase más específica es una subclase de otra clase. Esta última, más general, recibe el nombre de superclase.

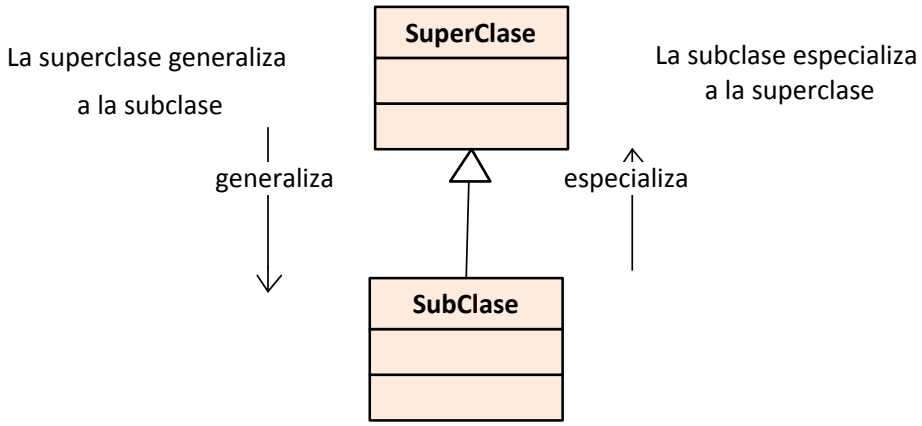


Figura 2.7 Superclase y subclase

En una generalización, las clases de nivel inferior son subclases que heredan los atributos y las operaciones de sus superclases. Entonces dichas clases de nivel inferior agregan atributos y operaciones más específicos.

2.1.3 Modelos de secuencia

Cuando se documenta un diseño, debe producirse un modelo de secuencia por cada interacción significativa. Si se desarrolló un modelo de caso de uso, entonces debe hacer un modelo de secuencia para cada caso de uso que identifique.

Los diagramas de secuencia se usan principalmente para modelar las interacciones entre los actores y los objetos en un sistema, así como las interacciones entre los objetos en sí. El UML tiene una amplia sintaxis para diagramas de secuencia, lo cual permite muchos tipos de interacción a modelar.

Un diagrama de secuencia muestra la sucesión de interacciones que ocurre durante un caso de uso particular o una instancia de caso de uso.

En el diagrama de secuencia que se muestra en la figura 2.8, los objetos y actores que intervienen se mencionan a lo largo de la parte superior del diagrama, con una línea punteada que se dibuja verticalmente a partir de éstos. Las intercepciones entre los objetos se indican con flechas dirigidas. El rectángulo sobre las líneas punteadas indica la línea de vida del objeto tratado. La secuencia de interacciones se lee de arriba abajo. Las anotaciones sobre las flechas señalan las llamadas a los objetos, sus parámetros y los valores que regresan. La notación para exponer alternativas, un recuadro marcado con “alt” se usa con las condicionales indicadas entre corchetes.

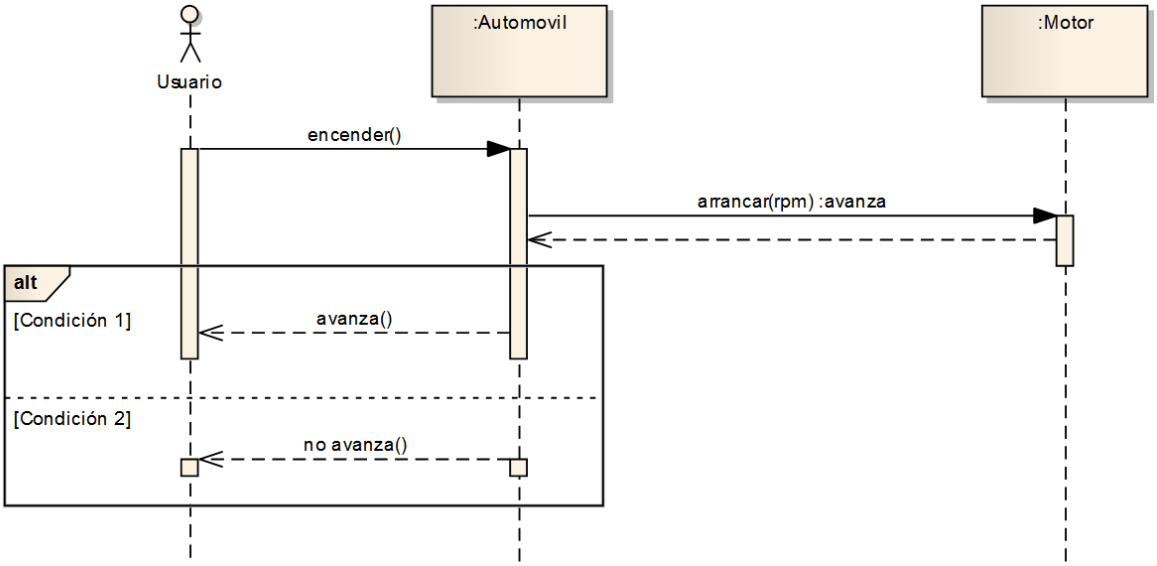


Figura 2.8 Diagrama de secuencia

2.1.4 Modelos de máquina de estado

Un modelo de máquina de estado resume el comportamiento de un objeto o un subsistema, en respuesta a mensajes y eventos. Este modelo muestra cómo la instancia objeto cambia de estado dependiendo de los mensajes que recibe.

Por lo general, no requiere un diagrama de estado para todos los objetos del sistema. Muchos de los objetos en un sistema son relativamente simples y un modelo de estado añade detalle innecesario al diseño.

Los diagramas de estado muestran estados y eventos que causan transacciones de un estado a otro. En los diagrama de estado, figura 2.9, los rectángulos redondeados representan estados del sistema. Pueden incluir una breve descripción de las acciones que se tomaran en dicho estado. Las flechas etiquetadas representan estímulos que fuerzan una transacción de un estado a otro. Puede indicar los estados inicial y final usando círculos rellenos.

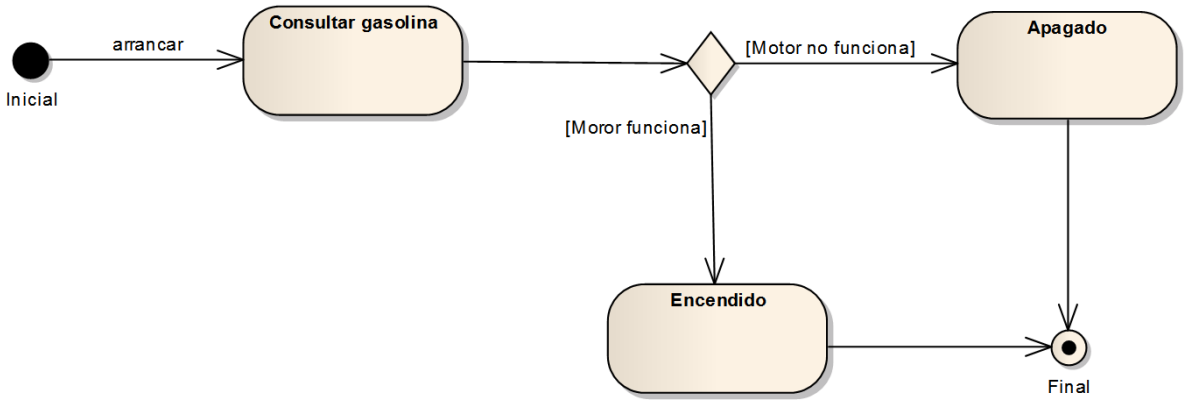


Figura 2.9 Diagrama de estado para un motor

2.2 Modelado de base de datos

2.2.1 El modelo entidad relación

La estructura lógica general de una base de datos[3] se puede expresar gráficamente mediante un *diagrama entidad relación.* El diagrama de entidad relación está basado en una percepción del mundo real que consta de una colección de objetos básicos, llamados *entidades*, y de *relaciones* entre estos objetos. En el cual:

1. Una entidad es una «cosa» u «objeto» en el mundo real que es distinguible de otros objetos.

2. Una relación es una asociación entre varias entidades.

Una entidad se representa mediante un conjunto de atributos. Los atributos describen propiedades que posee cada miembro de un conjunto de entidades. Cada entidad tiene un valor para cada uno de sus atributos. Para cada atributo hay un conjunto de valores permitidos, llamados el dominio, o el conjunto de valores, de ese atributo.

[3] Una base de datos es una organización de una colección de datos que se interrelacionan y se controlan.

Además de entidades y relaciones, el diagrama de entidad relación representa ciertas restricciones que los contenidos de la base de datos deben cumplir. Una restricción importante, mostrada en la figura 2.10 es la *correspondencia de cardinalidades* que expresa el número de entidades con las que otra entidad se puede asociar a través de un conjunto de relaciones.

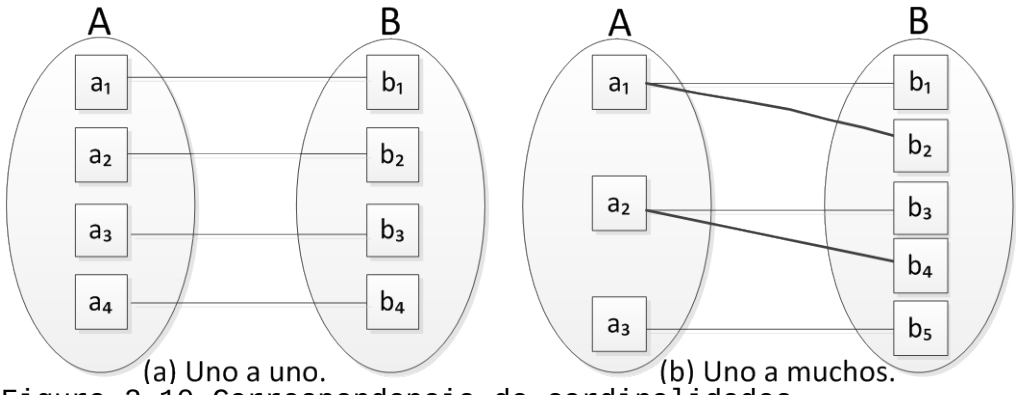


Figura 2.10 Correspondencia de cardinalidades

2.2.2 El modelo relacional

El modelo relacional organiza y representa los datos en forma de tablas o relaciones. Cada tabla contiene registros de un tipo particular. Cada tipo de registro define un número fijo de campos, o atributos. Las columnas de la tabla corresponden a los atributos del tipo de registro.

Los diseños de bases de datos a menudo se realizan en el diagrama entidad relación, y después se traducen al modelo relacional.

Para que la estructura de las tablas cumpla las leyes de la teoría relacional deben satisfacer las siguientes condiciones:

1. Todos los registros de la tabla deben tener el mismo número de campos, aunque alguno de ellos este vacío, deben ser registros de longitud fija.

2. Cada campo tiene un nombre o etiqueta que hay que definir previamente a su utilización.

3. La base de datos estará formada por muchas tablas, una cada tipo de registro.

4. Dentro de una tabla cada nombre de campo debe ser distinto.

5. Los registros de una misma tabla tienen que diferenciarse, al menos, en el contenido de alguno de sus campos, no puede haber dos registros “idénticos”.

6. Los registros de una tabla pueden estar dispuestos en cualquier orden.

7. El contenido de cada campo está delimitado por un rango de valores posibles.

8. Permite la creación de nuevas tablas a partir de las ya existentes, relacionando campos de distintas tablas anteriores.

El modelo de datos relacional utiliza llaves primarias y llaves secundarias (externas o foráneas) para representar las relaciones entre tablas. A la hora de definir las llaves primarias y secundarias es necesario tener presente de entrada lo siguiente:

1. Una llave primaria es una columna o combinación de columnas dentro de una tabla cuyo (s) valor (es) identifica (n) unívocamente a cada fila de la tabla. Cada tabla tiene una única llave primaria.

2. Una llave secundaria es una columna o combinación de columnas en una tabla cuyo (s) valor (es) es (son) un valor de la llave primaria para alguna otra tabla. Una tabla puede contener más de una llave secundaria, enlazándola a una o más tablas.

3. Una combinación llave primaria/llave secundaria crea una relación padre/hijo entre las tablas que las contienen.

2.2.3 Normalización

En una base de datos, la normalización tiene un significado matemático específico, realizando una separación de elementos de datos (tales como nombres, direcciones u oficios) en grupos afines y definiendo las relaciones normales o "correctas" entre ellos.

Con la técnica de normalización se trata de evitar la dependencia entre inserciones, actualizaciones y borrado de elementos de las tablas de la base de datos. También se reducen las operaciones de reorganización cuando hay que incorporar nuevos datos. La normalización tiene tres etapas que transforman las relaciones no normales en normalizadas y que se denominan primera, segunda y tercera formas normales (véase la Tabla 2.2).

1ª forma normal	El primer paso en la normalización es poner los datos en la primera forma normal. Esto se hace situando los datos en tablas separadas, de manera que los datos de cada tabla sean de un tipo similar, y dando a cada tabla una llave primaria y un identificador o etiqueta única. Esto elimina los grupos repetidos de datos.
2ª forma normal	El segundo paso, que es la segunda forma normal, se centra en aislar los datos que solo dependen de una parte de la llave.
3ª forma normal	El tercer paso, implica deshacerse de cualquier elemento de las tablas que no dependa únicamente de la llave primaria. Una vez que los datos están en tercera forma normal, están ya de manera automática en la primera y la segunda forma normal. Por tanto, el proceso total se puede acabar de una forma menos tediosa usando directamente la tercera forma normal, que haciéndola forma a forma. Simplemente, basta con organizar los datos para que las columnas de cada tabla, aparte de la llave primaria, dependan únicamente de toda la llave primaria.

Tabla 2.2 Normalización

Capítulo III

Fundamentos de marcos de trabajo

3.1 Implementación del diseño con Java EE

En la actualidad es posible distinguir cuatro distribuciones de Java (Java SE, Java ME, Java FX y JAVA EE), cada una de estas plataformas proveen una máquina virtual Java (VM) y una interfaz de programación de aplicaciones (API).

Una aplicación desarrollada con la plataforma Java EE, se diseña con una arquitectura de alto nivel con la cual se divide la aplicación en múltiples capas y se ubican los distintos componentes en cada una de ellas. Las capas en la plataforma Java EE se listan a continuación:

1. **Capa cliente**. Consiste de aplicaciones clientes que acceden a un servidor Java EE. El cliente hace *peticiones al servidor, éste procesa la petición y retorna una respuesta al cliente. Los clientes pueden ser un navegador Web o una aplicación estándar.*

2. **Capa de presentación**. Consiste de componentes que manejan la interacción entre la capa cliente y la capa de negocio. Sus principales tareas son:

   - Genera contenido dinámico en varios formatos para el cliente.

   - Colecta entradas de los usuarios de las interfaces cliente y retorna apropiados resultados de los componentes de la capa de negocio.

   - Controla el flujo de pantallas o páginas en el cliente.

- Mantiene el estado de datos para una sesión de usuarios.

- Ejecuta alguna lógica básica y almacena algunos datos temporalmente en componentes JavaBean[1].

3. **Capa de negocio**. Consiste de componentes que proveen la lógica del negocio para una aplicación, ésta lógica es código que provee funcionalidad para un particular dominio de negocio.

La plataforma Java EE cuenta con marcos de trabajo[2] como Hibernate, Struts, Spring, etc.

3.2 El marco de trabajo Hibernate

Un mapeador objeto-relacional (ORM), es una herramienta que provee un API simple para almacenar y recuperar objetos de algún modelo del dominio directamente en una base de datos relacional, véase la figura 3.1. Otro término que se utiliza frecuentemente cuando se habla de tecnologías ORM es persistencia[3] trasparente, esto debido a que es una técnica que permite escribir aplicaciones utilizando un lenguaje orientado a objetos que trata la información como objetos en lugar de usar conceptos específicos de base de datos.

---

[1] Los JavaBeans son componentes lógicos simples y reutilizables. Para que sea una clase JavaBean, deberá respetar ciertas convenciones: la clase debe ser serializable (para guardar y leer), la clase debe tener un constructor por defecto (sin argumento), las propiedades de los métodos deben ser accesibles a través de los métodos (descriptores de acceso).

[2] Un marco de trabajo es una pieza de un software estructural que:

- ✓ Es un conjunto de clases e interfaces que cooperan para resolver un tipo específico de problema de software.
- ✓ Provee una solución arquitectónica.
- ✓ Comprende múltiples clases o componentes, cada una de ellas pueden proveer una abstracción de algún concepto particular.
- ✓ Define como trabajan juntas estas abstracciones para resolver un problema.
- ✓ Los componentes son reusables.

[3] La persistencia usualmente en Java significa que se está usando una base de datos relacional.

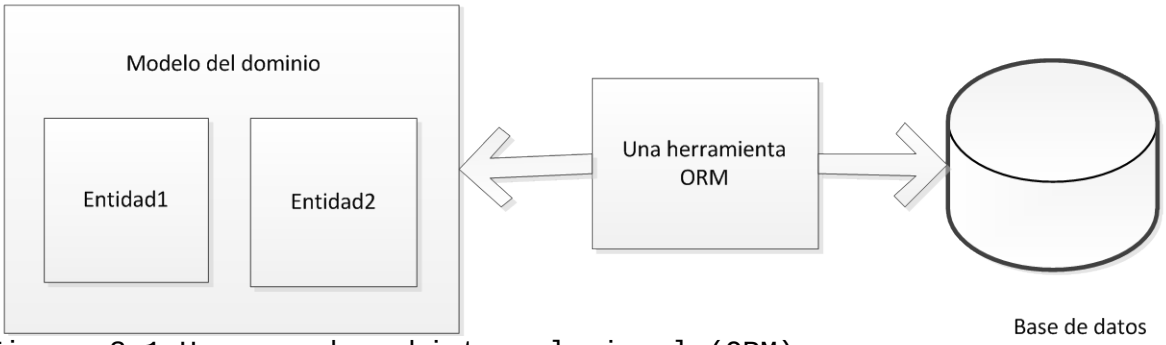


Figura 3.1 Un mapeador objeto-relacional (ORM)

El uso de un ORM tiene beneficios, en particular:

1. Elimina la escritura de código SQL, deja al desarrollador libre para concentrase en la lógica del negocio.

2. Habilita a crear un apropiado modelo del dominio, el desarrollador únicamente necesita pensar en términos de objetos, en lugar de tablas, renglones y columnas.

3. Reduce la dependencia de un SQL específico.

4. Reduce en más de un 30 % en la cantidad de código que se espera escribir.

Hibernate es una solución ORM para un ambiente Java y para .NET, el cual permite considerar que la base de datos almacena objetos Java, lo cual prácticamente no es verdad. Una base de datos almacena datos en forma de tablas, renglones y columnas.

3.2.1 Utilización de Hibernate

Lo que se necesita hacer para usar Hibernate es:

1. Escribir una simple clase POJO[4] (Plain Old Java Object).

2. Crear un archivo de mapeo XML que describa la relación entre la base de datos y los atributos de la clase.

3. Utilizar la API de Hibernate para cargar/almacenar los objetos persistentes.

Clases POJO

Las clases POJO representan las tablas de la base de datos. El patrón de diseño DAO[5] (Data Access Objects) puede ser usado, si se requiere, para tratar con las operaciones de la base de datos.

Configuración en Hibernate

La configuración en Hibernate puede ser definida en el archivo:

1. hibernate.properties o

2. hibernate.cfg.xml

Se recomienda usar el archivo XML, porque le da soporte a la configuración de los archivos de mapeo. Si se usa

[4] El acrónimo POJO se utiliza para hacer referencia a la simplicidad de utilización de un objeto en comparación con la dificultad de la utilización de un componente EJB (Enterprise Java Bean). La única diferencia real entre un POJO y un JavaBean es la posibilidad de los JavaBeans de gestionar eventos.

[5] Con el patrón de diseño DAO, se construyen objetos que accedan a los datos.

hibernate.properties se tendrá que realizar la configuración programándola en Java.

En el archivo hibernate.cfg.xml se configuran un conjunto de propiedades y archivos de mapeo, la convención de nombrar estos recursos es usar el nombre de la clase persistente seguido de la extensión hbm.xml.

Un archivo de configuración se puede formar de acuerdo a como lo especifica el documento que se encuentra en la siguiente dirección del sitio oficial de Hibernate: http://hibernate.-sourceforge.net/hibernate-mapping-3.0.dtd.

Archivos de mapeo

La estructura básica de un archivo de mapeo, se ejemplifica en la siguiente figura 3.2:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
                "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
                "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
[...]
</hibernate-mapping>
```

Figura 3.2 Código de la estructura de un archivo de mapeo.

3.2.2 Funcionamiento de Hibernate

Una típica aplicación de Hibernate (obsérvese la figura 3.3) usualmente sigue los siguientes pasos:

1. Se construye un objeto de tipo SessionFactory a partir de la lectura del archivo de configuración (hibernate.cfg.xml). Se llama a Configuration()-.configure() para que cargué el archivo de configuración e inicialice el medio ambiente de Hibernate.

Una instancia de SessionFactory es creada solo una vez y es usada para la creación de todos los objetos de tipo Session (una sesión) del contexto dado.

2. Se recupera y se abre una sesión.

3. Inicia una transacción con el objeto de tipo de Transaction a partir de la sesión.

4. Se realizan las operaciones deseadas con la base de datos tal como la persistencia de un objeto.

5. Finaliza una transacción.

6. Se cierra la sesión.

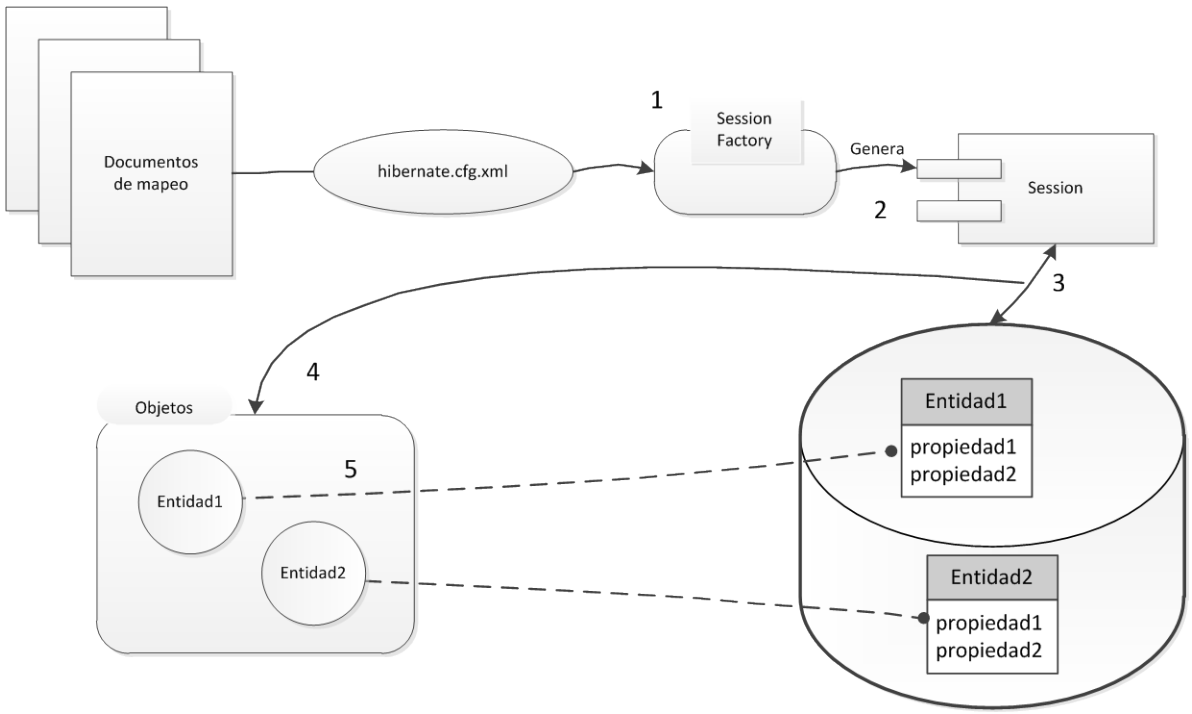


Figura 3.3. Funcionamiento de Hibernate.

3.2.3 Recuperar datos con Hibernate

Hay 3 diferentes formas de recuperar datos con Hibernate: con el API de Criteria, con HQL y con consultas de SQL. El API de Criteria provee un conjunto de objetos Java que se pueden usar para construir consultas. Hibernate provee su propio lenguaje de consultas, HQL, el cual permite usar una sintaxis parecida a SQL para recuperar objetos de la base de datos.

3.3. El marco de trabajo Struts

Struts 2 (referido a partir de aquí como Struts) es un marco de trabajo que permite una clara separación entre la lógica de negoció que interactúa con una base de datos de las paginas JSP que forman la respuesta. Struts es una fuerte implementación del patrón de diseño Modelo Vista Controlador[6] (MVC).

En Struts las tareas de patrón de diseño MVC; modelo, vista y controlador se implementan mediante acciones, resultados y el controlador FilterDispatcher respectivamente. En la figura 3.4 se muestra ésta implementación.

[6] El patrón de diseño MVC especifica cómo debe ser estructurada una aplicación, las capas que van a componer la misma y la funcionalidad de cada una. Además, propone separar el modelo, la vista y el controlador en tres componentes disjuntos y desacoplados en lo posible, a fin de poder modificar un componente independientemente de los demás y que el resultado impacte poco o nada sobre los otros componentes.

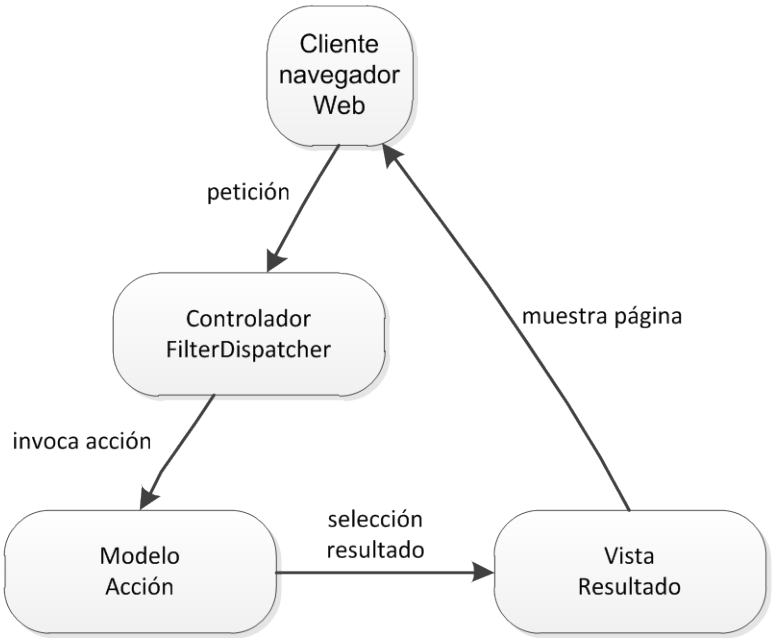


Figura 3.4 El MVC de Struts

El modelo

Implementar acciones en Struts es cosa sencilla, cualquier clase puede ser una acción si así se desea. La principal característica de las clases acción es que no tienen que heredar ninguna clase o interfaz del API, basta con que ofrezca un método de entrada para que el marco de trabajo las invoque cuando la acción se ejecute. Por convenio este método es llamado execute().

Una acción de Struts cumple dos funciones. En primer lugar, una acción es una encapsulación de las llamadas a la lógica del negocio. En segundo lugar, la acción sirve como ubicación de transferencia de datos, es decir, la acción es la responsable de elegir qué resultado debe mostrar la respuesta a partir de la elección de un número de resultados.

La vista

Normalmente, la vista se trata de páginas JSP, plantillas Velocity o cualquier otra tecnología de capa de presentación.

El controlador FilterDispatcher

Las funciones del controlador son:

1. Determina el URL para la acción que se va a ejecutar.

2. Utiliza una clase acción.

3. Ejecuta el método de acción de la clase si está asociada.

4. Si se han introducido datos, se crear un objeto y actualiza o se posiciona los valores de los parámetros.

5. Regresar a la vista (páginas JSP) para mostrar la respuesta.

El marco de trabajo está compuesto por algo más que sus componentes MVC, que participan en el procesamiento de cada petición. Los principales son los interceptores, el lenguaje OGNL y el controlador ValueStack (la figura 3.5 muestra estos nuevos componentes).

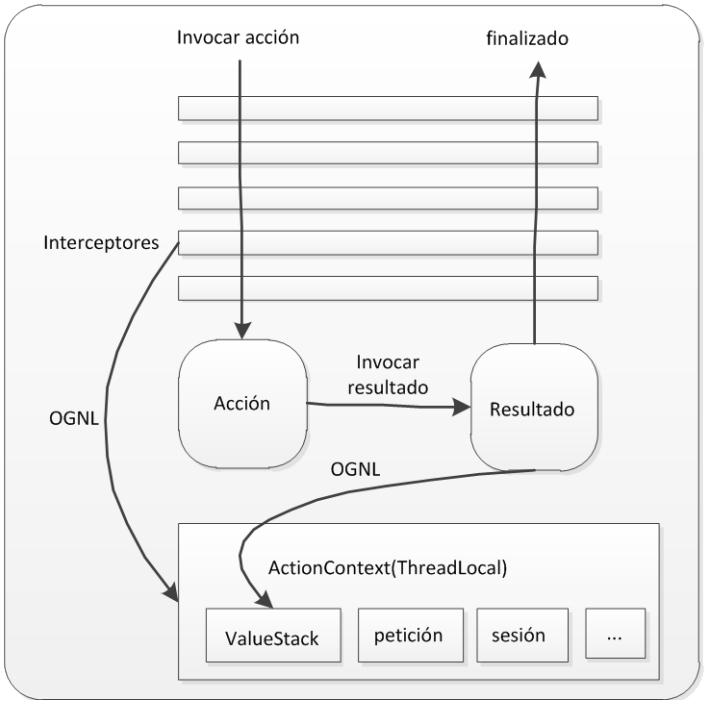


Figura 3.5 El procesamiento de peticiones de Struts

En la figura 3.5, el controlador FilterDispatcher ya ha realizado su función de controlador seleccionando la acción apropiada para gestionar la petición. Los primeros componentes que aparecen después de que se ha seleccionado una acción, son los interceptores[7], los siguientes componentes son ValueStack y el lenguaje OGNL.

Interceptores

Struts se basa en una lista de interceptores, que son llamados pila de interceptores que son los encargados de prestar servicios específicos para la gestión de los parámetros, la validación de formularios o incluso la carga de archivos.

Todas las acciones tendrán una pila de interceptores asociada a la misma. Éstos interceptores se invocan tanto antes como

[7] Un interceptor no es ni más ni menos que una clase Java que implementa la interface com.opensymphony.xwork2.Interceptor.

después de la acción[8], si bien en realidad se disparan después de que se haya ejecutado el resultado.

Un interceptor tiene un ciclo de ejecución condicional en tres fases:

4. Realiza algunas tareas de preprocesamiento.

5. Traslada el control a los interceptores sucesivos y en última instancia a la acción.

6. Realiza tareas de postprocesamiento.

ValueStack y OGNL

ValueStack es una zona de almacenamiento que contiene todos los datos asociados con el procesamiento de una petición. OGNL es la herramienta que permite acceder a los datos que se colocan en esa zona de almacenamiento. De forma más concreta, OGNL es un lenguaje de expresiones que permite referenciar y manipular los datos del ValueStack. El lenguaje OGNL se usa para vincular campos de formulario a propiedades Java.

El aspecto más curioso y potente de ValueStack y del lenguaje OGNL es que no pertenecen a ninguno de los componentes individuales del marco de trabajo. Si se examina de nuevo la figura 3.5, se observa que tanto interceptores como resultados utilizan OGNL para los valores de ValueStack. Los datos en ValueStack siguen el procesamiento de la petición a través de todas sus fases y se esparcen a lo largo de todo el marco. Esto es posible porque están almacenados en un contexto ThreadLocal llamado ActionContext.

[8] Los interceptores no tienen que hacer algo necesariamente las dos veces en que se disparan, pero existe la posibilidad.

ActionContext contiene todos los datos que conforman el contexto en que ocurre una acción determinada. Esto incluye, a ValueStack, pero también todo aquello que el entorno usará de forma interna, como por ejemplo la petición, la sesión y los mapas de la aplicación desde el API Servlet.

El entorno ofrece diversas formas elegantes para interactuar con los datos sin necesidad de tocar el ActionContext o el ValueStack. Fundamentalmente, se utilizara el OGNL para hacerlo.

En Struts, ninguna acción se invoca de manera aislada. La invocación de una acción es un proceso de varias capas que siempre incluye la ejecución de una pila de interceptores antes y después de la ejecución de la acción. En lugar de invocar directamente el método execute() de la acción, el marco de trabajo crea un objeto llamado ActionInvocation que encapsula la acción y todos los interceptores que se han configurado para dispararse antes y después de que dicha acción se ejecute, véase figura 3.6.

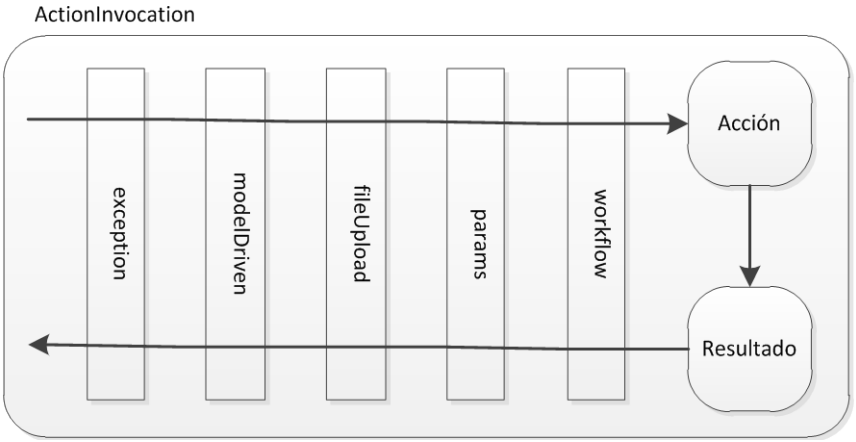


Figura 3.6 ActionInvocation encapsula la ejecución de una acción con sus interceptores y resultados asociados

El interceptor params es el responsable de mover todos los parámetros de la petición hacia el objeto de la acción.

3.3.1 Utilización de Struts.

Struts debe estar declarado en el archivo descriptor de la aplicación, es decir, en el archivo web.xml.

En una aplicación desarrollada con Struts se declaran los componentes (las acciones, los interceptores, los parámetros de configuración y los resultados de las acciones) en un archivo de configuración llamado struts.xml. Éste archivo de configuración debe colocarse en el directorio /WEB-INF/src (o /WEB-INF/classes después de la compilación) para que funcione como defecto. Además, éste archivo se lee al iniciar la aplicación.

En esta arquitectura declarativa se configuraran los componentes, en archivos XML o en anotaciones Java, a partir de los cuales el sistema creara la instancia en tiempo de ejecución de la aplicación.

El marco de trabajo utiliza el archivo struts.xml como punto de entrada, si bien es posible declarar todos los componentes en este archivo, con frecuencia se puede utilizar este archivo únicamente para incluir archivos secundarios XML para modularizar las aplicaciones.

Los documentos XML deben seguir la regla de ordenamiento (figura 3.7).

```
<!ELEMENT struts (package|include|bean|constant)*>

<!ELEMENT package (result-types?, interceptor?, default-interceptor-ref?,
  default-action-ref?, global-results?, global-exception-mappings?, action*)>
```

Figura 3.7 Código de la regla de ordenamiento de struts.xml.

Las acciones

Las acciones en Struts se reagrupan en paquetes según un principio semejante al de paquetes en Java. Una acción tendrá una URL asociada, en la figura 3.8 se muestra la anatomía de una URL:

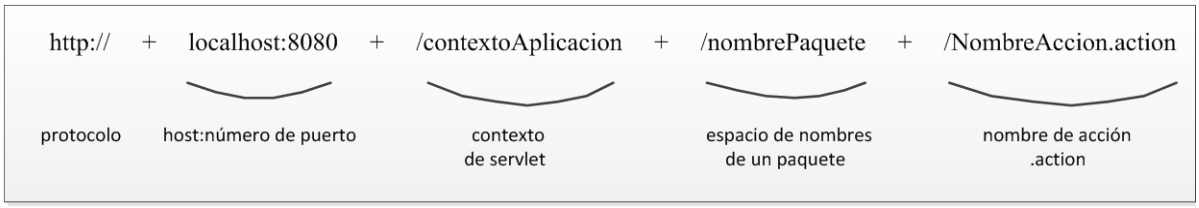


Figura 3.8 Anatomía de un URL

En un paquete, pueden definirse únicamente cuatro atributos: name, namespace, extends y abstract, en donde:

1. name: Es el nombre del paquete. Es obligatorio.

2. namespace: Espacio de nombres para todas las acciones del paquete.

3. extends: Paquete padre del que hereda.

4. abstract: Si true, este paquete se utilizara únicamente para definir componentes heredables, no acciones.

La clase com.opensymphony.xwork2.ActionSupport es la clase de acción predeterminada de Struts. El marco de trabajo crea una instancia de esta clase si no se ha especificado ninguna declaración. Si se hereda la clase ActionSupport, estarán disponibles las siguientes constantes: SUCCESS, NONE, ERROR, INPUT y LOGIN. Las clases de acción devuelven cadenas de caracteres en relación al procesamiento de la acción.

Las clases acción permiten el acceso a los recursos externos como los servlets de Java, estos recursos son los siguientes:

1. La clase ServletContext.

2. La clase HttpSession.

3. La clase HttpServletRequest.

4. La clase HttpServletResponse.

Existen dos clases que permiten acceder a los recursos, com.opensymphony.xwork2.ActionContext y org.apache.struts2.-ServletActionContext.

El marco de trabajo proporciona cuatro interfaces para acceder a los recursos externos:

1. org.apache.struts2.util.ServletContextAware

2. org.apache.struts2.interceptor.ServletRequestAware

3. org.apache.struts2.interceptor.ServletResponseAware

4. org.apache.struts2.interceptor.SessionAware

Los interceptores

El uso de XML es la única opción disponible a la hora de declarar los interceptores.

La etiqueta <interceptors/> se utiliza para definir interceptores en el archivo struts.xml. Antes de poder utilizar un interceptor, se debe declarar en la etiqueta <interceptors/>.

Ya que se hayan declarado los interceptores, se pueden utilizar con la etiqueta <interceptor-ref/> que es un elemento de la etiqueta action.

El orden de la declaración de los interceptores tiene una gran importancia. De hecho, determina el orden de ejecución de los interceptores para cada acción.

Los interceptores implícitos, declarados dentro del defaultStack del paquete struts-default, gestionan la mayor parte de tareas fundamentales, desde la transferencia de datos y la validación, a la gestión de excepciones.

Los parámetros de configuración

Se puede para simplificar y para volver a cargar el administrador, declarar la aplicación en modo de desarrollo. Con esta modalidad de diseñó el archivo se volverá a cargar cada vez que hay un cambio en la aplicación. Con esta etiqueta declarada dentro del archivo struts.xml, no es necesario recargar el contenedor.

Parámetro de configuración struts.devMode:

```
…
<constant name="struts.devMode" value="true">
...
```

Los resultados de las acciones

El marco de trabajo Struts incluye una librería de etiquetas, para las páginas JSP. La libreria de etiquetas de Struts se compone de una categoría para la gestión de datos, estructuras de control, formularios y para usar AJAX (véase http://struts.apache.org/2.0.14/docs/tag-reference.html).

3.4. AJAX

AJAX[9] permite que se actualicen ciertas partes de una página Web, ya sea un elevado número de datos o elementos gráficos; sin necesidad de recargar todo el contenido de la misma.

Las aplicaciones AJAX se apoyan en las siguientes tecnologías:

1. XHTML y CSS. El estándar XHTML se basa en la utilización de un conjunto de etiquetas para la construcción de la página, y el estándar CSS define una serie de propiedades de estilo que pueden aplicarse sobre las etiquetas XHTML, a fin de mejorar sus capacidades de presentación.

2. JavaScript. Las aplicaciones AJAX deben ser escritas en el lenguaje JavaScript.

3. XML. Cuando la aplicación del servidor tiene que enviar una serie de datos en forma estructurada al cliente, XML resulta la solución más práctica y sencilla, ya que se puede manipular fácilmente un documento de estas características y extraer sus datos desde el código JavaScript cliente.

4. El Modelo de Objeto de Documento (DOM). Mediante el DOM, tanto páginas Web como documentos XML pueden ser tratados como un conjunto de objetos organizados de forma jerárquica, cuyas propiedades y métodos pueden ser utilizados desde JavaScript para modificar el contenido de la página en un caso, o para leer los datos de la respuesta en otro.

5. El objeto XMLHttpRequest. Se trata del componente fundamental de una aplicación AJAX. A través de sus

[9] AJAX (Asynchronous JavaScript And XML)

propiedades y métodos es posible lanzar peticiones en modo asíncrono al servidor y acceder a la cadena de texto enviada en la respuesta.

Estas aplicaciones, también se apoyan para su funcionamiento en los siguientes estándares y tecnologías:

1. HTTP. El objeto XMLHttpRequest utiliza el protocolo HTTP para realizar las solicitudes al servidor, manejando también las respuestas recibidas mediante este protocolo.

2. Tecnologías de servidor. Una aplicación AJAX no tendría sentido sin la existencia de un programa en el servidor que atendiera las peticiones enviadas desde la aplicación y devolviera resultados al mismo.

3.4.1 Fases en la ejecución de una aplicación AJAX

El proceso de ejecución de una aplicación AJAX se desencadena al producirse un evento sobre la página Web, como el click de un botón, la selección de un elemento en una lista o la carga de la propia página.

Durante este proceso de ejecución pueden distinguirse las siguientes etapas:

Etapa 1: Creación y configuración del objeto XMLHttpRequest.

El punto de arranque de una aplicación AJAX es la creación del objeto XMLHttpRequest, cuya instancia será almacenada en la variable pública “xhr” a fin de que pueda ser utilizada por el resto del código AJAX (obsérvese la figura 3.9).

La instrucción de la creación del objeto es encerrada dentro de un bloque try, de modo que si se produce un error en esta

línea se mostrara un mensaje de aviso al usuario y no se ejecutará el resto del código.

```
if(window.ActiveXObject) {
        // navegador IE
        xhr = new ActiveXObject("Microsoft.XMLHttp");
} else if((window.XMLHttpRequest) || (typeof XMLHttpRequest) != undefined) {
        // navegador Firefox, Opera y Safari
        xhr = new XMLHttpRequest();
} else {
    // navegador sin soporte AJAX
        alert("Su navEgador no tiene soporte para AJAX");
        return;
}
```

Figura 3.9 Código de la creación de un objeto XMLHttpRequest

El objeto ActiveXObject es utilizado por Internet Explorer para crear instancias de componentes COM registrados en la máquina cliente, a partir de la cadena de registros de los mismos. En el caso de XMLHttpRequest, el componente COM que lo implementa es Microsoft.XMLHttp, aunque es muy posible que el cliente disponga además de versiones más modernas de éste, como la MSXM2.XMLHttp, la MSXM2.XMLHttp.3.0 o incluso la MSXM2.XMLHttp.5.0.

El resto de los navegadores más comúnmente utilizados por los usuarios de Internet, como Opera, Firefox o Safari, implementan XMLHttpRequest como un objeto nativo.

Tras la creación del objeto XMLHttpRequest se invoca a la función de la realización de petición para continuar con el proceso de ejecución de la aplicación.

Etapa 2: Realización de la petición.

Tras configurar los parámetros adecuados del objeto XMLHttpRequest, se procede a lanzar la petición al servidor, operación ésta que puede realizarse en modo síncrono o asíncrono, siendo este último el modo de funcionamiento mayoritariamente utilizado por las aplicaciones AJAX.

En la función de la realización de petición se prepara la petición. Tal preparación consiste en preparar el objeto para la realización de la petición, operación que se realiza invocando:

1. El método open()
2. La propiedad onreadystatechange
3. El método setRequestHeader()
4. El método send()

El método open().

El método open() se encuentra sobrecargado como se muestra en la figura 3.10, las cuatro versiones de este método son:

```
void open(in DOMString method, in DOMString uri, in boolean async,
          in DOMString user, in DOMString password)
void open(in DOMString method, in DOMString uri)
void open(in DOMString method, in DOMString uri, in boolean async)
void open(in DOMString method, in DOMString uri, in boolean async,
                                                  in DOMString user)
```

Figura 3.10 Código del método open() del objeto XMLHttpRequest

Todos los parámetros vienen precedidos por la palabra “in”, lo que significa que deben ser parámetros de entrada. El significado de estos parámetros es el siguiente:

1. method. Se trata de una cadena de caracteres que establece el tipo de petición que se va a realizar. Los más utilizados son:
   - GET
   - POST

2. url. Cadena de caracteres que representa la dirección relativa del programa de servidor que se va a solicitar.

3. async. Consiste en un tipo de dato booleano, utilizado para establecer el modo en que se va a procesar la petición, esto es, en forma asíncrona (true) o síncrona (false).

4. user. Es una cadena de caracteres que representa el nombre del usuario que se debe proporcionar para ser autentificados en el servidor, en caso de que sea necesario.

5. password. Al igual que el anterior, este parámetro se utiliza para la autentificación de un usuario, representando en este caso la contraseña del mismo.

La propiedad onreadystatechange

Cuando la petición realizada desde la aplicación AJAX se lleva a cabo en modo asíncrono es necesario indicarle al objeto XMLHttpRequest cuál es la función de retrollamada que se encargará del procesamiento de la respuesta.

El tipo de la propiedad está definido como EventListener, lo que significa que debe contener la definición de una función manejadora de evento.

Propiedad onreadystatechange del objeto XMLHttpRequest:

Attribute EventListener onreadystatechange

El método setRequestHeader()

Una petición HTTP está formada por una cabecera y un cuerpo. La cabecera, además de la URL del recurso a solicitar, puede incluir otros datos adicionales que permitan informar al

servidor sobre determinadas características del cliente. Estos datos, conocidos como encabezados o datos de cabecera, se definen mediante un nombre y un valor asociado.

Para establecer los encabezados que van a ser incluidos en una petición AJAX utilizaremos el método setRequestHeader() del objeto XMLHttpRequest:

```
void setRequestHeader(in DOMString header, in DOMString
value)
```

El primero de los parámetros es una cadena de caracteres que representa el nombre del encabezado, mientras que el segundo es el valor asignado a éste.

El método send()

Una vez configurados todos los parámetros que afectan a la petición, se procede a su envío mediante la llamada al método send() del objeto XMLHttpRequest. Se proporcionan tres versiones de este método, los cuales son:

```
void send()
void send(in DOMString data)
void send(in Document data)
```

El parámetro indicado en la segunda versión del método representa los datos que van a ser enviados al servidor en el cuerpo de la respuesta, por lo que sólo se utilizará cuando la petición sea de tipo “POST”.

En este caso, los datos son enviados como una cadena de caracteres formada por la unión de parejas nombre=valor. Se debe indicar además en el encabezado “content-type” que el tipo de información que figura en el cuerpo corresponde a datos procedentes de un formulario HTML, para lo cual deberá establecerse su valor a “application/x-www-form-urlencoded”.

En la tercera versión es posible enviar en el cuerpo de la petición un documento XML, indicando en este caso el objeto Document que apunta al nodo raíz del árbol de objetos DOM que representa el documento.

Etapa 3: Procesamiento de la petición en el servidor.

El servidor recibe la petición y ejecuta el componente correspondiente que, a partir de los datos recibidos, deberá realizar algún tipo de procesamiento, incluyendo consultas a bases de datos, y generará una respuesta con los resultados obtenidos.

Etapa 4: Recepción de los datos de respuesta.

Una vez completada la ejecución de código de servidor, se envía una respuesta HTTP al cliente con los resultados obtenidos en el formato adecuado para su manipulación. En éste momento, el navegador invoca a la función de retrollamada definida por el objeto XMLHttpRequest.

La función de retrollamada indicada en la propiedad onreadystatechange, es la encargada de la recepción y procesamiento de los datos enviados por el servidor en la respuesta.

Para comprobar en qué momento se ha completado la recepción de los datos del cliente, se utiliza la propiedad readyState del objeto XMLHttpRequest. El cambio de valor de ésta propiedad es lo que genera el evento onreadystatechange.

Los posibles valores que puede tomar readyState durante el ciclo de vida de una petición AJAX son:

- 0. El objeto XMLHttpRequest se ha creado pero aún no se ha configurado la petición.

- 1. La petición se ha configurado pero aún no se ha enviado.
- 2. La petición se acaba de enviar, aunque aún no se recibido respuesta.
- 3. Se ha recibido la cabecera de la respuesta pero no el cuerpo.
- 4. Se ha recibido el cuerpo de la respuesta. Es el momento en que ésta puede procesarse.

La mayoría de las ocasiones sólo interesa conocer si el estado de la petición ha adquirido el valor 4 para ver si es posible procesar los datos de la respuesta.

A través de la propiedad status del objeto XMLHttpRequest es posible conocer el estado de la respuesta, lo que permite saber si se ha producido algún tipo de error en el servidor o si por el contrario los datos recibidos son correctos:

```
Readonly attribute unsigned short status;
```

El valor contenido en esta propiedad representa el código de estado del servidor. Un código de estado igual a 200 significa que la petición se ha procesado correctamente.

Por otro lado, la propiedad statusText nos devuelve, en caso de error, un mensaje descriptivo asociado al mismo.

Manipulación de la página cliente.

A partir de los datos recibidos en la respuesta y mediante código JavaScript del cliente, se modifican las distintas zonas de la página XHTML que sea necesario actualizar.

El objeto XMLHttpRequest es capaz de recuperar la respuesta recibida desde el servidor como una cadena de caracteres (propiedad responseText) o como un documento XML (propiedad responseXML), en función del formato que se haya dado a ésta desde el servidor.

La sintaxis definida para los tipos de respuestas del objeto XMLHttpRequest son:

```
readonly attribute DOMString responseText;
readonly attribute Document responseXML;
```

Una vez disponibles, el acceso a los datos de la respuesta enviada por el componente en el servidor se lleva a cabo utilizando las distintas propiedades del objeto XMLHttpRequest. Si el componente en el servidor envía los datos como texto plano, se debe utilizar la propiedad responseText para recuperar la información. Hay muchas otras circunstancias en la que la información enviada en la respuesta debe tener cierta estructura y los datos que la componen deben poderse recuperar de una forma individualizada por el código de script del cliente. En estos casos, el servidor debe enviar los datos al cliente en un formato que permita su manipulación, este formato es XML.

Cuando el servidor envía los datos como un documento XML, para recuperar dicho documento y poder manipularlo desde el código JavaScript cliente se utiliza el DOM.

Todos los navegadores compatibles con DOM generan, a partir de un documento XHTML bien formado, un conjunto de objetos que representan las distintas partes del documento

y que proporcionan una serie de propiedades y métodos para que desde una aplicación JavaScript se pueda acceder al contenido del mismo e incluso manipular su estructura.

DOM es una especificación en la que se definen una serie de interfaces (interfaces DOM) con las propiedades y métodos que deben exponer los distintos objetos que componen un documento con estructura XML bien formado, ya se XHTML o simplemente XML. En la página http://www.w3.org/dom se puede encontrar toda la información referida a la especificación DOM.

Aunque XML es el estándar para la transmisión de la información estructurada en la Web, no siempre resulta la solución más adecuada para el envío de grandes cantidades de datos desde el servidor al cliente, pudiendo utilizarse otros formatos alternativos y mucho más eficientes para las aplicaciones AJAX como en el caso de JSON. Entre los inconvenientes de la utilización de XML es el elevado ancho de banda y la lentitud en el procesamiento de los datos codificados en un documento con este formato que pueden mermar el rendimiento de las aplicaciones.

JSON son las siglas de JavaScript Object Notation y hace referencia a un formato de datos, cuya sintaxis se basa en definir objetos JavaScript como cadenas de texto que encapsulen los datos que van a ser transmitidos entre el servidor y la aplicación AJAX cliente.

Este formato reduce el número de bytes a transmitir respecto al formato XML, además, dado que la información se encuentra codificada directamente en JavaScript, el proceso de la manipulación de los datos JSON resulta mucho más rápido que el tratamiento de documentos XML mediante DOM.

3.4.2 JSON

Son dos las estructuras de datos que podemos utilizar para codificar la información en JSON:

1. Objetos

2. Arreglos

Los objetos JSON

Los objetos JSON almacenan la información como una cadena de caracteres formada por parejas nombre-valor, de modo que al reconstruir el objeto JavaScript a partir del objeto JSON cada pareja estará representada por una propiedad con su correspondiente valor.

La sintaxis para definir un objeto JSON es la siguiente:

```
objetoJSON = {propiedad1:valor, propiedad2:valor,
propiedad3:valor};
```

Donde propiedad1, propiedad2 y propiedad3 representan el nombre de la propiedad y valor su valor correspondiente, que puede ser cualquier tipo válido JavaScript: texto, numérico, lógico, objeto o arreglo.

Si se quiere mostrar en un cuadro de diálogo la propiedad1 del objetoJSON, se tendría que escribir la instrucción:

```
alert(objetoJSON.propiedad1);
```

Los objetos JSON también admiten la definición de una función como valor en una pareja nombre-valor, lo que equivale a crear un método en el objeto JavaScript equivalente:

```
objetoJSON ={propiedad1:valor, propiedad2:valor, metodo1:
function() {
          alert(this.propiedad1 + ",”+ this.propiedad2);
}
};
```

Por lo que al ejecutar la siguiente instrucción:

```
objetoJSON.metodo1();
```

Se mostraría un cuadro de diálogo con las propiedades del objetoJSON.

Obsérvese que this se utiliza para acceder desde la función a las propiedades del propio objeto.

Arreglos JSON

Un arreglo JSON no es más que una sucesión de valores cuya sintaxis es:

```
arrayJSON = [valor1, valor2, valor3];
```

Pudiendo ser cada uno de estos valores cualquier tipo válido JavaScript, incluso de distinto tipo cada uno.

Los arrays JSON se definen utilizando la sintaxis de literales de array de JavaScript.

Se puede acceder individualmente a cada dato a través de un índice:

```
alert(datos[0]);
```

3.4.3 Interpretación de JSON en cliente

Un objeto JSON es generado desde una aplicación del servidor y enviado al cliente como una cadena de caracteres en la respuesta, por tanto, para ser recuperado desde el script cliente deberá recurrirse a la propiedad responseText del objeto XMLHttpRequest.

Para que la cadena de texto contenida en esta propiedad sea interpretado como un objeto JavaScript, será necesario recurrir a la función eval() de JavaScript:

```
objetoJSON = eval("("+xhr.responseText+")");
```

Una vez interpretado el objetoJSON, podemos utilizar la expresión:

```
objetoJSON.propiedad
```

3.5 El marco de trabajo Prototype Window Class

Prototype Window Class es un marco de trabajo en JavaScript que apunta al desarrollo sencillo y dinámico de aplicaciones web, permite añadir ventanas a una página HTML. Este marco de trabajo está basado en la biblioteca Prototype y el código se inspira en la biblioteca script.aculo.us para mostrar y ocultar las ventanas. Se puedes incluso utilizar todos los efectos de script.aculo.us para mostrar y para ocultar ventanas si se incluye el archivo de effects.js.

Entre sus características, se pueden destacar las siguientes:

1. Ventanas de redimensionables
2. Posibilidad de minimizar y maximizar
3. Efectos visuales
4. Aspecto configurable
5. Fácil implementación
6. Cuadros de diálogo

Capítulo IV

Metodología genérica utilizada

4.1 La actividad de comunicación.

Inicialmente se identificó la necesidad del cliente del laboratorio de Dispositivos Lógicos Programables (llamado a partir de aquí, cliente).

Se obtuvieron los requisitos del sistema de control de tarjetas por medio de una actividad de comunicación. Tal actividad se realizó a través de entrevistas con el cliente, centrándose tales entrevistas en el dominio del sistema a construir.

Se realizó el modelado de casos de usos del sistema para decidir sobre las fronteras del sistema. Con este modelado además de conocer todo lo que se va construir, sirvió para determinar requisitos obligatorios y deseables debido a que se cometió una breve descripción y una descripción paso por paso para cada caso de uso. Éste modelado fue el documento inicial de la especificación del sistema.

Con las descripciones de los casos de uso se validaron los requisitos, se encontró que faltaban algunas funcionalidades; además, se determinaron atributos faltantes de entidades de la base de datos.

Por otra parte, se realizó un prototipo el cual se le mostro al cliente en varias ocasiones, con la finalidad de determinar funcionalidades y atributos no especificados en el documento de requerimientos. Además, al utilizar tal prototipo se comprueba que los requerimientos puedan implementarse.

4.2 La actividad de planeación.

En la actividad de planeación se utilizaron los casos de uso para dar seguimiento de lo que se construyó.

También, se utilizaron otros modelos para dar seguimiento: clases, secuencia, entidad-relación y relacional con los cuales incluso se pudo estimar la cantidad de trabajo a realizar.

Con los diagramas de clases y secuencia se pudo determinar la cantidad de clases y métodos que se tenían que realizar; y con los modelos entidad-relación y relacional se supo los tipos de datos que se tenían que mapear en Hibernate.

4.3 La actividad de modelado.

Se utilizaron los modelos de casos de uso para aclarar lo que el sistema hará desde el punto de vista del usuario. Los diagramas de clases se utilizaron para identificar los objetos que compondrán el sistema. Y los modelos de secuencia se utilizaron para poder determinar la comunicación entre varios objetos para lograr una funcionalidad deseada por el cliente, es decir, se determinaron los métodos que lograran tal funcionalidad.

El modelo entidad-relación se utilizó para definir la base de datos a utilizar en el sistema. Se determinaron las entidades con sus atributos y relaciones posibles entre ellas, para cada entidad en la base de datos, se definió un objeto en el sistema que se comunica con tal entidad.

Todos los diagramas UML anteriormente mencionados se han utilizado para complementar el documento de la especificación del sistema.

4.4 La actividad de construcción y despliegue.

En la actividad de construcción todos los modelos realizados se utilizaron como una herramienta de ayuda para construir el sistema.

En esta etapa se realizó el prototipo utilizando la plataforma Java EE, particularmente los marcos de trabajo: Hibernate y Struts. Una característica de una aplicación realizada con la tecnología Java EE es que se puede dividir en componentes lo más independientes posibles de modo que un cambio en una parte del sistema no provoque fallas en una parte del código aparentemente no relacionada. Entre estos componentes unos tendrán la responsabilidad de realizar operaciones con la base datos, otros componentes tendrán la responsabilidad de realizar el gestionar el control de que páginas se van a visualizar a una determinada acción de un usuario y otros componentes tendrán la responsabilidad de que se genere una página dinámica a partir de recursos como paginas JSP y vistas realizadas utilizando el marco de trabajo Prototype Window Class. A partir de esta división de responsabilidades hace que el sistema sea robusto ya que la aplicación usa el patrón de diseño Modelo Vista Controlador con el cuál Struts está diseñado.

Un objetivo de realizar el prototipo es que llegara a ser el producto final a lo largo de varios cambios debido a la retroalimentación aportada por el cliente. Sin embargo, la parte de la vista no se modelo todo lo que involucra a AJAX y el marco de trabajo Prototype Window Class debido a que se tuvo que programar para comprobar que lo que se quería hacer era posible. Pero, si se estableció a partir del modelo de secuencia que vista se mostraría.

Se realizaron pruebas de desarrollo, es decir, se efectuaron pruebas de componentes y pruebas de sistema.

Con las primeras pruebas se verifico que las operaciones con la base de datos fuera correcta. Lo elaborado fue comprobar que los mapeos de Hibernate fueran correctos, y en algunos casos se encontró que se habían cometido errores. La forma de realizar lo anterior se hizo ejecutando el archivo de automatización de Ant.

Con las segundas pruebas se probó que la información pasa a través de distintos objetos hasta poder cumplir con las funciones deseadas por el cliente. Una de éstas pruebas fue ejecutar los métodos DAO y Transaccion solamente utilizando el marco de trabajo Hibernare, se encontraron algunos errores cuando no se transmitían correctamente los datos de éstos métodos a la base de datos, y viceversa. Otro tipo de estas pruebas fue comprobar que los datos transferidos de la vista a los métodos de las clases acción llegaran correctamente, y de la forma inversa tuviera el mismo resultado. Y finalmente, se probó que cada función deseada del sistema funcionara cabalmente.

En la actividad de despliegue se consideró instalar un servidor de aplicaciones en la plataforma anfitrión, en la cual se ejecutara el sistema. También se meditó sobre la entrega de un CD con todo lo referente al sistema de control de tarjetas tal como la documentación técnica del sistema, el software y por último se pensó en un posible soporte a futuro que ayude a la realización de posibles cambios o agregar nuevas funcionalidades en un futuro.

Capítulo V

Modelado del sistema de control de tarjetas

Se realizó el documento de especificación del sistema el cual contiene los modelos utilizando UML y el modelado de la base de datos del control de tarjetas.

Los diagramas de clase y el modelo entidad relación, se realizaron de manera conjunta, debido a su relación entre ambos modelos.

5.1 Modelado del sistema utilizando UML

5.1.1 Modelado con casos de uso.

El modelo con casos de uso se utilizaron para representar los requerimientos del cliente y también para delimitar el sistema. En la figura 5.1 se dividió el sistema en una parte de control y otra de estadísticas. Todo lo anterior con el objetivo de realizar primero la parte de control y después la parte de estadísticas.

Este modelado fue la conclusión de varios análisis que se realizaron con los cuales se fue agregando información, hasta llegar al descrito anteriormente, además conforme fue incrementándose el prototipo se fueron obteniendo requerimientos faltantes, así como la validación de los requisitos ya existentes en el modelo.

En el diagrama se incluyen relaciones entre los casos de uso (extensión e inclusión), estas se usaron para indicar el comportamiento adicional que debe incluir un caso con el cual interactúa el administrador. Por ejemplo en el caso de uso Ingresar Datos de Nueva Tarjeta es el caso de uso base, este caso tiene el comportamiento adicional de Comparar con Tarjetas Existentes. Esto significa llanamente que cuando el administrador ingrese una nueva tarjeta se tiene que comparar con las ya existentes en el sistema. Al incluirle comportamiento adicional a cada caso de uso con el que interactúa el Administrador, lo que se determina con esto son los delimitantes del sistema.

En la figura 5.1 se observa que el Administrador interactúa con los siguientes casos de uso: Ingresar Nuevo Tipo de Tarjeta, Ingresar Nuevo Semestre, Ingresar Datos de Nueva Tarjeta, Inscribir Nuevo Alumno, Prestar Tarjeta, Reanudar Funcionamiento de Tarjeta, Poner Tarjeta en Reparación, Devolver Tarjeta, Ver Tarjetas Prestadas, Ver Historial de Reparación y Solicitar Estadísticas.

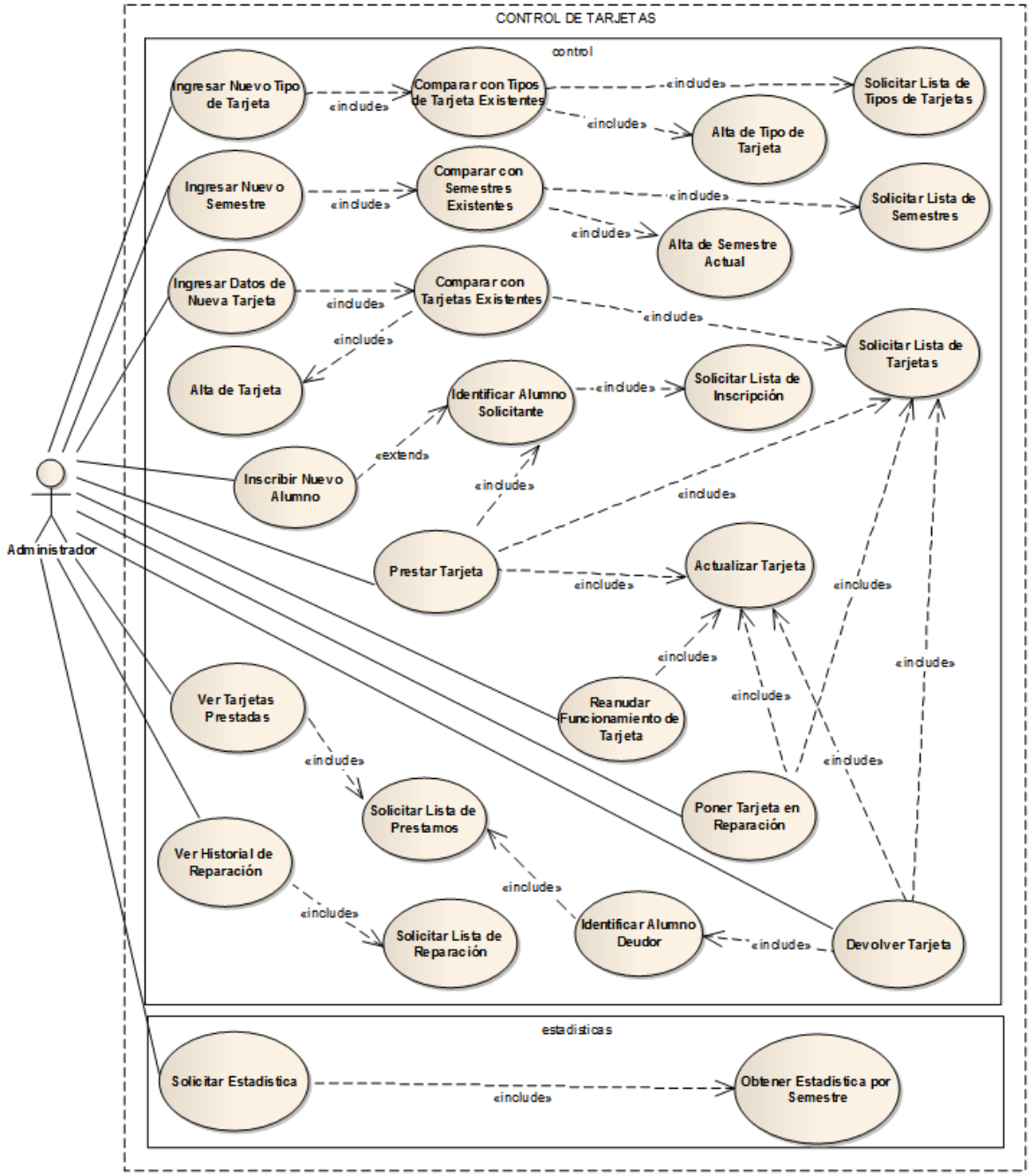


Figura 5.1 Modelado del sistema con casos de uso

Para cada caso de uso en este modelado, se realizaron descripciones textuales con dominio de la aplicación con las cuales se pudieron determinar los requerimientos del cliente y se comprendió todo lo relacionado al control de tarjetas.

Para cada caso de uso se realizó una breve descripción, así como una descripción paso a paso.

En la breve descripción se narra su principal objetivo del caso de uso, y en la descripción paso por paso se narra éste objetivo en detalle.

Ingresar Nuevo Tipo de Tarjeta
Breve descripción El caso de uso Ingresar Nuevo Tipo de Tarjeta permite al administrador ingresar un nuevo tipo de tarjeta.
Descripción paso por paso 1. El administrador introduce el nombre de un nuevo tipo de tarjeta. 2. El sistema de información verificará si no existe el nuevo tipo de tarjeta (Ver caso de uso Comparar con Tipos de Tarjetas Existentes).

Comparar con Tipos de Tarjetas Existentes
Breve descripción El caso de uso Comparar con Tipos de Tarjetas Existentes permite al caso de uso Ingresar Nuevo Tipo de Tarjeta comparar si el nombre introducido no coincide con algun tipo de tarjeta ya existente.
Descripción paso por paso 1. El sistema de información solicitará la lista de tipos de tarjetas (Ver el caso de uso Solicitar Lista de Tipos de Tarjetas). 2. Para cada elemento de tal lista, el sistema de información: 2.1. Compara el nombre ingresado con el nombre de cada elemento de la lista. 2.2. Si el resultado del paso 2.1. no coincide con ningún elemento, se procede a dar de alta el nuevo tipo de tarjeta (Ver caso de uso Alta de Tipo de Tarjeta). 2.3. De lo contrario, no se ingresa y se indica al administrador que el tipo de tarjeta existe en el catálogo de tipos de tarjetas.

Solicitar Lista de Semestre
Breve descripción El caso de uso Solicitar Lista de Tipos de Tarjetas permite al caso de uso Comparar con Tipos de Tarjetas Existentes solicitar la lista de tipos de tarjetas.
Descripción paso por paso 1. El sistema de información obtendrá en una lista el contenido del catálogo de semestres. Cada elemento de la lista contendrá información de cada semestre tal como el periodo.

Alta de Tipo de Tarjeta
Breve descripción El caso de uso Alta de Tipo de Tarjeta permite al caso de uso Comparar con Tipos de Tarjetas Existentes dar de alta un tipo de tarjeta.
Descripción paso por paso 1. Para cada tipo de tarjeta el sistema de información: 1.1. Ingresará los datos de un nuevo tipo de tarjeta en el catálogo de tipo de tarjetas.

- -
- -

Ingresar Nuevo Semestre
Breve descripción El caso de uso Ingresar Nuevo Semestre permite al administrador ingresar un nuevo semestre.
Descripción paso por paso 1. El administrador introduce el periodo para un nuevo semestre. 2. El sistema de información verificará si no existe el nuevo semestre (Ver caso de uso Comparar con Semestres Existentes).

Comparar con Semestres Existentes

Breve descripción

El caso de uso Comparar con Semestres Existentes permite al caso de uso Ingresar Nuevo Semestre comparar si el periodo introducido no coincide con algún semestre ya existente.

Descripción paso por paso

1. El sistema de información solicitará la lista de semestres (Ver el caso de uso Solicitar Lista de Semestres).
2. Para cada elemento de tal lista, el sistema de información:
   2.1. Compara el periodo ingresado con el periodo de cada elemento de la lista.
   2.2. Si el resultado del paso 2.1. no coincide con ningún elemento, se procede a dar de alta el nuevo semestre (Ver caso de uso Alta de Semestre).
   2.3. De lo contrario, no se ingresa y se indica al administrador que el semestre existe en el catálogo de semestres.

Solicitar Lista de Semestre

Breve descripción

El caso de uso Solicitar Lista de Tipos de Tarjetas permite al caso de uso Comparar con Tipos de Tarjetas Existentes solicitar la lista de tipos de tarjetas.

Descripción paso por paso

1. El sistema de información obtendrá en una lista el contenido del catálogo de semestres. Cada elemento de la lista contendrá información de cada semestre tal como el periodo.

Alta de Semestre Actual

Breve descripción

El caso de uso Alta de Semestre Actual permite al caso de uso Comparar con Semestres Existentes dar de alta un semestre.

Descripción paso por paso

1. Para cada semestre el sistema de información:
   1.1. Ingresará los datos de un nuevo semestre en el catálogo de semestres.

- -
- -

Ingresar Datos de Nueva Tarjeta
Breve descripción El caso de uso Ingresar Datos de Nueva Tarjeta permite al administrador ingresar los datos de una nueva tarjeta.
Descripción paso por paso 1. El administrador introduce los detalles de una nueva tarjeta. Éstos son: - Numero de serie - Tipo de cable - Tipo de tarjeta 2. El sistema de información verificará si no existe la nueva tarjeta (Ver caso de uso Comparar con Tarjetas Existentes).

Comparar con Tarjetas Existentes
Breve descripción El caso de uso Comparar con Tarjetas Existentes permite al caso de uso Ingresar Datos de Nueva Tarjeta comparar el número de serie introducido con alguna tarjeta ya existente.
Descripción paso por paso 1. El sistema de información solicitará la lista de tarjetas (Ver el caso de uso Solicitar Lista de Tarjetas). 2. Para cada elemento del catálogo de tarjetas, el sistema de información: 2.1. Compara el número de serie ingresado con el números de serie de cada elemento de la lista. 2.2. Si el resultado del paso 1.1. no coincide con ningún registro, se procede a dar de alta la tarjeta (Ver caso de uso Alta de Tarjeta). 2.3. De lo contrario, no se ingresa y se indica al administrador que la tarjeta existe en el catálogo de tarjetas.

Solicitar Lista de Tarjetas
Breve descripción El caso de uso Solicitar Lista de Tarjetas permite a los caso de uso Comparar con Tarjetas Existentes, Poner Tarjeta en Reparación, Prestar Tarjeta y Devolver tarjeta solicitar la lista de tarjetas.
Descripción paso por paso 1. El sistema de información obtendrá en una lista el contenido del catálogo de tarjetas. Cada elemento de la lista contendrá información de cada tarjeta tal como: - Número de serie - Descripción - Tipo de tarjeta - Si la tarjeta esta prestada - Si la tarjeta esta en reparación

Alta de Tarjeta

Breve descripción

El caso de uso Alta de Tarjeta permite al caso de uso Comparar con Tarjetas Existentes dar de alta una tarjeta.

Descripción paso por paso

1. Para cada tarjeta el sistema de información:
   1.1. Ingresará los datos de la nueva tarjeta en el catálogo de tarjetas.

- -

Prestar Tarjeta

Breve descripción

El caso de uso Prestar Tarjeta permite al administrador prestar una tarjeta.

Descripción paso por paso

1. El administrador selecciona una tarjeta de la lista de tarjetas (Ver el caso de uso Solicitar Lista de Tarjetas).
   1.1. El administrador seleccionará una de las tarjetas disponibles para prestamo.
2. El administrador solicita la identificación del alumno solicitante (Ver caso de uso Identificar Alumno Solicitante).
3. Para cada prestamo de una tarjeta, el sistema procederá a:
   3.1 Actualizar la tarjeta seleccionada (Ver caso de uso Actualizar Tarjeta).
   3.2. El sistema creará un nuevo elemento en el catalogo de prestamo, tal elemento asociará el alumno con la tarjeta.
4. El sistema informará al administrador que el prestamo se realizó con éxito

Identificar Alumno Solicitante

Breve descripción

El caso de uso Identificar Alumno Solicitante permite al caso de uso Prestar Tarjeta identificar al alumno solicitante.

Descripción paso por paso

1. El administrador introduce el número de cuenta del alumno solicitante.
2. El sistema de información obtiene los alumnos inscriptos en el semestre actual (Ver caso de uso Solicitar Lista de Inscripción).
3. Para cada identificación de un alumno solicitante, el sistema:
   3.1. Compará el número de cuenta del solicitante con cada uno de los números de cuenta de los alumnos inscriptos en el semestre actual.
   3.2. Si el resultado del paso 3.1. no corresponde a ningún alumno inscripto, se procede a inscribir al alumno en el semestre actual (Ver caso de uso Inscribir Nuevo Alumno). Una vez inscripto, se puede identificar al nuevo alumno solicitante.
   3.3. De lo contrario, se identifica al alumno solicitante.

Solicitar Lista de Inscripción

Breve descripción

El caso de uso Solicitar Lista de Inscripción permite al caso de uso Identificar Alumno Solicitante solicitar la lista de inscripción de alumnos en el semestre actual.

Descripción paso por paso

1. El sistema determinará la lista de inscripción.
   1.1. Se determinarán los elementos del catálogo de inscripción, que correponden al semestre actual.

Actualizar Tarjeta

Breve descripción

El caso de uso Actualizar Tarjeta permite a los casos de uso Prestar Tarjeta, Poner Tarjeta en Reparación y Devolver tarjeta actualizar la tarjeta.

Descripción paso por paso

1. El sistema actualizará la tarjeta si:
   1.1. La tarjeta se presta.
   1.2. La tarjeta se devuelve.
   1.3. La tarjeta se va reparar.
   1.4. La tarjeta se reparo.

- -

Inscribir Nuevo Alumno

Breve descripción

El caso de uso Inscribir Nuevo Alumno permite al caso de uso Identificar Alumno Solicitante y al administrador inscribir un nuevo alumno solicitante en el semestre actual.

Descripción paso por paso

1. Para cada alta de alumno, el sistema:
   1.1. Buscará si el alumno solicitante no existe en el catalogo de alumnos,
   1.2. Si el resultado del paso 1.1. no encuentra el alumno solicitante en éste catalogo, para cada inscripción el sistema:
      1.2.1. Ingresará los datos del nuevo alumno en éste catalogo.
         Los datos introducidos por el administrador serán:
         - Nombre(s)
         - Apellido paterno y materno
         - Teléfono
         - Celular
   1.3. De lo contrario, obtendrá los datos del alumno solicitante en éste catalogo.
   1.4. El sistema elegirá el semestre actual al que se va incribir el nuevo alumno.
      1.4.1. El sistema creará un nuevo elemento en el catalogo de inscripción, tal elemento asociará el semestre actual con el alumno.
2. El sistema informará al administrador que la inscrición se realizó con éxito.

Poner Tarjeta en Reparación
Breve descripción El caso de uso Poner Tarjeta en Reparación permite al administrador poner una tarjeta en reparación.
Descripción paso por paso 1. El administrador selecciona una tarjeta de la lista de tarjetas (Ver el caso de uso Solicitar Lista de Tarjetas). 2. El administrador introducirá una descripción de la reparación a realizar. 3. El sistema de información, actualizará la tarjeta (Ver el caso de uso Actualizar Tarjeta). 4. El sistema informará al administrador que la tarjeta no esta disponible para prestamo.

Poner Tarjeta en Funcionamiento
Breve descripción El caso de uso Poner Tarjeta en Funcionamiento permite al administrador poner una tarjeta en funcionamiento.
Descripción paso por paso 1. El administrador selecciona una tarjeta de la lista de tarjetas (Ver el caso de uso Solicitar Lista de Tarjetas). 2. El administrador seleccionará la opción de poner en uncionamiento la tarjeta. 3. El sistema de información, actualizará la tarjeta (Ver el caso de uso Actualizar Tarjeta). 4. El sistema informará al administrador que la tarjeta esta disponible para prestamo.

Devolver tarjeta
Breve descripción El caso de uso Devolver tarjeta permite al administrador realizar la devolución de una tarjeta prestada.
Descripción paso por paso 1. El administrador selecciona una tarjeta de la lista de tarjetas (Ver el caso de uso Solicitar Lista de Tarjetas). 2. Para cada tarjeta, el sistema: 2.1. Identificará al alumno deudor (Ver el caso de uso Identificar Alumno Deudor). 2.2. Ingresará un nuevo elemento en el catálogo de devolución, tal elemento indicará el prestamo al que corresponde dicha devolución. 3. El sistema informará al administrador que la devolución se realizó con éxito

Identificar Alumno Deudor

Breve descripción

El caso de uso Identificar Alumno Solicitante permite al caso de uso Prestar Tarjeta identificar al alumno solicitante.

Descripción paso por paso

1. El administrador introduce el número de cuenta del alumno deudor.
2. El sistema de información obtiene los alumnos que tienen alguna tarjeta prestada (Ver caso de uso Solicitar Lista de Prestamos).
3. Para cada identificación de un alumno deudor, el sistema:
   3.1. Comprará el número de cuenta del deudor con cada uno de los números de cuenta de los alumnos que tienen alguna tarjeta prestada.
   3.2. Si el resultado del paso 3.1. no corresponde a ningún alumno, se procede a informar al administrador que el alumno no es un deudor.
   3.3. De lo contrario, se identifica al alumno deudor.

Solicitar Lista de Prestamos

Breve descripción

El caso de uso Solicitar Lista de Prestamos permite a los casos de uso Identificar Alumno Deudor y Ver Tarjetas Prestadas solicitar una lista de las tarjetas en prestamo.

Descripción paso por paso

1. El sistema determinará la lista de prestamos.
   1.1. Se determinarán los elementos del catalogo de prestamos, que correponden a las tarjetas prestadas.
   1.2. Cada elemento contendrá la información relativa a quien se le presto determina tarjeta.

- -

- -

Ver Tarjetas Prestadas

Breve descripción

El caso de uso Ver Tarjetas Prestadas permite al administrador ver las tarjetas prestadas del semestre actual.

Descripción paso por paso

1. El administrador solicita ver las tarjetas prestadas en un determinado instante (Ver caso de uso Solicitar Lista de Prestamos).

- -

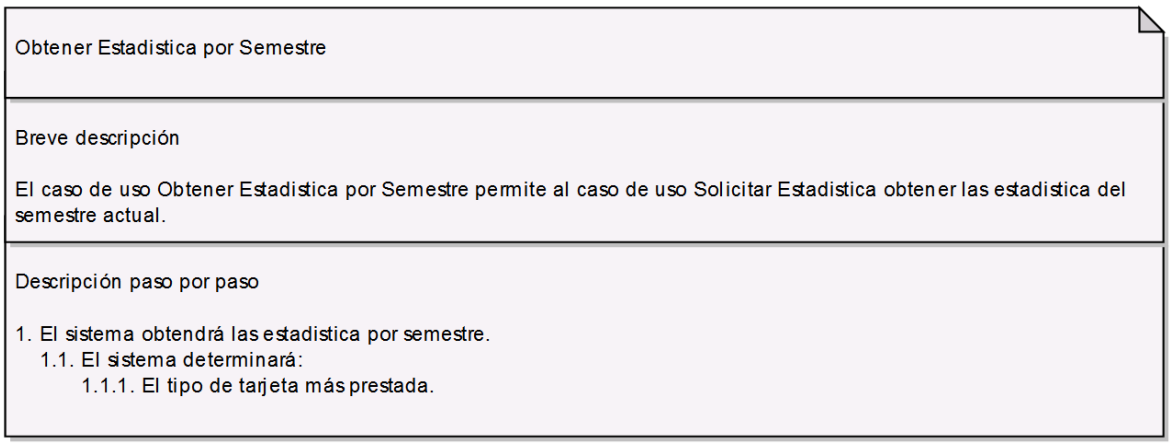
- -

Ver Historial de Reparación
Breve descripción El caso de uso Ver Historial de Reparación permite al administrador ver el historial de la tarjetas que se han puesto en reparación .
Descripción paso por paso 1. El administrador solicita ver el historial de las tarjetas en reparación en un determinado instante (Ver caso de uso Solicitar Lista de Reparación).

Solicitar Lista de Reparación
Breve descripción El caso de uso Solicitar Lista de Reparación permite al caso de uso Ver Historial de Reparación solicitar una lista de las tarjetas que han sido puestas en reparación.
Descripción paso por paso 1. El sistema determinará la lista de reparación. 1.1. Se determinarán los elementos del catalogo de reparaciones, que correponden a las tarjetas en reparación. 1.2. Cada elemento contendrá la información relativa al dianóstico de la reparación.

\- -

Solicitar Estadistica
Breve descripción El caso de uso Solicitar Estadistica permite al Administrador solicitar las estadistica.
Descripción paso por paso 1. El administrador solicita las estadistica por semestre (Ver caso de uso Obtener Estadistica por Semestre).



Con estas descripciones se validarón los requisitos en tanto a consistencia, realismo y verificabilidad. Además, con estas descripciones se abstrayeron entidades y sus relaciones entre estas entidades que forman parte del modelado orientado a objetos del sistema de control de tarjetas.

Por ejemplo, si consideramos las descripciones del caso de uso Inscribir Nuevo Alumno, se determina que atributos son necesarios cuando se inscribe un alumno. En este caso de uso se determinó la entidad alumno y las relaciones con otras entidades, como la entidad inscripción.

5.1.2 Diagramas de clases

La figura 5.2, muestra el diagrama de clases POJO, cuyos componentes tienen la responsabilidad de representar el dominio del sistema de control de tarjetas. Estas clases (Alumno, Devolucion, Estadistica Inscripcion, Personal, Prestamo, Reparacion, Semestre, Tarjeta, TipoTarjeta) son el modelado de la información perdurable, es decir, es la información que se almacenará en la base de datos.

En este diagrama se identificaron las clases del dominio del sistema de control de tarjetas, y sus respectivas relaciones entre tales clases. Si se observa la figura 5.2, se tiene asociaciones y relaciones todo/parte. Las asociaciones que se determinaron son relaciones con multiplicidad uno a muchos,

uno a uno. Las relaciones todo/parte se refiere a relaciones de generalizacion (herencia e implementacion de interfaces).

En este modelo, para cada entidad, se diseño una interface y una clase, donde la interface tiene el nombre de la entidad y la clase que implementa esta entidad tiene el nombre de la entidad concatenado con "Impl". Ademas, se definio que tales clases le corresponda el paquete "tarjetas.control.pojo".

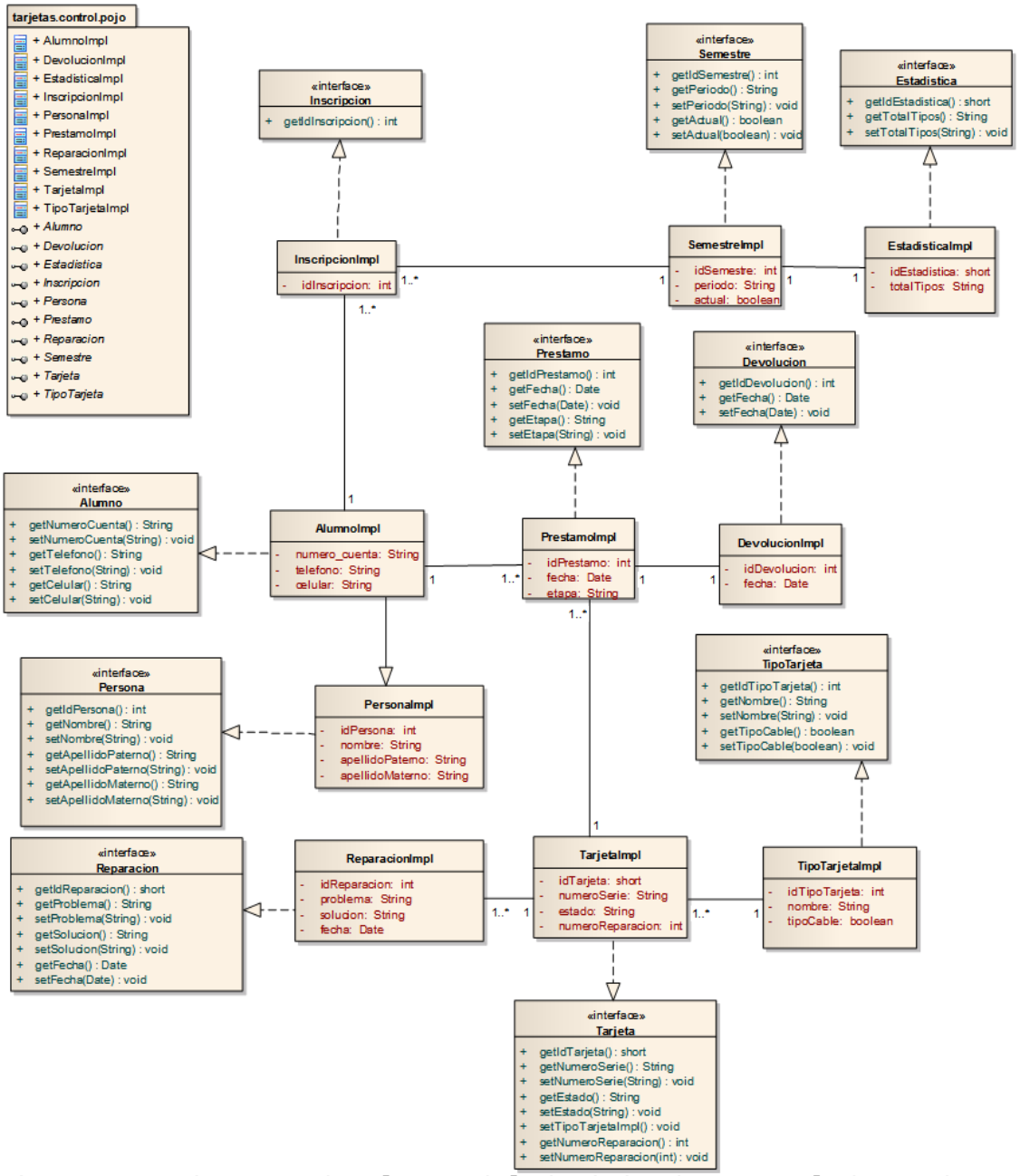


Figura 5.2 Diagrama de clases del dominio de control de tarjetas

La figura 5.3, muestra el diagrama de clases DAO, cuyos componentes tienen la responsabilidad de realizar operaciones con la base de datos. Tales operaciones pueden ser guardar, actualizar o consultar una entidad cualquiera en la base de datos.

Estas clases (AlumnoDAO, DevolucionDAO, EstadisticaDAO, InscripcionDAO, PrestamoDAO, SemestreDAO, TarjetaDAO, TarjetaReparacionDAO, TipoTarjetaDAO) representan el modelado que son las encargadas de la interccion con la base de datos.

En este diagrama solo se identifican relaciones todo/parte. Las relaciones todo/parte que se determinaron, relaciones de generalizacion (implementacion de interfaces).

En este modelo, para cada entidad, se diseño una interface y una clase, donde la interface tiene el nombre de la entidad y la clase que implementa esta entidad tiene el nombre de la entidad concatenado con “DAOImpl”. Ademas, se definio que tales clases le corresponda el paquete “tarjetas.control.dao”.

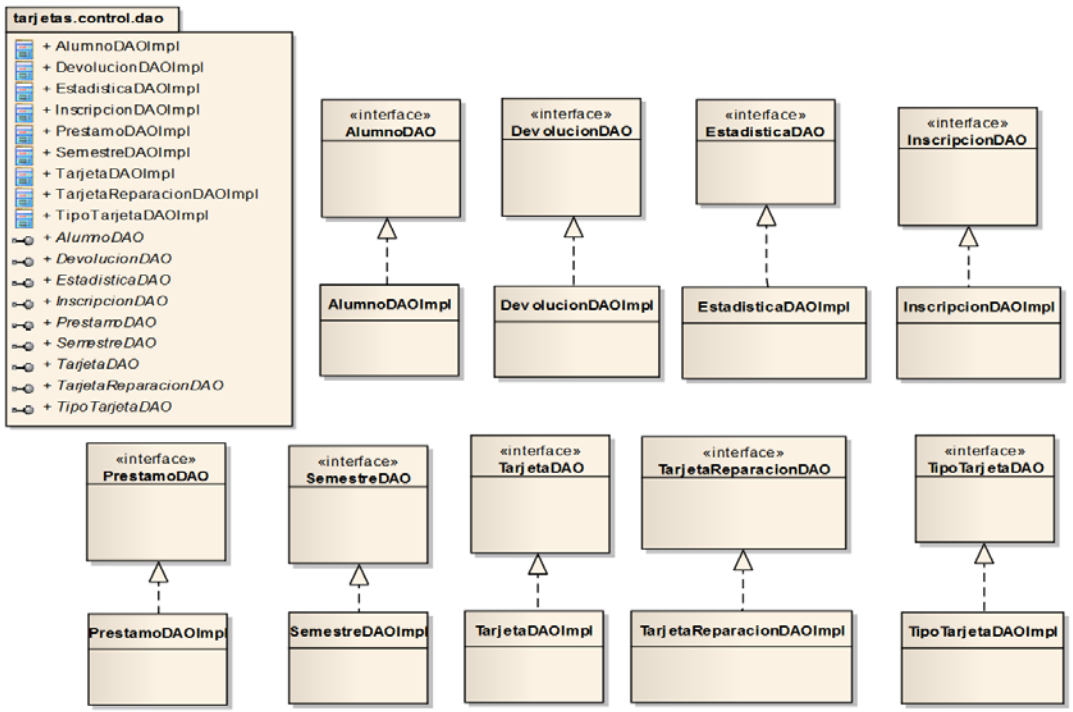


Figura 5.3 Diagrama de clases DAO

La figura 5.4, muestra el diagrama de clases TRANSACCION, cuyos componentes tienen la responsabilidad de verificar que finalicen correctamente o fallen las operaciones con la base de datos. Tales operaciones pueden ser guardar y actualizar un registro cualquiera en una tabla de la base de datos; si finaliza correctamente se realizará un commit, caso contrario un rollback.

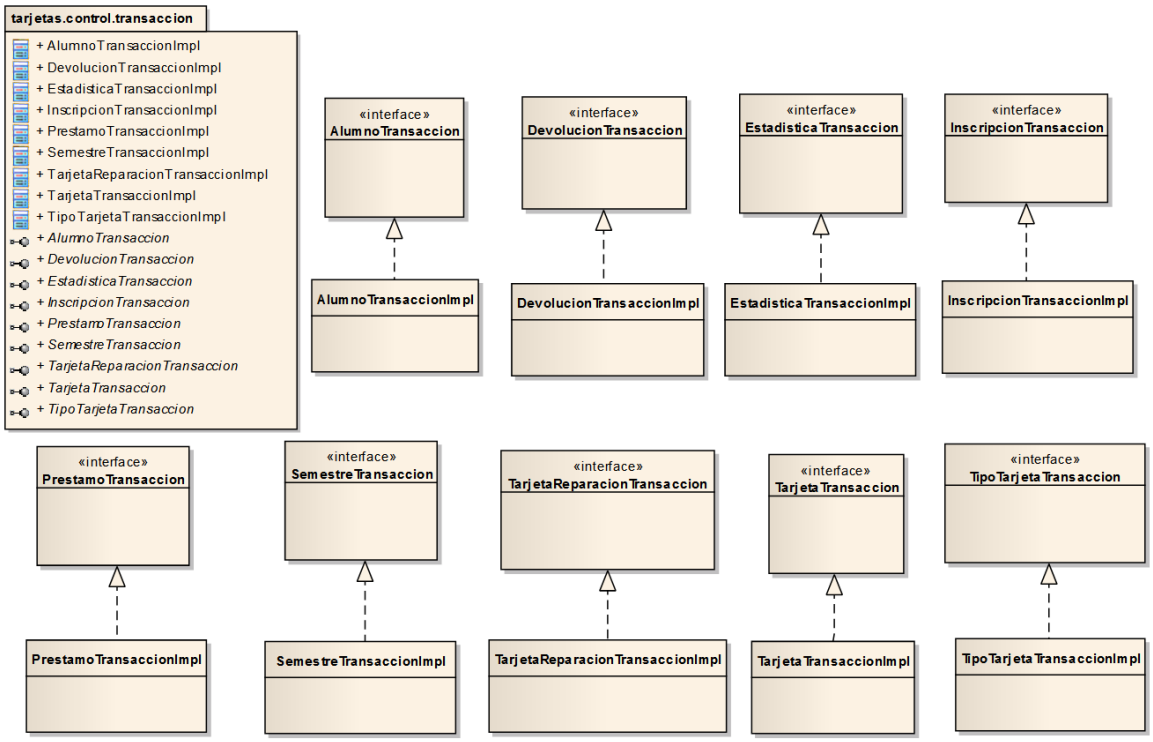


Figura 5.4 Diagrama de clases TRANSACCION

Estas clases (AlumnoTransaccion, DevolucionTransaccion, EstadisticaTransaccion, InscripcionTransaccion, Prestamo-Transaccion, SemestreTransaccion, TarjetaReparacion-Transaccion, TarjetaTransaccion, TipoTarjetaTransaccion) representan el modelado que se encargan del éxito o falla de la interccion con la base de datos.

En este diagrama, tambien solo se identifican relaciones todo/parte. Las relaciones todo/parte que se determinaron, relaciones de generalizacion (implementacion de interfaces).

En este modelo, para cada entidad, se diseño una interface y una clase, donde la interface tiene el nombre de la entidad y la clase que implementa esta entidad tiene el nombre de la entidad concatenado con “TransaccionImpl”. Ademas, se definio que tales clases le corresponda el paquete “tarjetas.control. transaccion”.

La figura 5.5, muestra el diagrama de clases ACTION, cuyos componentes tienen la responsabilidad de realizar el control de las vistas a mostrar.

Estas clases (DevolucionAction, EstadisticaAction, PrestamoAction, ReparacionAction , SemestreAction, TarjetaAction, TipoTarjetaAction) representan el modelado que se encarga del control de que vista se mostrará al Administrador dependiendo de una determinada acción. Ademas, se definio que tales clases le corresponda el paquete “tarjetas.control. action”.

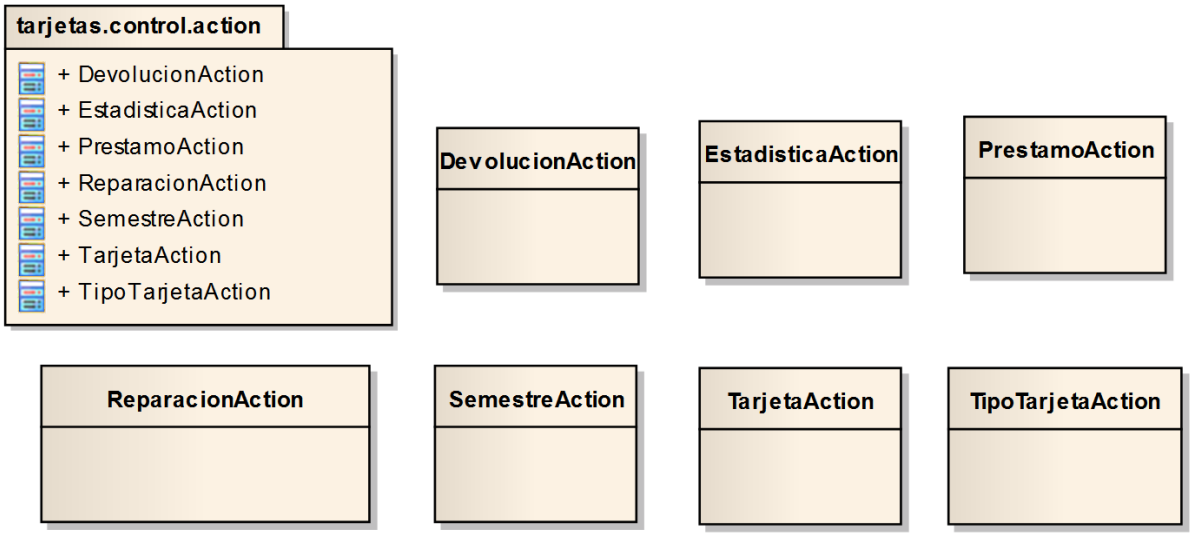


Figura 5.5 Diagrama de clases ACTION

En estas tres ultimas figuras, cuando se modelaron no se definieron sus metodos. En la siguente sección trata sobre diagramas de secuencia, con estos diagramas se determinaron los metodos necesarios para los componentes que componen las tres diagramas anteriores mencionados (diagrama de clases

DAO, diagrama de clases TRANSACCION y diagrama de clases ACTION).

5.1.3 Modelado con diagramas de secuencia

Ademas, de determinar los métodos necesarios para los componente DAO, TRANSACCION, ACTION los diagramas de secuencia se utilizaron para encontrar atributos faltantes de clases, o atributos faltantes de las entidades de la base de datos.

Los modelos de secuencia se utilizaron para programar el sistema de control de tarjetas, esto se expone una parte en este capitulo y otra parte en el siguiente.

Cada diagrama de secuencia representa la interaccion entre el Administrador y el sistema de control de tarjetas. El objetivo de cada diagrama es modelar cada caso de uso con el cual el Administrador interactúa directamente (Figura 5.1), a continuacion se presentan los modelos que corresponden a cada de estos casos de uso, el unico que no tiene un diagrama es el caso de uso Incribir Nuevo Alumno.

Diagrama de secuencia para Ingresar Datos de Nueva Tarjeta

En los diagramas de secuencia, cuando una flecha apunta hacia una instancia, significa que se deberá crear el metodo en tal componente (clase e interface). En la figura 5.6 se muestra el diagrama de secuencia para Ingresar Datos de Nueva Tarjeta, se puede observar que el diagrama especifica que se debe crear el metodo consultar(numeroSerie) en la interface TarjetaDAO y en la clase TarjetaDAOImpl. Si el atributo numeroSerie no existiera en la clase Tarjeta (vease el tipo de retorno para este metodo: List<Tarjeta>) se debe agregar tal nuevo atributo a esta clase. De una manera similar se realiza un proceso similar en cada instancia del diagrama.

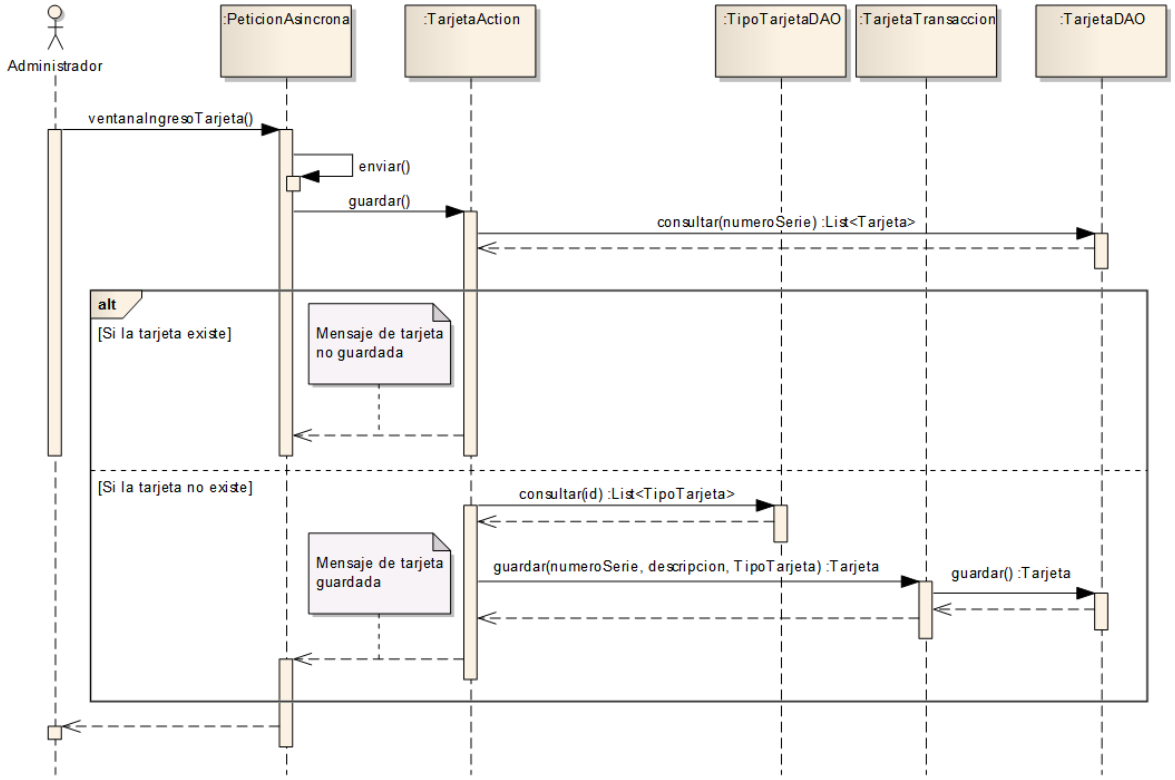


Figura 5.6 Diagrama de secuencia para Ingresar Datos de Nueva Tarjeta

En la figura 5.7, se muestra el diagrama de secuencia para Ingresar Nuevo Tipo de Tarjeta. En este diagrama es semejente al anterior, se observa un cuadro que representa una condición de acuerdo al resultado de un método. En este caso, si el método consultar(nombre) retorna que el tipo de tarjeta existe, se le indicará al Administrador la existencia.

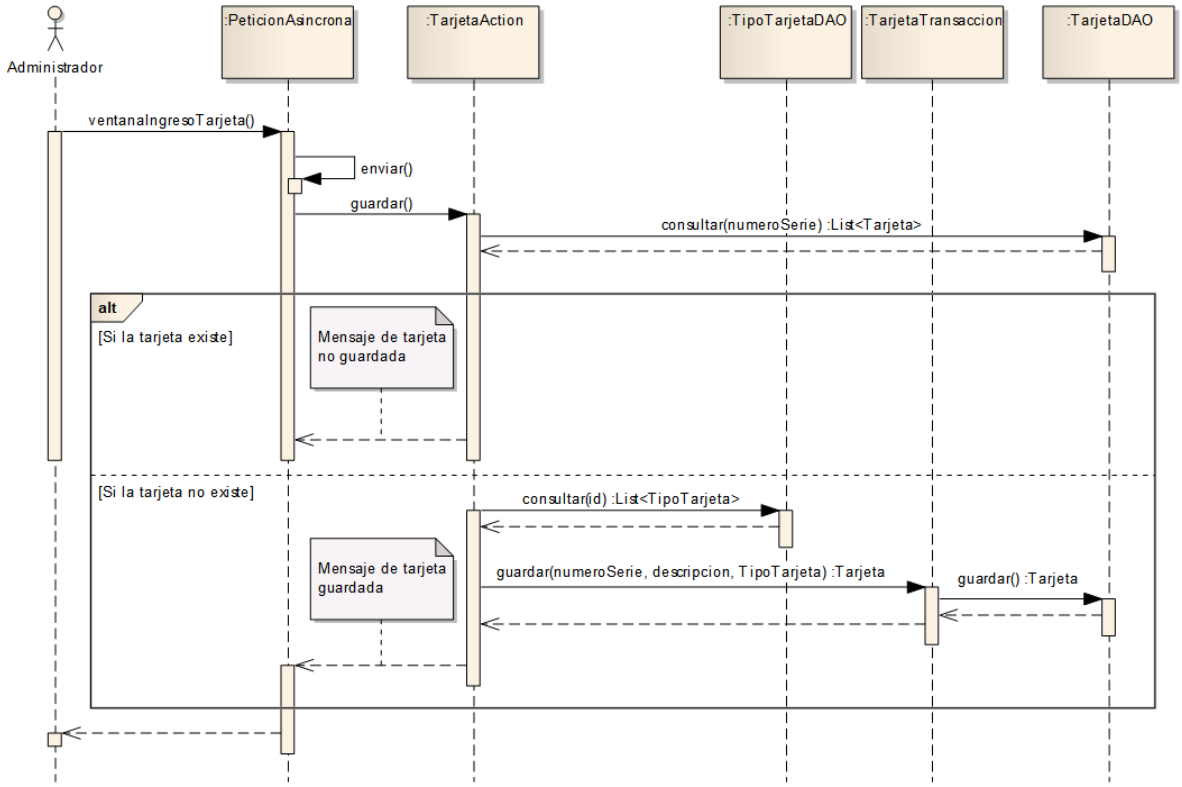


Figura 5.7 Diagrama de secuencia para Ingresar Nuevo Tipo de Tarjeta

En las siguientes figura, se muestra los diagrama de secuencia para Ingresar Nuevo Semestre (5.8), Prestar Tarjeta (5.9), Devolver Tarjeta (5.10), Reanudar Funcionamiento de Tarjeta (5.11), Poner Tarjeta en Reparación (5.12), Ver Tarjetas Prestadas (5.13), Ver Tarjetas en Reparación (5.14) y Solicitar Estadistica (5.15).

Es importante mencionar que los diagrama de secuencia para Ingresar Nuevo Semestre y Prestar Tarjeta se les agrego la parte para llevar la cuenta de las tarjetas prestadas, con esto se registro por separado el numero total para cada tipo de tarjeta prestada en el semestre en curso. La partes que aparecen en rojo en estos dos diagramas mencionados, son las nuevas partes que se le agrego al diagrama original.

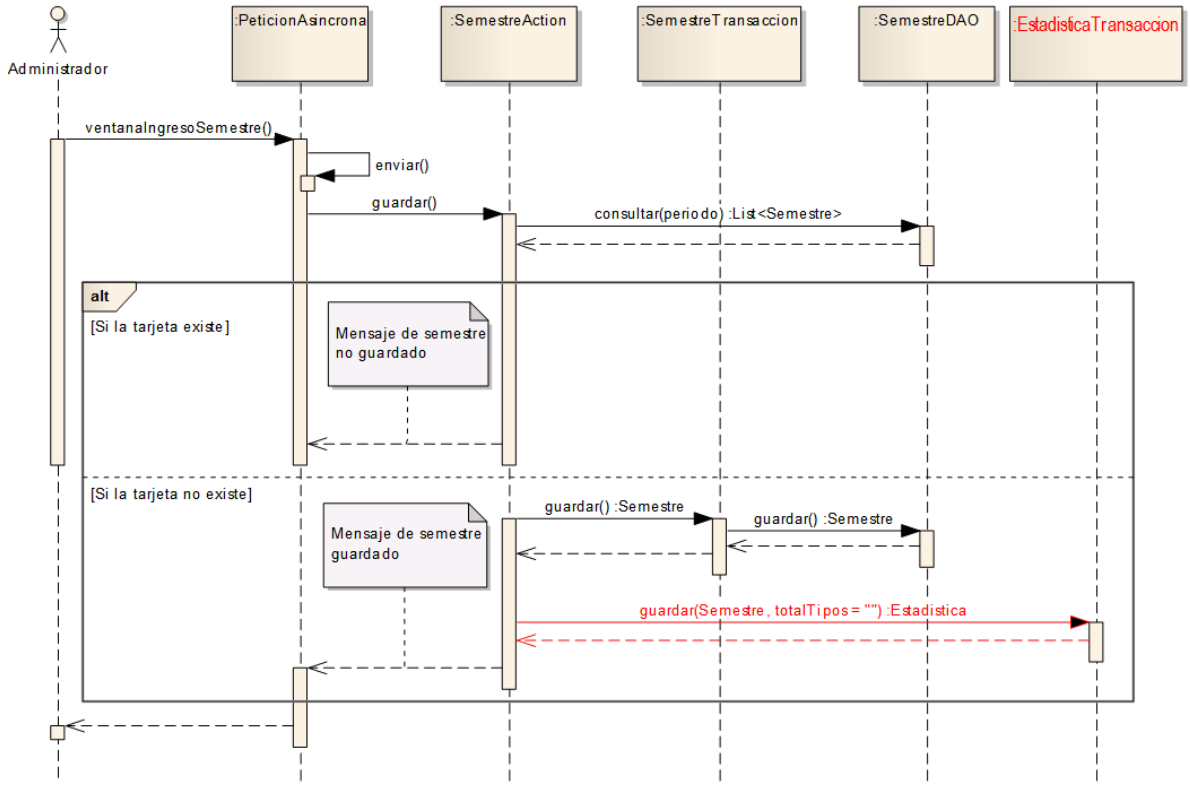


Figura 5.8 Diagrama de secuencia para Ingresar Nuevo Semestre

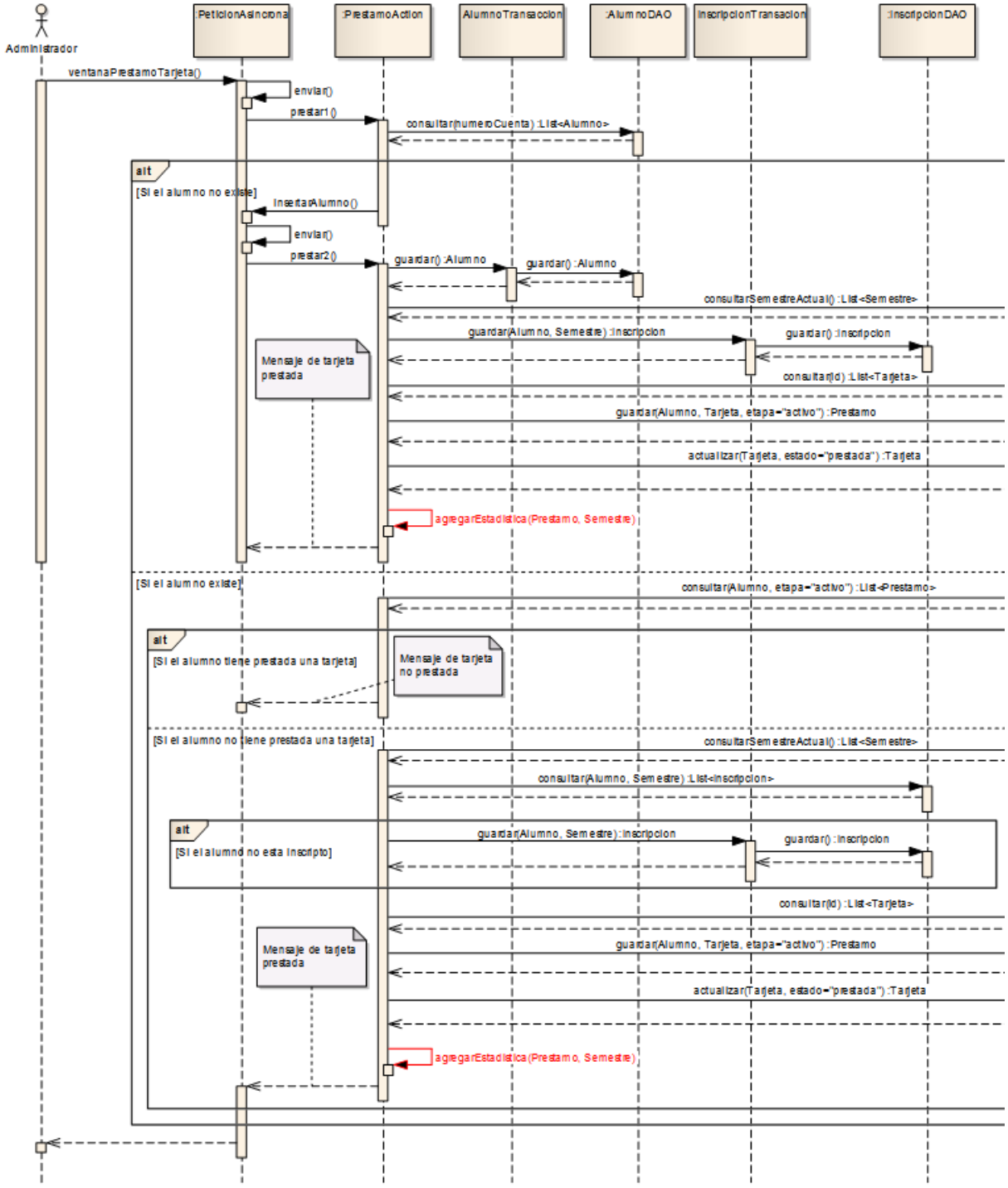
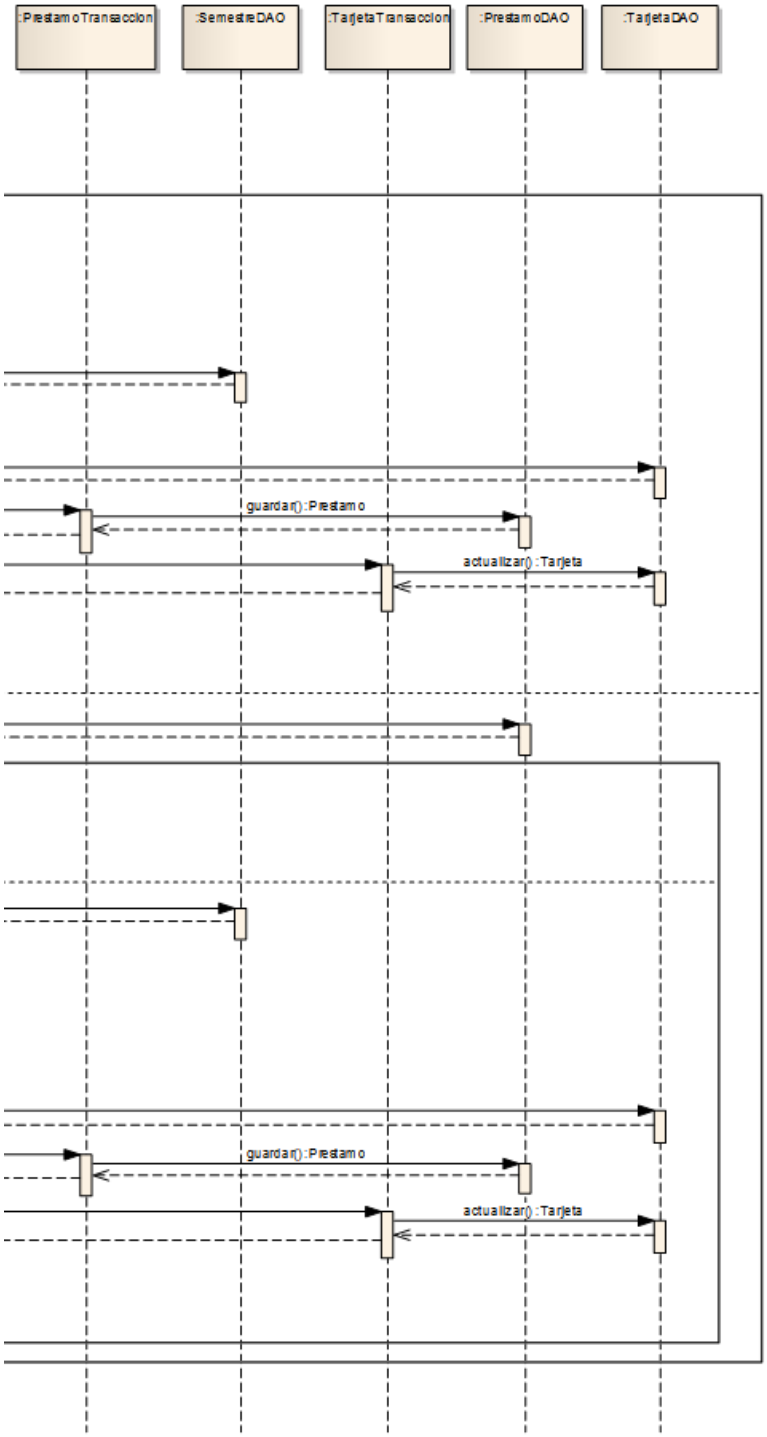


Figura 5.9 Diagrama de secuencia para Prestar Tarjeta


:PrestamoTransaccion
:SemestreDAO
:TarjetaTransaccion
:PrestamoDAO
:TarjetaDAO
guardar():Prestamo
actualizar():Tarjeta
guardar():Prestamo
actualizar():Tarjeta

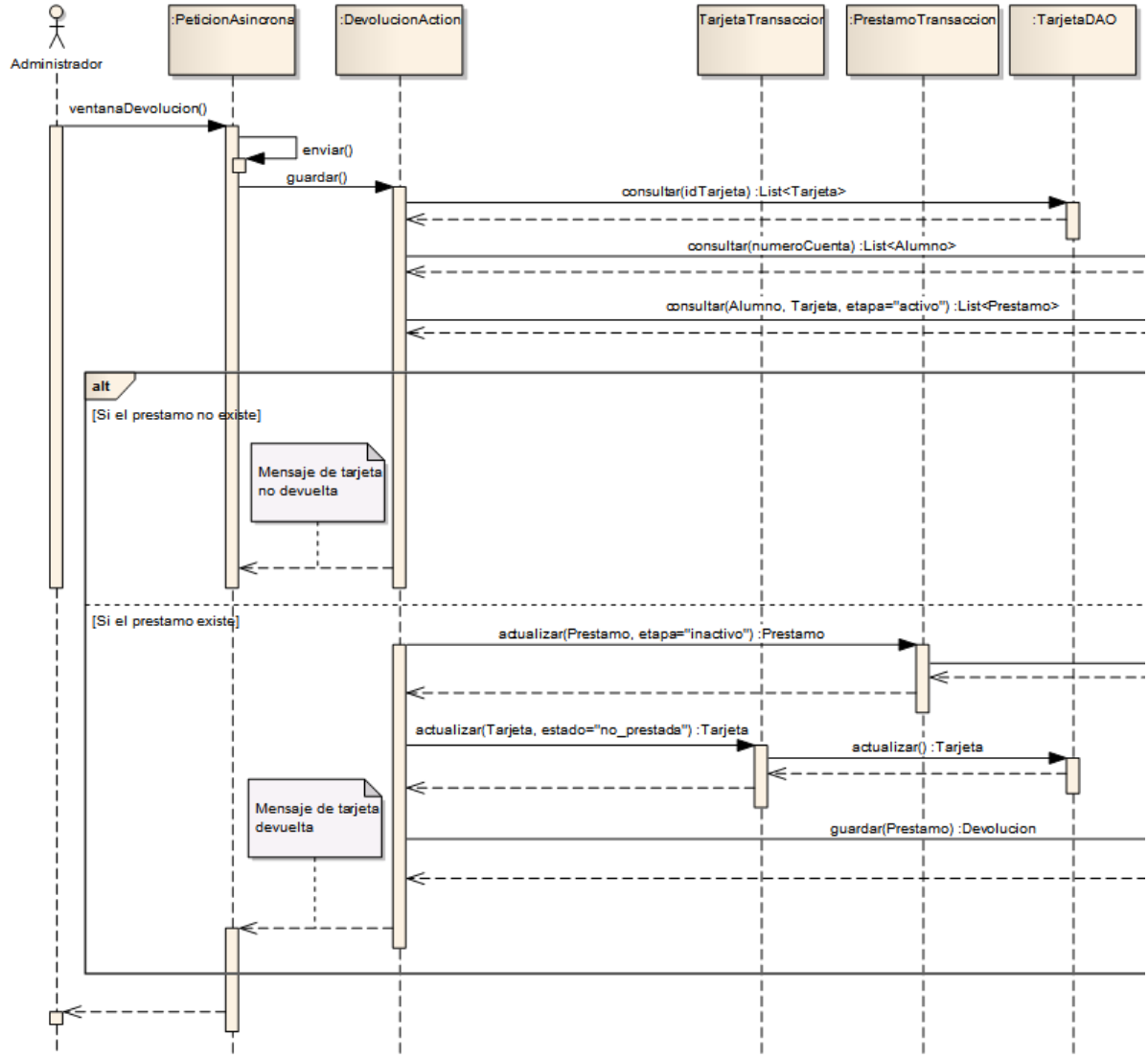
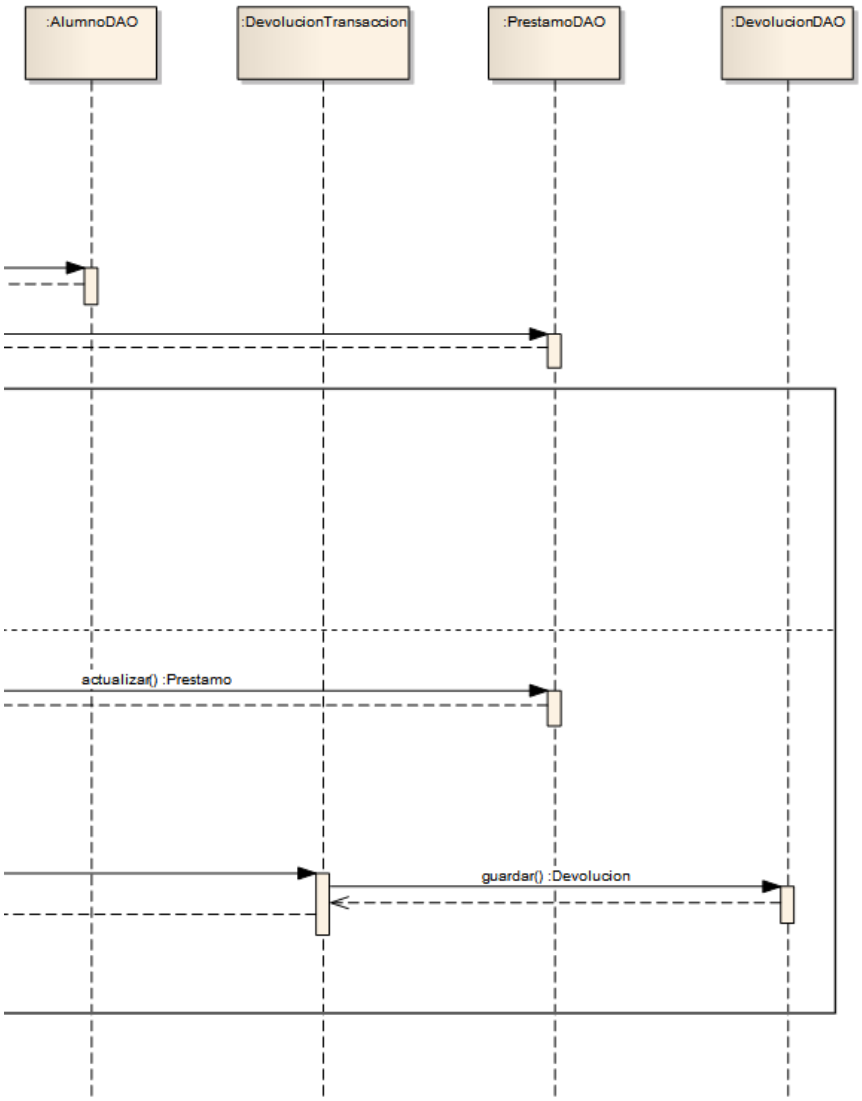


Figura 5.10 Diagrama de secuencia para Devolver Tarjeta


:AlumnoDAO
:DevolucionTransaccion
:PrestamoDAO
:DevolucionDAO
actualizar() :Prestamo
guardar() :Devolucion

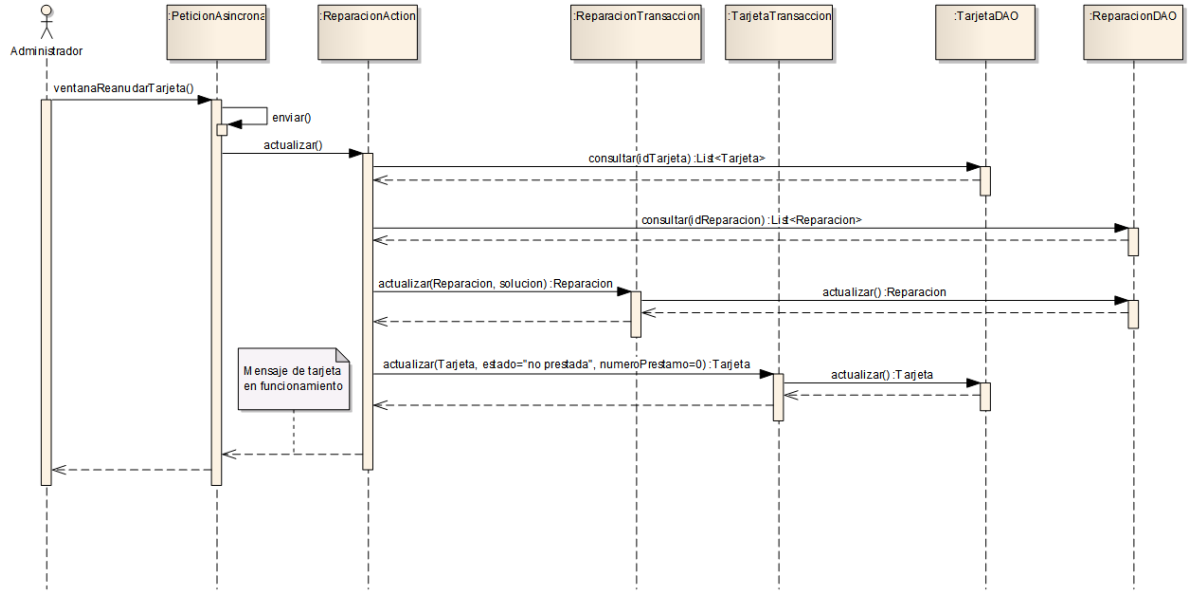


Figura 5.11 Diagrama de secuencia para Reanudar Funcionamiento de Tarjeta

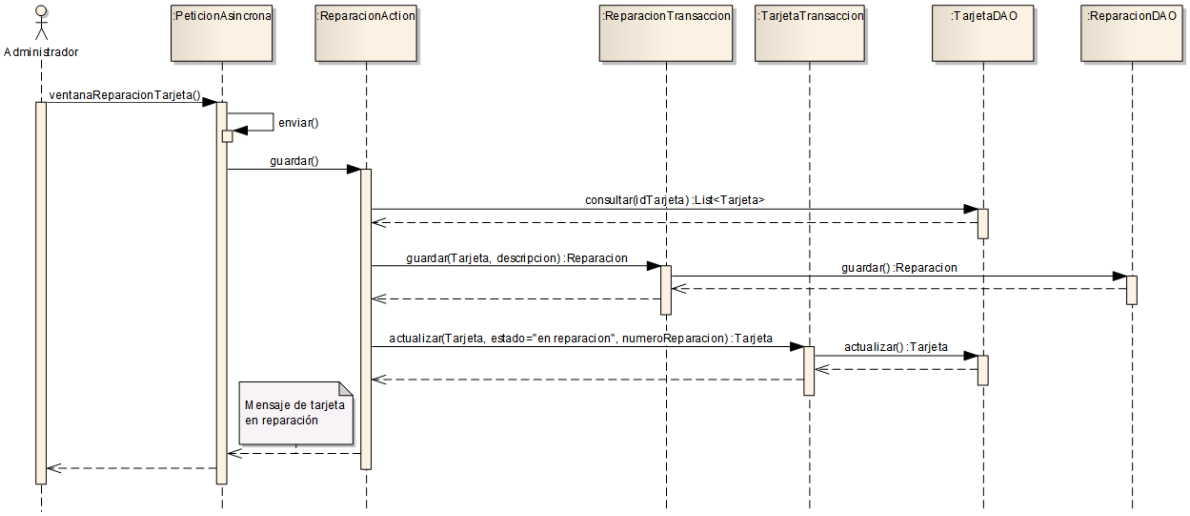


Figura 5.12 Diagrama de secuencia para Poner Tarjeta en Reparación

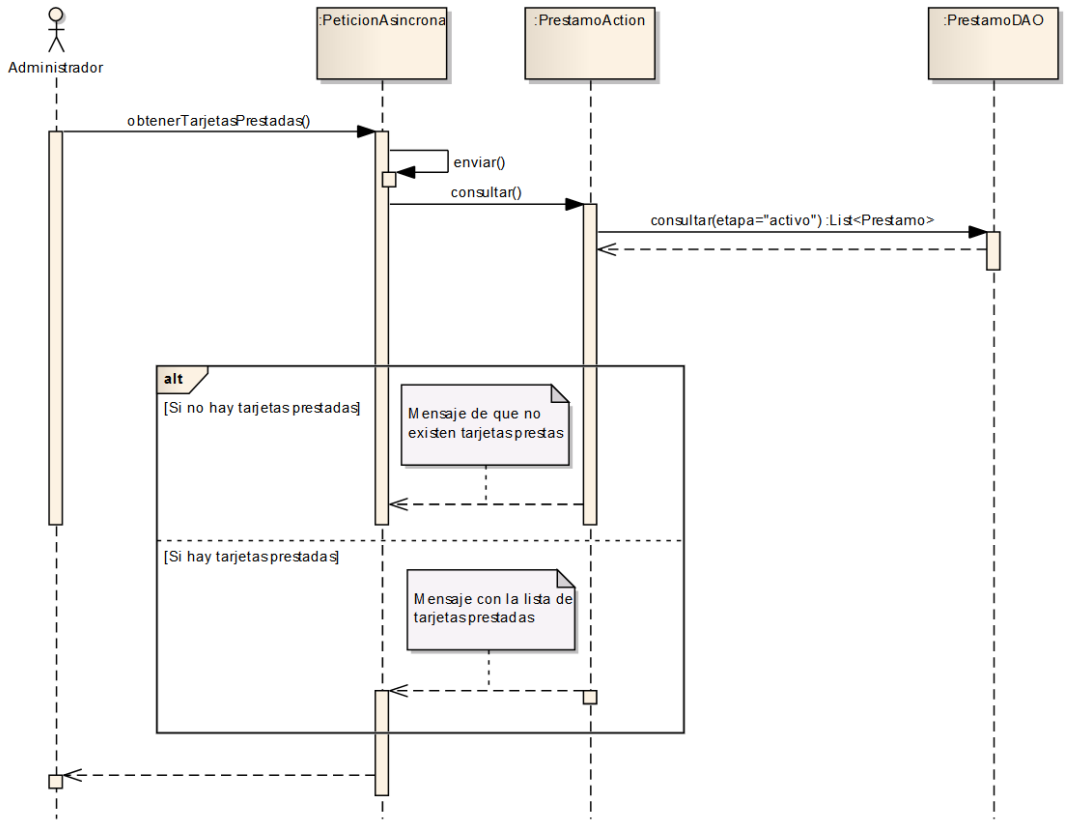


Figura 5.13 Diagrama de secuencia para Ver Tarjetas Prestadas

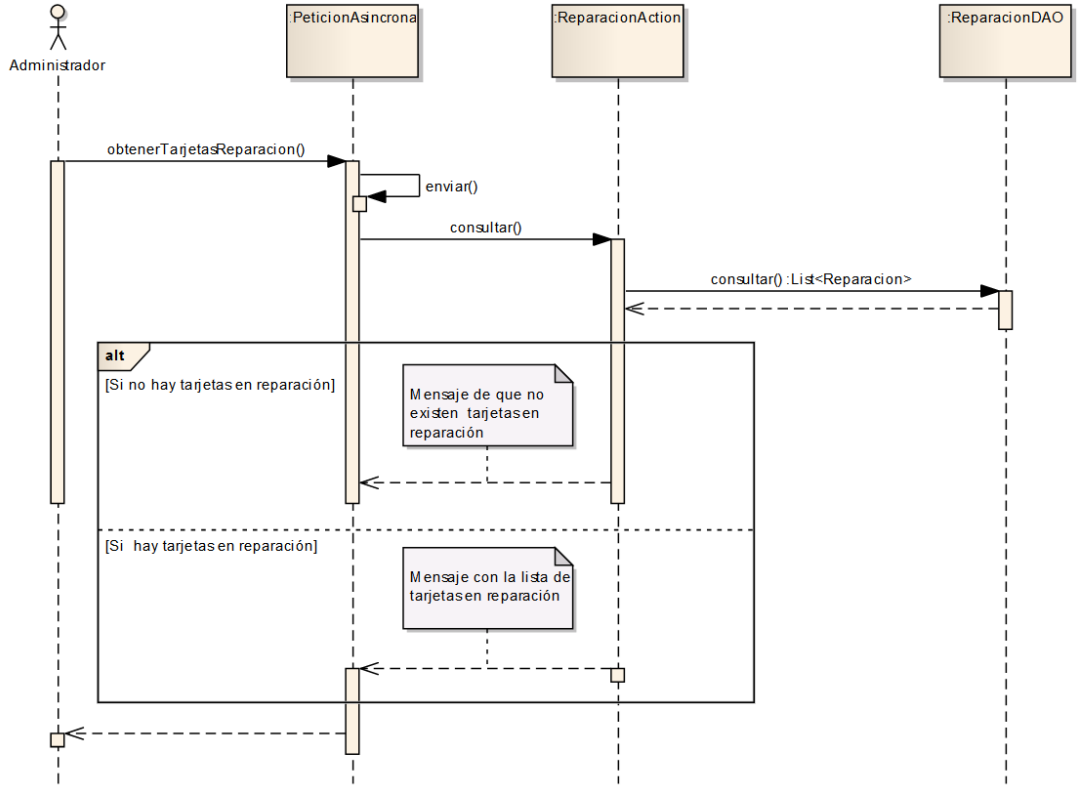


Figura 5.14 Diagrama de secuencia para Ver Tarjetas en Reparación

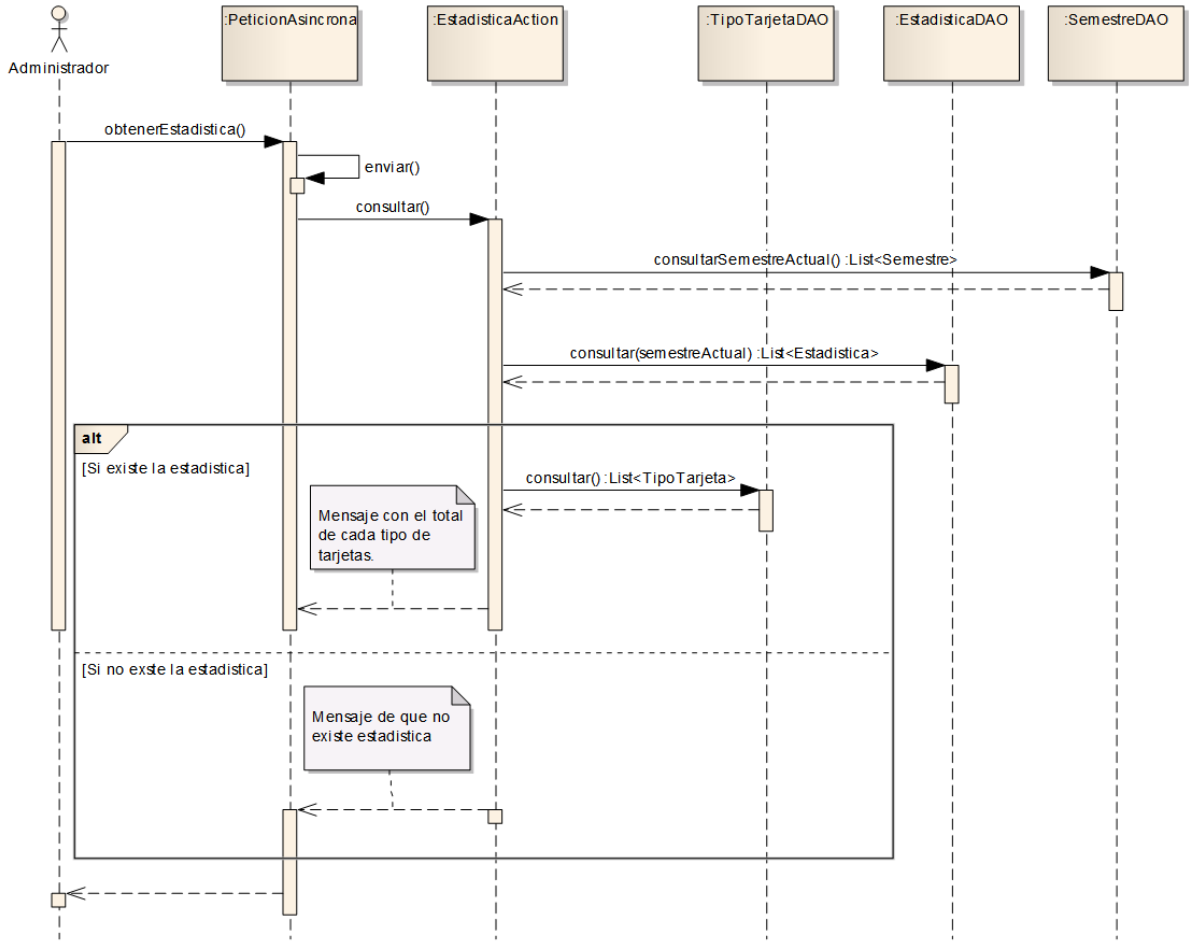


Figura 5.15 Diagrama de secuencia para Solicitar Estadistica

5.2 Modelado de la base de datos

5.2.1 Modelado con diagrama entidad-relación

Se realizo el modelado del diagrama entidad-relación, vease figura 5.16. Se partio de un primer modelado donde se consideraron las entidades Alumno, Tarjeta y Tipo_Tarjeta, en este modelado se determinaron las relaciones entre Alumno y Tarjeta; y Tarjeta y Tipo_Tarjeta, la multiplicidad hallada para las relaciones fueron muchos a muchos y muchos a uno, respectivamente. La relacion muchos a muchos se rompio en dos relaciones de muchos a uno, estas relaciones van de la nueva entidad Prestamo a las entidades Alumno y Tarjeta.

Posteriormente, se agregaron a este modelo otras entidades: Devolucion, Semestre y Reparacion, con sus respectivas relaciones con las entidades existentes. Las relaciones encontradas entre Alumno y Semestre, Tarjeta y Reparacion, y Prestamo y Devolucion, la multiplicidad hallada para las relaciones fueron muchos a muchos, uno a muchos y uno a uno, respectivamente. Tambien, se rompio la relacion entre las entidades Alumno y Semestre, teniendo una nueva entidad llamada Inscripcion.

Y por ultimo se agrego otra entidad llamada Estadistica, con su respectiva relación con la entidad Semestre, la multiplicidad descubierta para esta relación fue uno a uno.

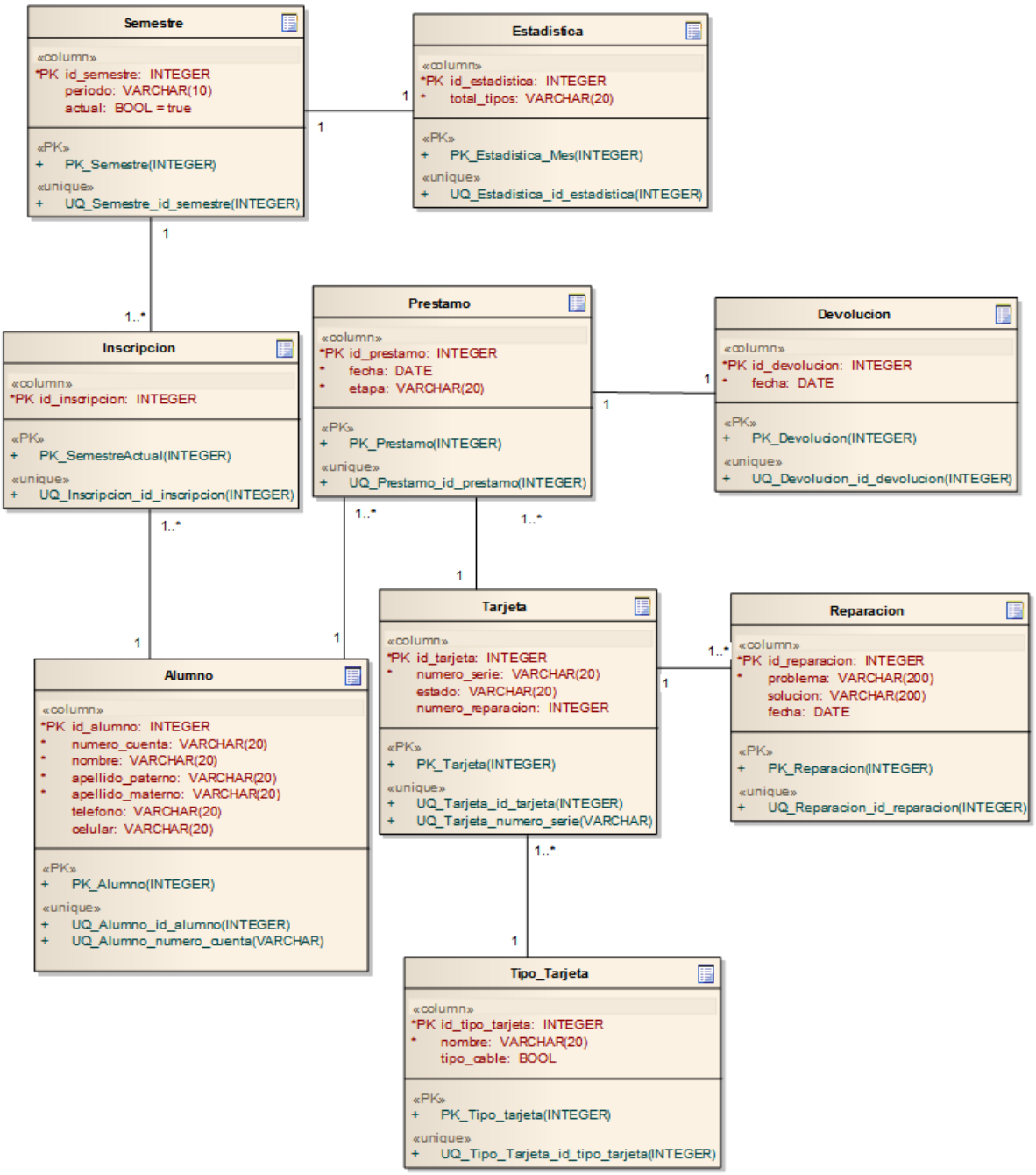


Figura 5.16 Modelado del diagrama entidad-relación

5.2.2 Modelado relacional

La figura 5.17 representa el modelo entidad-relacion en tablas, este modelo relacional traduce cada entidad a una tabla. Los atributos, las llaves primarias y las relaciones

de una entidad son en este modelo columnas, llaves primarias (pk) y llaves foraneas (fk), respectivamente.

Las tablas obtenidas en este modelo relacional son semestre, incripcion, alumno, estadistica, tipo_tarjeta, tarjeta, prestamo, reparacion y devolucion.

Para determinar en donde se debe ubicar en las tablas las llaves foraneas, se parte de conocer cuál es la entidad fuerte y debil. La llave foranea se va colocar en la tabla que corresponda a la entidad debil.

La entidad Prestamo es la entidad debil porque depende de las entidades Alumno y Tarjeta, es decir, si un alumno o una tarjeta no existen, no se puede realizar ningún prestamo. La entidad Prestamo en relación con la entidad Devolucion, es la fuerte debido a que debe existir el prestamo para poder realizar la devolucion.

Todas las tablas del modelo relacional se les aplico la normalizacion, es decir, las tres formas normales. Se verifico que las columnas dependan unicamente de la llave primaria.

En el caso de la tabla tarjeta, las columnas numero_serie, estado, numero_reparacion dependen del llave primaria, sólo van a tener un único valor para una determinada llave primaria. Son el numero de serie, estado (cuyo valor puede tomar “prestada”, “no prestada” y “en reparacion”), numero de reparación de la tarjeta.

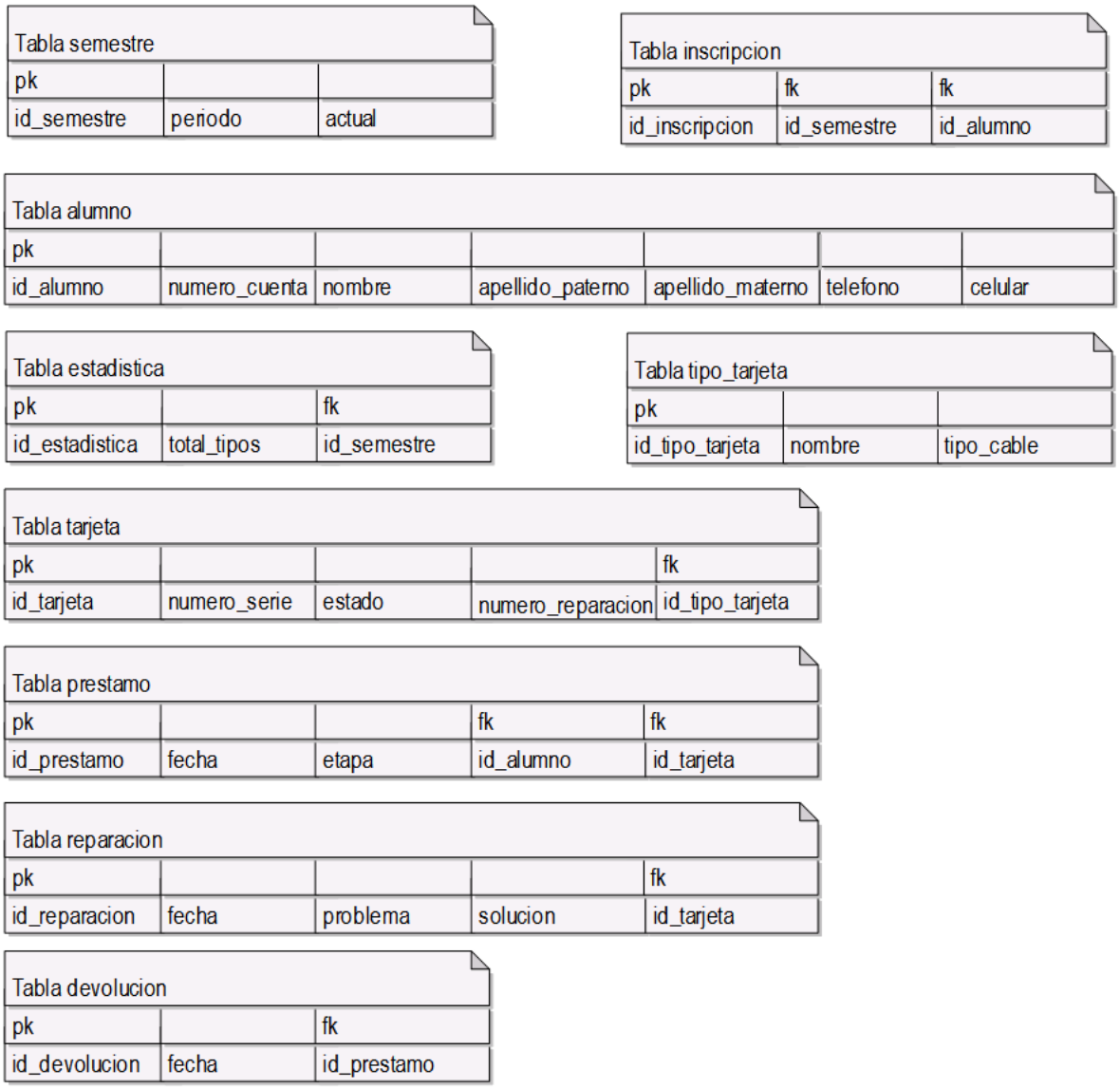


Figura 5.17 Modelo entidad-relacion en tablas.

Capítulo VI

Construcción del sistema de control de tarjetas

El sistema realizado, véase la figura 6.1, se dividió en varios conjuntos de componentes. Cada conjunto se diseñó con una responsabilidad específica, que al conjuntarlos forman el sistema funcional. En los siguientes apartados se describe cada uno de ellos; se explica primero el dominio del sistema de control de tarjetas, a continuación se revisan las clases que controlan las peticiones HTTP y por último, se exponen las ventanas generadas como respuesta a estas peticiones.

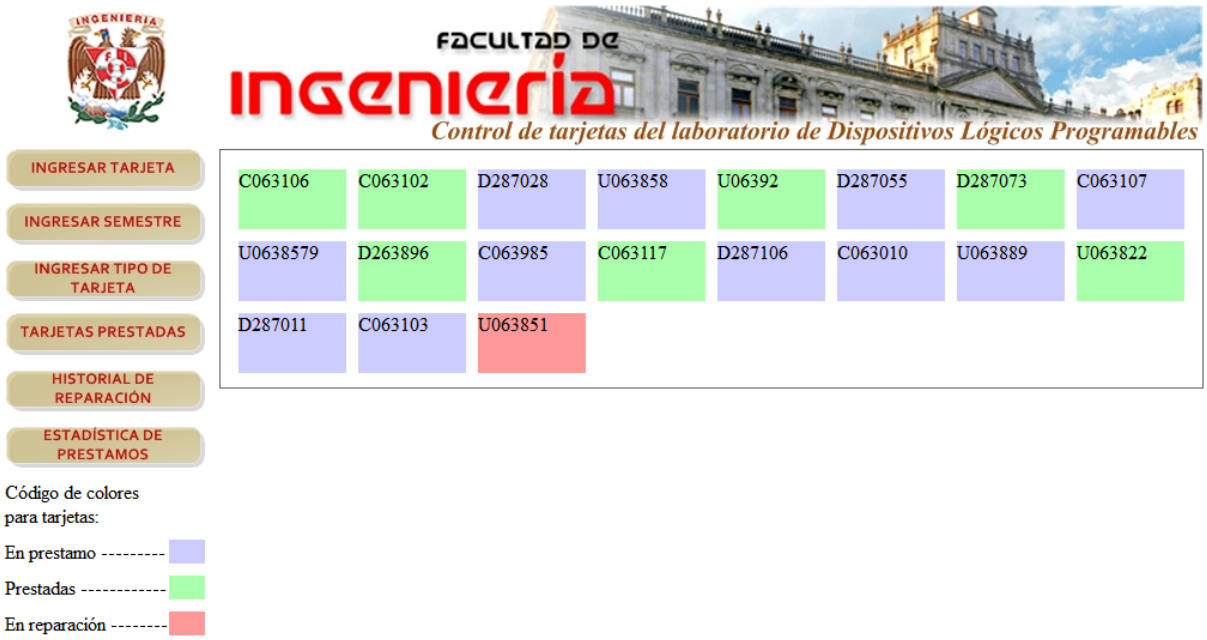


Figura 6.1 Sistema de control de tarjetas.

6.1. Construcción del modelo utilizando Hibernate

Las variables de instancia de las clases POJO representan la información que se almacena en la base de datos. Para lograr este objetivo se utilizó Hibernate.

Lo realizado utilizando Hibernate en el sistema de control de tarjetas es lo siguiente:

1. Instalación del API de Hibernate

2. Configuración de Hibernate.

3. Archivos de mapeo.

4. Creación de una sesión.

5. Guardado de datos con Hibernate.

6. Actualización de datos con Hibernate.

7. Recuperación de datos con Hibernate.

8. Automatización para generar la base de datos.

6.1.1 Instalación del API de Hibernate

La aplicación del sistema de control de tarjetas, se inició a partir de darle un nuevo nombre al archivo struts-blank-version.war. Además, a este nuevo proyecto se le agregaron las librerías .jar de Hibernate que a continuación se presentan:

- hibernate3.jar (librería completa de Hibernate, es la biblioteca principal).
- antlr-version.jar
- commons-collections-version.jar
- dom4j-version.jar
- javassist-version.GA.jar
- jta-version.jar
- slf4j-api-versión.jar
- cglib-version.jar
- servlet-api.jar
- slf4j.jar

6.1.2 Configuración de Hibernate

En el archivo hibernate.cfg.xml (figura 6.2) de la aplicación del sistema de control de tarjetas se configuran un conjunto de propiedades[1] como las siguientes:

1. connection.driver_class: La clase driver JDBC

2. connection.url: El URL JDBC de la instancia de la base de datos.

3. connection.username: El usuario de la base de datos.

4. connection.password: La contraseña del usuario de la base de datos.

5. hibernate.dialect: El dialecto SQL permite que se genere un SQL optimizado para una base de datos relacional en particular en este caso MySQL. Cada base de datos relacional soporta un diferente conjunto y uso ligeramente de SQL.

6. hibernate.show_sql: Muestra el SQL generado.

[1] Para obtener más información sobre la configuración de Hibernate se puede visitar la siguiente liga: http://docs.jboss.org/hibernate/core-/3.3/reference/en/html/session-configuration.html 3.3/reference/en/html/session-configuration.html

```
<hibernate-configuration>
   <session-factory>

        <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="connection.url">jdbc:mysql://localhost/tarjetas</property>
        <property name="connection.username">usuario</property>
        <property name="connection.password">contraseña</property>
        <property name="connection.pool_size">50</property>
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
         <property name="show_sql">false</property>

        <mapping resource="tarjetas/control/pojo/hbm/Semestre.hbm.xml"/>
        <mapping resource="tarjetas/control/pojo/hbm/Inscripcion.hbm.xml"/>
        <mapping resource="tarjetas/control/pojo/hbm/Alumno.hbm.xml"/>
        <mapping resource="tarjetas/control/pojo/hbm/Devolucion.hbm.xml"/>
        <mapping resource="tarjetas/control/pojo/hbm/Prestamo.hbm.xml"/>
        <mapping resource="tarjetas/control/pojo/hbm/TipoTarjeta.hbm.xml"/>
        <mapping resource="tarjetas/control/pojo/hbm/Tarjeta.hbm.xml"/>
        <mapping resource="tarjetas/control/pojo/hbm/Reparacion.hbm.xml"/>
        <mapping resource="tarjetas/control/pojo/hbm/Estadistica.hbm.xml"/>

   </session-factory>
</hibernate-configuration>
```

Figura 6.2 Archivo de configuración hibernate.cfg.xml

Además, se agrega a este archivo los recursos de mapeo. Cada archivo de mapeo que se declaró representan los tipos de objetos que se harán persistir.

6.1.3 Archivos de mapeo

En la figura 6.3, se observa el mapeo que existe entre un atributo de la clase PrestamoImpl y una columna de una tabla prestamo. El mapeo mencionado es la parte en XML, en este caso es una parte del contenido del archivo Prestamo.hbm.xml.

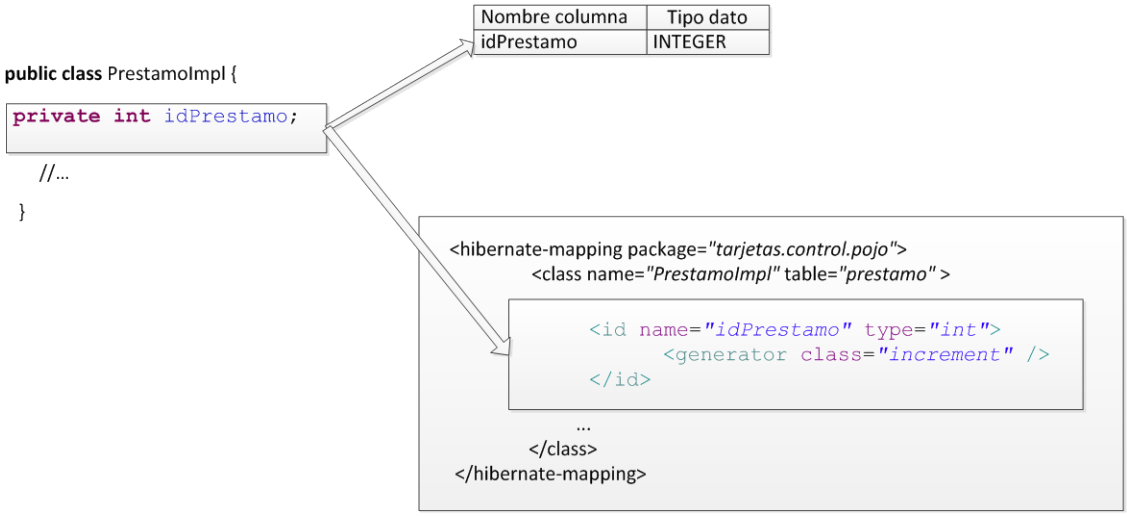


Figura 6.3 Mapeo de un atributo a una columna

Otra característica que se puede observar, es que los tipos de datos en Java y datos en una base de datos relacional no son iguales. En la figura 6.3 el tipo de dato utilizado en Java es int y el tipo de dato utilizado en el manejador de la base de datos es INTEGER. Continuando con la observación, en la parte en XML el dato int en Java se mapeo con el tipo de dato int de Hibernate.

En cualquier archivo de mapeo, se tienen que mapear los tipos de datos Java a los correspondientes tipos de datos Hibernate. A continuación se presentan en la tabla 6.1, los tipos de datos de Hibernate equivalentes para los tipos de datos primitivos, Wrapper Class y String de Java.

Tipo de dato de Hibernate	**Equivalencia a tipo de datos Java**
int	int, java.lang.Integer
short	short, java.lang.Short
long	long, java.lang.Long
float	float, java.lang.Float
double	double, java.lang.Double
character	char, java.lang.Character
byte	byte, java.lang.Byte
boolean, yes_no, true_false	boolean, java.lang.Boolean
string	java.lang.String

Tabla 6.1 Equivalencias entre tipos de datos de Hibernate y Java

Los archivos de configuración de este proyecto son los siguientes: Alumno.hbm.xml, Devolucion.hbm.xml, Inscripcion-.hbm.xml, Prestamo.hbm.xml, Semestre.hbm.xml, Tarjeta-.hbm.xml, TarjetaReparacion.hbm.xml, TipoTarjeta.hbm.xml y Estadistica.hbm.xml. Cada uno mapea su respectiva clase POJO.

A continuación se muestran los archivos de mapeo para las clases AlumnoImpl, PrestamoImpl, TarjetaImpl (vease las imágenes 6.4, 6.5 y 6.6).

```
<hibernate-mapping package="tarjetas.control.pojo">
      <class name="AlumnoImpl" table="alumno" >

            <id name="idPersona" type="int" column="idAlumno" >
                  <generator class="increment" />
            </id>

            <property name="nombre" type="string" length="40"/>
            <property name="apellidoPaterno" type="string" length="40"/>
            <property name="apellidoMaterno" type="string" length="40"/>
             <property name="numeroCuenta" type="string" length="20" unique="true"/>
             <property name="telefono" type="string" length="20"/>
             <property name="celular" type="string" length="20"/>

      </class>
</hibernate-mapping>
```

Figura 6.4 Archivo de mapeo Alumno.hbm.xml

```
<hibernate-mapping package="tarjetas.control.pojo">
       <class name="PrestamoImpl" table="prestamo" >

              <id name="idPrestamo" type="int">
                     <generator class="increment" />
              </id>

              <property name="fecha" type="date"/>
              <many-to-one name="tarjeta" class="TarjetaImpl" column="idTarjeta"/>
              <many-to-one name="alumno" class="AlumnoImpl" column="idAlumno"/>
              <property name="etapa" type="string" length="20" />

       </class>
</hibernate-mapping>
```

Figura 6.5 Archivo de mapeo Prestamo.hbm.xml

```
<hibernate-mapping package="tarjetas.control.pojo">
      <class name="TarjetaImpl" table="tarjeta" >

            <id name="idTarjeta" type="short">
                  <generator class="increment" />
            </id>

            <property name="numeroSerie" type="string" length="20" unique="true" />
            <property name="estado" type="string" length="20" />
            <property name="numeroReparacion" type="int"/>
            <many-to-one name="tipoTarjeta" class="TipoTarjetaImpl" column="idTipoTarjeta"/>

      </class>
</hibernate-mapping>
```

Figura 6.6 Archivo de mapeo Tarjeta.hbm.xml

En éstos archivos de mapeo, se pueden apreciar los siguientes elementos:

1. El elemento <hibernate-mapping />
   Es la raíz de cualquier archivo de mapeo. Éste elemento tiene varios atributos de configuración, en este caso se utilizo el atributo mostrado en la tabla 6.2.

Atributo	Descripción
package	Específica un paquete el cual se asumirá que es el paquete de las clases que no se especifique explícitamente.

Tabla 6.2 Atributos de la etiqueta <hibernate-mapping />

2. El elemento <class />

   El elemento <class /> es el elemento hijo de <hibernate-mapping /> se configuro con los siguientes atributos mostrados en la tabla 6.3.

Atributo	Descripción
name	Indica el nombre del objeto que se va a persistir.
table	Indica el nombre de la tabla asociada al objeto. Si no se especifica el nombre asignado al atributo name será asignado como nombre de la tabla.

Tabla 6.3 Atributos de la etiqueta <class />

3. El elemento <id />

Cuando se mapea una clase Java a una tabla en la base de datos, un ID es requerido. Las clases mapeadas deben declarar la columna que sea llave primaria en la tabla de la base de datos. El elemento <id /> permite especificar información sobre esta columna.

El elemento <id /> se configuro con los siguientes atributos mostrados en la tabla 6.4.

Atributo	**Descripción**
name	Específica el nombre del atributo del objeto que representa la llave primaria. Si no se especifica, Hibernate asume que el objeto no tiene un atributo directo que represente la llave primaria.
type	Es un nombre, el cual indica el tipo de dato de Hibernate y es opcional. Si no se especifica, Hibernate determina el tipo.
column	Es el nombre de la columna de la llave primaria y es opcional. Si no se especifica, entonces Hibernate usa el valor dado al atributo name.
length	Es el tamaño de la columna a ser usado y es opcional.

Tabla 6.4 Atributos de la etiqueta <id />

4. El elemento <generator />

El elemento <generator /> es el hijo del elemento <id>.

Cuando se inserta un nuevo renglón en una tabla de una base de datos, la columna ID debe ser poblada con un único valor en el orden del identificador único que persiste el objeto. El elemento <generator /> ayuda a definir cómo se genera una nueva llave primaria para una nueva instancia de una clase.

En éste elemento se utilizó el atributo: class, el cual se configuro con la clase increment (véase la tabla 6.5).

Clase	Descripción
increment	Genera el valor de la llave primaria agregándole 1 al último valor de la llave primaria. Esta clase genera identificadores de tipo: short, int o long.

Tabla 6.5 Atributos de la etiqueta <generator />

5. El elemento <property />

En el elemento <property /> se configuraron los siguientes atributos mostrados en la tabla 6.6.

Atributo	Descripción
name	Es el nombre de la propiedad en una clase POJO.
column	Es el nombre de la columna de la tabla en la base de datos donde el atributo debe ser salvado y es opcional. Si el atributo no se especifica, entonces Hibernate usa el valor dado al atributo name.
type	Es un nombre, el cual indica los tipos de datos de Hibernate. Este atributo es opcional.
unique	Específica si los valores duplicados son permitidos para una columna y es opcional. Por defecto, el valor de este atributo es false.
not-null	Específica si la columna es permitida a contener valores NULL y es opcional. Por defecto, el valor de este atributo es false.
length	Es el tamaño de la columna a ser usado y es opcional.

Tabla 6.6 Atributos de la etiqueta <property />

6. El elemento <many-to-one />

El elemento <many-to-one /> representa la relación en la cual múltiples instancias de una clase pueden referir a una simple instancia de una clase.

Con el elemento <many-to-one /> se configuraron los atributos mostrados en la tabla 6.7.

Atributo	Descripción
name	Es el nombre de la asociación Java con la relación.
column	Es el nombre de la columna llave foránea y es opcional. Por defecto, el atributo toma el valor del atributo name.
class	Es el nombre de la clase asociada y es opcional.

Tabla 6.7 Atributos de la etiqueta <many-to-one />

6.1.4 Creación de una sesión

En la siguiente clase (figura 6.7) se implementó la creación de una sesión.

```
public class HibernateUtil {

	private static final ThreadLocal session = new ThreadLocal();
	private static final SessionFactory sessionFactory = new Configuration()
				.configure().buildSessionFactory();

	private HibernateUtil() {
	}

	public static Session getSession() {
		Session session = (Session)HibernateUtil.session.get();
		if( session == null ) {
			session = sessionFactory.openSession();
			HibernateUtil.session.set(session);
		}
		return session;
	}
}
```

Figura 6.7 Código de la creación de una sesión

Un objeto de tipo Session tiene diversos métodos que se pueden utilizar de acuerdo a su API (véase http://docs.jboss-.org/hibernate/annotations/3.5/api/org/hibernate/Session.html).

6.1.5 Guardado de datos en Hibernate

El objeto Session contiene un método llamado save(), a continuación se muestra un fragmento extraído de la API en la figura 6.8:

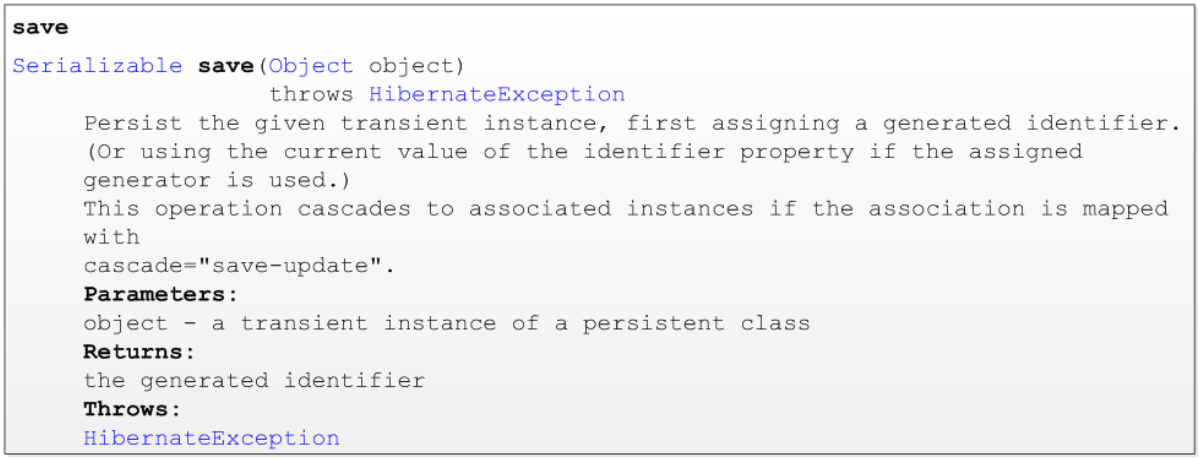


Figura 6.8 El método save() en la API de Hibernate

Con este fragmento se puede obtener la siguiente información:

1. A este método se le tiene que pasar como parámetro cualquier objeto que se quiera persistir, éste objeto persistido debe haber sido previamente mapeado en un archivo de mapeo.

2. Retorna el identificador generado, esto de acuerdo al tipo de identificador que se haya declarado en el archivo de mapeo.

3. Se debe tratar la excepción HibernateException.

En el diagrama de secuencia de Ingresar Datos de Nueva Tarjeta se especifica que métodos se tienen que construir en las clases e interfaces[2] DAO y Transaccion. A continuación, se muestra la implementación del método guardar() (obsérvese la figura 6.9) en la clase TarjetaDAOImpl:

```
public Tarjeta guardar(String numeroSerie, String tipoCable, TipoTarjeta tipoTarjeta)
               throws Exception {
      Tarjeta tarjeta = null;

        try {
                  tarjeta = new TarjetaImpl();
                  tarjeta.setNumeroSerie(numeroSerie);
                  tarjeta.seTipoCable(tipoCable);
                  tarjeta.setEstado("no prestada");
                  tarjeta.setTipoTarjeta(tipoTarjeta);
                  tarjeta.setNumeroReparacion(0);

                  // Se persiste una nueva tarjeta.
                  HibernateUtil.getSession().save(tarjeta);

              return tarjeta;
      } catch(HibernateException e) {
            debug.log(Level.SEVERE, "No pudo guardarse la Tarjeta", e);
            throw new Exception("No pudo guardarse la Tarjeta", e);
      }
}
```

Figura 6.9 El método guardar() de la clase TarjetaDAOImpl

Si éste método no lanza una excepción de tipo HibernateException, el método guardar() implementado en la clase TarjetaImplTransaccion realizará un commit, en caso contrario realizará un rollback concluyéndose la transacción en cualquiera de los dos casos. En seguida, se muestra éste código en la figura 6.10:

[2] Una técnica muy común usada para reducir el acoplamiento es ocultar los detalles de implementación detrás de interfaces de forma que la clase de implementación real pueda intercambiarse sin impactar la clase de cliente.

```
public Tarjeta guardar(String numeroSerie, String tipoCable, TipoTarjeta tipoTarjeta)
                        throws Exception {

      TarjetaDAO tarjetaDAO = new TarjetaDAOImpl();
      Transaction tx = HibernateUtil.getSession().beginTransaction();

      Tarjeta tarjeta = null;

      try {
            tarjeta = tarjetaDAO.guardar(numeroSerie, tipoCable, tipoTarjeta);
            tx.commit();
            return tarjeta;
      } catch(HibernateException e) {
            tx.rollback();
            debug.log(Level.SEVERE, "No se pudo hacer la transacción", e);
            throw new Exception("No se pudo hacer la transacción", e);
      } catch(Exception e) {
            tx.rollback();
            debug.log(Level.SEVERE, "No se pudo hacer la transacción", e);
            throw new Exception("No se pudo hacer la transacción", e);
      }
}
```

Figura 6.10 El método guardar() en la clase TarjetaImplTransaccion

6.1.6 Actualización de datos en Hibernate

El objeto Session contiene un método llamado update(), a continuación se muestra otro fragmento extraído de la API en la figura 6.11:

update

```
void update(Object object)
            throws HibernateException
```

Update the persistent instance with the identifier of the given detached instance. If there is a persistent instance with the same identifier, an exception is thrown. This operation cascades to associated instances if the association is mapped with cascade="save-update".

Parameters:

object - a detached instance containing updated state

Throws:

HibernateException

6.11 El método update() en la API de Hibernate

Con este fragmento se puede obtener la siguiente información:

1. A este método se le tiene que pasar como parámetro cualquier objeto que se quiera actualizar su persistencia.

2. Se debe tratar la excepción HibernateException.

Para el diagrama de secuencia de Prestar Tarjeta, se especificó que se debía implementar el método actualizar (obsérvese la figura 6.12) en la clase TarjetaDAOImpl:

```
public Tarjeta actualizar(Tarjeta tarjeta, String estado) throws Exception {

      try {
            // Se modifica el estado de la tarjeta.
            tarjeta.setEstado(estado);

            // Se persiste la tarjeta.
            HibernateUtil.getSession().update(tarjeta);

            return tarjeta;
      } catch(HibernateException e) {
            debug.log(Level.SEVERE, "No pudo actualizarse la Tarjeta", e);
            throw new Exception("No pudo actualizarse la Tarjeta", e);
      }
}
```

Figura 6.12 El método actualizar() de la clase TarjetaDAOImpl

Si éste método no lanza una excepción de tipo HibernateException, el método actualizar() implementado en la clase TarjetaImplTransaccion realizará un commit, en caso contrario realizará un rollback. En seguida, se muestra éste código en la figura 6.13:

```
public Tarjeta actualizar(Tarjeta tarjeta, String estado) throws Exception {

	TarjetaDAO tarjetaDAO = new TarjetaDAOImpl();
	Transaction tx = HibernateUtil.getSession().beginTransaction();

	try {
		tarjeta = tarjetaDAO.actualizar(tarjeta, estado);
		tx.commit();
		return tarjeta;
	} catch(HibernateException e) {
		tx.rollback();
		debug.log(Level.SEVERE, "No se pudo hacer la transacción", e);
		throw new Exception("No se pudo hacer la transacción", e);
	} catch(Exception e) {
		tx.rollback();
		debug.log(Level.SEVERE, "No se pudo hacer la transacción", e);
		throw new Exception("No se pudo hacer la transacción", e);
	}
}
```

Figura 6.13 El método actualizar() de la clase TarjetaImplTransaccion

6.1.7 Recuperación de datos con Hibernate

Se recuperaron los datos con Hibernate utilizando la API de Criteria y la API de Query.

La manera más fácil de recuperar datos con Hibernate es usando el API de Criteria. El objeto Session contiene un método llamado createCriteria(), la manera simple de usar Criteria es:

```
Criteria criteria = sesión.createCriteria(NombrePojo.class);
List resultados = criteria.list();
```

Para desplegar el total de tarjetas existentes, se implementó el método consultar() (obsérvese la figura 6.14) en la clase TarjetaDAOImpl:

```
public List<Tarjeta> consultar()   {
      try {
            // Se consultan todas las tarjetas.
            Criteria criteria = HibernateUtil.getSession().createCriteria
                              (TarjetaImpl.class);
            List<Tarjeta> lista = criteria.list();
            return lista;
      } catch(Exception e) {
            debug.log(Level.SEVERE, "Problema al realizar la consulta de
                                    los registros de Tarjeta", e);
            return null;
      }
}
```

Figura 6.14 El método consultar() en la clase TarjetaDAOImpl

En éste código, se puede observar que se utilizó el método createCriteria() de la manera más simple. En el tipo de retorno se obtiene la lista de todos los objetos de tipo Tarjeta que se hayan persistido.

El objeto Session, también contiene un método llamado createQuery(), el cual se le pasa como parámetro algo parecido al SQL, es decir, el HQL. La manera más simple de usar éste método es:

```
Query query = sesión.createQuery("from ObjetoPersistido");
List results = query.list();
```

Para el diagrama de secuencia de Devolver Tarjeta, se especificó que se debía implementar el método consultar (obsérvese la figura 6.15) en la clase PrestamoDAOImpl:

```
public List<Prestamo> consultar(Alumno alumno, Tarjeta tarjeta, String etapa) {

      try {
            // Se consulta el prestamo por alumno and tarjeta and etapa.
            Query query = HibernateUtil.getSession()
                        .createQuery("from PrestamoImpl where alumno =
                                    :alumno and tarjeta = :tarjeta and etapa = :etapa");
            query.setParameter("alumno", alumno);
            query.setParameter("tarjeta", tarjeta);
            query.setString("etapa", etapa);
            List<Prestamo> lista = query.list();
            return lista;
      } catch(HibernateException e) {
            debug.log(Level.SEVERE, "Problema al realizar la consulta por ID de los
                                    registros de Prestamo", e);
            return null;
      }
}
```

Figura 6.15 El método consultar() en la clase PrestamoDAOImpl

En éste código, se puede observar que se utilizó el método createQuery(), el parámetro que se le pasa es una cadena escrita en el lenguaje HQL “from PrestamoImpl where alumno = :alumno and tarjeta = :tarjeta and etapa = :etapa”.

En el HQL se realiza una consulta de todos los objetos Prestamo, en donde los atributos alumno, tarjeta y etapa de éstos objetos coincidan con los parámetros (alumno, tarjeta y etapa) pasados al método consultar(). El tipo de retorno es una lista con todos los objetos de tipo Prestamo que resulten de ésta consulta y el tamaño de la lista de retorno dependerá de la cantidad de coincidencias.

6.1.8 Automatización para generar la base de datos

Uno de los puntos críticos de cualquier proyecto de software es la herramienta de construcción del mismo. Dos de las herramientas de construcción más famosas son Maven[3] y Ant[4], ambas proyectos son de código abierto de la fundación Apache.

Con Ant se realizó la automatización para generar la base de datos, en la figura 6.16 se muestra el fragmento del script para realizar esto. Lo que se hace éste fragmento es leer el

[3] http://maven.apache.org/
[4] http://ant.apache.org/

archivo de configuración de Hibernate y generar la base de datos cada vez que se ejecuta.

```
<!--
Definicion de la tarea: generar base
-->
<target name="generar_base">
      <java classname="org.hibernate.tool.hbm2ddl.SchemaExport"
                        classpathref="classpath">
            <arg line="--config=hibernate.cfg.xml --output=${name}.sql"/>
      </java>
</target>
```

Figura 6.16 Fragmento de script para construir la base de datos

Una de las propiedades del archivo de configuración que se utilizó en el script es hbm2ddl. Si se usa en el archivo hibernate.cfg.xml sería algo así:

```
<property name=”hdm2ddl.auto”>create</property>
```

Lo que hace esta propiedad es borrar y generar el esquema de la base de datos definida a través de los archivos de mapeo.

En seguida, se muestra la salida cuando se ejecuta el script:

```
Buildfile: C:\Users\tesis\Desktop\TARJETAS\tarjetas\AntAutomatizacion.xml
generar_base:
     [java] SLF4J: The requested version 1.5.11 by your slf4j
            binding is not compatible with [1.5.5, 1.5.6, 1.5.7, 1.5.8]
     [java] SLF4J: See http://www.slf4j.org/codes.html#version_mismatch for further details.
     [java] 297 [main] INFO org.hibernate.cfg.Environment -
            Hibernate 3.3.2.GA
     [java] 312 [main] INFO org.hibernate.cfg.Environment -
            hibernate.properties not found
     [java] 359 [main] INFO org.hibernate.cfg.Environment -
            Bytecode provider name : javassist
     [java] 453 [main] INFO org.hibernate.cfg.Environment -
            using JDK 1.4 java.sql.Timestamp handling
     [java] 1202 [main] INFO org.hibernate.cfg.Configuration -
            configuring from resource: hibernate.cfg.xml
     [java] 1202 [main] INFO org.hibernate.cfg.Configuration -
            Configuration resource: hibernate.cfg.xml
     [java] 1716 [main] INFO org.hibernate.cfg.Configuration -
            Reading mappings from resource : tarjetas/control/pojo/hbm/Semestre.hbm.xml
     [java] 2028 [main] INFO org.hibernate.cfg.HbmBinder -
            Mapping class: tarjetas.control.pojo.SemestreImpl -> semestre
     [java] 2075 [main] INFO org.hibernate.cfg.Configuration -
            Reading mappings from resource : tarjetas/control/pojo/hbm/Inscripcion.hbm.xml
     [java] 2153 [main] INFO org.hibernate.cfg.HbmBinder -
            Mapping class: tarjetas.control.pojo.InscripcionImpl -> inscripcion
     [java] 2418 [main] INFO org.hibernate.cfg.Configuration -
            Reading mappings from resource : tarjetas/control/pojo/hbm/Alumno.hbm.xml
     [java] 2450 [main] INFO org.hibernate.cfg.HbmBinder -
            Mapping class: tarjetas.control.pojo.AlumnoImpl -> alumno
     [java] 2465 [main] INFO org.hibernate.cfg.Configuration -
            Reading mappings from resource : tarjetas/control/pojo/hbm/Devolucion.hbm.xml
     [java] 2496 [main] INFO org.hibernate.cfg.HbmBinder -
            Mapping class: tarjetas.control.pojo.DevolucionImpl -> devolucion
     [java] 2496 [main] INFO org.hibernate.cfg.Configuration -
            Reading mappings from resource : tarjetas/control/pojo/hbm/Prestamo.hbm.xml
     [java] 2528 [main] INFO org.hibernate.cfg.HbmBinder -
            Mapping class: tarjetas.control.pojo.PrestamoImpl -> prestamo
     [java] 2528 [main] INFO org.hibernate.cfg.Configuration -
            Reading mappings from resource : tarjetas/control/pojo/hbm/TipoTarjeta.hbm.xml
     [java] 2543 [main] INFO org.hibernate.cfg.HbmBinder -
            Mapping class: tarjetas.control.pojo.TipoTarjetaImpl -> tipo_tarjeta
     [java] 2543 [main] INFO org.hibernate.cfg.Configuration -
            Reading mappings from resource : tarjetas/control/pojo/hbm/Tarjeta.hbm.xml
     [java] 2590 [main] INFO org.hibernate.cfg.HbmBinder -
            Mapping class: tarjetas.control.pojo.TarjetaImpl -> tarjeta
     [java] 2606 [main] INFO org.hibernate.cfg.Configuration -
            Reading mappings from resource : tarjetas/control/pojo/hbm/Reparacion.hbm.xml
     [java] 2637 [main] INFO org.hibernate.cfg.HbmBinder -
            Mapping class: tarjetas.control.pojo.ReparacionImpl -> reparacion
     [java] 2637 [main] INFO org.hibernate.cfg.Configuration -
            Reading mappings from resource : tarjetas/control/pojo/hbm/Estadistica.hbm.xml
     [java] 2668 [main] INFO org.hibernate.cfg.HbmBinder -
            Mapping class: tarjetas.control.pojo.EstadisticaImpl -> estadistica
     [java] 2668 [main] INFO org.hibernate.cfg.Configuration -
            Configured SessionFactory: null
     [java] 3152 [main] INFO org.hibernate.dialect.Dialect -
            Using dialect: org.hibernate.dialect.MySQL5Dialect
     [java] 3604 [main] INFO org.hibernate.tool.hbm2ddl.SchemaExport -
            Running hbm2ddl schema export
     [java] 3666 [main] INFO org.hibernate.tool.hbm2ddl.SchemaExport -
            writing generated schema to file: esteticas.sql
     [java] 3682 [main] INFO org.hibernate.tool.hbm2ddl.SchemaExport -
            exporting generated schema to database
     [java] 3698 [main] INFO org.hibernate.connection.DriverManagerConnectionProvider -
            Using Hibernate built-in connection pool (not for production use!)
     [java] 3698 [main] INFO org.hibernate.connection.DriverManagerConnectionProvider -
            Hibernate connection pool size: 50
     [java] 3698 [main] INFO org.hibernate.connection.DriverManagerConnectionProvider -
            autocommit mode: false
     [java] 3744 [main] INFO org.hibernate.connection.DriverManagerConnectionProvider -
            using driver: com.mysql.jdbc.Driver at URL: jdbc:mysql://localhost/tarjetas
     [java] 3744 [main] INFO org.hibernate.connection.DriverManagerConnectionProvider -
            connection properties: {user=tarjetas, password=****}
```

```
[java] alter table devolucion drop foreign key FK7C391548D124FEEA
     [java] alter table estadistica drop foreign key FK6A33832612EAEE8
     [java] alter table inscripcion drop foreign key FK183A418986B43B74
     [java] alter table inscripcion drop foreign key FK183A418912EAEE8
     [java] alter table prestamo drop foreign key FKB3EE3EFF86B43B74
     [java] alter table prestamo drop foreign key FKB3EE3EFFF10B0C2E
     [java] alter table reparacion drop foreign key FK8D8237D8F10B0C2E
     [java] alter table tarjeta drop foreign key FKA4511E0D8A0138A6
     [java] drop table if exists alumno
     [java] drop table if exists devolucion
     [java] drop table if exists estadistica
     [java] drop table if exists inscripcion
     [java] drop table if exists prestamo
     [java] drop table if exists reparacion
     [java] drop table if exists semestre
     [java] drop table if exists tarjeta
     [java] drop table if exists tipo_tarjeta
     [java] create table alumno (idAlumno integer not null, nombre varchar(40),
            apellidoPaterno varchar(40), apellidoMaterno varchar(40),
            numeroCuenta varchar(20) unique, telefono varchar(20), celular varchar(20),
            primary key (idAlumno))
     [java] create table devolucion (idDevolucion integer not null, fecha date,
            idPrestamo integer, primary key (idDevolucion))
     [java] create table estadistica (idEstadistica integer not null, idSemestre integer,
            totalTipos varchar(255), primary key (idEstadistica))
     [java] create table inscripcion (idInscripcion integer not null, idSemestre integer,
            idAlumno integer, primary key (idInscripcion))
     [java] create table prestamo (idPrestamo integer not null, fecha date,
            idTarjeta  smallint, idAlumno integer, etapa varchar(20),
            primary key (idPrestamo))
     [java] create table reparacion (idReparacion integer not null, fecha date,
            idTarjeta smallint, problema varchar(200), solucion varchar(200),
            primary key (idReparacion))
     [java] create table semestre (idSemestre integer not null, periodo varchar(20) unique,
            actual char(1), primary key (idSemestre))
     [java] create table tarjeta (idTarjeta smallint not null, numeroSerie varchar(20) unique,
            estado varchar(20), numeroReparacion integer, idTipoTarjeta smallint,
            primary key (idTarjeta))
     [java] create table tipo_tarjeta (idTipoTarjeta smallint not null,
            nombre varchar(50) unique, tipoCable varchar(10), primary key (idTipoTarjeta))
     [java] alter table devolucion add index FK7C391548D124FEEA (idPrestamo), add constraint
            FK7C391548D124FEEA foreign key (idPrestamo) references prestamo (idPrestamo)
     [java] alter table estadistica add index FK6A33832612EAEE8 (idSemestre), add constraint
            FK6A33832612EAEE8 foreign key (idSemestre) references semestre (idSemestre)
     [java] alter table inscripcion add index FK183A418986B43B74 (idAlumno), add constraint
            FK183A418986B43B74 foreign key (idAlumno) references alumno (idAlumno)
     [java] alter table inscripcion add index FK183A418912EAEE8 (idSemestre), add constraint
            FK183A418912EAEE8 foreign key (idSemestre) references semestre (idSemestre)
     [java] alter table prestamo add index FKB3EE3EFF86B43B74 (idAlumno), add constraint
            FKB3EE3EFF86B43B74 foreign key (idAlumno) references alumno (idAlumno)
     [java] alter table prestamo add index FKB3EE3EFFF10B0C2E (idTarjeta), add constraint
            FKB3EE3EFFF10B0C2E foreign key (idTarjeta) references tarjeta (idTarjeta)
     [java] alter table reparacion add index FK8D8237D8F10B0C2E (idTarjeta), add constraint
            FK8D8237D8F10B0C2E foreign key (idTarjeta) references tarjeta (idTarjeta)
     [java] alter table tarjeta add index FKA4511E0D8A0138A6 (idTipoTarjeta), add constraint
            FKA4511E0D8A0138A6 foreign key (idTipoTarjeta) references
            tipo_tarjeta (idTipoTarjeta)
     [java] 12574 [main] INFO org.hibernate.tool.hbm2ddl.SchemaExport - schema export
            complete
     [java] 12605 [main] INFO org.hibernate.connection.DriverManagerConnectionProvider -
            cleaning up connection pool: jdbc:mysql://localhost/tarjetas
BUILD SUCCESSFUL
```

En esta salida se puede observar que se genera la base de datos tal como se realizó en el modelado de la base de datos.

6.2 Construcción con Struts

Lo realizado utilizando Struts en el sistema de control de tarjetas es lo siguiente:

1. Instalación del API de Struts.
2. El archivo descriptor web.xml.
3. Configuración de Struts.
4. Las clases Action.
5. El lenguaje OGNL en la vista principal.

6.2.1 Instalación del API de Struts

La instalación del API es sencilla, basta con copiar los archivos descargados en los directorios de una aplicación Web Java EE tradicional. Se puede crear un nuevo proyecto y copiar los archivos necesarios (librerías .jar) o bien utilizar una aplicación vacía de Struts, es decir, el archivo struts-blank-version.war.

El marco de trabajo Struts está compuesto por varias librerías .jar, que contienen todas las clases utilizadas:

1. commons-fileupload.jar (librería de gestión de la carga de archivos).
2. commons-logging.jar (librería de registros de mensajes).
3. commons-io-version.jar (librería de gestión de entradas/salidas).

4. freemarker-version.jar (librería utilizada para la presentación y el motor de plantillas).

5. ognl-version.jar (librería utilizada para la manipulación de objetos Java).

6. junit-version.jar (librería del marco de trabajo de gestión de las pruebas unitarias).

7. struts2-core-version.jar (librería completa de Struts, es la biblioteca principal).

8. xwork-version.jar (librería de XWork con las dependencias).

6.2.2 El archivo descriptor web.xml

El controlador que utiliza Struts debe estar declarado en el archivo descriptor de la aplicación, es decir, en el archivo web.xml. Este controlador se define en la clase org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter, a continuación se presenta en la figura 6.17 parte del contenido del archivo web.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_9" version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
         http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <display-name>Sistema de control de tarjetas</display-name>

    <filter>
        <filter-name>struts2</filter-name>
        <filter-class>org.apache.struts2.dispatcher.ng.filter
                      .StrutsPrepareAndExecuteFilter</filter-class>
    </filter>

    <filter-mapping>
        <filter-name>struts2</filter-name>
        <url-pattern>*.action</url-pattern>
    </filter-mapping>

</web-app>
```

Figura 6.17 Fragmento del archivo web.xml

En éste fragmento el elemento <filter-mapping /> permite administrar las relaciones entre un nombre y el controlador o una URL. Solamente los URL con el sufijo .action serán procesados por el controlador. El controlador no gestiona los recursos estáticos, como por ejemplo imágenes, archivos JavaScript u hojas de estilo CSS.

6.2.3 Configuración de Struts

La configuración de Struts se realizó en el archivo struts.xml y un archivo secundario de la aplicación del sistema de control de tarjetas (véanse las figuras 6.18 y 6.19).

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>

    <constant name="struts.devMode" value="true" />

    <package name="publico" namespace="/" extends="struts-default">

      <action name="inicio">
            <result>/inicio.jsp</result>
      </action>

    </package>

    <include file="tarjetas/control/action/struts2.xml"/>

</struts>
```

Figura 6.18 El archivo de configuración struts.xml

En éste archivo de configuración se puede observar que aquí se incluyen los archivos de configuración secundarios con la etiqueta <include />.

En el archivo secundario de configuración de Struts, se organizan las acciones en el paquete control. El espacio de nombres del URL al que se asignaron las acciones es /, es decir, la anatomía de la URL es la siguiente:

http://localhost:8080/control/NombreAccion.action

En donde NombreAccion puede ser el nombre de una acción definida en la figura 6.19.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>

  <package name="control" namespace="/" extends="struts-default">

    <action name="tarjetas" class="tarjetas.control.action.TarjetaAction"
                            method="consultarTodo">
    <result name="exito">/menu_tarjetas.jsp</result>
    </action>
    <action name="control" class="tarjetas.control.action.TarjetaAction"
                            method="consultarTodo">
    <result name="exito">/control_tarjetas.jsp</result>
    </action>
    <action name="prestarTarjeta1" class="tarjetas.control.action.PrestamoAction"
                                   method="prestar1">
    </action>
    <action name="prestarTarjeta2" class="tarjetas.control.action.PrestamoAction"
                                   method="prestar2">
    </action>
    <action name="guardarTarjeta" class="tarjetas.control.action.TarjetaAction"
                                   method="guardar">
    </action>
    <action name="guardarSemestre" class="tarjetas.control.action.SemestreAction"
                                    method="guardar">
    </action>
    <action name="guardarTipoTarjeta" class="tarjetas.control.action.TipoTarjetaAction"
                                      method="guardar">
    </action>
    <action name="guardarDevolucion" class="tarjetas.control.action.DevolucionAction"
                                      method="guardar">
    </action>
    <action name="guardarReparacion" class="tarjetas.control.action.ReparacionAction"
                                      method="guardar">
    </action>
    <action name="actualizarReparacion" class="tarjetas.control.action.ReparacionAction"
                                        method="actualizar">
    </action>
    <action name="consultarPrestamos" class="tarjetas.control.action.PrestamoAction"
                                      method="consultar">
    </action>
    <action name="consultarReparaciones" class="tarjetas.control.action.ReparacionAction"
                                         method="consultar">
    </action>
    <action name="consultarEstadistica" class="tarjetas.control.action.EstadisticaAction"
                                        method="consultar">
    </action>
  </package>

</struts>
```

Figura 6.19 El archivo secundario de configuración de Struts

6.2.4 Las clases Action

Todos los diagramas de secuencia, especifican que clases Action se debían implementar. Las clases Action implementadas son las siguientes: DevolucionAction, EstadisticaAction,

PrestamoAction, ReparacionAction, SemestreAction, TarjetaAction y TipoTarjetaAction; todas estas clases se diseñaron para que hereden la clase PeticionRespuestaAction que se presenta en la figura 6.20:

```
public class PeticionRespuestaAction extends ActionSupport
            implements ServletRequestAware, ServletResponseAware{

      private HttpServletResponse response;
      private HttpServletRequest request;
      private ServletOutputStream sos;

      //...
}
```

Figura 6.20 La clase PeticionRespuestaAction

En ésta clase, se puede observar que se realizó la extensión de la clase ActionSupport e implemento las interfaces ServletRequestAware y ServletResponseAware. El objetivo de hacer esto, es que todas las clases Action puedan acceder a los recursos externos como lo son los servlets de Java con la meta de poder utilizar AJAX.

Para el diagrama de secuencia de Ingresar Nuevo Tipo de Tarjeta, se especificó que se debía implementar el método guardar() en la clase TipoTarjetaAction. Recordando que el diagrama de secuencia se lee de arriba abajo y de izquierda a derecha, en este diagrama lo que señala a continuación que se debe hacer es crear una instancia de tipo TipoTarjetaDAO en éste método. Una vez creado el objeto TipoTarjetaDAO, se debe mandar a llamar al método consultar(). A continuación se presenta este fragmento de código:

```
TipoTarjetaDAO tipoTarjetaDAO = new TipoTarjetaDAOImpl();
Iterator<TipoTarjeta> iter1 =
tipoTarjetaDAO.consultar(nombre).iterator();
```

De este pequeño fragmento, se puede observar lo siguiente: debido a que el tipo de retorno del método consultar(nombre) es de tipo List<TipoTarjeta>, quiere decir que el método

retorna una lista de TipoTarjeta. Lo que se tiene que realizar en el código es iterar sobre ésta lista, si existe por los menos un elemento se concluirá que el tipo de tarjeta ya existe en el sistema:

```
if(iter.hasNext()) { // Si el tipo de tarjeta existe.

    // Se notifica el cliente que el tipo de tarjeta ya existe
    TextoJSON = {función:'existeTipoTarjeta'};
} else { // Si el tipo de tarjeta no existe.
         // Se procede a guardar el tipo de tarjeta.

    // Código para guardar tarjeta
}
```

Éste código, sigue la lógica del diagrama de secuencia para Ingresar Nuevo Tipo de Tarjeta.

La manera de enviar al cliente una respuesta al Administrador, es enviar datos en formato JSON. En este caso, se envía una cadena llamada textoJSON que se le da el formato JSON: {funcion:'existeTipoTarjeta'}.

En todas las clases Action se incluyó la cadena textoJSON, no en todos los casos la cadena es tan simple como la anterior. En algunos casos se genera dinámicamente dependiendo del contenido de la base de datos.

El método guardar() en la clase TipoTarjetaAction, a continuación se presenta en la figura 6.21:

```
public String guardar() throws IOException {
       getResponse().setContentType("text/html;charset=ISO-8859-1");
       setSos(getResponse().getOutputStream()); // 1
       String textoJSON = null;

       // Se consultan los tipos de tarjeta por nombre.
       TipoTarjetaDAO tipoTarjetaDAO = new TipoTarjetaDAOImpl();
       Iterator<TipoTarjeta> iter1 = tipoTarjetaDAO.consultar(nombre).iterator();

      try {
            if(iter1.hasNext()) { // Si el tipo de tarjeta existe.

                  // Se notifica al cliente que el tipo de tarjeta ya existe.
                  textoJSON = "{funcion:'existeTipoTarjeta'}";

            } else { // Si el tipo de tarjeta no existe.
                        // Se procede a guardar el tipo de tarjeta.

                  // Se guarda el tipo de tarjeta.
                  TipoTarjetaTransaccion
                  tarjetaTransaccion = new TipoTarjetaTransaccionImpl();

                  tarjetaTransaccion.guardar(nombre, descripcion);

                  // Se notifica al cliente que el tipo de tarjeta se guardo.
                  textoJSON = "{funcion:'guardoTipoTarjeta'}";
            }
      } catch (Exception e) {
            e.printStackTrace();

            // Se notifica al cliente que se lanzo una excepción.
            textoJSON = "{funcion:'lanzoExcepcion'}";
            getSos().print(textoJSON); // 2
            return null;
      } finally {
            getSos().print(textoJSON); // 2
            return null;
      }
}
```

Figura 6.21 El método guardar() en la clase TipoTarjetaAction

En este código, se hace uso de los recursos externos, ver líneas 1 y 2 en la figura 6.21. En la línea 1 se obtiene el objeto que envía texto al cliente y en las líneas 2 se imprime la respuesta. Pueden pasar dos cosas: que se lance una excepción o se guarde el tipo de tarjeta. Finalmente, la cadena textoJSON se procesa en el código JavaScript en el cliente.

6.2.5 El lenguaje de expresiones OGNL en la vista principal

Cuando una petición llega al marco de trabajo, una de las primeras cosas que Struts hace es crear los objetos que almacenaran todos los objetos importantes de la petición. Si se habla de los datos específicos del dominio de la aplicación (es decir, los datos a los que con mayor frecuencia se acceden con las etiquetas de Struts), estos se almacenan en ValueStack. Pero el procesamiento de una petición exige más que los datos de dominio de una aplicación. Es necesario almacenar otro tipo de datos más relacionados con la infraestructura. Todos estos datos, conjuntamente con el ValueStack, se almacenan en el llamado ActionContext.

ActionContext contiene todos los datos disponibles para el procesamiento de la petición del marco de trabajo, incluido aspectos que van desde los datos de la aplicación hasta mapas de sesión o de aplicación. Todos los datos específicos de una aplicación, tales como las propiedades expuestas sobre la acción, quedaran contenidos en ValueStack, uno de los objetos de ActionContext.

Todas las expresiones OGNL deben resolverse contra uno de los objetos contenidos en ActionContext. ValueStack será el elegido por defecto para la resolución OGNL.

Las expresiones OGNL apuntan hacia propiedades sobre objetos específicos. La resolución de cada expresión OGNL exige un objeto raíz contra el que comenzará la resolución de las referencias. ValueStack sirve por defecto como objeto raíz para la resolución de todas las expresiones OGNL que no nombren de forma explícita un objeto inicial.

Vea la siguiente expresión OGNL:

tarjeta.tipoTarjeta.nombre

Aquí se apunta a la propiedad nombre del objeto tipoTarjeta en el objeto tarjeta.

Las expresiones OGNL pueden empezar con una sintaxis especial que nombra al objeto del contexto contra el cual debe resolverse. La siguiente expresión demuestra esta sintaxis:

```
#listaTarjeta.get(0).numeroSerie
```

El operador # ordena al lenguaje OGNL utilizar el objeto con nombre ubicado en su contexto como objeto inicial para la resolución del resto de la expresión.

Para generar la vista principal, es decir, el despliegue de todas las tarjetas existentes (figura 6.22) en la aplicación del sistema de control de tarjetas se utilizó el lenguaje OGNL

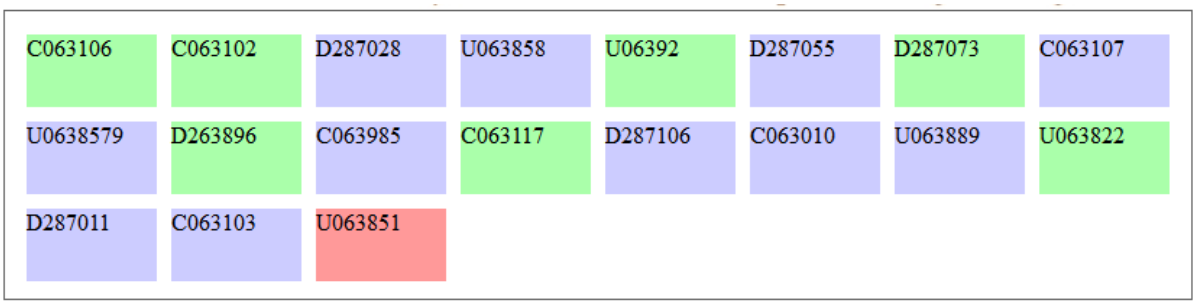


Figura 6.22 Despliegue de todas las tarjetas existentes

Cuando la aplicación inicia la acción que hace que se muestre la pantalla de la figura 6.22, es la acción llamada tarjetas. Ésta acción definida en el archivo secundario de configuración en seguida se muestra:

```
<action name="tarjetas" method="consultarTodo"
class="tarjetas.control.action.TarjetaAction">
        <result name="exito">/menu_tarjetas.jsp</result>
</action>
```

En esta configuración se puede observar el método consultarTodo() de la clase TarjetaAction, dicho método se ejecuta cuando el cliente solicita la acción. A continuación se muestra éste método en la figura 6.23.

En este método, se puede observar que se realizaron tres consultas: consultar todas las tarjetas, consultar el semestre actual y consultar todos los tipos de tarjetas. Estas tres consultas se guardan en las tres variables de instancia listaSemestre, listaTarjeta y listaTipoTarjeta con la finalidad de que los datos de las tres consultas se guarden en el ValueStack.

El interceptor params moverá los datos desde el objeto de la petición a ValueStack. La parte más difícil es la de asignar el nombre del parámetro a una propiedad real de ValueStack. Es en este punto donde el lenguaje OGNL entra en acción. El interceptor params interpreta el nombre del parámetro de la petición como una expresión OGNL que debe localizar la propiedad de destino correcta en el ValueStack.

```
private List<Tarjeta> listaTarjeta;
private List<Semestre> listaSemestre;
private List<TipoTarjeta> listaTipoTarjeta;

public String consultarTodo() {

	// Se consultan todas la tarjetas.
	TarjetaDAO tarjetaDAO = new TarjetaDAOImpl();
	setListaTarjeta(tarjetaDAO.consultar());

	// Se consulta el semestre actual.
	SemestreDAO semestreDAO = new SemestreDAOImpl();
	setListaSemestre(semestreDAO.consultarSemestreActual());

	// Se consulta todos los tipos de tarjetas.
	TipoTarjetaDAO tipoTarjetaDAO = new TipoTarjetaDAOImpl();
	setListaTipoTarjeta(tipoTarjetaDAO.consultar());

	return "exito";
}
```

Figura 6.23 El método consultarTodo() de la clase TarjetaAction

Una vez que se ejecutó el método consultarTodo(), el controlador de Struts regresará a la vista para mostrar la respuesta a tal petición. La vista que se mostrará será la ejecución de la página menu_tarjetas.jsp, en la que se utilizó la librería de etiquetas de Struts.

Para poder utilizar la librería de etiquetas de Struts, se debe incluir la siguiente declaración en la parte superior de cada página e incluir el prefijo <s:nombre_etiqueta /> a las etiquetas:

```
<%@ taglib prefix="s" uri="/struts-tags" %>
```

A continuación se presenta un fragmento de código de la página menu_tarjetas.jsp (miré la figura 6.24).

```
<s:set var="listaTarjeta" value="listaTarjeta"></s:set>
<s:set var="tamano" value="listaTarjeta.size"></s:set>
<s:set var="posx" value="180"></s:set>
<s:set var="posy" value="160"></s:set>

<s:if test="8 > #tamano">
    <s:iterator begin="0" end="7" id="m">
        <s:if test="#tamano > #m"> ...
        <s:else> ...
      <s:set var="posx" value="#posx + 100"></s:set>
    </s:iterator>
</s:if>
<s:else>
    <s:iterator begin="0" end="(#tamano - 1)" id="k">
        <s:if test="#k % 8 !=  0">
            <s:if test="#listaTarjeta.get(#iter2).estado.equals('no prestada')">
            <td class='colorTarjeta1' width='90px' height='50px' align='left'
                  valign='top' onclick="menuTarjetasNoPrestadas({numeroSerie:'<s:property
                  value='#listaTarjeta.get(#k).numeroSerie' />',idTarjeta:<s:property
                  value='#listaTarjeta.get(#k).idTarjeta' />,periodo:'<s:property
                  value='#listaSemestre.get(0).periodo' />',idSemestre:<s:property
                  value='#listaSemestre.get(0).idSemestre' />,posicionx:<s:property
                  value='#posx' />,posiciony:<s:property value='#posy' />})">
                  <s:property value="#listaTarjeta.get(#k).numeroSerie" />
            </td>
            </s:if>
            <s:elseif test="#listaTarjeta.get(#k).estado.equals('prestada')"> ...
            <s:elseif test="#listaTarjeta.get(#k).estado.equals('en reparacion')"> ...
            <s:if test="#posx >=  840"> ...
          </s:if>
        <s:else>
            <s:if test="#listaTarjeta.get(#k).estado.equals('no prestada')"> ...
            <s:elseif test="#listaTarjeta.get(#k).estado.equals('prestada')"> ...
            <s:elseif test="#listaTarjeta.get(#k).estado.equals('en reparacion')"> ...
          </s:else>
        <s:set var="posx" value="#posx + 100"></s:set>
    </s:iterator>
<s:else>
```

Figura 6.24 Fragmento de código de menu_tarjetas.jsp

En éste fragmento, las etiquetas muestran contenido dinámico y estático con la ayuda de expresiones OGNL. Las etiquetas utilizadas son las etiquetas para la gestión de los accesos a los datos, las condicionales y estructuras de control.

En el orden que aparecen las etiquetas en el fragmento de código, tenemos las etiquetas:

1. <s:set />
2. <s:if />, <s:else /> y <s:elseif />
3. <s:iterator />
4. <s:property />

<s:set />

La etiqueta <s:set /> permite crear una propiedad asociada a su valor dinámico (tipo primitivo u objeto). La variable se puede crear en el contexto de la aplicación, en la sesión, en la solicitud actual o en la página. En la siguiente tabla 6.8, se muestran los parámetros de la etiqueta.

Atributo	Requerido	Tipo	Descripción
id	No	String	Obsoleto. Utilice var en su lugar.
name	No	String	Obsoleto. Utilice var en su lugar.
scope	No	String	El ámbito a la cual se asigna la variable. Puede ser aplication, session, request, page, o action.
value	No	Object	El valor que es asignado al nombre de la variable.
var	No	String	Nombre que se utiliza para hacer referencia al valor dentro de ValueStack.

Tabla 6.8 Parámetros de la etiqueta <s:set />

En las primeras dos líneas del fragmento de la página menu_tarjetas.jsp, se declaran dos variables:

```
<s:set var=”listaTarjeta” value=”listaTarjeta”></s:set>
<s:set var=”tamano” value=”listaTarjeta.size”></s:set>
```

El lenguaje de expresiones OGNL es la parte que está entre comillas dentro del atributo value. En este caso, la etiqueta <s:set var=”listaTarjeta” value=”listaTarjeta”></s:set> toma un valor de unos de los objetos Java. En esto consiste la función del lenguaje de expresiones: permite usar una sintaxis simplificada para referenciar objetos que residen en el entorno Java.

El lenguaje de expresiones OGNL puede ser mucho más complejo que la expresión de un único elemento; da incluso soporte a funciones tan avanzadas como la invocación a llamadas al método sobre los objetos Java a los que puede acceder, pero en realidad el lenguaje de expresiones tiene como propósito simplificar el acceso a los datos.

<s:if />, <s:elseif> y <s:else>

Las etiquetas <s:if />, <s:elseif> y <s:else> se utilizan para realizar pruebas condicionales. En las tablas 6.9 y 6.10 se muestran los parámetros de estas etiquetas.

Atributo	Requerido	Tipo	Descripción
test	verdadero	Boolean	Expresión para determinar si el cuerpo de la etiqueta se mostrará.

Tabla 6.9 Parámetros de la etiqueta <s:if />

Atributo	Requerido	Tipo	Descripción
test	verdadero	Boolean	Expresión para determinar si el cuerpo de la etiqueta se mostrará.

Tabla 6.10 Parámetros de la etiqueta <s:elseif />

En las siguientes líneas del fragmento de la página menu_tarjetas.jsp, en seguida se muestran algunas de las pruebas condicionales que se realizaron:

```
<s:set var="tamano" value="listaTarjeta.size"></s:set>
<s:if test="8 > #tamano">
     <s:iterator begin="0" end="7" id="m">
          <s:if test="#tamano >= #m"> …
          <s:else>
```

En la etiqueta <s:if test="8 > #tamano" /> se prueba que el tamaño de la lista de tarjetas sea menor a 8, la etiqueta <s:iterator begin="0" end="7" id="m" /> se ejecutará si la

condición se cumple. Se puede observar que en la prueba “8 > #tamano”, #tamano es la manera de referenciar a la variable declarada en la etiqueta <s:set />.

<s:iterator />

La etiqueta <s:iterator /> permite recorrer colecciones de objetos con facilidad. La etiqueta está diseñada para recorrer cualquier elemento Collection, Map, Enumeration, Iterator o arreglo. También, ofrece la capacidad de definir una variable que permite determinar cierto nivel básico de información sobre el estado de la iteración. En la tabla 6.11, se muestran los parámetros de la etiqueta.

Atributo	Requerido	Tipo	Descripción
id	No	String	Obsoleto. Utilice var en su lugar.
value	No	String	Objeto que debe iterarse.
var	No	String	Nombre que se utiliza para hacer referencia al valor dentro de Value-Stack.
begin	No	String	Inicio de donde se debe empezar a iterar.
end	No	String	Final de donde se debe terminar de iterar.

Tabla 6.11 Parámetros de la etiqueta <s:iterator />

La etiqueta <s:iterator /> se muestra en la figura 6.25. En éste fragmento de código se imprime en pantalla una celda por cada elemento de la lista de tarjetas. Cada celda representa una tarjeta.

Cada celda está formada por una etiqueta <td /> con un color de fondo y un evento asociado dependiendo del estado de la tarjeta, es decir, si la tarjeta esta: prestada, no prestada o en reparación.

El valor de #m se inicializa en 0, se incrementará de uno en uno hasta llegar a 7. Bien, si el tamaño de la lista de tarjetas es menor a 7 la condición <s:if test="*#tamano* >=

#m"> no se cumplirá y se imprimirán celdas vacías con la finalidad de que el total de celdas en el renglón sean siempre ocho.

```
<s:iterator begin="0" end="7" id="m">
  <s:if test="#tamano >= #m">
    <td width='5px'></td>
    <s:if test="#listaTarjeta.get(#m).estado.equals('no prestada')">

    <td class='colorTarjeta1'  width='90px' height='50px' align='left'
      valign='top' onclick="menuTarjetasNoPrestadas({numeroSerie:
      '<s:property value='#listaTarjeta.get(#m).numeroSerie' />',
      idTarjeta:<s:property value='#listaTarjeta.get(#m).idTarjeta' />,
      periodo:'<s:property
      value='#listaSemestre.get(0).periodo' />',idSemestre:<s:property
      value='#listaSemestre.get(0).idSemestre' />,posicionx:<s:property
      value='#posx'/>,posiciony:<s:property value='#posy' />})">
        <s:property value="#listaTarjeta.get(#m).numeroSerie" />
    </td>
    </s:if>
    <s:elseif test="#listaTarjeta.get(#m).estado.equals('prestada')">
    <td class='colorTarjeta2' width='90px' height='50px' align='left'
      valign='top' onclick="menuTarjetasPrestadas({numeroSerie:'<s:property
      value='#listaTarjeta.get(#m).numeroSerie' />',idTarjeta:<s:property
      value='#listaTarjeta.get(#m).idTarjeta' />,periodo:'<s:property
      value='#listaSemestre.get(0).periodo' />',idSemestre:<s:property
      value='#listaSemestre.get(0).idSemestre' />,posicionx:<s:property
      value='#posx'/>,posiciony:<s:property value='#posy' />})">
        <s:property value="#listaTarjeta.get(#m).numeroSerie" />
    </td>
    </s:elseif>
    <s:elseif test="#listaTarjeta.get(#m).estado.equals('en reparacion')">
    <td class='colorTarjeta3' width='90px' height='50px' align='left'
      valign='top'onclick="menuTarjetasEnReparacion({numeroSerie:'<s:property
      value='#listaTarjeta.get(#m).numeroSerie' />',idTarjeta:<s:property
      value='#listaTarjeta.get(#m).idTarjeta' />,numeroReparacion:<s:property
      value='#listaTarjeta.get(#m).numeroReparacion' />,posicionx:<s:property
      value='#posx' />,posiciony:<s:property value='#posy' />})">

        <s:property value="#listaTarjeta.get(#m).numeroSerie" />
    </td>
    </s:elseif>
    <td width='5px'></td>
  </s:if>
  <s:else>
    <td width='5px'></td>
    <td width='100px' height='50px' align='left' valign='top'
      style='cursor: pointer; background-color: #fff'>
    </td>
    <td width='5px'></td>
  </s:else>
  <s:set var="posx" value="#posx + 100"></s:set>
</s:iterator>
```

Figura 6.25 La etiqueta <s:iterator /> en la condición <s:if />

En la figura 6.26, se muestra el código en ejecución de la etiqueta <s:iterator />. Cuando se realiza un evento sobre la celda, en este caso un click se despliega un menú.

Figura 6.26 El código en ejecución de la etiqueta <:iterator /> en la condición <s:if />

Se crearon las variables posx y posy para posicionar horizontalmente y verticalmente la ventana de menú. Al irse realizando la iteración, la variable posx se va incrementando con lo cual se tendrá una posición diferente del menú para cada tarjeta.

Para cada celda generada, se utiliza el parámetro onclick en la etiqueta <td /> respectiva. En el parámetro onclick de ésta etiqueta, se llama a una función JavaScript (menuTarjetasNoPrestadas, menuTarjetasPrestadas y menuTarjetasEnReparacion), a las cuales se les pasa entre otros valores los valores de las variables posx y posy.

A continuación se presenta el código de una celda no prestada:

```
  <td class='colorTarjeta1' width='90px' height='50px'
      align='left' valign='top' onclick="menuTarjetasNoPrestadas(
      {numeroSerie:'<s:property value='#listaTarjeta.get(#m).
      numeroSerie' />',idTarjeta:<s:property value='#listaTarjeta.
      get(#m).idTarjeta' />,periodo:'<s:property value=
      '#listaSemestre.get(0).periodo' />',idSemestre:<s:property
      value='#listaSemestre.get(0).idSemestre'/>,posicionx:
      <s:property value='#posx' />,posiciony:<s:property value='#posy'
/>})">
      <s:property value="#listaTarjeta.get(#m).numeroSerie" />
  </td>
```

Lo realizado en éste código es lo siguiente: se utilizaron hojas de estilo CSS para controlar el color de la celda dependiendo el estado de la tarjeta, al darse el evento click sobre la celda se manda a llamar en este caso a la función menuTarjetasNoPrestadas() y se le envía la información de la tarjeta, del semestre y así como, la posición horizontal y vertical en que debe aparecer el menú.

Sin embargo, si la condición de la etiqueta <s:if test="8 > #tamano" /> no se cumple se ejecutara la etiqueta <s:else />. En este caso, el código 6.27 producirá la vista.

En la etiqueta <s:iterator /> el valor de #k se inicializa en 0, se incrementará de uno en uno hasta llegar a #tamano - 1. Después de ésta etiqueta se presenta la condición <s:if test="*#k % 8 != 0*">, si se cumple se imprimirán las celdas de la segunda hasta la *#k % 8*. En otro caso, se imprimirá la primera celda de cada renglón.

También, en este fragmento de código se utilizan las variables posx y posy para posicionar horizontalmente y verticalmente la ventana de menú. Al irse realizando la iteración, estas variables se van incrementando. La variable posy se incrementa al terminarse un renglón de ocho elementos.

```
<s:else>
  <s:iterator begin="0" end="(#tamano - 1)" id="k">
      <s:if test="#k % 8 !=  0">
        <td width='5px'></td>
            <s:if test="#listaTarjeta.get(#k).estado.equals('no prestada')">
            <td class='colorTarjeta1' width='90px' height='50px' align='left'
                  valign='top' onclick="menuTarjetasNoPrestadas({numeroSerie:
                  '<s:property value='#listaTarjeta.get(#k).numeroSerie' />',idTarjeta:
                  <s:property value='#listaTarjeta.get(#k).idTarjeta' />,periodo:
                  '<s:property
                  value='#listaSemestre.get(0).periodo' />',idSemestre:
                  <s:property value='#listaSemestre.get(0).idSemestre' />,posicionx:
                  <s:property
                  value='#posx' />,posiciony:<s:property value='#posy' />})">
                  <s:property value="#listaTarjeta.get(#k).numeroSerie" />
            </td>
            </s:if>
            <s:elseif test="#listaTarjeta.get(#k).estado.equals('prestada')">...

            <s:elseif test="#listaTarjeta.get(#k).estado.equals('en reparacion')">...

            <td width='5px'></td>
            <s:if test="#posx >=  840">
                  <s:set var="posx" value="110"></s:set>
                  <s:set var="posy" value="#posy + 70"></s:set>
            </s:if>
      </s:if>
      <s:else>
            <tr>
                  <td  colspan="1" height="10px" style='background-color: #fff'></td>
            <tr/>
            <tr>
                  <td width='5px'></td>
                  <s:if test="#listaTarjeta.get(#k).estado.equals('no prestada')">
                  <td class='colorTarjeta1'  width='90px' height='50px' align='left'
                        valign='top' onclick="menuTarjetasNoPrestadas({numeroSerie:
                        '<s:property
                        value='#listaTarjeta.get(#k).numeroSerie' />',idTarjeta:
                        <s:property value='#listaTarjeta.get(#k).idTarjeta' />,periodo:
                        '<s:property
                        value='#listaSemestre.get(0).periodo' />',idSemestre:
                        <s:property value='#listaSemestre.get(0).idSemestre' />,posicionx:
                        <s:property
                        value='#posx' />,posiciony:<s:property value='#posy' />})">
                        <s:property value="#listaTarjeta.get(#k).numeroSerie" />
                  </td>
                  </s:if>
                  <s:elseif test="#listaTarjeta.get(#k).estado.equals('prestada')">...

                  <s:elseif test="#listaTarjeta.get(#k).estado.equals('en reparacion')">...

                  <td width='5px'></td>
            </s:else>
      <s:set var="posx" value="#posx + 100"></s:set>
  </s:iterator>
</s:else>
```

Figura 6.27 La etiqueta <s:iterator /> en la condición <s:else />

<s:property />

La etiqueta <s:property> permite mostrar, a partir de expresiones OGNL, datos presentes en la accion asociada con

sus descriptores de acceso o en el contexto de la aplicación (application, sesión, request, parameters, attr). El atributo value permite nombrar la propiedad y el parámetro espace permite evitar los caracteres HTML especiales (“,”, &, < y >).

En los fragmentos de código anteriores (figuras 6.25 y 6.27) se utiliza ésta etiqueta para mostrar en pantalla el número de serie de la tarjeta en cada celda.

6.3 Construcción con AJAX

Se empleó la reutilización de código JavaScript en la aplicación, para esto se encapsulo dentro de una clase el proceso de configuración y realización de peticiones asíncronas.

La clase PeticionAsincrona está formada por los siguientes métodos:

1. **enviar()** Realiza el envío de la petición.

2. **respuestaTexto()** Devuelve la respuesta recibida en formato de texto plano.

3. **estado()** Devuelve el estado HTTP de la respuesta.

4. **textoEstado()** Devuelve el texto asociado al estado HTTP de la respuesta.

El constructor (figura 6.28) de la clase PeticionAsincrona quedo de la siguiente manera:

```
function PeticionAsincrona() {

      //Creación de un objeto XMLHttpRequest
      var xhr;
      if(window.ActiveXObject) {
            xhr=new ActiveXObject("Microsoft.XMLHttp");
      } else if((window.XMLHttpRequest) || (typeof XMLHttpRequest)!=undefined) {
            xhr=new XMLHttpRequest();
      } else {
       xhr=null;
      }

      // Declaración de métodos de la clase
      this.enviar=m_enviar;
      this.respuestaTexto=m_texto;
      this.estado=m_estado;
      this.textoEstado=m_textoEstado;

      // Funciones para la implementacion de los métodos
      // ...
}
```

Figura 6.28 Constructor de la clase PeticionAsincrona

El envío de la petición

Con una única llamada al método enviar se realizará todo el trabajo de configuración y petición AJAX. Éste método tiene el siguiente formato:

```
enviar(accion, funcionRetorno, datos)
```

1. **accion.** Representa la acción que procesará la petición asíncrona.

2. **funcionRetorno.** Cadena de caracteres que representa el nombre de la función de retorno donde se implementarán las instrucciones para el tratamiento de la respuesta.

3. **datos**. Cadena de caracteres que sigue el formato de parejas nombre-valor:

   nombre1=valor1&nombre2=valor2

El método enviar es implementado mediante la función interna m_enviar, la figura 6.29 corresponde a dicha función:

```
//Implementación del método enviar
function m_enviar(accion, funcionRetorno, datosEnviar) {

	// Realiza siempre la petición en modo asíncrono y en método POST
	xhr.open("POST", accion, true);

	xhr.onreadystatechange=function() {
		// Si la respuesta está disponible
		// se ejecuta la función de retorno
		if(xhr.readyState==4) {
			eval(funcionRetorno+"("+")");
		}
	};

	// Establece el encabezado apropiado para el
	// envío de datos de un formulario
	xhr.setRequestHeader('Content-Type','application/x-www-form-urlencoded');

	xhr.send(datosEnviar);
}
```

Figura 6.29 Implementación del método enviar

Los métodos respuestaTexto, textoEstado y estado son implementados mediante la funciones internas m_texto, m_estado y m_textoEstado, respectivamente. En la figura 6.30 se muestran dichas funciones:

```
// Implementación del método texto
function m_texto() {
	return xhr.responseText;
}

// Implementación del método estado
function m_estado() {
	return xhr.status;
}

// Implementación del método textoEstado
function m_textoEstado() {
	return xhr.statusText;
}
```

Figura 6.30 Implementación de las funciones respuestaTexto, textoEstado y estado

6.4 Construcción utilizando el marco de trabajo Prototype Window Class

En todos los ingresos (figuras 6.31, 6.32 y 6.33) en el sistema de control de tarjetas se utilizó la ventana de dialogo confirm del marco de trabajo Prototype Window Class.

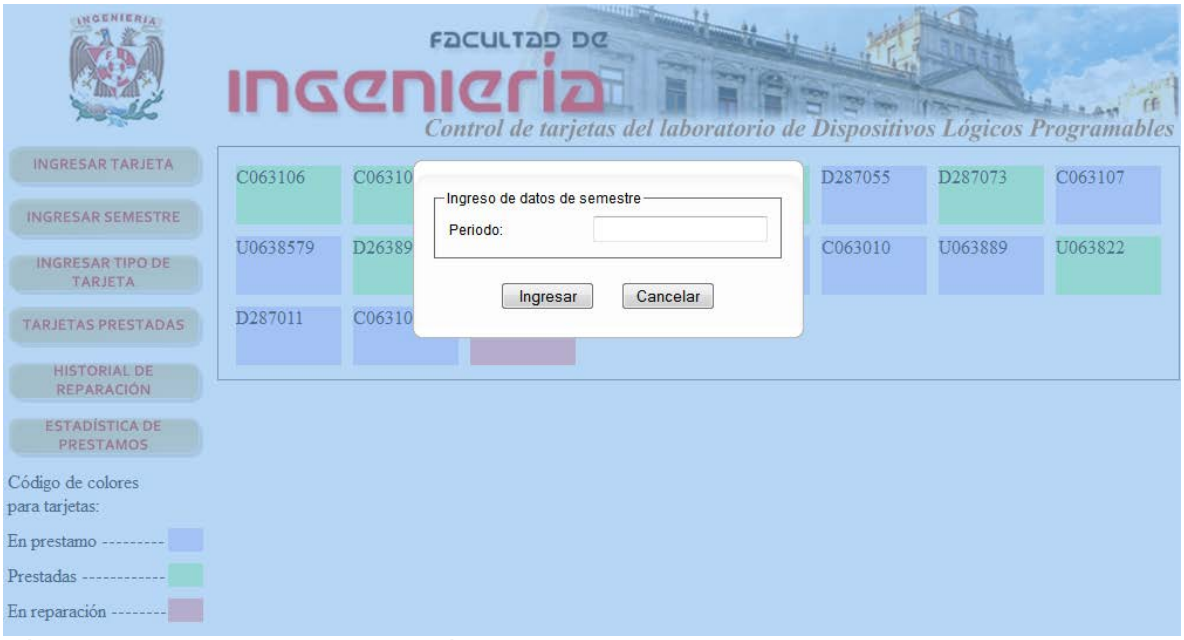


Figura 6.31 Ventana para ingresar nuevo semestre

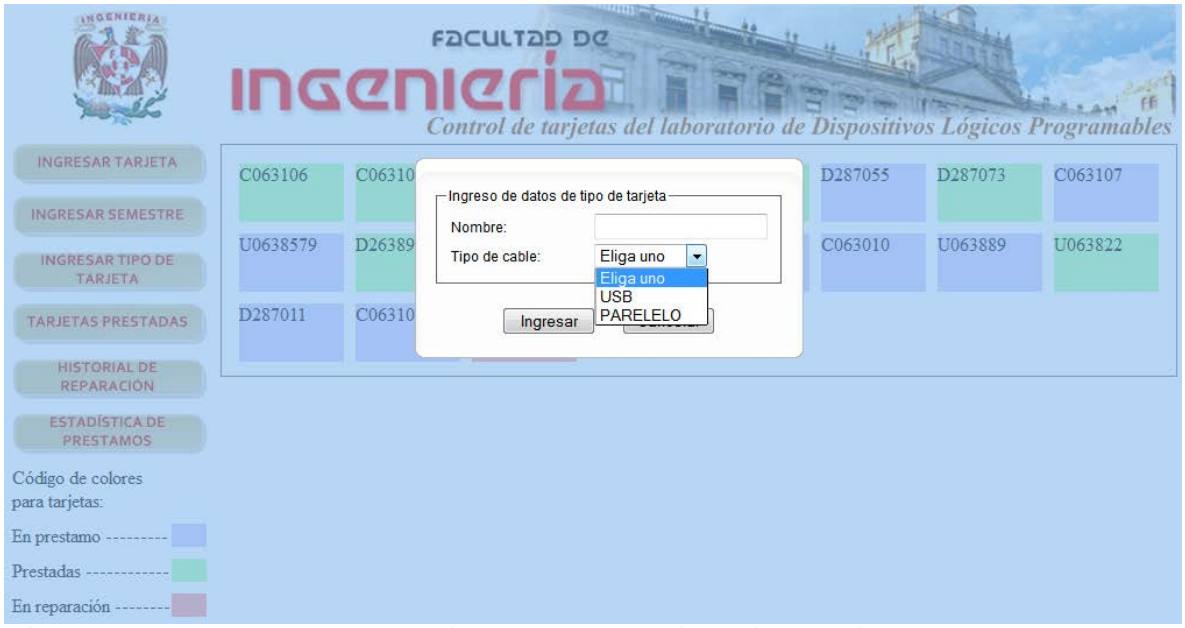
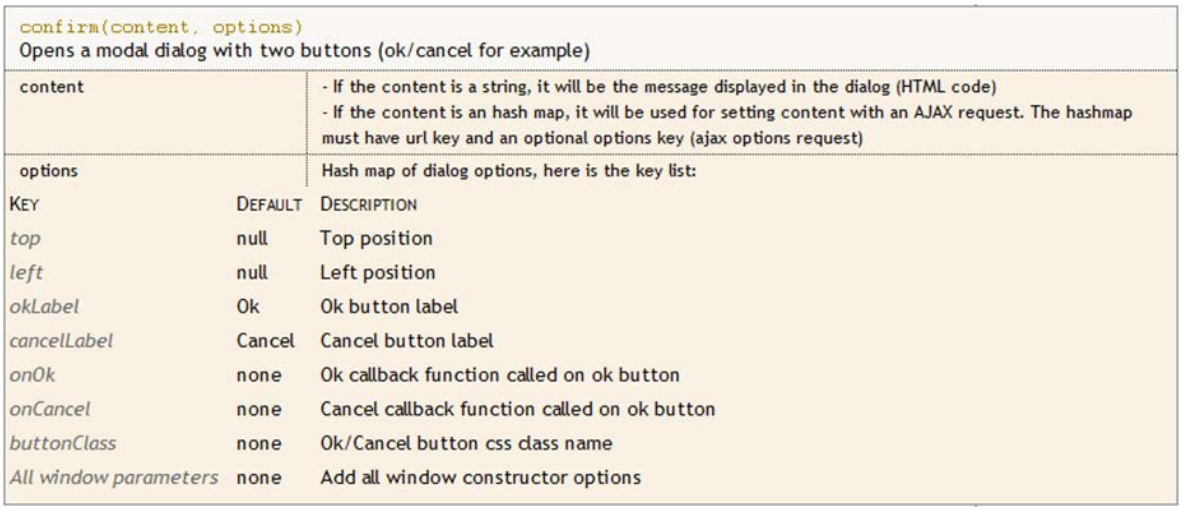


Figura 6.32 Ventana para ingresar un tipo de tarjeta

Figura 6.33 Ventana para ingresar nueva tarjeta

La documentación se puede revisar en la siguiente liga (http://prototype-window.xilinus.com/documentation.html). A continua-ción, en la figura 6.34 se muestra un fragmento extraído de esta documentación para la ventana de dialogo confirm:



confirm(content, options)
Opens a modal dialog with two buttons (ok/cancel for example)

content		- If the content is a string, it will be the message displayed in the dialog (HTML code) - If the content is an hash map, it will be used for setting content with an AJAX request. The hashmap must have url key and an optional options key (ajax options request)
options		Hash map of dialog options, here is the key list:
KEY	DEFAULT	DESCRIPTION
top	null	Top position
left	null	Left position
okLabel	Ok	Ok button label
cancelLabel	Cancel	Cancel button label
onOk	none	Ok callback function called on ok button
onCancel	none	Cancel callback function called on ok button
buttonClass	none	Ok/Cancel button css class name
All window parameters	none	Add all window constructor options

Figura 6.34 El método confirm() en la API Prototype Window Class

En esta documentación se puede obtener la siguiente información: el método confirm() se la pasan dos parámetros: el contenido y una serie de opciones entre {}. El contenido puede ser una cadena que contenga código HTML para que se despliegue en la ventana. Las opciones pueden ser posición superior, posición izquierda, los botones ok y cancel (se les puede poner el nombre de etiqueta que se desee), etc.

En la función onOk, se agrega el código que se ejecutará si el Administrador da click en el botón ok. También, en la función cancel se agrega el código que se ejecutará si el Administrador da click en el botón cancel.

Para acceder a cualquiera de las ventanas de ingreso el Administrador lo que tiene que hacer es dar click en alguna opción del menú vertical (figura 6.35):

Figura 6.35 Parte del menú vertical: opciones de ingreso

Lo que se realizó en la página JSP es llamar a una función JavaScript cuando ocurra el evento click en cualquiera de los botones. En seguida se muestra el fragmento de código que realiza esto en la figura 6.36:

```
<tr>
<td colspan="2" style="cursor: pointer" onclick=
                    "ventanaIngresoTarjeta('<s:property value='#select' />')">
      <img alt="" src="botones/Ingresar_Tarjeta.jpg"></img>
</td>
</tr>
<tr>
<td colspan="2" style='cursor: pointer' onclick="ventanaIngresoSemestre()">
      <img alt="" src="botones/Ingresar_Semestre.jpg"></img>
</td>
</tr>
<tr>
<td colspan="2" style='cursor: pointer' onclick="ventanaIngresoTipoTarjeta()">
      <img alt="" src="botones/IngresarTipoTarjeta.jpg"></img>
 </td>
</tr>
```

Figura 6.36 El código del menú vertical en la página JSP

En este código se puede observar que se llama a las funciones ventanaIngresoTarjeta, ventanaIngresoSemestre y ventana-IngresoTipoTarjeta. En estas funciones se utilizó la función confirm(), en seguida se muestra en la figura 6.37 la primera función:

```
function ventanaIngresoTarjeta(select) {

    timeout = 0;
    if (timeout > 0) {
        setTimeout(infoTimeout, 5000);
    } else {
          Dialog.closeInfo();
          timeout = 30;
          setTimeout(infoTimeout4, 200);
    }

    var HTML = "<center>";
    HTML = HTML + "<fieldset style='width: 280px; border: 1px solid
                   #666666'>";
    HTML = HTML + "<legend>Ingreso de datos de tarjeta</legend>";
    HTML = HTML + "<table border='0' cellpadding='2' cellspacing='4'>";
    HTML = HTML + "<tr>";
    HTML = HTML + "<td width='150px' align='left'>";
    HTML = HTML + "Número de serie:";
    HTML = HTML + "</td>";
    HTML = HTML + "<td align='left'>";
    HTML = HTML + "<input type='text' name='numeroSerie'
                   id='numeroSerie' maxlength='40' size='20'/>";
    HTML = HTML + "</td>";
    HTML = HTML + "</tr>";
    HTML = HTML + "<tr>";
    HTML = HTML + "<td width='150px' align='left'>";
    HTML = HTML + "Tipo de tarjeta:";
    HTML = HTML + "</td>";
    HTML = HTML + "<td align='left'>";
    HTML = HTML + select;
    HTML = HTML + "</td>";
    HTML = HTML + "</tr>";
    HTML = HTML + "</table>";
    HTML = HTML + "</fieldset>";
    HTML = HTML + "</center>";

    Dialog.confirm(HTML,
       {top: 140,
       width:320,
               height:140,
       className: "alphacube",
       onOk: function(win) {

                      enviarDatosTarjeta($('numeroSerie').value,
                                         $('tipoTarjeta').value);

                      return true;
                          },
                cancel: function() {
                                        return true;
                          },
                      okLabel: "Ingresar",
                      cancelLabel:"Cancelar"
                });
}
```

Figura 6.37 La función ventanaIngresoTarjeta()

Lo que se realizó en éste código (figura 6.37), además de generar la ventana, es llamar a la función setTimeout() para dejar que dure un determinado tiempo la ventana desplegada. Y al terminar ese tiempo se cerrara si el Administrador no realiza ninguna acción. En seguida se definió la variable HTML y por último, se genera la ventana con la función confirm().

El código que llama a la función setTimeout() es el siguiente:

```
timeout = 0;
if (timeout > 0) {
setTimeout(infoTimeout, 5000);
} else {
      Dialog.closeInfo();
      timeout = 30;
      setTimeout(infoTimeout4, 200);
}
```

Cuando se termine el tiempo, se llamará a la función closeInfo(). En seguida, se presenta en la figura 6.38 el fragmento de la documentación correspondiente a ésta función:

Figura 6.38 El método closeInfo() en la API Prototype Window Class

En la variable HTML se forma el formulario que se mostrara en la ventana de la figura 6.39. Ésta variable se le pasa como primer parámetro a la función confirm() y como segundo parámetro una serie de opciones, entre ellas la función onOk().

En la función onOk() se llama a enviarDatosTarjeta(), en la figura 6.39 se muestra esta función. En ésta función se creó una instancia de la clase PeticionAsincrona para poder llamar a su método enviar. A la funcion enviar() se le pasa como parámetros la acción que procesara la petición asíncrona, el nombre de la funcion de retrollamada que se ejecutará cuando

termine de procesarse la petición y la informacion enviada al servidor.

```
var obj;

function enviarDatosTarjeta(numeroSerie, idTipoTarjeta) {
	obj = new PeticionAsincrona();
	var valores = "numeroSerie="+numeroSerie+"&idTipoTarjeta="+idTipoTarjeta;
	obj.enviar("guardarTarjeta.action","procesaResultadoGuardadoTarjeta",valores);
}

function procesaResultadoGuardadoTarjeta() {
	var objetoJSONin = eval("("+obj.respuestaTexto()+")");

	if(objetoJSONin.funcion=="existeTarjeta") {
		existeTarjeta(objetoJSONin);
	} else if(objetoJSONin.funcion=="guardoTarjeta") {
		guardoTarjeta(objetoJSONin);
	}
}
```

Figura 6.39 La función enviarDatosTarjeta()

La función que se ejecuta dependerá del procesamiento en el servidor, si en el servidor se determina que la tarjeta existe se llamará a la función existeTarjeta() y se verá la ventana de la figura 6.40. Y en caso contrario, se verá la ventana de la figura 6.41.

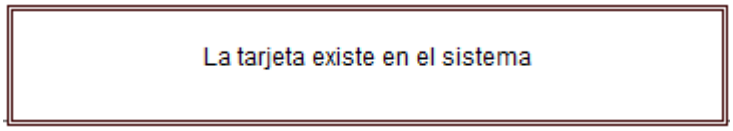


Figura 6.40 Mensaje de que existe la tarjeta

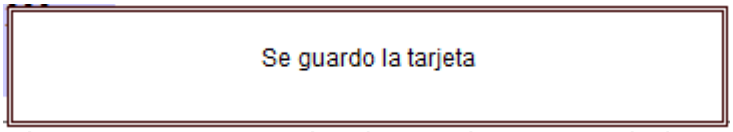


Figura 6.41 Mensaje de tarjeta guardada

En la figura 6.42 se muestra el menú correspondiente a una tarjeta dependiendo del estado en el que se encuentre:

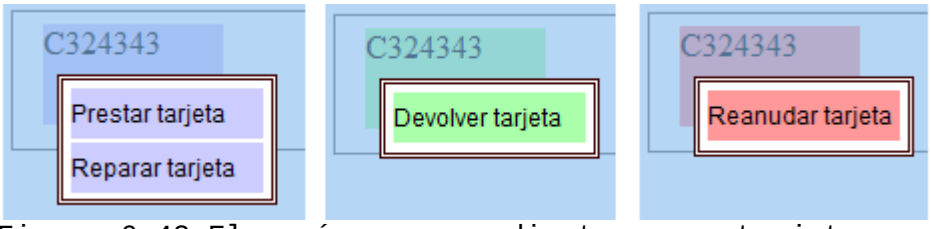


Figura 6.42 El menú correspondiente a una tarjeta

Para desplegar estos menús se utilizó la función info(), en este caso del primer menú de la figura 6.42, se genera con el código que a continuación se presenta en la figura 6.43:

En este código se le pasa un objeto JSON al método menuTarjetasNoPrestadas(). Se le envía el objeto JSON a los métodos ventanaPrestamoTarjeta() y ventanaReparacionTarjeta() para conservar la información relativa a una tarjeta.

```
function menuTarjetasNoPrestadas(objetoJSONin) {
   timeout=2;
   Dialog.info("<table height='50' width='100' border='0' cellpadding='1' cellspacing='2'>
               <tr style='color:black'><td height='25' class='colorTarjeta1'
               onclick='ventanaPrestamoTarjeta("+objetoJSONin.toJSONString()+")'>
               Prestar tarjeta</td></tr><tr style='color:black'><td height='25'
               class='colorTarjeta1' onclick='ventanaReparacionTarjeta
               ("+objetoJSONin.toJSONString()+")'>Reparar tarjeta</td></tr></table>",
               {left: objetoJSONin.posicionx, minHeight:20, minWidth:100, top:
               objetoJSONin.posiciony, width:100, height:55, showProgress: false});
   setTimeout(infoTimeout, 200);
}
```

Figura 6.43 La función que genera el menú de tarjetas no prestadas

Si se revisa la documentación sobre la función info(), se encontrará con lo que se muestra la figura 6.44:

info(content, options)
Opens a modal info panel without any button. It can have a progress image (Used to display submit waiting message for example)

content		- If the content is a string, it will be the message displayed in the dialog (HTML code) - If the content is an hash map, it will be used for setting content with an AJAX request. The hashmap must have url key and an optional options key (ajax options request)
options		Hash map of dialog options, here is the key list:
KEY	DEFAULT	DESCRIPTION
top	null	Top position
left	null	Left position
showProgress	false	Add a progress image (info found in the css file)
All window parameters	none	Add all window constructor options

Figura 6.44 El método info() en la API Prototype Window Class

De una manera similar a las ventanas anteriores, se le pasa a ésta función el contenido a mostrar en la ventana y las opciones. En la figura 6.43 el contenido mostrado es una tabla de HTML con dos celdas, las cuales llaman a una función JavaScript al darse el evento click sobre ellas: una función es para llamar la ventana de préstamo de tarjeta (figura 6.45) y la otra función es para visualizar la ventana de tarjetas en reparación (figura 6.46).

Figura 6.45 Ventana para préstamo de tarjeta

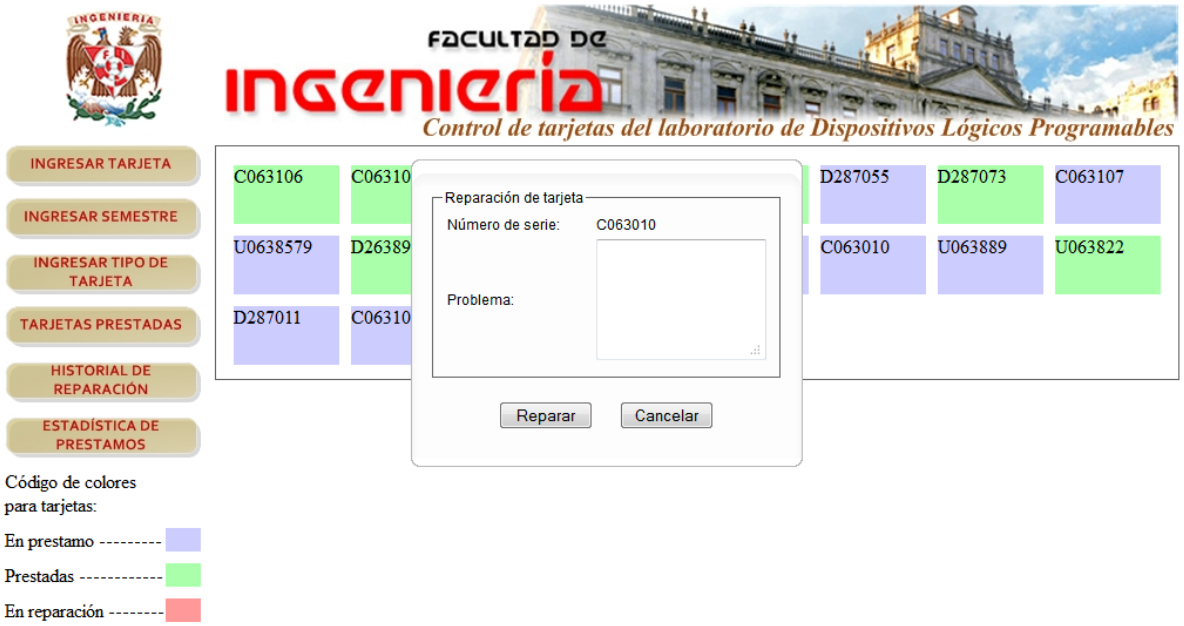


Figura 6.46 Ventana para reparación de tarjeta

En la figura 6.47, se muestra el código en el cual se definió la función ventanaPrestamoTarjeta(). En esta función lo primero que aparece es el código para que la ventana permanezca un determinado tiempo, en seguida se definió la variable HTML y por último, se genera la ventana con la función confirm().

También, en la variable HTML se forma el formulario que se mostrara en la ventana de la figura 6.45. Ésta variable se le pasa como primer parámetro a la función confirm() y como segundo parámetro una serie de opciones, entre ellas la función onOk().

```
function ventanaPrestamoTarjeta(objetoJSONin) {

    timeout = 0;
    if (timeout > 0) {
        setTimeout(infoTimeout, 5000);
    } else {
          Dialog.closeInfo();
          timeout = 30;
          setTimeout(infoTimeout4, 200);
    }

    var HTML = "<center>";
    HTML = HTML + "<fieldset style='width: 280px; border: 1px solid #666666'>";
    HTML = HTML + "<legend>Prestamo de tarjeta del semestre "+objetoJSONin.periodo+
                  "</legend>";
    HTML = HTML + "<table border='0' cellpadding='2' cellspacing='4'>";
    HTML = HTML + "<tr>";
    HTML = HTML + "<td width='150px' align='left'>";
    HTML = HTML + "Número de serie:";
    HTML = HTML + "</td>";
    HTML = HTML + "<td>";
    HTML = HTML + objetoJSONin.numeroSerie;
    HTML = HTML + "</td>";
    HTML = HTML + "</tr>";
    HTML = HTML + "<tr>";
    HTML = HTML + "<td align='left'>";
    HTML = HTML + "Número de cuenta:";
    HTML = HTML + "</td>";
    HTML = HTML + "<td align='left'>";
    HTML = HTML + "<input type='text' name='numeroCuenta' id='numeroCuenta'
                   maxlength='40' size='20'/>";
    HTML = HTML + "</td>";
    HTML = HTML + "</tr>";
    HTML = HTML + "</table>";
    HTML = HTML + "</fieldset>";
    HTML = HTML + "</center>";

    Dialog.confirm(HTML,
         { top: 140,
             width:320,
                           height:150,
           className: "alphacube",
             onOk: function(win) {

                      enviarPantalla1($('numeroCuenta').value,objetoJSONin.idSemestre,
                      objetoJSONin.idTarjeta);

                      return true;
                           },
             cancel: function() {
                                    return true;
                           },
                       okLabel: "Prestar",
                       cancelLabel:"Cancelar"
           });
}
```

Figura 6.47 La función ventanaPrestamoTarjeta()

En la función onOk() se llama a enviarDatosPrestamo1(), en la figura 6.48 se muestra esta función. En ésta función se creó una instancia de la clase PeticionAsincrona para poder llamar

a su método enviar. A la funcion enviar() se le pasa como parámetros la acción que procesara la petición asíncrona, el nombre de la funcion de retrollamada que se ejecutará cuando termine de procesarse la petición y la informacion enviada al servidor.

```
var obj;

function enviarDatosPrestamo1(numeroCuenta,idSemestre,idTarjeta) {
      obj = new PeticionAsincrona();
      var valores = "numeroCuenta="+numeroCuenta+"&idTarjeta="+
      idTarjeta+"&idSemestre="+idSemestre;
      obj.enviar("prestarTarjeta1.action","procesaResultadoPrestamo1",valores);
}
function procesaResultadoPrestamo1() {
      var objetoJSONin = eval("("+obj.respuestaTexto()+")");

      if(objetoJSONin.funcion=="insertarAlumno") {
            ventanaIngresoAlumno(objetoJSONin);
      } else if(objetoJSONin.funcion=="tarjetaPrestada") {
            tarjetaPrestada(objetoJSONin);
      } else if (objetoJSONin.funcion=="noPrestamoAlumno") {
            noPrestamoAlumno(objetoJSONin);
      } else if (objetoJSONin.funcion=="lanzoExcepcion") {
            lanzoExcepcion(objetoJSONin);
      }
}
```

Figura 6.48 La función enviarDatosPrestamo1()

La función que se ejecuta en el código 6.48 dependerá del procesamiento en el servidor, si en el servidor se procesa el préstamo de la tarjeta se llamará a la función tarjetaPrestada() y se verá la ventana de la figura 6.49.

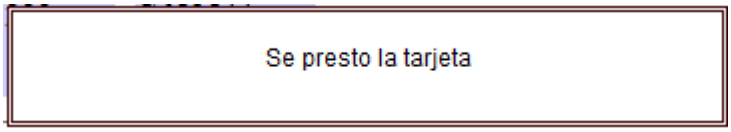


Figura 6.49 Mensaje de tarjeta prestada

Si en el servidor se determina que no existe el alumno en la base de datos, se mostrará la ventana de la figura 6.50.

Figura 6.50 Ventana para ingresar datos de un alumno

El código para generar la ventana de la figura 6.50, se exhibe a continuación en la figura 6.51:

```
function ventanaIngresoAlumno(objetoJSONin) {
    timeout = 0;
    if (timeout > 0) {
        setTimeout(infoTimeout, 5000);
    } else {
         Dialog.closeInfo();
         timeout = 30;
         setTimeout(infoTimeout4, 200);
    }
    var HTML = "<center>";
    HTML = HTML + "<fieldset style='width: 280px; border: 1px solid #666666'>";
    HTML = HTML + "<legend>Ingrese los datos del nuevo alumno para continuar con prestamo</legend>";
    HTML = HTML + "<table border='0' cellpadding='2' cellspacing='4'>";
    HTML = HTML + "<tr>";
    HTML = HTML + "<td width='150px' align='left'>";
    HTML = HTML + "Número de cuenta:";
    HTML = HTML + "</td>";
    HTML = HTML + "<td>";
    HTML = HTML + objetoJSONin.numeroCuenta;
    HTML = HTML + "</td>";
    HTML = HTML + "</tr>";
    HTML = HTML + "<tr>";
    HTML = HTML + "<td align='left'>";
    HTML = HTML + "Nombre:";
    HTML = HTML + "</td>";
    HTML = HTML + "<td align='left'>";
    HTML = HTML + "<input type='text' name='nombre' id='nombre' maxlength='40' size='20'/>";
    HTML = HTML + "</td>";
    HTML = HTML + "</tr>";
    HTML = HTML + "<tr>";
    HTML = HTML + "<td align='left'>";
    HTML = HTML + "Apellido paterno:";
    HTML = HTML + "</td>";
    HTML = HTML + "<td align='left'>";
    HTML = HTML + "<input type='text' name='apellidoPaterno' id='apellidoPaterno' maxlength='40' size='20'/>";
    HTML = HTML + "</td>";
    HTML = HTML + "</tr>";
    HTML = HTML + "<tr>";
    HTML = HTML + "<td align='left'>";
    HTML = HTML + "Apellido materno:";
    HTML = HTML + "</td>";
    HTML = HTML + "<td align='left'>";
    HTML = HTML + "<input type='text' name='apellidoMaterno' id='apellidoMaterno' maxlength='40' size='20'/>";
    HTML = HTML + "</td>";
    HTML = HTML + "</tr>";
    HTML = HTML + "<tr>";
    HTML = HTML + "<td align='left'>";
    HTML = HTML + "Telefono:";
    HTML = HTML + "</td>";
    HTML = HTML + "<td align='left'>";
    HTML = HTML + "<input type='text' name='telefono' id='telefono' maxlength='40' size='20'/>";
    HTML = HTML + "</td>";
    HTML = HTML + "</tr>";
    HTML = HTML + "<tr>";
    HTML = HTML + "<td align='left'>";
    HTML = HTML + "Celular:";
    HTML = HTML + "</td>";
    HTML = HTML + "<td align='left'>";
    HTML = HTML + "<input type='text' name='celular' id='celular' maxlength='40' size='20'/>";
    HTML = HTML + "</td>";
    HTML = HTML + "</tr>";
    HTML = HTML + "</table>";
    HTML = HTML + "</fieldset>";
    HTML = HTML + "</center>";
    Dialog.confirm(HTML,
                   {top: 140,
                        width:320,
                            height:270,
                        className: "alphacube",
                        onOk: function(win) {
            enviarDatosPrestamo2(objetoJSONin.numeroCuenta,objetoJSONin.idSemestre,objetoJSONin.idTarjeta,$('nombre').
            value,$('apellidoPaterno').value,$('apellidoMaterno').value,$('telefono').value,$('celular').value);
                              return true;
                              },
                        cancel: function() {
                                        return true;
                              },
                              okLabel: "Prestar",
                              cancelLabel:"Cancelar"
                        });
}
```

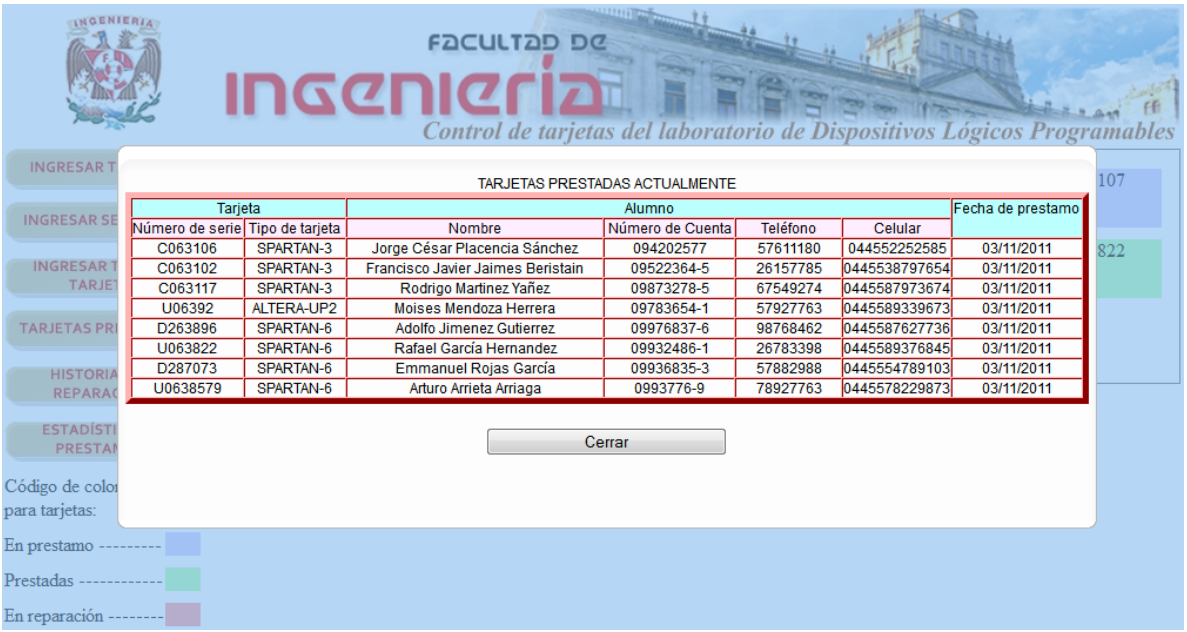
Figura 6.51 La función ventanaIngresoAlumno()

En la función ventanaIngresoAlumno() de una forma similar a la funciones de las figuras 6.37 y 6.47 se deja un tiempo determinado la ventana desplegada, se prepara el HTML que se vi-sualizará en la ventana y se envía la petición asíncrona a la respectiva acción que la procesara. Además de estas funciones, para generar las ventanas de reparar tarjeta, devolver tarjeta, reanudar tarjeta y tarjetas prestadas se realizaron las funciones ventanaReperacionTarjeta(), ventanaDevolverTarjeta(), y ventanaReanudarTarjeta() respec-tivamente.

Para visualizar las ventanas de tarjetas prestadas, historial de tarjetas en reparación y estadísticas de préstamos el Administrador lo que tiene que hacer es dar click en alguna opción del menú vertical (figura 6.52). Básicamente lo que se realiza con estas tres opciones es la lectura de la base de datos.

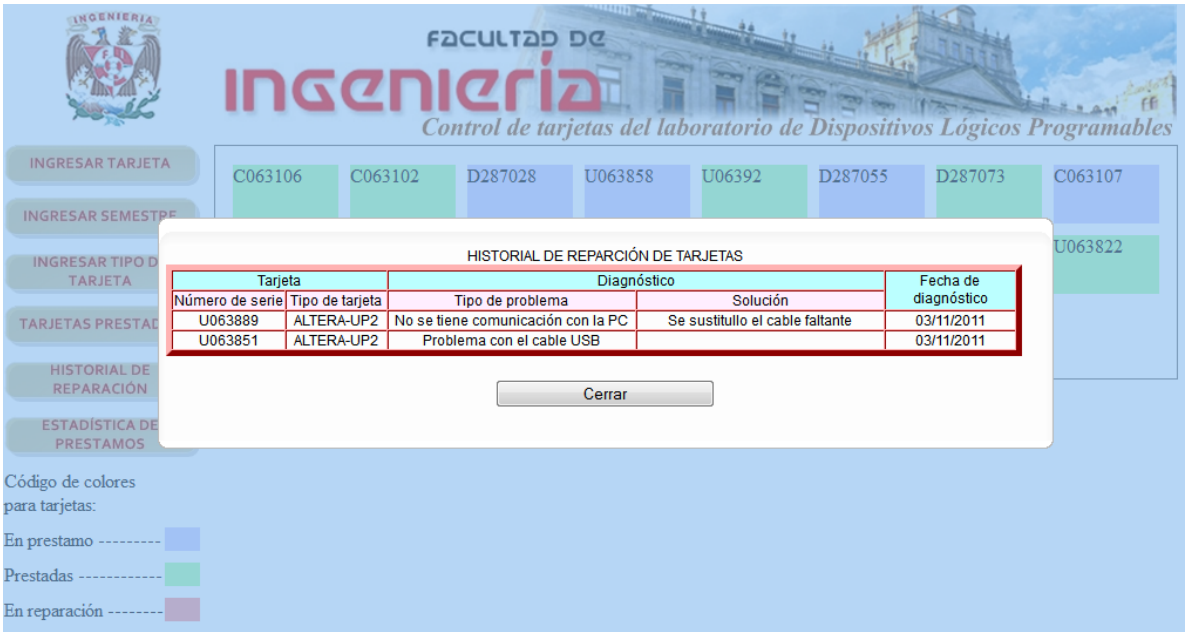
Figura 6.52 Parte del menú vertical: lectura de la base de datos

En las tres opciones de lectura de la base de datos (figuras 6.53, 6.54 y 6.55) en el sistema de control de tarjetas se utilizó la ventana de dialogo alert del marco de trabajo Prototype Window Class.



Tarjeta		Alumno				Fecha de prestamo
Número de serie	Tipo de tarjeta	Nombre	Número de Cuenta	Teléfono	Celular	
C063106	SPARTAN-3	Jorge César Placencia Sánchez	094202577	57611180	044552252585	03/11/2011
C063102	SPARTAN-3	Francisco Javier Jaimes Beristain	09522364-5	26157785	0445538797654	03/11/2011
C063117	SPARTAN-3	Rodrigo Martinez Yañez	09873278-5	67549274	0445587973674	03/11/2011
U06392	ALTERA-UP2	Moises Mendoza Herrera	09783654-1	57927763	0445589339673	03/11/2011
D263896	SPARTAN-6	Adolfo Jimenez Gutierrez	09976837-6	98768462	0445587627736	03/11/2011
U063822	SPARTAN-6	Rafael García Hernandez	09932486-1	26783398	0445589376845	03/11/2011
D287073	SPARTAN-6	Emmanuel Rojas García	09936835-3	57882988	0445554789103	03/11/2011
U0638579	SPARTAN-6	Arturo Arrieta Arriaga	0993776-9	78927763	0445578229873	03/11/2011

Figura 6.53 La ventana de tarjetas prestadas



Tarjeta		Diagnóstico		Fecha de diagnóstico
Número de serie	Tipo de tarjeta	Tipo de problema	Solución	
U063889	ALTERA-UP2	No se tiene comunicación con la PC	Se sustitullo el cable faltante	03/11/2011
U063851	ALTERA-UP2	Problema con el cable USB		03/11/2011

Figura 6.54 La ventana del historial de tarjetas en reparación.

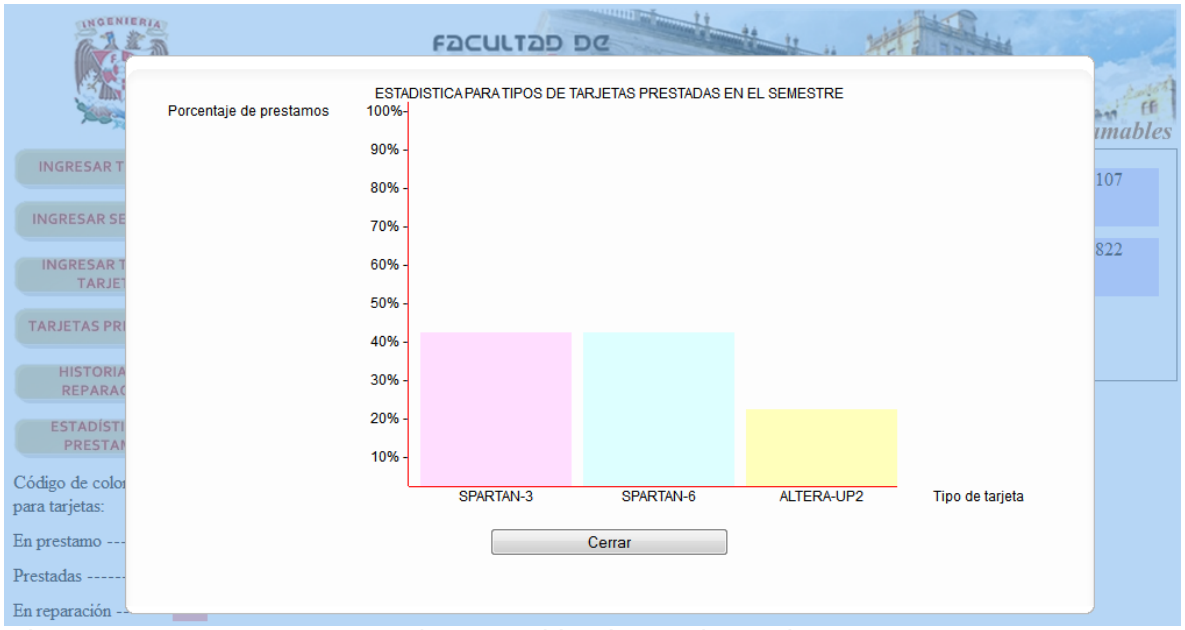


Figura 6.55 La ventana de estadísticas de préstamos

El fragmento del código de la página JSP en el que se llaman a las funciones JavaScript es el siguiente:

```
<td colspan="2" style='cursor: pointer'
onclick="obtenerTarjetasPrestadas()">
<img src="botones/Alumnos_Prestamo.jpg"></img>
</td>

<td colspan="2" style='cursor: pointer'
onclick="obtenerTarjetasReparacion()">
<img src="botones/TarjetasReparacion.jpg"></img>
</td>

<td colspan="2" style='cursor: pointer'
onclick="obtenerEstadistica()">
     <img src="botones/Estadisticas.jpg"></img>
</td>
```

En este fragmento de código se puede observar las funciones JavaScript que se ejecutan al darle click en cada una de las opciones de la figura 6.52. En la figura 6.56 se muestra el código de la funcion obtenerTarjetasPrestadas().

```
function obtenerTarjetasPrestadas() {
	enviarConsultaPrestamos();
}

var obj;

function enviarConsultaPrestamos() {
	obj = new PeticionAsincrona();
	var valores = "";
	obj.enviar("consultarPrestamos.action","procesaResultadoConsultaPrestamos",valores);
}
```

Figura 6.56 La funcion obtenerTarjetasPrestadas()

Lo que realiza la función obtenerTarjetasPrestadas es llamar a la funcion enviarConsultaPrestamos(), la cual crea una instancia de la clase PeticionAsincrona y envía la petición a la acción consultarPrestamos. Si se revisa la acción definida en el archivo de configuración, se obtendrá:

```
<action name="consultarPrestamos" method="consultar"
	class="tarjetas.control.action.PrestamoAction">
</action>
```

En la figura 6.57, se expone el método consultar() de la clase PrestamoAction. Si existen prestamos, además de mostrarse la ventana de la figura 6.53 el código del método generará una cadena textoJSON dinámicamente que dependerá del contenido de la base de datos.

```
public String consultar() throws IOException {
            getResponse().setContentType("text/html;charset=ISO-8859-1");
            setSos(getResponse().getOutputStream());
            String textoJSON = null;

            PrestamoDAO prestamoDAO = new PrestamoDAOImpl();
            Iterator<Prestamo> iter1 = prestamoDAO.consultar("activo").iterator();


            if(!iter1.hasNext()) {

                  // Se notifica al cliente que no existen tarjetas prestadas.
                  textoJSON = "{funcion:'noExistenPrestamos'}";
            } else {

                  StringBuilder json = new StringBuilder("[");

                  while(iter1.hasNext()) {
                        Prestamo prestamo = iter1.next();

                        json.append("{numeroCuenta:'"+prestamo.getAlumno().getNumeroCuenta()+
                                    "',alumno:'"+prestamo.getAlumno().getNombre()+" "+
                                       prestamo.getAlumno().getApellidoPaterno()+" "+
                                       prestamo.getAlumno().getApellidoMaterno()+
                                    "',numeroSerie:'"+prestamo.getTarjeta().getNumeroSerie()+
                                    "',fechaPrestamo:'"+new java.text.SimpleDateFormat("dd/MM/yyyy").
                                       format(prestamo.getFecha())+
                                    "',telefono:'"+prestamo.getAlumno().getTelefono()+
                                    "',celular:'"+prestamo.getAlumno().getCelular()+
                                    "',tipoTarjeta:'"+prestamo.getTarjeta().getTipoTarjeta().getNombre()+
                                    "'},");
                  }

                  // Eliminación de la última coma
                  json.deleteCharAt(json.length() - 1);
                  json.append("]");

                  textoJSON = json.toString();
            }

            getSos().print(textoJSON);
            return null;
      }
```

Figura 6.57 El método consultar() de la clase PrestamoAction

Una vez, que el servidor tenga una respuesta a la petición se ejecutara la funcion de retrollamada procesaResultado-ConsultaPrestamos() publicada en la figura 6.58.

```
function procesaResultadoConsultaPrestamos() {
      var objetoJSONin = eval("("+obj.respuestaTexto()+")");
      if(objetoJSONin.funcion=="noExistenPrestamos") {
            noExistenPrestamos(objetoJSONin);
      } else if (objetoJSONin.funcion=="lanzoExcepcion") {
            lanzoExcepcion(objetoJSONin);
      } else {
      var tabla = "<table border='5' bordercolor=red cellpandding='0' cellspacing='0'>" +
                  "<tr><td colspan='2' class='tdclass1' >Tarjeta</td><td colspan='4'
                        class='tdclass1'>Alumno</td><td rowspan='2' class='tdclass1'
                        valign='top'>Fecha de prestamo</td></tr>" +
                  "<tr><td class='tdclass2'>Número de serie</td><td class='tdclass2'>
                      Tipo de tarjeta</td><td class='tdclass2'>Nombre</td><td class='tdclass2'>
                      Número de Cuenta</td><td class='tdclass2'>Teléfono</td><td class='tdclass2'>
                      Celular</td></tr>";
      for(var i = 0; i < objetoJSONin.length; i++) {
            tabla = tabla+"<tr>" +
                                    "<td width='95'>"+objetoJSONin[i].numeroSerie+"</td>" +
                                    "<td width='85'>"+objetoJSONin[i].tipoTarjeta+"</td>" +
                                    "<td width='220px'>"+objetoJSONin[i].alumno+"</td>" +
                                    "<td width='110'>"+objetoJSONin[i].numeroCuenta+"</td>" +
                                    "<td width='90'>"+objetoJSONin[i].telefono+"</td>" +
                                    "<td width='90'>"+objetoJSONin[i].celular+"</td>" +
                                    "<td width='110'>"+objetoJSONin[i].fechaPrestamo+"</td>" +
                                          "</tr>";
      }
      tabla = tabla+"</table>";

      ventanaSalidaPrestamos(tabla);
      }
```

Figura 6.58 La función de retrollamada procesaResultadoConsulta-Prestamos()

En este código, se obtiene la cantidad de objetos JSON con:

objetoJSONin.length

Y se genera una tabla con los objetos encontrados. Cada atributo de un objeto JSON se colocan en cada columna y cada objeto se coloca en un renglón. En seguida se manda a llamar a la función ventanaSalidaPrestamos() (figura 6.59) la cual se le pasa como parámetro la tabla generada. En este código se utiliza la funcion alert() para crear la ventana de tarjetas prestadas.

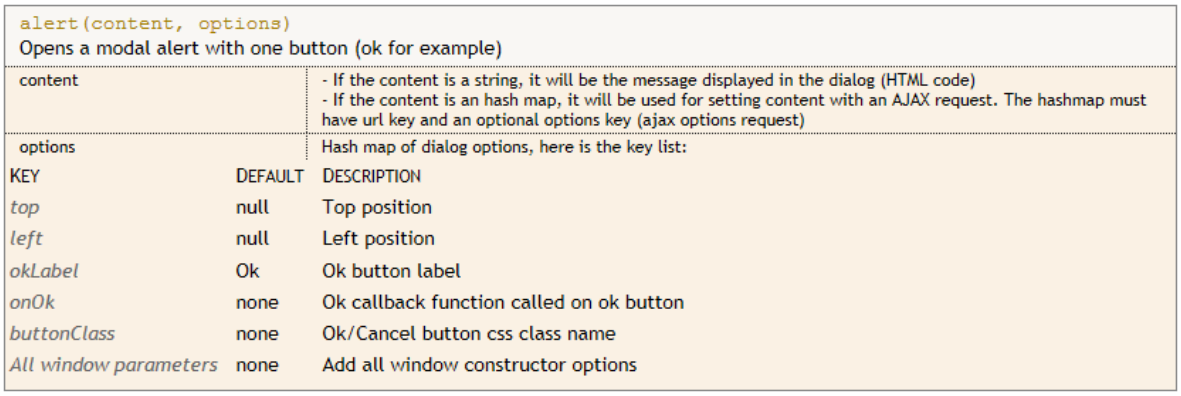
```
function ventanaSalidaPrestamos(htm) {

       var HTML = 'TARJETAS PRESTADAS ACTUALMENTE';
       HTML += htm;

       Dialog.alert(HTML, {className: "alphacube", width:820, okLabel: "Cerrar"});
}
```

Figura 6.59 La función ventanaSalidaPrestamos()

Si se revisa la documentación sobre la función alert(), se encontrará con lo que se muestra la figura 6.60:



alert(content, options)

Opens a modal alert with one button (ok for example)

content		- If the content is a string, it will be the message displayed in the dialog (HTML code) - If the content is an hash map, it will be used for setting content with an AJAX request. The hashmap must have url key and an optional options key (ajax options request)
options		Hash map of dialog options, here is the key list:
KEY	DEFAULT	DESCRIPTION
top	null	Top position
left	null	Left position
okLabel	Ok	Ok button label
onOk	none	Ok callback function called on ok button
buttonClass	none	Ok/Cancel button css class name
All window parameters	none	Add all window constructor options

Figura 6.60 El método alert() en la API Prototype Window Class

De una manera similar a otras funciones del API Prototype Window Class, se le pasa a ésta función el contenido a mostrar en la ventana y las opciones.

Finalmente, para las otras dos opciones del menú de la figura 6.52 su funcionamiento es similar a la concepción de la ventana de tarjetas prestadas.

Conclusiones

El desarrollo del presente trabajo de tesis nos ha permitido aportar una solución para el desarrollo del sistema del control de préstamos de tarjetas del Laboratorio de Dispositivos Lógicos Programables. El cual se desarrolló como una aplicación Web, siendo ésta fácil de entender y manejar debido a que es una aplicación muy visual.

La solución que propusimos consiste en la realización de un modelado y creación de un prototipo para facilitar la realización de un proyecto. Con esta propuesta adquirimos experiencia profesional debido a que aprendimos como organizar el trabajo que involucra la creación de software.

Para lograr el objetivo se utilizo una metodología genérica de desarrollo de software, la cual fue crucial durante el desarrollo del proyecto ayudándonos a realizar el trabajo de un modo sistemático y por actividades, la metodología nos ayudo a definir cuatro áreas fundamentales: la fase de comunicación en la cual obtuvimos las necesidades del cliente, las funcionalidades y sus requisitos; la fase de planeación donde estimamos la cantidad de trabajo a realizar y se pudo tener un seguimiento de lo realizado; la fase de modelado la cual nos ayudo a describir el comportamiento y las propiedades del sistema; y por último, la construcción del sistema utilizando la plataforma Java EE, AJAX y Prototype Windows Class.

Los marcos de trabajo Hibernate y Struts nos permitieron dividir el sistema en varios componentes y que cada uno de ellos tuviera una responsabilidad, lo cual nos ayudo en gran medida, ya que la programación se realizo por bloques y fue posible realizar pruebas intermedias e ir acoplando el sistema hasta llegar al producto final.

El marco de trabajo Hibernate nos permitió la creación de la base de datos en el manejador de base de datos MySQL y no trabajar directamente con el manejador, al utilizar esta API también nos ayudó a realizar la persistencia de objetos, actualizar la persistencia de un objeto y consultar los objetos persistidos. Al realizar las consultas con la API de

Criteria y Query el tiempo en esta actividad es menor que hacerlo con SQL.

La introducción de AJAX a la escena de nuestra aplicación nos ofreció grandes beneficios en las peticiones al servidor, demostró ser una opción viable y con resultados debido a que se obtuvo un software ligero en sus comunicaciones a través de peticiones asíncronas por medio del objeto XMLHttpRequest, logrando así no cargar la página por cada petición realizada al servidor. Por lo que el tiempo de respuesta es eficiente y no tarda en dar respuesta el servidor a una petición.

Con la utilización del marco de trabajo Prototype Windows Class nos permitió la introducción de las ventanas en la aplicación, las cuales se crearon de acuerdo a las necesidades de la aplicación desarrollada, así como los efectos visuales y los cuadros de dialogo, resultando en un proyecto atractivo visualmente. Y debido a que este marco de trabajo se utilizó en conjunto con AJAX, la aplicación da al usuario la sensación de que está utilizando una aplicación de escritorio.

Los marcos de trabajo Hibernate y Struts permiten la reutilización con lo cual la construcción de un sistema resulta ser más fácil por la reutilización de componentes, por lo que se pueden aplicar a nuevos proyectos. Además, se reutilizo el conocimiento de abstracciones exitosas tal como el patrón de diseño Modelo Vista Controlador, el patrón DAO y el patrón Transaction.

Finalmente, el sistema de control de tarjetas consistió en representar cada tarjeta disponible en el sistema como una celda, a la cual se le puede prestar a un alumno, se puede poner en reparación. Y cuando este prestada una tarjeta se puede devolver, en el caso que este en reparación se puede poner en reanudar. También, en el sistema de control se puede leer toda la información de la base de datos sobre las tarjetas, es decir, se puede obtener una pantalla que muestra los datos de las tarjetas que están en reparación, siendo posible desplegar una pantalla que muestra un historial de las tarjetas que se han puesto en reparación y se puede mostrar otra pantalla que despliegue la estadística del tipo de tarjetas más prestadas en el semestre en curso.

Como conclusión general podemos decir que contar con una plataforma de desarrollo estable y unificada (Java EE) permitió integrar todos los módulos del desarrollo que contribuyo en la finalización exitosa del proyecto.

Índice de figuras

Índice de tablas

Referencias

Roger S. Pressman “Ingenieria de Software, un enfoque práctico” McGRAW-HILL Sexta edición 2006

Sommerville “Ingeniería de software” novena edición Pearson 2011

Dante Cantone “La biblia del programador, implementación y debugging” Users.Code 2006

Debrauwer Laurent y Heyde Fien “UML 2 Iniciación, ejemplos y ejercicios corregidos” segunda edición ediciones eni 2009

Rumbaugh, Jacobson y Booch “El Lenguaje Unificado de Modelado, Manuel de referencia” Addison Wesley 2006

Simon Bannett, Steve y Ray “Análisis y diseño orientado a objetos de sistemas usando UML” McGrawHill tercera edición 2007

Silberschatz, Abraham “Fundamentos de Bases de datos” cuarta edición 2002 McGRAW-HILL

Pérez López, César “Oracle 10g Administración y Análisis de Bases de Datos” Alfaomega 2005

Batini Cere, Navathe “Diseño conceptual de base de datos” Addison-Wesley 1992

Hansen, Gary W. “Diseño y administración de base de datos” Prentice Hall, 2ª Edición, págs: 51, 2000.

Ceballos Fco. Javier “Java 2 Curso de Programación” cuarta edición Alfaomega 2003

Thierry Groussard “JAVA 6 Los fundamentos del lenguaje java “ediciones eni 2009

Andrés Martínez Quijano “Programación Web Java” Users.Code 2006

Ceballos Fco. Javier “Java 2 Interfaces gráficas y aplicaciones para internet” segunda edición Alfaomega 2006

Jason Brittain, Ian F. Darwin “Tomcat 6.0 La guía definitiva” Anaya Multimedia 2008

David Roldan Martínez “Aplicaciones Web un enfoque práctico” Alfaomega 2010

Thierry Groussard “Java Enterprise Edition, Desarrollo de aplicaciones Web con JEE 6 “ediciones eni 2010

Christian Bauer Gavin king “Hibernate In Action” Manning 2005

Sharanam Shah, Vaishali “Hibernate 3 for Beginners” The Team 2010

Donald Brown, Chad Davis “Struts 2” Anaya multimedia 2008

Jerome Lafosse “Struts 2 El framework de deserrollo de aplicaciones Java EE” ediciones eni 2010

Sharanam Shah, Vaishali Shah “Struts 2 for Beginners” segunda edicion 2009 The Team

Antonio J. Martín Sierra “Struts” Alfaomega segunda edición 2011

Sharanam Shah, Vaishali “Struts 2 for Beginners” The Team 2009

Gutierrez Emmanuel “JavaScript Conceptos básicos y avanzados” ediciones eni 2009

Antonio J. Martín Sierra “AJAX en J2EE” Alfaomega segunda edición 2011

Paul J. Deitel “AJAX, rRich internet Applications y desarrollo Web para programadores” Anaya Multimedia 2008

Guilherme somera “Entrenamiento Practico de CSS” Serie Pocket 22 2007

Christophe Aubry “CSS 2.1 Adopte las hojas de estilo “ediciones eni 2009

Luc Van Lancker “HTML 4 dominar el código fuente” Tercera edición ediciones eni 2009

http://www.oracle.com/technetwork/java/index.html

http://www.hibernate.org/

http://hibernate.-sourceforge.net/hibernate-mapping-3.0.dtd.

http://docs.jboss.org/hibernate/annotations/3.5/api/org/hibernate/Session.html

http://struts.apache.org/2.x/

http://struts.apache.org/2.0.14/docs/tag-reference.html

http://www.w3.org/dom

http://prototype-window.xilinus.com/documentation.html

http://www.java2s.com/Code/Java/Hibernate/CatalogHibernate.htm