Universidad de las Ciencias Informáticas

Facultad 3

Implementación de la capa de persistencia de datos de los módulos de Presentación y Contratación del proyecto Convenio Integral de Cooperación Cuba Venezuela **CICCV.**

Trabajo de Diploma para optar por el título de Ingeniero en Ciencias Informáticas

Autores

Alberto López Milán.

Edgardo Juan Chung La Rosa

Tutor

Ing. Dalgis Rogelio López Góngora

Ciudad de la Habana, Cuba

Mayo, 2009

Agradecimientos

De Alberto

A:

La Revolución y a nuestro Comandante en Jefe Fidel Castro Ruz porque sin su visión y guía no hubiese sido posible nada de esto.

Mis padres Alberto y Marcia. E que son mi razón de existir, por guiarme en todo momento hacia un camino correcto, por estar siempre a mi lado en todos los momentos brindándome su apoyo y amor.

Mi hermana Elenita que la quiero muchísimo y siempre ha estado a mi lado.

Mi abuela Miriam (abue), que ha sido mi segunda madre, siempre se ha mantenido a mi lado brindándome su apoyo y cariño.

Mi tía Miriam. E (mi madrina), por aconsejarme en todo momento, por quererme mucho y tenerme presente siempre.

Mis tíos que los quiero mucho a todos y siempre han estado apoyándome en lo que me ha hecho falta.

Mi primo Carlos Miguel y su familia que siempre me han dado su apoyo.

Mi pareja que ha estado a mi lado durante algún tiempo y ha sabido aconsejarme y compartir conmigo todo tipo de momentos.

Mis compañeros de grupo y profesores.

Mis compañeros de proyecto, mi tutor y mi compañero de tesis.

Toda mi familia y todas mis amistades.

Agradecimientos

De Edgardo

A:

Mis padres Eduardo y Ofelia por la educación recibida, el cariño y por estar en los momentos en que los he necesitado.

Mis hermanos

Mis tíos y abuelos que los quiero muchísimo

Mi familia en general

Mis compañeros de grupo

Los amigos y amigas que casi considero como hermanos y que han estado en los momentos buenos y malos.

Los profes que me han acompañado durante mi vida escolar

Mi tutor y los desarrolladores del proyecto

Dedicatoria

De Alberto

A:

Mis padres, mi hermana y abuela Miriam (abue).

Mi familia.

Mis compañeros y amigos.

Dedicatoria

De Edgardo

A:

Mis padres

Mi familia

Mis amigos

Resumen

En este trabajo se describe el proceso de implementación de la capa de persistencia de datos de los módulos de Presentación y Contratación del Convenio Integral de Cooperación Cuba-Venezuela, utilizando para la interacción objeto-relacional las funcionalidades que brinda el framework hibernate 3.2, guiado por la metodología de desarrollo Rational Unified Process (RUP), el Visual Paradigm como herramienta de modelado del ciclo de desarrollo, Java como lenguaje de programación incluyendo la máquina virtual JDK 6.2 , Eclipse 3.3 como Entorno de Desarrollo Integrado y subeclipse para la integración con el repositorio de versiones. De esta forma se logra la gestión y persistencia de los datos a través de la vinculación del paradigma orientado a objetos de la aplicación y el paradigma utilizado por las bases de datos relacionales.

Indice

INTRODUCCION

La República de Cuba y la República Bolivariana de Venezuela con el afán de fortalecer los lazos de amistad entre ambos países y conscientes de su interés común por fomentar el progreso de sus respectivas economías han iniciado una serie de programas y proyectos de colaboración con el fin de obtener resultados efectivos en el ámbito económico y social de los respectivos países y fomentar la integración de América Latina y el Caribe. Para lograrlo se han concebido espacios para presentar dichos proyectos denominados Mixtas.

Los proyectos pueden ser propuestos en las Mixtas por entes ejecutores (EE) o por ministerios (Min), cubanos o venezolanos, pero para que se realice este proceso, como en la mayoría de éstos, ambas entidades deben ponerse de acuerdo. Esto resulta un problema teniendo en cuenta que las formas de comunicación, ya sean llamadas telefónicas o correo electrónico, no tienen la misma efectividad que una visita presencial puesto que esta última brinda un mayor nivel de detalle y de entendimiento de los procesos. Debido a esto, la forma en que se manipula la información de los proyectos no garantiza el control y seguimiento de sus datos, situación que resulta insostenible teniendo en cuenta que la República Bolivariana de Venezuela invierte anualmente varios millones en estos proyectos.

Esta situación determinó que dentro del marco de la 8va mixta se propusiera como unos de sus proyectos la creación de un sistema capaz de controlar la gestión de los procesos de la información de los mismos con el fin de evitar la pérdida de recursos. Luego de un estudio y debido a la importancia y la experiencia de la Universidad de las Ciencias Informáticas en el desarrollo de software se decidió darle la responsabilidad de ofrecer soporte tecnológico a este problema. Se resolvió crear un software que respondiera a las necesidades antes expuestas, por lo que se acordó insertarlo dentro del inmenso grupo de softwares colaborativos y dentro de este, como software de gestión.

Así comenzó el desarrollo del producto utilizando un conjunto de herramientas de avanzada, tal es el caso, de Visual Paradigm como herramienta de modelado del ciclo de desarrollo, Java2 Enterprise Edition como lenguaje de programación incluyendo la maquina virtual JDK 6.2 y como servidor de aplicaciones web Apache Tomcat, debido a su compatibilidad y adaptabilidad con el ambiente del lenguaje utilizado , Eclipse 3.3 como Entorno de Desarrollo Integrado(IDE), subeclipse para la integración con el repositorio de versiones, entre otras. Estas herramientas fueron el soporte automático

entre la forma de construir el software y el fundamento de la Ingeniería de Software como bien define el Proceso Unificado de Desarrollo (RUP); metodología utilizada para desarrollar el producto y que le dedica especial atención al control y aseguramiento de la calidad del producto, a la documentación de sus procesos, que facilita el ajuste del proceso de desarrollo a las necesidades del proyecto y que se encuentra dividido por fases y flujos de trabajos. Es precisamente en la fase implementación, luego de haber capturado los requisitos y haber realizado el análisis y diseño, donde se implementan, integran y prueban los elementos provenientes desde el diseño. De esta forma se van creando componentes como bases de datos, clases y subsistemas de implementación.

En la actualidad es una necesidad el uso de base de datos (BD) debido a que garantizan la independencia de los datos, mejora su disponibilidad y posee una gestión de almacenamiento bastante eficiente comparado con las aplicaciones orientadas a objetos que mantienen la persistencia de los datos mientras la sesión del servidor de aplicación no sea cerrada. La mayoría de las BD son relacionales, debido a esto no existe una forma directa que le permita comunicarse con estas aplicaciones, lo que ha traído como consecuencia la búsqueda de alternativas y herramientas para vincular el mundo objetual y el relacional.

La agrupación de clases en subsistemas y la aplicación de una arquitectura en capas garantizan una amplia reutilización, optimizaciones, refinamientos y la independencia entre los programadores de las capas definidas. Resulta esencial el uso de una de estas capas para garantizar la persistencia de los datos de la aplicación en bases de datos relacionales realizando acciones como insertar, modificar, eliminar, realizar consultas que permitan responder a los reportes del sistema, y que a la vez sirva de entrada con la capa de negocio, interactuando directamente con esta.

Actualmente, en la capa de persistencia de datos(de persistencia objeto/relacional o de acceso a datos) de los módulos Presentación y Contratación del proyecto CICCV se hace muy complicado vincular el mundo orientado a objetos con el mundo relacional, debido a que existen diferencias en cuanto al manejo de los datos. Consecuentemente es necesario implementar la capa de persistencia de datos utilizando una herramienta que sirva como enlace entre estos dos paradigmas de programación y que a su vez proporcione seguridad en los datos.

Problema científico de la investigación

¿Cómo desarrollar la capa de persistencia de datos de los módulos de Presentación y Contratación del proyecto CICCV para facilitar el vínculo entre el mundo objetual y el

relacional, lograr una mayor calidad en la gestión y persistencia de los datos y garantizar entradas esperadas por la capa de lógica del negocio?

Objeto de estudio

Proceso de Desarrollo de Software

Campo de acción

Capa de persistencia de datos de los módulos de Presentación y Contratación del proyecto CICCV.

Objetivo

Implementar la capa de persistencia de datos de los módulos de Presentación y Contratación del proyecto CICCV utilizando una herramienta que permita el enlace entre el mundo objetual y el relacional.

Hipótesis

Si se implementa la capa de persistencia de datos para los módulos de Presentación y Contratación del proyecto CICCV a través del uso de la herramienta adecuada, se facilitará el vínculo entre el mundo objetual y el relacional, logrando una mayor calidad en la gestión y persistencia de los datos y ofreciendo las entradas esperadas por la capa de lógica del negocio.

Tareas de investigación

1. Estudiar el estado del arte de la disciplina de implementación.
    1.1. Estudiar las potencialidades del lenguaje de programación y el entorno de desarrollo a utilizar.
    1.2. Estudiar los patrones de implementación que puedan ser de utilidad en la implementación de la capa de acceso a datos.
    1.3. Analizar las herramientas candidatas para vincular el mundo objetual y el relacional.
    1.4. Seleccionar la herramienta para vincular el mundo objetual y el relacional.
    1.5. Analizar las habilidades y métodos que permitan disminuir el tiempo de implementación y producir programas de alta calidad

1.6. Estudiar herramientas que permitan realizar pruebas de unidad a los componentes de la capa de persistencia.

2. Desarrollar la solución propuesta.
   2.1. Desarrollar el modelo de componentes de la capa de persistencia
   2.2. Implementar los componentes de las capa de persistencia
   2.3. Aplicar patrones de implementación y algoritmos con complejidades aceptables.
   2.4. Aplicar las habilidades y métodos que permitan disminuir el tiempo de implementación y producir programas de alta calidad.
3. Validar de la solución propuesta.
   3.1. Aplicar pruebas de unidad a los componentes de la capa de persistencia.

Estructura del trabajo

El capítulo 1 abarca el estudio del arte de (objeto) y de la implementación de la capa de acceso a datos, el análisis de las facilidades de las herramientas, tecnologías, plataformas, lenguajes disponibles y de las bondades los paradigmas de programación existentes.

El capítulo 2 comprende la propuesta de solución al problema de la investigación. Esta encierra la implementación de los componentes de la capa de acceso a datos usando patrones de implementación, técnicas de programación y métodos que permitieron implementar componentes con calidad.

El capítulo 3 incluye la aplicación de pruebas de unidad a los componentes previamente implementados en la capa de acceso a datos.

Métodos científicos

Histórico-Lógico: Permitió conocer los antecedentes de las herramienta y tecnologías utilizadas para implementar la capa de acceso a datos en sistemas con características similares y analizar sus ventajas y desventajas. Conociendo su desarrollo histórico y las directrices, fue de gran ayuda para seleccionar la forma correcta de persistir los datos

Analítico-Sintético: Fue útil en el estudio del estado del arte de los sistemas de colaboración y de la implementación de la capa de persistencia, puesto que para

procesar el gran volumen de información en las actividades anteriores se necesitó descomponerla mediante el pensamiento en conceptos abstractos y sus relaciones, a través del análisis, y luego a través de la síntesis crear nueva información concreta, resumiendo en ella la características generales y sus relaciones.

Métodos empíricos

Observación: Permitió recoger información de las variables definidas en la hipótesis de forma consciente y objetiva.

Experimentación: Para realizarle pruebas de integración a los componentes, previamente integrados, de la capa de persistencia de datos para validar la solución propuesta corrigiendo los errores que puedan encontrarse y de esta forma comprobar la validez de la hipótesis.

CAPITULO 1. Fundamentación Teórica

1.1 Introducción

En este capítulo se analizan las características de algunas de las metodologías de desarrollo de software más utilizadas. Conjuntamente se efectúa un análisis de los paradigmas, lenguajes de programación y las herramientas que puedan proporcionar un producto que responda a las necesidades del cliente.

Además se realiza un estudio de las potencialidades de la plataforma de programación utilizada para vincularse con las herramienta de persistencia objeto/relacional y se selecciona el estándar de codificación a utilizar durante la implementación de los componentes.

A partir de ahí, se examinan los principales esquemas de persistencia y se propone el framework Hibernate como la solución a la persistencia de objetos en Java, logrando el vínculo entre el mundo relacional de las bases de datos y el mundo objetual de las aplicaciones orientadas a objetos, utilizando Data Access Object (DAO) como patrón de diseño en la capa de acceso a datos, por otra parte como mecanismo de validación se utiliza JUnit para implementar y automatizar las pruebas de unidad.

1.2 Metodología de desarrollo de software

En la concepción de cada proyecto es de gran importancia realizar un buen proceso de desarrollo de software. Un proceso de desarrollo de software es el conjunto de actividades necesarias para transformar los requisitos de un usuario en un sistema de software (**Ver Figura 1**). El proceso de desarrollo de software requiere por un lado un conjunto de conceptos, una metodología y un lenguaje propio.

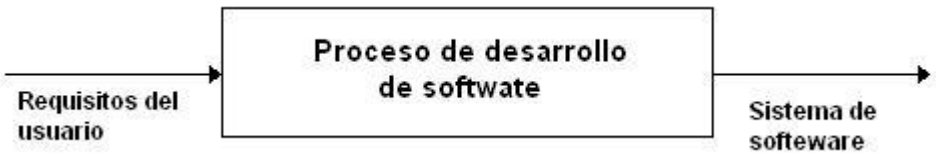


Figura 1. Proceso de desarrollo de software

Una metodología de desarrollo de software es un conjunto de pasos y procedimientos que deben seguirse para desarrollar software (Mendoza Sanchez, 2004).

Normalmente consiste en un conjunto de fases descompuestas en subfases (módulos, etapas, pasos, etc.). Esta descomposición del proceso de desarrollo guía a los desarrolladores en la elección de las técnicas que debe elegir para cada estado del proyecto, y facilita la planificación, gestión, control y evaluación de los proyectos. Dentro de las metodologías más utilizadas se encuentra Rational Unified Process (RUP).

RUP

RUP es el resultado de varios años de desarrollo y uso práctico en el que se han unificado técnicas de desarrollo, a través del trabajo de muchas metodologías utilizadas por los clientes. Fue creado por Jacobson, Rumbaugh y Booch, se inserta dentro de las metodologías orientadas a objetos, es una opción a tener en cuenta para desarrollar grandes y complejos proyectos.

RUP es un proceso de desarrollo de software. Sin embargo, el Proceso Unificado es más que un simple proceso; es un marco genérico que puede especializarse para una gran variedad de sistemas software, para diferentes áreas de aplicación, diferentes tipos de organizaciones, diferentes niveles de aptitud y diferentes tamaños de proyecto (Jacobson, et al., 2000)

RUP define **quién** (trabajadores) hace **qué** (artefactos), **cómo** los hace (Actividades) y **cuándo** los hace (flujo de actividades). Además está basado en componentes, lo cual quiere decir que el sistema software en construcción está formado por componentes interconectados a través de interfaces bien definidas. RUP utiliza el Lenguaje Unificado de Modelado (UML) para preparar todos los esquemas de un sistema software. De hecho UML es una parte esencial de él. (Jacobson, y otros, 2000)

Sin embargo, los rasgos que caracterizan el ciclo de vida de RUP son:

Dirigido por casos de uso: Los casos de uso reflejan lo que los usuarios futuros necesitan y desean, lo cual se capta cuando se modela el negocio y se representa a través de los requerimientos. Sin embargo, los casos de uso no son sólo una herramienta para especificar los requisitos de un sistema. También guían su diseño,

implementación, y prueba; esto es, guían el proceso de desarrollo. (Jacobson, y otros, 2000)

Centrado en la arquitectura: La arquitectura muestra la visión común del sistema completo en la que el equipo de proyecto y los usuarios deben estar de acuerdo, por lo que describe los elementos del modelo que son más importantes para su construcción. La arquitectura le da la forma necesaria al sistema para que los casos de uso aporten las funcionalidades requeridas por el usuario, con la necesaria adaptabilidad a cambios futuros. Además, aparte del estudio de la plataforma, de los protocolos para comunicación en red y del sistema de gestión de base de datos requeridos para que el sistema funcione, la arquitectura se desarrolla a partir de los caso de uso significativo, por lo que existe una fuerte relación entre ellos

Iterativo e Incremental: como sugiere Dijkstra, la técnica de dominar la complejidad se conoce desde tiempos remotos: divide et impera (divide y vencerás) (Dijkstra, 1979). De esta forma RUP divide un proyecto de software en partes más pequeñas o miniproyectos. Cada miniproyecto es una iteración que resulta en un incremento. Las iteraciones hacen referencia a pasos en los flujos de trabajo, y los incrementos, al crecimiento del producto. Cada iteración se realiza de forma planificada es por eso que se dice que son miniproyectos. (Jacobson, y otros, 2000) En cada iteración se realizar un proceso de gestión de riesgos, garantizando así que los costes por algún problema afecten sólo a la iteración y no al producto completo.

RUP define nueve flujos de trabajo, seis ingenieriles y tres de apoyo, a la vez estos flujos se encuentran relacionados de forma bidimensional con cuatro fases. El ciclo de vida de un producto de software en RUP está determinado por las iteraciones y los hitos de cada fase (**Ver Figura 2**).

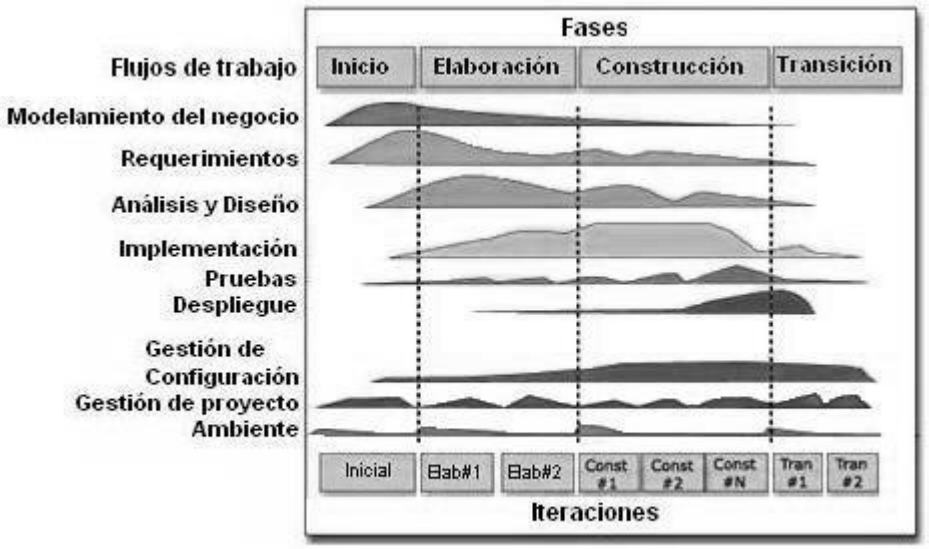


Figura 2. Gráfica bidimensional entre flujos de trabajo y fases de RUP

En RUP, cada ciclo produce una nueva versión del sistema, y cada versión es un producto preparado para su entrega. Consta de su cuerpo de código fuente incluido en componentes que puede compilarse y ejecutarse, además de manuales y otros productos asociados. Sin embargo, el producto terminado no sólo debe ajustarse a las necesidades de los usuarios, sino también a las de todos los interesados, es decir. Toda la gente que trabajara con el producto. El producto software debería ser algo más que le código maquina que se ejecuta. (Jacobson, y otros, 2000)

Así, a través de la abundante documentación y los artefactos que RUP ofrece, se hace más fácil afrontar futuros cambios en los requisitos, en el sistema operativo, base de datos u otros. De esta forma los mismos desarrolladores o incluso otros desarrolladores podrán partir de las funcionalidades los casos de usos ya descritos, fusionar los cambios y darle forma al sistema. Resulta un proceso menos costoso que si se tiene que rehacer el sistema desde el comienzo

Por otra parte el Proceso Unificado de Desarrollo realiza una mitigación temprana de posibles altos riesgos, sostiene un progreso visible en las primeras etapas, posee una temprana retroalimentación que se ajusta a las necesidades reales, manteniendo la gestión de la complejidad de los proyectos y hace posible que el conocimiento adquirido en una iteración puede aplicarse de iteración a iteración. (Díaz, et al., 2008)

El flujo de trabajo de implementación abarca la definición de la organización del código en términos de subsistemas de implementación organizados en capas, la implementación de los elementos de diseño en términos de elementos de

implementación (códigos fuentes, ejecutables, entre otros), la prueba de los componentes desarrollados como unidades y la integración de los resultados de los programadores o equipos de programadores, en un sistema ejecutable. Es precisamente, el rol del Programador, el encargado de implementar y probar componentes de acuerdo con los estándares definidos en el proyecto, para la integración en subsistemas más amplios (RUP, 2003) .

Este flujo de trabajo cuenta con varios roles, entre ellos el rol de implementador. El implementador debe tener conocimiento del paradigma y el lenguaje de programación escogidos, así como dominar el estándar de codificación que más se ajusta a este lenguaje. Además debe conocer con profundidad las potencialidades del IDE de desarrollo y saber aplicarlas a la hora de implementar. Debe dominar la programación utilizando una estructura de soporte definida con bibliotecas y librerías (frameworks) que le puedan ayudar a implementar los componentes que le corresponden. Asimismo el programador debe dominar los pasos para implementar los elementos de diseño necesarios para obtener componentes ejecutables que proporcionen funcionalidad al sistema. Además necesita saber cómo llevar los patrones modelados en el diseño a la implementación si desea obtener código reusable y legible. Paralelamente debe probar la implementación utilizando herramientas que permitan realizar pruebas de unidad a su código. También el programador debe llevar a cabo prácticas como el Proceso de Software Personal (PSP) que le permitan planificar su trabajo, detectar y corregir errores, y de esta forma lograr una disminución del tiempo en la implementación.

1.3 Paradigmas de programación

Un paradigma representa las directivas en la creación de abstracciones, y es un principio por el cual un problema puede ser comprendido y descompuesto en componentes manejables. Un paradigma fija las reglas y propiedades, pero también ofrece herramientas para el desarrollo de aplicaciones (Paola Pérez, et al., 2007)

Bobrow y Stefik definen un estilo de programación como «una forma de organizar programas sobre las bases de algún modelo conceptual de programación y un lenguaje apropiado para que resulten claros los programas escritos en ese estilo.

(Bobrow, et al., 1986) Sugieren además que hay cinco tipos principales de estilos de programación, que se listan aquí con los tipos de abstracciones que emplean:

- ***Orientados a procedimientos:*** Algoritmos.
- ***Orientados a objetos:*** Clases y objetos
- ***Orientados a lógica:*** Objetivos, a menudo expresados como cálculo de predicado
- ***Orientados a reglas:*** Objetivos, a menudo expresados como cálculo de predicados
- ***Orientados a restricciones:*** Relaciones invariantes

Por otra parte Zarate dice que los principales son imperativa; orientada a objetos; funcional, y lógica. (Zárate Rea, 2008)

Cada uno de esos estilos de programación se basa en su propio marco de referencia conceptual. Cada uno requiere una actitud mental diferente, una forma distinta de pensar en el problema. Para todas las cosas orientadas a objetos, el marco de referencia conceptual es el modelo de objetos. Hay cuatro elementos fundamentales en este modelo (Booch, 1996)

1.3.1 Paradigmas a comparar

Paradigma Imperativo

Los primeros lenguajes de programación (código maquina, ensamblador y de alto nivel) se basaron sobre el paradigma imperativo, que consistía en una secuencia de comandos/sentencias con los que se operan los datos almacenados en memoria. Uno de los aspectos más remarcables de la programación imperativa es el mecanismo de *side effect* realizado en la sentencia de asignación. Esta sentencia cambia el estado del programa alterando el contenido de las posiciones de memoria (Paola Pérez, et al., 2007) La programación imperativa consiste en una serie de comandos que una computadora ejecutará. Estos comandos detallan de forma clara y específica el cómo

hacer las cosas y llevarán al programa a través de distintos estados. (Zárate Rea, 2008)

Este paradigma posee algunos elementos fundamentales como variables, tipos de datos y expresiones. Excepto por los paradigmas lógico y funcional, todos los demás heredan muchas características de la programación imperativa. (Zárate Rea, 2008)

Con el tiempo se sumaron otros paradigmas a este estilo para facilitar la construcción de programas, entre ellos el *orientado a objetos.*

Paradigma Orientado a Objetos (POO)

Este estilo usa el modelo orientado a objetos y consta de cuatro elementos fundamentales: *abstracción, encapsulamiento, modularidad y jerarquía*. Estos elementos son necesarios para el estilo.

Una ***abstracción*** denota las características esenciales de un objeto que lo distinguen de todos los demás tipos de objetos y proporciona fronteras conceptuales nítidamente definidas respecto a la perspectiva del observador. (Booch, 1996). Para explicar algunas de sus características se defina a un *cliente* como cualquier objeto que usa los recursos de otros objetos (denominado *servidor*). La vista exterior del servidor ofrece servicios al cliente, estos servicios son llevados a cabo por la vista interior del propio servidor. Las operaciones que puede realizar un cliente sobre un objeto, junto con las formas de invocación u órdenes que admite se le denomina *protocolo.* Este denota las formas en que un objeto puede actuar y reaccionar, constituyendo la vista externa completa, estática y dinámica de la abstracción. Conjuntamente se encuentra la *invariancia* que son las condiciones que deben cumplir tanto el cliente como el servidor para que se mantenga la confianza de uno sobre otro y se mantenga la relación que existe entre ambos. Asimismo las abstracciones tienen propiedades estáticas y dinámicas, por ejemplo la propiedad nombre de un objeto y su valor respectivamente. Así se actúa sobre el nombre y cambiar su valor, generando una reacción del mismo.
La abstracción se centra en la visión externa y el comportamiento observable de un objeto, de la implementación que da lugar a este comportamiento se encarga el ***encapsulamiento***. (Booch, 1996) Según Liskov para que la abstracción funcione debe estar encapsulada, es decir, cada clase debe tener una interfaz que capture sólo su vista externa abarcando la abstracción que se ha hecho del comportamiento común de

todas sus instancias y la implementación que comprende los mecanismos que consiguen su comportamiento. (Liskov, 1988) Es posible además dividir un programa en partes más pequeñas logrando una disminución de la complejidad y la creación de una serie de fronteras bien definidas y documentadas dentro del programa que facilitan su comprensión. *La **modularidad** es la propiedad que tiene un sistema que ha sido descompuesto en un conjunto de módulos cohesivos y débilmente acoplados*. (Booch, 1996) De esta forma se lograr implementar y probar los módulos de formas separadas, reduciendo los costes de tiempo del software. Sin embargo, en un programa generalmente existen muchas abstracciones y es necesario organizarlas, para esto la programación orientada a objetos propone la ***jerarquía***. La jerarquía puede manifestarse a través de la herencia, que no es más que una relación entre clases, en la que una clase comparte la estructura de comportamiento definida en una o más clases (lo que se denomina herencia simple o herencia múltiple, respectivamente) representando una jerarquía de abstracción en la que una subclase hereda de una o más superclases y en la que distintivamente una subclase aumenta o redefine la estructura y el comportamiento de sus superclases, (Booch, 1996) la combinación de herencia con agregación(otra forma de jerarquía) es potente: la agregación permite el agrupamiento físico de estructuras relacionadas lógicamente, y la herencia permite que estos grupos de aparición frecuente se reutilicen con facilidad en diferentes abstracciones.

Además existen otros aspectos que son opcionales en este estilo. Los ***tipos*** son la puesta en vigor de la clase de los objetos, de modo que los objetos de tipos distintos no pueden intercambiarse o, como mucho, pueden intercambiarse sólo de formas muy restringidas. Un lenguaje de programación determinado puede tener comprobación estricta de tipos, comprobación débil de tipos, o incluso no tener tipos, y aún así ser considerado como orientado a objetos. En POO, los objetos (extraídos de una abstracción del mundo real) que representan un hilo separado de control (una abstracción de un proceso) son llamados activos, y es necesario tener mecanismos que garanticen exclusión mutua si dos objetos activos intentan enviar mensajes a un tercer objeto de forma que el estado del objeto sobre el que se actúa no está corrupto cuando los dos objetos activos intentan actualizarlo simultáneamente. Este es el punto en que se garantiza el control de la ***concurrencia***. Al mismo tiempo un objeto puede o no sobrevivir a la ejecución de un determinado programa, puede incluso continuar viviendo después que su creador deja de existir y/o si su posición varía con respecto al espacio de direcciones en el que fue creado, esta propiedad se llama ***persistencia.***

Paradigma Funcional

Este considera al programa como una función matemática, donde el dominio representaría el conjunto de todas las entradas posibles (inputs) y el rango sería el conjunto de todas las salidas posibles (outputs). Se basa en la teoría de funciones recursiva de computación. (Zárate Rea, 2008)

Como sus lenguajes no soportan el concepto de variables tampoco existen operaciones de asignación. Además una función sólo depende de sus parámetros y tiene efecto únicamente en su resultado, por lo que podríamos llamar a una función arbitrariamente sin tener efectos colaterales en el resto de las computaciones. Por otra parte, una función puede ser utilizada como parámetros y resultados de cualquier otra función

La programación funcional también presenta similitudes con la programación orientada a objetos, provee tipos de datos abstractos, clases, encapsulamiento, subtipos y estructuras básicas. (Paola Pérez, et al., 2007)

Este estilo permite definir una amplia variedad de estructuras de datos de uso genérico, la posibilidad de definir funciones que aceptan otras funciones como argumentos y devuelven funciones como resultado, facilidades para definir y manipular estructuras de datos complejas, un modelo computacional simple, claro y bien fundamentado, datos potencialmente infinitos, etc.

De no menor importancia es la posibilidad de razonar, de forma sencilla, acerca de las propiedades de los programas. Sin embargo no se encuentra estandarizado y no posee un alto rendimiento en los programas.

Paradigma Lógico

Es un paradigma bastante diferente a los demás. No sólo en su sintaxis o semántica, sino que en él la lógica representa conocimiento, el cual es manipulado mediante inferencias. A diferencia de los demás paradigmas, trabajar en este significa especificar qué hacer y no cómo hacerlo, por ello son llamados *lenguajes declarativos*.

El proceso general de la programación lógica es que a partir de un conjunto de reglas (axiomas) e inferencias podamos comprobar nuevas proposiciones que nos sean relevantes. Este proceso está basado en reglas de lógica de primer orden. Aunque Prolog es el lenguaje más representativo en este paradigma, también se puede insertar Structured Query Language (SQL) dentro del estilo. Por ejemplo en una consulta SQL, se declara únicamente que es lo que se quiere hacer con los datos en

la base de datos, pero no se define como, cuantas veces iterar alguna instrucción, especificaciones para hacer cada comparación, etc. (Zárate Rea, 2008)

1.3.2 Selección del/los paradigma(s) de programación a utilizar

No hay un estilo de programación que sea el mejor para todo tipo de aplicaciones. *Por ejemplo, la programación orientada a reglas sería la mejor para el diseño de una base de conocimiento, y la programación orientada a procedimientos sería la más indicada para el diseño de operaciones de cálculo intensivo. Por nuestra experiencia, el estilo orientado a objetos es el más adecuado para el más amplio conjunto de aplicaciones; realmente, este paradigma de programación sirve con frecuencia como el marco de referencia arquitectónico en el que se emplean otros paradigmas*. (Booch, 1996)

Después de haber profundizado en las características de algunos de los estilos de programación se decidió utilizar dos estilos de programación:

Lógico

- Es el más indicado para desarrollar las consultas HQL puesto que a través de este se declara únicamente que es lo que se quiere hacer con los datos en la base de datos, pero no se define cómo, cuantas veces iterar alguna instrucción, ni las especificaciones para hacer cada comparación.

Orientado a objetos

- No existe necesidad de realizar operaciones de cálculo intensivo por lo que se excluye la posibilidad de usar el paradigma funcional.
- El paradigma orientado a objetos es un tipo de paradigma imperativo que evolucionó para enfrentarse a software más complejo, debido a esto se prescinde la posibilidad de utilizarlo
- Permite la reutilización, el aumento de la comprensión del programa, facilidad en el mantenimiento del código, bajo acoplamiento, alta cohesión,

independencia del código a la hora de implementar, probar o recompilar un cambio.(a través de la ***modularidad***)

- Posibilita el ahorro y la claridad en el código realizando acciones comunes e individuales según la necesidad.(a través de la ***herencia***)
- Permite un mayor grado de mantenibilidad y reusabilidad.(a través del ***polimorfismo*** y el ***encapsulamiento***)

1.4 Lenguajes de programación

En los tiempos actuales, donde la empresa del software toma auge y la producción de productos de software ha incrementado su demanda, la programación y la creación de software (lenguajes) que permitan traducir procesos de la vida real al lenguaje de las computadoras resultan de gran importancia en la síntesis de la información y el agilizamiento de los procesos sistemáticos de los sistemas que trabajan por medio de estas.

Los lenguajes de programación forman un subconjunto de los lenguajes formales. Mediante ellos el ser humano puede comunicarse con una computadora y decirle qué es lo que tiene que hacer y, al mismo tiempo, sirven para comunicarse con otros seres humanos y expresarles qué es lo que se quiere que haga una computadora (Zárate Rea, 2008) Además, son un conjunto de reglas, herramientas y condiciones que nos permiten crear programas o aplicaciones dentro de una computadora. Estos programas son los que permiten ordenar distintas acciones a la computadora en un *"idioma"* comprensible por esta. Como su nombre lo indica, un lenguaje tiene su parte *sintáctica* y su parte *semántica*, es decir, todo lenguaje de programación posee reglas acerca de cómo se deben escribir las sentencias y de qué forma. A su vez, los lenguajes de programación se dividen en tres grandes grupos: los ***lenguajes de máquina***, los de ***bajo nivel*** y los de ***alto nivel*** (Bonanata, 2003)**.** Estos grupos están dados por el *nivel de abstracción*.

Según la *manera de ejecutarse* los lenguajes pueden ser clasificados en:

- ***Lenguajes compilados***: Un programa traductor traduce el código del programa (código fuente) en código máquina (código objeto). Otro programa, el enlazador, unirá los ficheros de código objeto del programa principal con los de las librerías para producir el programa ejecutable. Ejemplo: C.

- ***Lenguajes interpretados***: Un programa (intérprete), ejecuta las instrucciones del programa de manera directa. Ejemplo: Lisp
- También los hay ***mixtos***, como Java, que primero pasan por una fase de compilación en la que el código fuente se transforma en "bytecode", y este "bytecode" puede ser ejecutado luego (interpretado) en ordenadores con distintas arquitecturas (procesadores) que tengan todos instalados la misma "Maquina Virtual Java"

Por otra parte los programas pueden agruparse en paradigmas o estilos de programación (Bobrow, et al., 1986) explicados en el epígrafe anterior. Existe una amplia gama de lenguajes que se encuentran agrupados dentro del estilo orientado a objetos, entre los más utilizados se encuentran C#, PHP y Java.

1.4.1 Lenguajes a comparar

C#

C# es un lenguaje de programación que toma las mejores características de lenguajes preexistentes como Visual Basic, Java o C++ y las combina en uno solo.. (Gonzalez Seco, 2001). Es un lenguaje moderno orientado a objetos, soporta el concepto de tipos de datos, flujo de declaraciones de control, operadores, colecciones, propiedades, y excepciones (Turtschi, et al.). Además incluye el concepto de clases y las características del paradigma orientado a objeto, tales como encapsulamiento herencia y polimorfismo

Este lenguaje usa la *plataforma .NET* para desarrollar aplicaciones tanto Web como tradicionales. Todo lo necesario para programar en C# se encuentra agrupado en el denominado kit de desarrollo de software llamado .NET Framework SDK, este contiene algunas herramientas importantes para lograr su distribución y ejecución y *Visual Studio.NET* que permite hacer todo lo anterior desde una interfaz visual basada en ventanas.

Plataforma .NET

La plataforma .NET es mucho más que un nuevo lenguaje, Software Development

Kit (SDK) o incluso un sistema operativo. Este ofrece nuevos servicios importantes, un nuevo formato binario de procesadores independientes, nuevas gestión de lenguajes, extensión de la gestión de lenguajes existentes, entre otras.
(Turtschi, et al.)
Por otra parte, el Common Language Runtime (CLR) o núcleo de la plataforma .NET permite ejecutar cualquier aplicación .NET en cualquier plataforma donde exista una versión CLR, logrando así las portabilidad del código entre diferentes versiones de Windows. Además permite la ***gestión de memoria*** a través del recolector de basura, restando preocupación al programador de acciones como crear o destruir objetos. También realiza importantes funciones en la ***seguridad de tipos, tratamiento de excepciones y soporte multihilo, distribución, seguridad y la interoperabilidad con el código antiguo.***

Otras características de C#

Las anteriores características son aplicables sobre los lenguajes que soporta la plataforma .NET, asimismo C# es un lenguaje **orientado a componentes, ofrece extensibilidad de tipos básicos, operadores y modificadores.** Además permite crear nuevas versiones de tipos sin temor a que la introducción de nuevos miembros provoque errores difíciles de detectar en tipos hijos previamente desarrollados y ya extendidos con miembros de igual nombre a los recién introducidos (Gonzalez Seco, 2001). Al mismo tiempo, brinda una eficiencia aceptable debido a que todo el código incluye numerosas restricciones para asegurar su seguridad y no permite el uso de punteros. Sin embargo, y a diferencia de Java, en C# es posible saltarse dichas restricciones manipulando objetos a través de punteros. Para ello basta marcar regiones de código como inseguras (modificador *unsafe*) y podrán usarse en ellas punteros de forma similar a cómo se hace en C++, lo que puede resultar vital para situaciones donde se necesite una eficiencia y velocidad procesamiento muy grandes (Gonzalez Seco, 2001)

PHP

PHP (acrónimo de "PHP: Hipertexto Preprocessor") es un lenguaje interpretado de alto nivel embebido en páginas HTML y ejecutado en el servidor. (Aulbach, et al., 2001)
En PHP en vez de escribir un programa con muchos comandos para crear una salida en HTML, escribimos el código HTML con cierto código PHP embebido (introducido)

en el mismo, que producirá cierta salida (en nuestro ejemplo, producir un texto). El código PHP se incluye entre etiquetas especiales de comienzo y final que nos permitirán entrar y salir del modo PHP.

Lo que distingue a PHP de la tecnología Javascript, la cual se ejecuta en la máquina cliente, es que el código PHP es ejecutado en el servidor. Si tuviésemos un script similar al de nuestro ejemplo en nuestro servidor, el cliente solamente recibiría el resultado de su ejecución en el servidor, sin ninguna posibilidad de determinar que código ha producido el resultado recibido. El servidor web puede ser incluso configurado para que procese todos los ficheros HTML con PHP (Aulbach, et al., 2001)

Poco a poco el PHP se ha convertido en un lenguaje que permite hacer de todo. En un principio diseñado para realizar poco más que un contador y un libro de visitas, PHP ha experimentado en poco tiempo una verdadera revolución y, a partir de sus funciones, en estos momentos se pueden realizar una multitud de tareas útiles para el desarrollo de la web.

Al nivel más básico, PHP puede hacer cualquier cosa que se pueda hacer con un script CGI, como procesar la información de formularios, generar páginas con contenidos dinámicos, o mandar y recibir cookies.
Quizás la característica más potente y destacable de PHP es su soporte para una gran cantidad de bases de datos. Escribir un interfaz vía web para una base de datos es una tarea simple con PHP. Las siguientes bases de datos están soportadas actualmente:

- Adabas D Ingres Oracle (OCI7 and OCI8)
- dBase InterBase PostgreSQL
- Empress FrontBase Solid
- FilePro mSQL Sybase
- IBM DB2 MySQL Velocis
- Informix ODBC Unix dbm

PHP también soporta el uso de otros servicios que usen protocolos como IMAP, SNMP, NNTP, POP3, HTTP y derivados.
Tambien se pueden abrir sockets de red directos (raw sockets) e interactuar con otros protocolos. (Aulbach, et al., 2001)

Java

Java es un lenguaje de desarrollo de propósito general, y como tal es válido para realizar todo tipo de aplicaciones profesionales. Entonces, ¿es simplemente otro lenguaje más? Definitivamente no, incluye una combinación de características que lo hacen único y está siendo adoptado por multitud de fabricantes como herramienta básica para el desarrollo de aplicaciones comerciales de gran repercusión. (Microsystems)

Java puede utilizarse para crear programas del tipo aplicaciones o applets. Un applet es un *programa inteligente,* no solamente una animación o archivo de sonido. En otras palabras, un applet es un programa que puede reaccionar ante las acciones del usuario y cambiar dinámicamente, no solamente ejecutar repetidamente una y otra vez la misma animación o sonido. (Schildt, 2001). Sin embargo es necesario que Java garantice parámetros como *portabilidad y seguridad* para lograr un funcionamiento adecuado de los applets.

Java no es exactamente un lenguaje interpretado o compilado, ni una cosa ni la otra. Primero pasa por un proceso de compilación a través de *Javac*, el compilador. Una vez "compilado" el programa, se crea un fichero que almacena lo que se denomina bytecodes o j_code (pseudocódigo prácticamente al nivel de código máquina). Para ejecutarlo, es necesario un "intérprete", la JVM (Java Virtual Machine) máquina virtual Java (**Ver Figura 3**). De esta forma, es posible compilar el programa en una estación UNIX y ejecutarlo en otra con Windows95 utilizando la máquina virtual Java para Windows95.

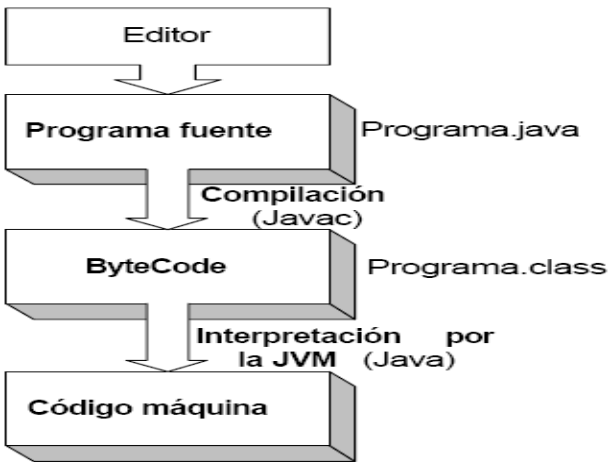


Figura 3. Proceso compilación e interpretación del código Java

La maquina Virtual Java(JVM)

La JVM se encarga de leer los *bytecodes* y traducirlos a instrucciones ejecutables directamente en un determinado microprocesador, de una forma bastante eficiente (Microsystems), garantizando la ***portabilidad*** de los programas Java. (**Ver Figura 4**)

El hecho de que Java sea interpretado ayuda a serlo ***seguro***, como la ejecución de cada programa está bajo el control del intérprete Java, este puede contener al programa e impedir que se generen efectos no deseados en el resto del sistema. (Schildt, 2001)

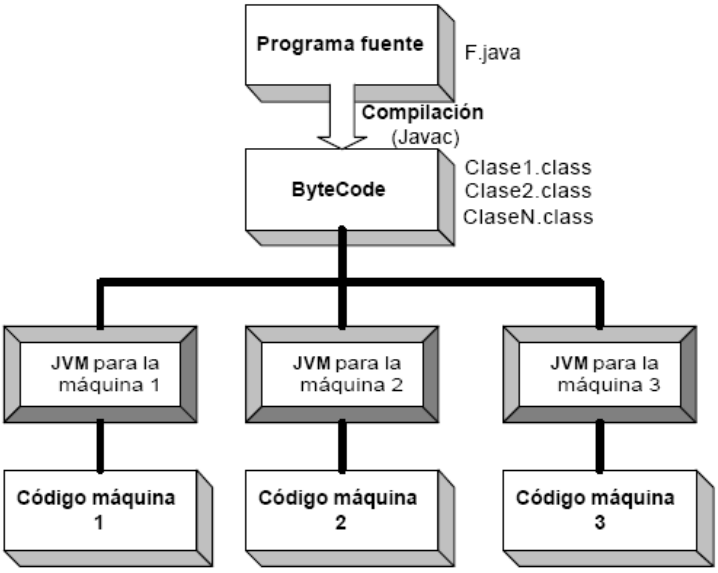


Figura 4. La Maquina Virtual de Java

Este proceso le proporciona a Java una velocidad mayor que algunos de los otros lenguajes interpretados como Visual Basic pero resulta ser más lento que lenguajes compilados como C++.

Aun así, puede aminorar la velocidad a través del compilador Just-In-Time (JIT). Este transforma los bytecodes de un programa o un applet en código nativo de la plataforma ejecutándose de forma más rápida. Suelen ser incorporados por los navegadores, como Netscape o Internet Explorer.

Otras características de Java

Java resulta ser un lenguaje ***simple***, no es difícil dominarlo si se tiene experiencia programando. Los desarrolladores de Java recibieron grandes influencias del paradigma ***orientado a objetos,*** esto ha devenido en que tenga un modelo de objetos sencillo y de fácil ampliación. Además, Java restringe al programador en algunas áreas clave, y así se consigue encontrar rápidamente los errores en el desarrollo del programa. Al mismo tiempo Java le libera de preocuparse por las causas más comunes de errores de programación. Como Java es un lenguaje estrictamente tipificado, comprueba el código durante la compilación. Sin embargo, también comprueba el código durante la ejecución. (Schildt, 2001), ejemplo son errores en tiempo de ejecución como la gestión de memoria y las condiciones de excepción no controladas. Java soporta programación ***multihilo***, pues permite escribir programas que realicen varias cosas a la vez y posibilita la construcción de sistemas interactivos a través de la exquisita solución que brinda su intérprete en la sincronización de múltiples procesos. Por otra parte se ha conseguido en gran parte que el código Java además de ser portable sea duradero, esta característica permite que el código pueda ejecutarse en cualquier momento, manteniendo una ***arquitectura neutral***. Asimismo, el hecho de que el bytecode haya sido diseñado por una fácil interpretación por la JVM, le proporciona al lenguaje un ***alto rendimiento***. Tambien se considera un lenguaje ***distribuido***, puesto que en la versión original de Java (Oak) incluía facilidades que permitía a objetos situados en ordenadores diferentes ejecutar procedimientos de forma remota. Java ha retomado estas interfaces en un paquete denominado de *invocación a remoto (RMI, Remote Method Invocation)*. Esta característica aporta un nuevo nivel de abstracción en la programación cliente/servidor (Schildt, 2001)

El entorno de desarrollo JDK.

La herramienta básica para empezar a desarrollar aplicaciones o applets en Java es el JDK (Java Developer's Kit) o Kit de Desarrollo Java, que consiste, básicamente, en un compilador y un intérprete (JVM) para la línea de comandos. No dispone de un entorno de desarrollo integrado (IDE), pero es suficiente para aprender el lenguaje y desarrollar pequeñas aplicaciones. (Microsystems)

1.4.2 Selección del lenguaje a utilizar

A nivel mundial el lenguaje Java es ampliamente utilizado por las facilidades que ofrece al programador (**Ver figura 5**).

TOP10: Lenguajes de Programación.

SkillMarket			TIOBE			SourceForge		
Lugar	Lenguaje	Score	Lugar	Lenguaje	Score	Lugar	Lenguaje	Score
1	Java	15194	1	Java	22%	1	Java	18%
2	C++	7298	2	C	16%	2	C++	18%
3	C#	5608	3	VBasic	11%	3	C	17%
4	VBasic	5534	4	C++	10%	4	PHP	13%
5	Perl	4515	5	PHP	10%	5	Perl	7%
6	Assembler	1905	6	Perl	5%	6	Python	5%
7	PHP	1295	7	C#	4%	7	C#	3%
8	C	1240	8	Python	3%	8	JScript	3%
9	Python	755	9	JScript	3%	9	VBasic	2%
10	Ruby	287	10	Ruby	2%	10	Assembler	2%

Figura 5. Utilización mundial de los lenguajes de programación (Pacheco, 2008)

Luego de haber profundizado en las características de algunos de los lenguajes de programación más utilizados en la producción de software en la universidad se decidió utilizar Java en la implementación del proyecto, por las siguientes razones:

- ***sencillez:*** resulta más fácil para el equipo de desarrollo aplicar las técnicas de programación al lenguaje.
- ***distribuido:*** es de suma importancia debido a que el software beneficia tanto a Cuba como Venezuela.
- ***alto rendimiento:*** posibilita implementar el software de forma rápida y con calidad.
- ***portabilidad:*** permite ejecutar el software en cualquier sistema operativo, característica que impulsa el uso de la tecnología multiplataforma, directriz seguida por la universidad desde hace algún tiempo.
- ***seguridad:*** la JVM puede contener al programa e impedir que se generen efectos no deseados en el resto del sistema.
- el cliente y el equipo de arquitectura del proyecto decidieron escoger el lenguaje Java para el desarrollo del proyecto.

1.5 Entornos de Desarrollo Integrado (IDE)

Un Entorno de Desarrollo Integrado, es la integración de un conjunto de tecnologías, en una herramienta o plataforma de desarrollo de aplicaciones de software, para escribir, editar, compilar y ejecutar programas. Específicamente para Java, son diversos los IDE que lo soportan, entre los cuales, y más utilizados se encuentran: NetBeans, IntelliJ IDEA, Java Studio Creator 2, JCreator y el popularizado **Eclipse**. Eclipse, se ha convertido en un potencial IDE de desarrollo para los programadores de aplicaciones Java, por las capacidades de implementación que éste les brinda. (Fernandez de la Pera, y otros, 2007)

Eclipse es una plataforma universal para integrar herramientas de desarrollo, con una arquitectura abierta y basada en plug-ins. Además de que da soporte a todo tipo de proyectos que abarcan desde el ciclo de vida del desarrollo de aplicaciones, incluyendo soporte para modelado. (Abián, 2003)

La arquitectura de plug-ins permite integrar diversos lenguajes sobre un mismo IDE o introducir otras aplicaciones accesorias que pueden resultar útiles durante el proceso de desarrollo, como: herramientas UML, editores visuales de interfaces y ayuda en línea para librerías. Conservan el registro de las versiones y generan y mantienen la documentación de cada etapa del proyecto. (Abián, 2003)

Eclipse además incluye asistentes para la creación de interfaces y clases, y proporciona una integración perfecta con el sistema open source CVS (*Concurrent Version System*), que sirve de gran utilidad para llevar el control de la versiones con las que se trabaja, y conocer en todo momento sus respectivas diferencias.

1.6 Persistencia

La persistencia de la información es la parte más crítica en una aplicación de software. Si la aplicación esta diseñada con la orientación a objetos, la persistencia se logra por serialización del objeto o almacenamiento en una base de datos. Hoy en dia las bases de datos son relacionales, el modelo de objetos (estructura jerárquica) difiere en muchos aspectos del modelo relacional (estructura tabular), por lo que existe una

interface que une el modelo de objetos y el relacional llamada marco de mapeo objeto-relacional (ORM en ingles). (Pizarro, 2006)

"*Persistencia es la habilidad que tiene un objeto de sobrevivir al ciclo de vida del proceso en el que reside. Los objetos que mueren al final de un proceso se llaman transitorios.*" (Leyet Fernández, y otros, 2008)

Dentro de las alternativas de persistencia que se presentan para Java se pudiesen utilizar como mecanismos de persistencia, la serializacion, que tiene la capacidad de convertir un conjunto de instancias de objetos que contienen referencias a cada uno, en un conjunto de bytes, los cuales pueden ser enviados a traves de la red, almacenados en un fichero, o simplemente manipulados como un conjunto de datos pero su utilidad es muy limitida al igual que su aplicación. Otra forma común de lidiar con la persistencia en Java es trabajando directamente con SQL y Java Database Connectivity (JDBC).

1.6.1 Esquemas de persistencia

En la actualidad la mayoría de las aplicaciones Java tratan con datos persistentes. Esto significa interactuar con una base de datos relacional o un Sistema de Manejo de Bases de Datos (DBMS) estándar industrial. El API JDBC y los drivers proporcionan una forma de utilizar SQL para ejecutar consultas a bases de datos. Sin embargo, el interface se complica por la "diferencia de impedancia" entre el modelo de objetos de dominio de la aplicación y el modelo relacional de la base de datos. Ninguno de estos modelos es particularmente mejor que el otro, el problema es que son diferentes y no siempre se acoplan de forma confortable en la misma aplicación. (Paterson, 2004)

La forma de conseguir la persistencia de los objetos de forma eficiente es mediante la utilización de esquemas de persistencias. Un esquema de persistencia es un conjunto reutilizable de clases que presentan servicios a los objetos persistentes. Se utiliza para trabajar con bases de datos relacionales, una API de servicios de datos orientados a registros (Microsoft ODBC) u otro mecanismo de almacenamiento. No se utiliza en bases de datos orientadas a objetos y en general traduce los objetos a registros para guardarlos en una base de datos y viceversa. (**Ver Figura 6**)

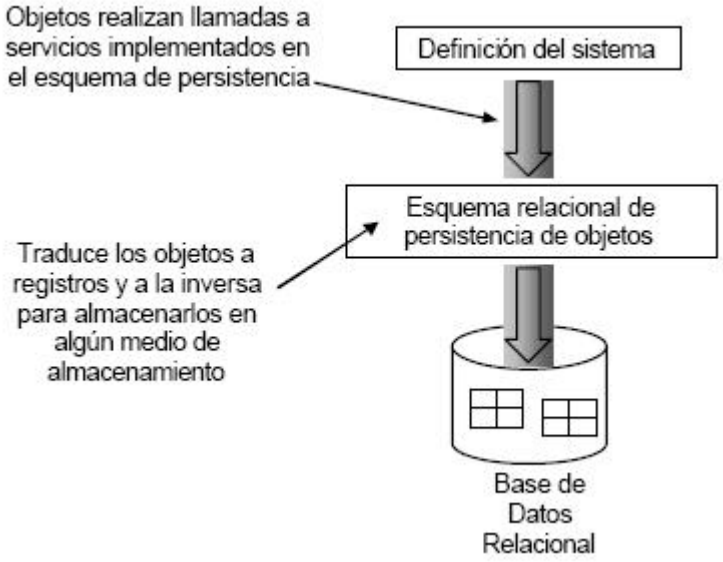


Figura 6. Esquema de persistencia

Al igual que existen IDEs para Java, existen tecnologías y frameworks que manejan el acceso a bases de datos relacionales (BDR), y proporcionan mecanismos que hacen corresponder la incompatibilidad entre el modelo de objetos y el modelo relacional. Para que se logre esta interacción y además sea lo más transparente y eficiente posible, se implementa una capa de persistencia o de acceso a datos que interactúe directamente con una o más BDR.

Existen muchas herramientas que manejan la persistencia en Java, como son: IBATIS, Kodo, Toplink, Castor, EJB 3.0, JPA e Hibernate. Este último es considerado el producto open source líder en este campo, gracias a la documentación, prestaciones y estabilidad que le ofrece a los programadores de la persistencia de objetos en Java. (Fernandez de la Pera, y otros, 2007)

Hibernate se integra al Eclipse mediante plug-ins de Hibernate Tools, proporcionándole un conjunto de herramientas para tareas Ant, realizar ingeniería inversa, generación de código y visualización e interacción con Hibernate.

Algunas encuestas realizadas en el sitio Web oficial javaHispano, sobre el impacto y tendencia de los más reconocidos manejadores de persistencia, reflejan los siguientes datos, en los cuales Hibernate es el indicador más relevante (**Ver Figura 7 y 8**). (Fernandez de la Pera, y otros, 2007)

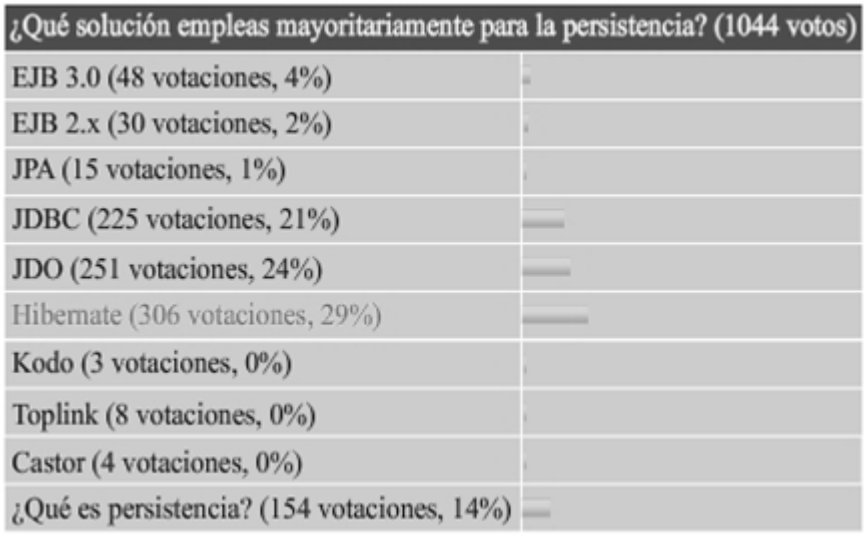


Figura 7. Soluciones de persistencia más utilizadas

Figura 8. Visión de la posible evolución de la persistencia en las aplicaciones Java

1.6.2 Hibernate: Solución para la persistencia de objetos en Java

Anteriormente se mencionaba el mapeo objeto-relacional (ORM) con Hibernate como solución óptima para la persistencia de objetos en Java.

El mapeo objeto/relacional no es más que la persistencia de objetos Java hacia las tablas de una base de datos relacional, usando metadatos que describen el mapeo entre los objetos y la base de datos. En esencia, transforma los datos de una representación a otra.

Esto implica ciertas penalidades en forma de rendimiento, aunque el ORM, implementado de forma correcta, ofrece muchas oportunidades para la optimización

que no existirían en una capa de persistencia con código manejado a través de SQL usando JDBC. (**Ver Figura 9**)

Una solución ORM contiene los siguientes elementos:

- API para realizar operaciones básicas con objetos persistentes.
- Lenguaje de consulta que ataca directamente a las clases y propiedades del modelo (HQL en caso de Hibernate).
- Facilidad para definir metadatos correspondientes al mapping.
- Técnicas que permitan interactuar con objetos transaccionales, permitiendo dirty checking, asociaciones perezosas (lazy) y otras funciones de optimización.
- Estrategias de obtención de datos pertenecientes a una asociación. (Bauer, y otros, 2005)

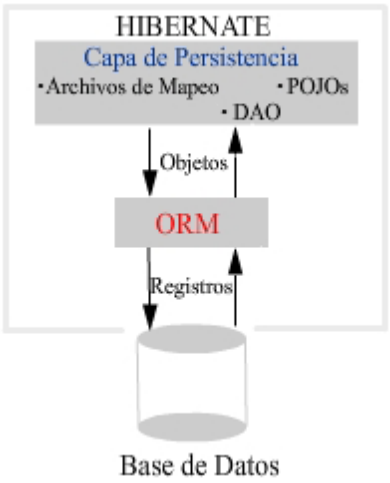


Figura 9. Hibernate como herramienta ORM

ORM e Hibernate además propician ventajas que evidencian la viabilidad de su utilización:

Productividad

Evita mucho del código farragoso de la capa de persistencia, permitiendo centrarse en la lógica de negocio. Permite una estrategia de desarrollo de aplicaciones topdown (empezar con el modelo de entidades) o bottom-up (trabajar con un modelo de datos existente).

Mantenimiento

Al tener pocas líneas de código permite que el código sea más comprensible. ORM proporciona un intermediario entre la representación relacional y la implementación del modelo de objetos, permitiendo más elegantemente el uso de la orientación a objetos en Java y aislando a cada modelo de los cambios del otro.

Rendimiento

Existe la tendencia a pensar que una solución “manual” es más eficiente que una “automática”. Hay que tener en cuenta que una solución automática, permite que dediques más tiempo a optimizaciones. Actualizar las columnas que cambian en una sentencia update es más rápido en unas bases de datos, pero más lentas en otras. Todo está lógica está embebida en el motor ORM. El motor está desarrollado por programadores con altos conocimientos de los SGBD y la conectividad con Java (JDBC y drivers).

Independencia de proveedor

Una solución ORM te abstrae del SGBD. Permite desarrollar con bases de datos ligeras sin implicación en el entorno de producción. (Bauer, y otros, 2005)

Hibernate hace uso de las APIs de Java, incluyendo JDBC, la API de transacciones Java (JTA), y Java Naming and Directory Interface (JNDI). Las APIs son los principales elementos que debemos tener en cuenta en cuanto al trabajo con Hibernate para utilizarlas en la capa de persistencia de la aplicación. (**Ver Figura 10**)

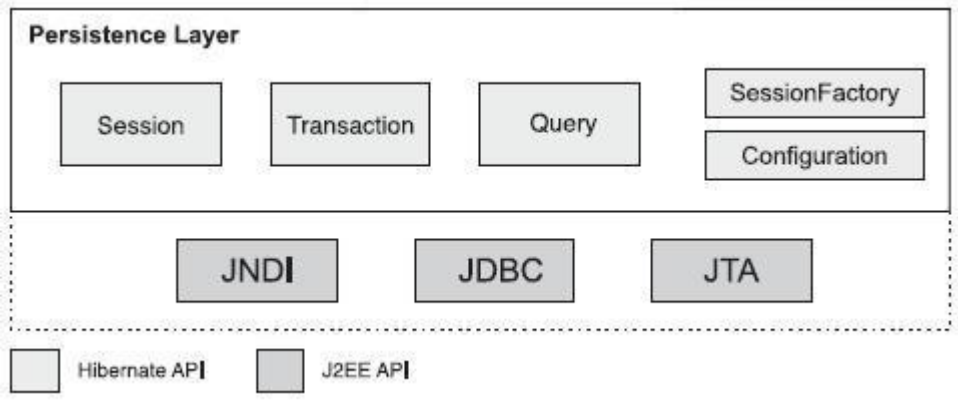


Figura 10. APIs de Hibernate

Con el uso de estas interfaces (APIs) se puede almacenar y recuperar información además de controlar las transacciones que se ejecuten en una aplicación.

Interface Session

Es la principal interface usada por las aplicaciones de Hibernate. Las instancias de la sesión se crean y se destruyen de forma ligera. Esto es muy importante debido a que las aplicaciones tienen que crear y destruir sesiones todo el tiempo, posiblemente en todas las solicitudes que se hagan. La noción de Hibernate de un período de sesiones es algo entre conexión y transacción. Puede ser más fácil pensar en un período de sesiones como caché o colección de objetos cargados relativos a una única unidad de trabajo. Hibernate puede detectar cambios en los objetos en esta unidad de trabajo. A veces la sesión es llamada gestor de persistencia, ya que es también la interfaz para las operaciones relacionadas con la persistencia, como el almacenamiento y la recuperación de objetos.

Interface SessionFactory

La sessionFactory es la que crea sessiones en las aplicaciones. Es compartida por muchos hilos de la aplicación. Las SessionFactorys son inmutables; el comportamiento de una SessionFactory es controlado por *properties* suplidas en tiempo de configuración. Estas properties son definidas en la clase Environment.

Interface Configuration

Una aplicación usa instancias de configuration para especificar donde se encuentran los mapping, las properties de Hibernate y la sessionFactory.

Además de las interfaces mencionadas anteriormente están la **Transaction**, **Query and Criteria**, **Callback** (*Lifecycle, Validatable, Interceptor*), **Types** y **Extension.**

Debido a que Hibernate está diseñado para operar en muchos entornos diferentes, es de suma importancia conocer como configurar a Hibernate para utilizarlo en una aplicación. Hibernate puede configurarse y ejecutarse en la mayoría de aplicaciones Java y entornos de desarrollo, generalmente se utiliza en aplicaciones cliente/servidor de tres capas, desplegándose únicamente en el servidor. Hibernate puede ser configurado para entornos gestionados (*Managed Environment*) y no gestionados (*Non-managed Environment*). Este último proporciona una gestión básica de la concurrencia a través de un *pooling* de *threads*. Los entornos no gestionados no proporcionan infraestructura para transacciones automáticas, gestión de recursos, o

seguridad. La propia aplicación es la que gestiona las conexiones con la base de datos y establece los límites de las transacciones. Hibernate intenta abstraerse del ambiente en el cual interviene, en el caso de un ambiente non-managed, Hibernate maneja transacciones y conexiones JDBC.

Generalmente, no es conveniente crear una conexión cada vez que se necesite interactuar con bases de datos. En lugar de eso, las aplicaciones Java utilizan un contenedor de conexiones JDBC (*connection pool*); tres razones para usar un contenedor son: *adquirir una nueva conexión es costoso*, *mantener muchas conexiones inactivas es costoso* y *crear declaraciones* (*consultas SQL*) *es también costoso para algunos drivers.* Sin Hibernate, el código de la aplicación usualmente llama al contenedor de conexiones para obtener conexiones JDBC y ejecutar declaraciones SQL. (Fernandez de la Pera, y otros, 2007)

Un mapeo objeto-relacional con Hibernate genera básicamente objetos de Java (POJOs) y ficheros de mapeo (XML) necesarios para manejar la persistencia y utilizados para la implementación de la capa de acceso a datos. En estos ficheros de mapeo de Hibernate se especifica el mapeo entre tablas y clases, columnas y propiedades, llaves foráneas y asociaciones y tipos de datos de SQL y de Java. Estos ficheros de mapeo soportan los elementos o tags (etiquetas) que hacen corresponder el modelo relacional con el modelo de objetos utilizando algunos como: **<class/>**, **<id/>**, **<composite-id/>**, **<generator/>** y **<property/>** elementos para las asociaciones entre entidades **<one-to-one/>**, **<many-to-many/>**, **<one-to-many/>** ; tags para el mapeo de colecciones **<set/>**, **<bag/>**, **<list/>**, **<map/>** . Como elementos componentes de estas etiquetas, se definen atributos para las entidades a las cuales se hace referencia, tales como: **unsaved-value**, **type**, **length**, **fetch**, **lazy**, **unique**, **inverse**, **cascade** entre otros. (Fernandez de la Pera, y otros, 2007)

Para lograr solucionar el problema de recuperar datos de bases de datos relacionales, Hibernate esta equipado con un potente lenguaje de consulta orientado a objetos (HQL). El uso de HQL nos permite usar un lenguaje intermedio que según la base de datos que usemos y el dialecto que especifiquemos será traducido al SQL dependiente de cada base de datos de forma automática y transparente. Este lenguaje es utilizado sólo para recuperar objetos, y no para actualizar, insertar o eliminar datos.

En adición, Hibernate permite utilizar cuatro estrategias para cargar los datos de cualquier asociación entre entidades persistentes, especificadas por atributos en los ficheros de mapeo en tiempo de ejecución:

• **Immediate fetching** (*carga inmediata*)**:** ocurre cuando se recupera una entidad de la base de datos e inmediatamente se recuperan otras entidades asociadas en más peticiones a la base de datos o a la caché. La carga inmediata no es usualmente una estrategia eficiente a menos que se espere que las entidades asociadas sean cargadas casi siempre.

• **Lazy fetching** (*carga perezosa*)**:** cuando se hace una petición a una entidad y los objetos asociados a ésta en la base de datos, no es generalmente necesario recuperar todos los objetos de cada (indirectamente) objeto asociado. La carga perezosa permite determinar cuántos objetos son cargados en una interacción inicial con la base de datos y cuales asociaciones deben ser cargadas. Este es un concepto fundacional en la persistencia de objetos y el primer paso para alcanzar un rendimiento aceptable. Se recomienda que para comenzar, todas las asociaciones sean configuradas para una carga perezosa en los ficheros de mapeo.

• **Eager** (*outer join*) **fetching** (*carga ávida*)**:** permite explícitamente especificar cuáles objetos asociados deben ser cargados conjuntamente con el objeto referenciado. Hibernate puede devolver los objetos asociados en una única petición a la base de datos, utilizando un outer join de SQL. La optimización del rendimiento en Hibernate a menudo involucra un bien ponderado uso de la carga ávida para transacciones particulares. Por lo tanto, aunque se puede declarar por defecto una carga ávida en los ficheros de mapeo, es más común especificar el uso de esta estrategia en tiempo de ejecución para una consulta particular en HQL o Criteria. 31

• **Batch fetching** (*carga por lotes*)**:** no es estrictamente una estrategia de recuperación de asociaciones; es una técnica que puede ayudar a mejorar el rendimiento de la carga perezosa o inmediata. Usualmente, cuando se carga un objeto o una colección, la cláusula WHERE de SQL especifica el identificador del objeto o los objetos que pertenecen a la colección. Si la carga por lotes es habilitada, Hibernate determina que otras instancias (*proxied*) o colecciones no inicializadas son referenciadas en la presente Session e intenta cargarlas al mismo que especifica valores múltiples del identificador en la cláusula WHERE. (Bauer, y otros, 2005)

No cabe duda que trabajar con Hibernate puede reducir el coste en cuanto al trabajo de la interacción con bases de datos relacionales y permite solucionar todos los problemas de persistencia manteniendo la portabilidad de la aplicación que se esté desarrollando.

1.7 Pasos para implementar elementos de diseño

Después de conocer el lenguaje a utilizar, el IDE y el framework que facilitará la persistencia de los datos es necesario conocer y aplicar algunos pasos que permitan escribir código flexible y reusable a la hora de implementar los elementos de diseño. Con este objetivo se hará un enfoque en la implementación de los atributos, las operaciones y las asociaciones.

Implementación de los atributos

La implementación de los atributos se puede realizar de tres formas: usando tipos primitivos, usando clases existentes o creando clases nuevas. En el caso de las clases persistentes se definió utilizar clases existentes que brinda el sistema que se corresponden con el atributo primitivo. Por ejemplo String, Integer, Boolean.

Implementación de las operaciones

Las operaciones de más peso en la implementación de acceso a datos se encuentran en las clases de acceso a datos (DAOs), es ahí donde las siguientes acciones tienen más peso:

- ***Seleccionar algoritmos*:** Es de gran importancia saber seleccionar los algoritmos porque muchas veces se trata de implementar algoritmos que son brindados por el lenguaje y se encuentran ya optimizados. Esto trae consigo pérdida de tiempo en la implementación y de rendimiento en la operación. Debido a esto para persistir y cargar los datos de las base de datos fueron utilizados los métodos *save()*, *saveOrUpdate()*, *delete()*, *findByNamedQueryAndNamedParam()* que brinda la plantilla de Hibernate HibernateDaoSupport, a través del método *getHibernateTemplate().*
- ***Seleccionar estructuras de datos apropiadas con los algoritmos:*** Una estructura de datos es cualquier representación de datos y sus operaciones asociadas. En dependencia del algoritmo seleccionado y las operaciones básicas que deben ser soportadas se debe decidir que estructura de datos a utilizar. Cada estructura de datos tiene asociados costos y beneficios, debido a esto es incorrecto decir que una estructura de datos es mejor que otra en todos los casos. Además una estructura de datos requiere una cierta cantidad de espacio para cada dato que almacena, una cierta cantidad de tiempo para realizar una operación básica y un cierto esfuerzo de programación. Debido a

esto se decidió utilizar las interfaces *Set* para las asociaciones one-to-many y *List* e *Iterator* para las consultas.

Implementación de las asociaciones

Las asociaciones unidireccionales con multiplicidad ***one*** pueden ser implementadas con un atributo que contiene una referencia del objeto de la clase correspondiente. Si la multiplicidad es ***many*** entonces el atributo es de tipo *Set.* En caso de necesitarse que los datos sean ordenados se usaría *List* en lugar de *Set.* También existen las asociaciones bidireccionales, que son asociaciones unidireccionales en ambos sentidos de la relación. Teniendo en cuenta la necesidad de navegar en ambos sentidos de las relaciones, sobre todo a la hora de realizar consultas se decidió utilizar las asociaciones bidireccionales.

1.8 Estándar de codificación utilizado

Los estándares, estilos o convenciones de codificación son importantes para los programadores debido a que ofrecen:

- **Extensibilidad**: La facilidad con que se adapta el software a cambios de especificación. Un buen estilo de código fomenta programas que no sólo resuelven el problema, sino que también reflejan claramente la relación problema/solución. Esto tiene como efecto que muchos cambios simples en el problema reflejen de forma obvio los cambios a hacer en el programa.
- **Verificabilidad**: la facilidad con que pueden comprobarse propiedades de un sistema. Si el estilo de código hace obvia la estructura del programa, eso ayuda a verificar que el comportamiento sea el esperado.
- **Reparabilidad**: la posibilidad de corregir errores sin demasiado esfuerzo.
- **Capacidad de evolución**: la capacidad de adaptarse a nuevas necesidades.
- **Comprensibilidad**: la facilidad con que el programa puede ser comprendido.

El estándar utilizado en el proyecto, fue la convención para el lenguaje de programación Java brindada por **Sun Microsystems**. (Hommel) A continuación se muestra una tabla con las convenciones de nombres pertenecientes a este estilo de codificación.

Tipos de identificadores	Reglas para nombras	Ejemplos
Paquetes	El prefijo del nombre de un paquete se escribe siempre con letras ASCII en minúsculas, y debe ser uno de los nombres de dominio de alto nivel, actualmente *com*, *edu*, *gov*, *mil*, *net*, *org*, o uno de los códigos ingleses de dos letras que identifican cada país como se especifica en el ISO Standard 3166, 1981. Los subsecuentes componentes del nombre del paquete variarán de acuerdo a las convenciones de nombres internas de cada organización. Dichas convenciones pueden especificar que algunos nombres de los directorios correspondan a divisiones, departamentos, proyectos o máquinas.	cu.uci.ccv.dao
Clases	Los nombres de las clases deben ser sustantivos, cuando son compuestos tendrán la primera letra de cada palabra que lo forma en mayúsculas. Intentar mantener los nombres de las clases simples y descriptivas. Usar palabras completas, evitar acrónimos y abreviaturas (a no ser que la abreviatura sea mucho más conocida que el nombre completo, como URL o HTML).	class DFichaProyecto; class DContrato;
Interfaces	Los nombres de las interfaces siguen la misma regla que las clases.	Interface FichaDao; Interface ContratoDaoImpl;
Métodos	Los métodos deben ser verbos, cuando son compuestos tendrán la primera letra en minúscula, y la primera letra de las siguientes palabras que lo forma en mayúscula.	crear(); cargarPorId();
Variables	Excepto las constantes, todas las instancias y variables de clase o método empezarán con minúscula. Las palabras internas que lo forman (si son compuestas) empiezan con su primera letra en mayúsculas. Los nombres de variables no deben empezar con los caracteres subguión	Integer i; String s; Double d; DFichaProyecto fichaProyecto;

	"_" o signo del dólar "$", aunque ambos están permitidos por el lenguaje. Los nombres de las variables deben ser cortos pero con significado. La elección del nombre de una variable debe ser un mnemónico, designado para indicar a un observador casual su función. Los nombres de variables de un solo carácter se deben evitar, excepto para variables índices temporales. Nombres comunes para variables temporales son i, j, k, m, y n para enteros; c, d, y e para caracteres.	

Constantes	Los nombres de las variables declaradas como constantes deben ir totalmente en mayúsculas separando las palabras con un subguión ("_"). (Las constantes ANSI se deben evitar, para facilitar su depuración.)	static final int ANCHURA_MINIMA = 4;
		static final int ANCHURA_MAXIMA = 999;
		static final int COGER_LA_CPU = 1;

Figura 111. Estilo de codificación brindada por Sun Microsystems

Es necesario aclarar que se realizaron algunos cambios en la nomenclatura de las *clases.* Se determinó:

- Separar los nombres compuestos de las clases persistentes con un subguión ("_").
- Comenzar cada clase persistente con la letra “D” o “N” correspondiendo con el tipo de entidad persistente que representa dicha clase, entidad de datos o entidad nomenclador respectivamente.

Los ejemplos de clases resultantes quedaron de la siguiente forma:
class DFicha_Proyecto;
class DContrato

1.9 Patrones de diseño

- Los patrones de diseño (design patterns en ingles) son la base para la búsqueda de soluciones a problemas comunes que se repiten y se presentan en situaciones particulares del diseño durante el proceso de desarrollo de software. Entre la variedad tan amplia que existen de patrones de diseño que pueden utilizarse en la capa de acceso a datos, se encuentran los patrones J2EE dentro de los cuales está el patrón Data Access Object (DAO).
- El patrón DAO consiste en trabajar con objetos de acceso a datos para abstraer y encapsular los accesos a las distintas fuentes de datos (base de datos, archivos o servicios externos) que pueda tener una aplicación. El DAO maneja las conexiones con las fuentes de datos para obtener y almacenar datos persistentes o transitorios. Para acceder a bases de datos relacionales se pueden utilizar Interfaces de Programación de Aplicaciones (APIs) como JDBC que permite de forma estándar el trabajo con bases de datos utilizando sentencias SQL, la idea del patrón DAO además de ocultar las fuentes de datos como se comentó anteriormente, es ocultar la complejidad del uso de JDBC a las capas superiores.

1.10 JUnit como herramienta para pruebas

JUnit es el estándar para desarrollar test en Java. Forma parte de una arquitectura más amplia conocida como XUnit, con desarrollos para varios lenguajes de programación, pero ha sido en el mundo Java donde se ha hecho más popular. Esta fama se debe a que ha conseguido simplificar una de las tareas más vitales e importantes en el ciclo de vida de un programa, **comprobar la calidad del código**, esto es, probarlo. Todo programa deber ser pensado, codificado y probado. **JUnit** proporciona una manera fácil de completar este requerimiento, permitiendo estandarizar este proceso.

JUnit no es más que un marco donde se ejecutan programas que prueban otros programas, o mejor dicho, trozos de un programa, mediante un proceso conocido como **test**. Cada uno de estos tests está asociado a ciertas partes de nuestra aplicación cuyo comportamiento hay que someter a prueba. Cada uno de estos

análisis del comportamiento debe ser independiente de los otros para que el test sea bueno. Es decir, el test y no el programa es lo que debe ser independiente del resto. Estas pruebas deben codificarse con esta premisa siempre en mente, para que al final del ciclo de nuestra aplicación, se disponga de una batería lo suficientemente amplia como para poder haber capturado todos los posibles comportamientos erráticos de nuestra aplicación. Lo que se consigue con todo esto es saber cuál es el estado de nuestra aplicación, y mucho más importante, saber cómo una modificación en una parte del código afecta a las demás partes sin necesidad de depurar o rellenar infinidad de datos.

Lo que **JUnit** aporta a los programadores es una manera rápida y estándar de escribir los test, y lo que es más importante, una manera rápida de visualizar los datos para saber si todo está bien o si está mal (incluso dónde se halla el fallo) sin necesidad de perder tiempo poniendo trazas o puntos de ruptura.

Sin embargo, el único problema de **JUnit** es el tiempo que hay que emplear en escribir y pensar los test, pero es un esfuerzo que se compensa sobradamente con las ventajas que aporta. Además, cuanto más grande sea el proyecto más vale la pena dicho esfuerzo.

1.11 Conclusiones

En este capítulo se hizo un estudio del arte de las principales metodologías de desarrollo de software, paradigmas de programación, lenguajes, entornos de desarrollo integrado y tecnologías o frameworks de persistencia existentes, haciendo una caracterización profunda, comparando y seleccionando en cada caso la más adecuada para utilizarla en el proceso de desarrollo del software y lograr de esta forma obtener un producto con los costes de tiempo y calidad requeridos, además de que responda a las necesidades del cliente.

Además se trabajó utilizando las referencias bibliográficas necesarias para garantizar la autenticidad de la información expuesta en el mismo, así el lector podrá auxiliarse de las diferentes fuentes de información consultadas en caso de que requiera mayor información acerca del tema expuesto.

CAPITULO 2. Propuesta de solución

2.1 Introducción

En este capítulo se abordará la descripción de la propuesta para solucionar el problema de esta investigación.

Para ello se tomará como punto de partida el modelo de datos y el diagrama de clases previamente diseñados en el flujo de trabajo de diseño, además de las clases y los ficheros de mapeo objeto-relacional que permitirán la vinculación de la aplicación con la base de datos usando Hibernate. De gran importancia será además la realización de un modelo de componentes que sirva como guía para implementar las operaciones necesarias como insertar, modificar, eliminar y la creación de reportes que respondan a las necesidades del cliente.

Paralelamente se utilizaran métodos de ingeniería como PSP que incrementen las habilidades y el rendimiento de los programadores en cuanto a las tareas a realizar, además de que garanticen la calidad en la capa de acceso a datos de los módulos de Presentación y Contratación de proyectos.

2.2 Elementos de diseño utilizados en la implementación

Para comenzar a implementar los artefactos de la capa de acceso a datos se hace imprescindible tener como entrada algunos elementos de diseño tales como el modelo de datos y el diagrama de clases.

El modelo de datos es un conjunto de conceptos, reglas y convenciones que nos permiten describir y manipular los datos que deseamos almacenar en la base de datos. Al producto del modelo de datos se le llama esquema (descripción de la estructura de la base de datos) y a los datos almacenados en la base de datos, ocurrencia del esquema.

El modelo de datos está formado por dos componentes, componente estático, relacionado con el lenguaje de definición de datos (LDD) y dinámico, relacionado con el lenguaje de manipulación de datos (LMD). La parte estática se refiere a la estructura

y la dinámica a qué operaciones puedo realizar sobre cada objeto. A continuación se presenta el modelo de datos integrado por las entidades del módulo Común (color azul), Presentación (color naranja) y Contratación (color verde). (**Ver Anexo I**)

Los diagramas de clases del diseño son diagramas de estructura estática que muestran las clases, junto con sus métodos y atributos, así como las relaciones estáticas entre ellas: qué clases *conocen* a qué otras clases o qué clases *son parte* de otras clases, pero no muestran los métodos mediante los que se invocan entre ellas. Son los diagramas más comunes en el modelo de sistemas orientados a objetos, además de que son importantes no sólo para visualización de modelos, especificación y documentación estructurales, sino también para construir sistemas ejecutables. (**Ver Anexo II, III y IV**)

2.3 Modelo de componentes de la capa de persistencia

El modelo de componentes muestra la estructura de los componentes de software que se usarán para construir el sistema, incluyendo los clasificadores que especifican los componentes, y artefactos que los implementan. También se pueden utilizar para mostrar el alto nivel de la estructura del Modelo de implementación en términos de subsistemas de implementación, y las relaciones entre los elementos de implementación. Se puede construir a partir del modelo de clases y escribir desde cero para el nuevo sistema o se puede importar de otros proyectos y de productos de terceros. Los componentes son agregaciones de alto nivel de las piezas de software más pequeñas y proveen un enfoque de construcción de bloques de “caja negra” para la elaboración de software. (**Ver Anexo V, VI y VII**)

2.4 Estructura de la capa de persistencia

La capa de acceso a datos está compuesta por un conjunto de clases e interfaces que permiten realizar funciones de persistencia y recuperación de datos sobre la base de datos. La implementación de dicha capa en el proyecto CICCV incluye las capas de acceso a datos de los módulos de Presentación y Contratación de proyectos.

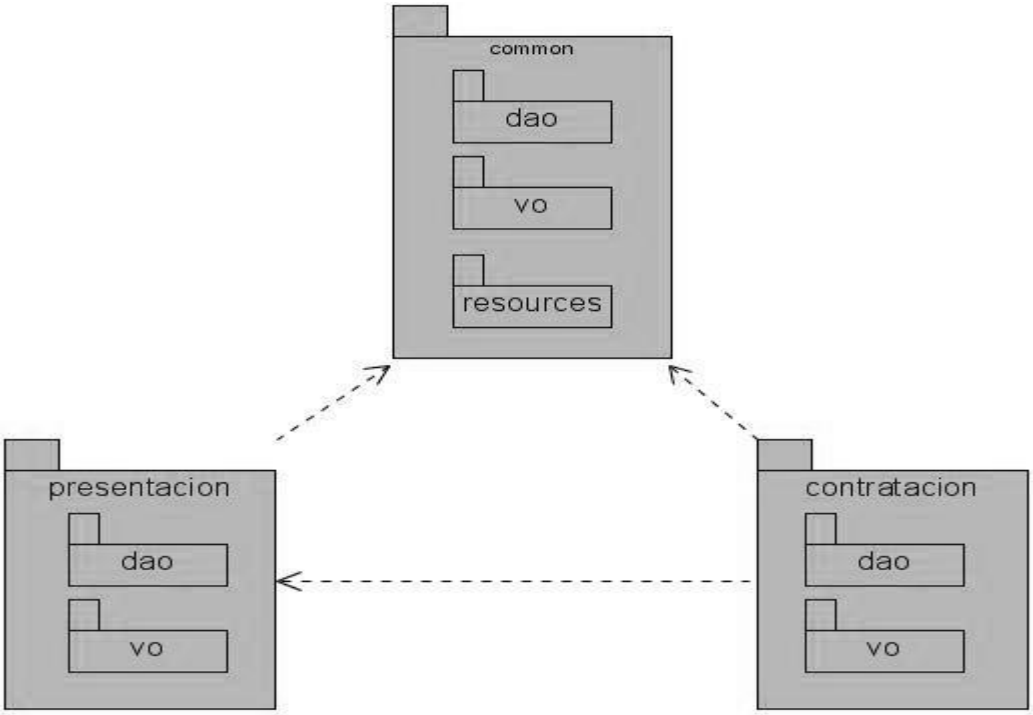


Figura 122. Estructura de la capa de persistencia

La implementación de la capa de acceso a datos del modulo Contratación dependen de los elementos de la capa de acceso a datos del modulo Presentación y estas dos a su vez dependen de los elementos incluidos en dicha capa del modulo común. (**Ver Figura 12**)

Desde el IDE Eclipse estos paquetes se muestran de la siguiente forma:

Figura 133. Estructura de la capa de persistencia desde el Eclipse

2.5 Implementación de los elementos de diseño

En esta tarea se describe cómo producir una implementación para una parte del diseño (por ejemplo, una clase, una realización de un caso de uso o una entidad de base de datos), o para solucionar uno o varios defectos. El resultado son archivos nuevos o modificados de datos y de código fuente, que se conocen generalmente como elementos de implementación. En el caso de los módulos de Presentación y Contratación de Proyectos los archivos como los POJOs (Plain Old Java Object en ingles) y los archivos de mapeos hbm.xml se generaran con la ayuda de la herramienta Visual Paradigm.

Por el contrario las clases DAOs (Data Access Object) serán creados por el programador.

2.5.1 Mapeo objeto/relacional

Como se mencionaba anteriormente Hibernate y el ORM facilitan el trabajo con la persistencia en la capa de acceso a datos debido a las potencialidades que ofrecen con respecto a la compresibilidad, manejabilidad y optimización del código permitiendo minimizar el tiempo empleado en la implementación de la capa de acceso a datos.

Archivos de configuración de hibernate

La interfaz principal para interactuar con Hibernate es *org.hibernate.Session.* La interfaz Session proporciona funcionalidades básicas de acceso a datos tales como salvar, actualizar, eliminar y cargar objetos de la base de datos.
La forma de obtener una referencia del objeto Session de Hibernate es a través de la implementación de la interfaz SessionFactory de Hibernate. Su responsabilidad radica en abrir, cerrar, y gestionar sesiones. Existen dos formas de configurar el bean SessionFactory en los ficheros de mapeo de Hibernate, a través de las clases LocalSessionFactoryBean o AnnotationSessionFactoryBean, ambas realizan funciones similares, excepto que la última crea una sessionfactory basada en anotaciones en una o más clases del dominio. (**Ver Anexo VII**)

La SessionFactory está compuesta por diferentes propiedades, entre ellas "mappingDirectoryLocations", que especifica la ubicación de los archivos de las clases del dominio, además la propiedad "hibernateProperties", que permite configuraciones avanzadas de hibernate como el dialecto de la base de datos utilizada y las trazas en el lenguaje SQL generadas por las operaciones realizadas sobre la base de datos. El datasource es otra de las propiedades de la SessionFactory.
Existen varias opciones para configurar el bean datasource, estas son:

- Datasource definidos por drivers JDBC
- Datasource que son buscados por el JNDI
- Datasource por piscina de conexiones

Este último se realiza incluyendo en el bean datasource de los archivos de hibernate la clase BasicDataSource. (**Ver Anexo VIII**)

Las primeras cuatro propiedades son básicas en la configuración del BasicDataSource. La propiedad *url* especifica la URL JDBC completa de la base de datos, el *driverClassName* indica el nombre completo del driver de la base de datos. El *username* y *password* se utilizan para la autentificación de la conexión a la base de datos. Por otra parte la propiedad *initialSize* expresa el número de las conexiones creadas cuando comienza la piscina, *maxActive* muestra el número de conexiones que pueden ser permitidos al mismo tiempo, *maxIdle* y *minIdle* muestran la cantidad máxima de conexiones activas que permanecerán desocupadas en el pool, y *maxWait contiene e*l tiempo en milisegundos que el pool espera para lanzar una excepción cuando no hay conexiones disponibles.
Los valores de las propiedades de la SessionFactory y el datasource se pueden especificar directamente o pueden ser cargados desde un fichero externo y ser cableados hacia variables de posición especificadas en dichas propiedades mediante los caracteres ${…}. Este último proceso se realiza a través de la clase PropertyPlaceholderConfigurer. (**Ver Anexo IX**)

La única información que necesita es la ubicación del archivo que posee el contenido la cual se especifica mediante la propiedad *locations* en un fichero externo que almacena el valor de las propiedades. (**Ver Anexo X**)

POJOs y archivos HBM. XML

La herramienta visual Paradigm permite realizar la ingeniería inversa a la base de datos y generar los POJOs y los archivos XML (hbm.xml) necesarios para mapear las clases persistentes con las tablas de la base de datos. Por cada tabla en la base de datos se genera un XML y una clase .java (**Ver Figura 14**) cuyos atributos corresponden a los campos de dichas tablas y a sus respectivas relaciones y pueden ser configurados en dependencia de las necesidades del desarrollador.

Módulo Presentación	
Archivos de mapeo	Clases persistentes(POJOs)
DFicha_Proyecto.hbm.xml	DFicha_Proyecto.java
DActividad.hbm.xml	DActividad.java
DFinanciamiento.hbm.xml	DFinanciamiento.java
DGasto_Administrativo.hbm.xml	DGasto_Administrativo.java

DOtro_Recurso.hbm.xml	DOtro_Recurso.java
DRecurso_Humano.hbm.xml	DRecurso_Humano.java
DRecurso_Material.hbm.xml	DRecurso_Material.java
NFuente_Financiamiento.hbm.xml	NFuente_Financiamiento.java
NModalidad.hbm.xml	NModalidad.java
Módulo Contratación	
DContrato.hbm.xml	DContrato.java
DCronograma.hbm.xml	DCronograma.java
DDesembolsoInv.hbm.xml	DDesembolsoInv.java
DDesembolsoTransf.hbm.xml	DDesembolsoTransf.java

Figura 14. Tabla con las clases persistentes y los archivos de mapeo de los módulos de Presentación y Contratación

A continuación veremos algunas características de estos componentes.

```
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN" "http://h
<hibernate-mapping>
 (1) <class name="cu.uci.ccv.contratacion.vo.DContrato" table="dContrato"
        lazy="false">
     (2) <id name="idContrato" column="documento" type="integer">
          (3) <generator class="foreign">
                  <param name="property">documento</param>
              </generator>
         </id>
     (4) <one-to-one name="documento"
              class="cu.uci.ccv.common.vo.DDocumento" cascade="all"
              constrained="true" />
     (5) <property name="documentoAdjunto" column="documentoAdjunto"
              type="binary" not-null="true" lazy="true" />
         <property name="nombredoc" column="nombredoc"
              type="string" not-null="false" lazy="true" />
         <property name="fechaFirma" column="fechaFirma" type="date"
              not-null="false" />
     (6) <set name="fichaProyecto" lazy="false" cascade="save-update"
              inverse="true">
              <key column="contrato" not-null="false" />
              <one-to-many
                  class="cu.uci.ccv.presentacion.vo.DFicha_Proyecto" />
         </set>
    </class>
</hibernate-mapping>
```

Figura 15. Archivo de mapeo DContrato.hbm.xml

(1) especifica el nombre de la clase y la tabla que representa en la base de datos. El atributo ***lazy*** define la estrategia de carga de los objetos de dicha clase. En este caso está definida como "false" indicando que siempre se va a cargar las relaciones en el mismo momento de cargar una instancia de una clase que esté relacionada con ella.
(2) es la propiedad estilo JavaBeans que tiene el identificador único de una instancia, define el mapeo de esa propiedad a la columna de clave primaria.
(3) nombra una clase Java usada en generar identificadores únicos para instancias de la clase persistente. De requerirse algún parámetro para configurar o inicializar la instancia del generador, se pasa usando el elemento <param>. En este caso se uso ***class***=”foreign” mostrando el uso del identificador del objeto de la clase DDocumento. Y representado por la asociación de llave primaria <one-to-one>.
(4) se mapea en el caso de una relación en la base de datos de uno-a-uno con *identificación*. El atributo ***constrained***=”true” especifica que existe una restricción de llave foránea con la llave primaria del la instancia de la clase DDocumento. Puede servir como alternativa para la herencia en tres dos clases.
(5) es la representación de una columna de la tabla de la base de datos. Especificando el tipo de datos a través del atributo ***type*** y la restricción de permitir o no valores nulos con el atributo ***not-null.***
(6) es el elemento de mapeo de Hibernate usado para mapear una colección, un elemento <set> se usa para mapear propiedades de la interfaz Set representado el mapeo de una relación <one-to-many> con los objetos de la clase DFicha_Proyecto. El atributo cascade define como se realizaran las operaciones *save()*, *update()* y *delete()* de los objetos relacionados. Además el atributo ***inverse***=”true” expresa que la relación es de tipo bidireccional, o sea que existe navegabilidad en ambos sentidos. La etiqueta <key> es la clave foránea que indica cual es el identificador de la tabla DContrato dentro de DFicha_Proyecto. (**Ver Figura 16)**

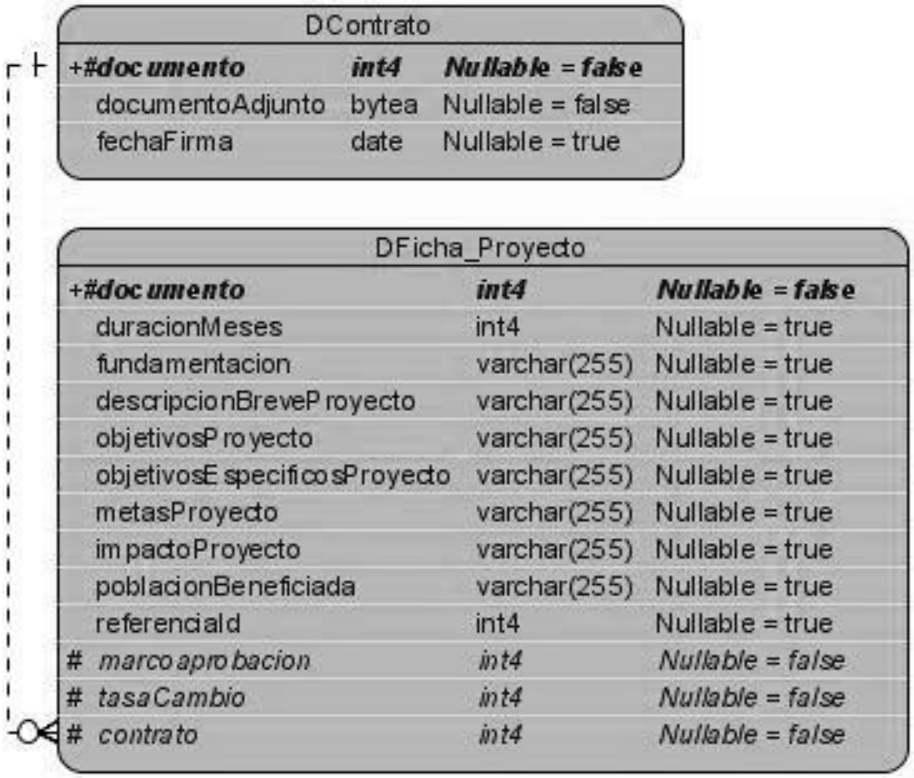


Figura 16. Relación one-to-one con identificación entre la DContrato y DFicha_Proyecto

La clase persistente correspondiente al fichero DContrato.hbm.xml muestra algunas particularidades. (**Ver Anexo XI**)

Como se observa en anexo XI la clase DContrato hereda de la clase DDocumento. Además en ella se representan cada una de las propiedades mapeadas en el archivo XML correspondiente a través de atributos. Los nombres deben ser iguales y los tipos deben corresponderse con su mapeo. En este caso el objeto ***documento*** muestra la relación con DDocumento. Asimismo ***fichaProyecto*** es un Set de DFicha_Proyecto que representa a la etiqueta <set> y los atributos restantes se corresponden con la <property> mapeada en el hbm.xml correspondiente. Los tipos de datos de estos últimos no deben ser primitivos sino su clase correspondiente, por ejemplo *Integer, Byte y Date*. También es necesario implementar los métodos *get()* y *set()* de cada unos de los atributos de las clase. Por otra parte el XML de DFicha_Proyecto algunas etiquetas de mapeo diferentes, la siguiente figura lo demuestra.

```
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN" "http
<hibernate-mapping>
    <class name="cu.uci.ccv.presentacion.vo.DFicha_Proyecto"
        table="dFicha_Proyecto" lazy="false">
        <!-- <cache usage="read-only"/>  -->
        <id name="idFicha" column="documento" type="integer">
            <generator class="foreign">
                <param name="property">documento</param>
            </generator>
        </id>
        <one-to-one name="documento"
            class="cu.uci.ccv.common.vo.DDocumento" cascade="all"
            constrained="true" />
        <property name="referenciaId" column="referenciaId"
            type="integer" not-null="false" lazy="false" />
        <many-to-one name="tasaCambio" cascade="none"
            column="tasaCambio" class="cu.uci.ccv.presentacion.vo.NTasa_Cambio"
            not-null="false" access="field">
        </many-to-one>
(1)     <many-to-one name="contrato" column="contrato"
            class="cu.uci.ccv.contratacion.vo.DContrato" cascade="none"
            not-null="false" access="field">
        </many-to-one>
        <set name="DActividad" lazy="true" cascade="all-delete-orphan"
            inverse="true">
            <key column="fichaProyecto" not-null="true" />
            <one-to-many class="cu.uci.ccv.presentacion.vo.DActividad" />
        </set>
(2)     <set name="DFinanciamiento" lazy="false"
            cascade="all-delete-orphan" inverse="true">
            <key column="fichaProyecto" not-null="true" />
            <one-to-many
                class="cu.uci.ccv.presentacion.vo.DFinanciamiento" />
        </set>
        <one-to-one name="DCronograma"
            class="cu.uci.ccv.contratacion.vo.DCronograma" cascade="all"
            property-ref="fichaProyecto" access="field" />
        <many-to-one name="marcoAprobacion" cascade="none"
            column="marcoAprobacion" class="cu.uci.ccv.presentacion.vo.DMixta"
            not-null="false" access="field">
        </many-to-one>
(3)     <set name="Modalidad" table="dFicha_Proyecto_Modalidad"
            lazy="false" cascade="none">
            <key column="fichaProyecto" not-null="true" />
            <many-to-many column="modalidad"
                class="cu.uci.ccv.presentacion.vo.NModalidad">
            </many-to-many>
        </set>
    </class>
    </hibernate-mapping>
```

Figura 17. Archivo de mapeo DFicha_Proyecto.hbm.xml

(1) representa una asociación bidireccional muchos-a-uno y es el tipo más común de asociación.
A nivel de bases de datos para expresar relaciones mucho a mucho se crea una tabla intermedia que posee como identificadores los respectivos identificadores de las tablas que la generan. Esta puede o no tener campos adicionales. (**Ver Figura 18**)
(2) se utiliza para mapear una relación mucho a mucho con atributos intermedios. Esta relación se mapea a través de la etiqueta <set> que contiene un <one-to-many> con DFinanciamiento que es la clase que representa la tabla intermedia en la base de datos.
(3) es la forma de mapear una relación mucho a mucho sin atributos intermedios. En este caso la etiqueta <set> contiene un <many-to-many> con NModalidad, debido a que en este caso la tabla intermedia no está representada por una clase persistente.

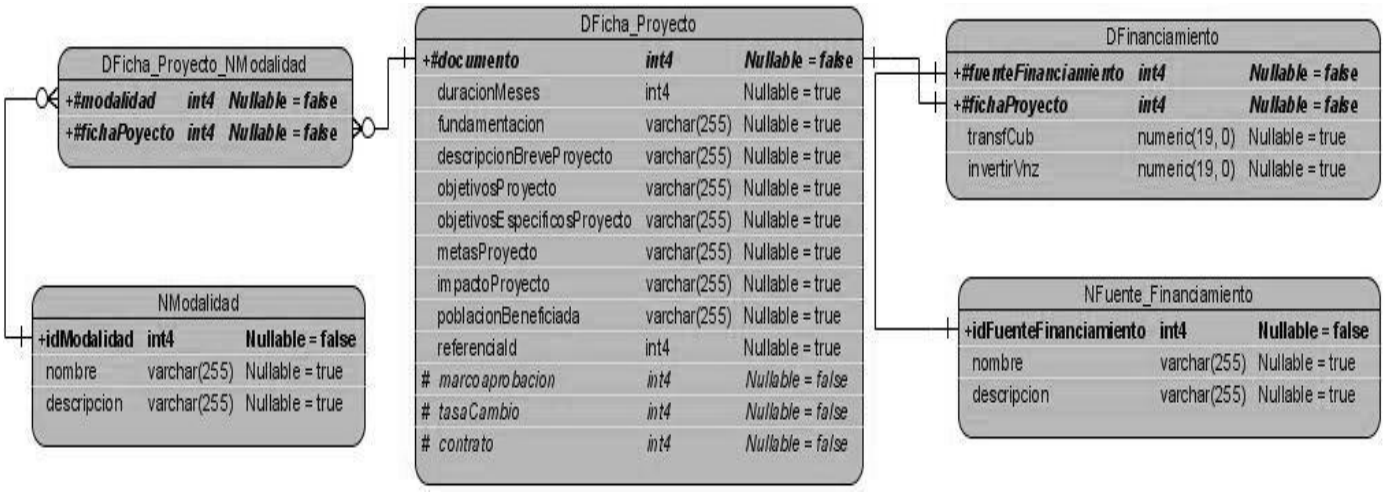


Figura 18. Relación many-to-many con y sin atributos en las talas intermedias

La clase persistente correspondiente al fichero DFicha_Proyecto.hbm.xml muestra algunas particularidades. (**Ver Anexo XII**)

Para garantizar la bidireccionalidad entre las clases DContrato y DFicha_Proyecto es necesario que al setearle un contrato a la ficha_proyecto este adicione la ficha a su Set, permitiendo la navegavilidad en ambos sentidos.

Tanto la relación con NFuente_Financiamiento como la relación con NModalidad se representan con objetos de tipo Set, que es la interface utilizada para mapear colecciones.

Para usar la interface Set es necesario tener redefinido los métodos *equals()* y *hashCode()* en las clases que representan la colección. Estos métodos permiten que no se dupliquen objetos teniendo en cuenta la clave primaria que funciona como identificador del objeto, además permiten realizar acciones como buscar y eliminar objetos en dicha colección basándose en dicha clave. (**Ver Anexo XIII**)

2.5.2 Objetos de acceso a datos (DAOs)

Con el objetivo de aislar la lógica del negocio de las operaciones con la base de datos se decidió utilizar un patrón de diseño que encapsulara la implementación de la capa de acceso a datos llamado DAO (*Data Access Object*). Este patrón se rige por una interfaz y una implementación, la interfaz tiene como objetivo servir de fachada entre la capa de negocio y la capa de acceso a datos. En la implementación se hace usa la API de hibernate por medio de una herencia de la clase abstracta HibernateDaoSupport que requiere una SessionFactory para ser fijada y brinda funcionalidades que permiten salvar, modificar, eliminar y cargar elementos de la base de datos mediante la plantilla de hibernate HibernateTemplate a través del método *getHibernateTemplate()*.

La implementación de los DAOs se realizó de forma que pudiese ser lo más reutilizable posible para todos los módulos del proyecto, implementando una clase BaseDaoImpl con su respectiva interfaz de la que heredan todos los DAOs, que contiene métodos básicos para lograr la persistencia.

```
package cu.uci.ccv.common.dao.impl;

import java.io.Serializable;

public class BaseDaoImpl<DomainObject, KeyType extends Serializable> extends
        HibernateDaoSupport implements BaseDao<DomainObject, KeyType> {

 (1) public void crear(DomainObject object) throws DAOException {
        try {
            this.getHibernateTemplate().save(object);
        } catch (HibernateException e) {
            String entidad = (object == null) ? "con parametro nulo" : object.getClass().getSimpleName();
            throw new DAOException("Error al intentar crear la entidad " + entidad, e);
        }
    }
```

```
(2) public void modificar(DomainObject object) throws DAOException {
        try {
            this.getHibernateTemplate().saveOrUpdate(object);
        } catch (HibernateException e) {
            String entidad = (object == null) ? "con parametro nulo" : object.getClass().getSimpleName();
            throw new DAOException("Error al intentar modificar la entidad " + entidad, e);
        }
    }

(3) public void eliminar(DomainObject object) throws DAOException {
        try {
            this.getHibernateTemplate().delete(object);
        } catch (HibernateException e) {
            String entidad = (object == null) ? "con parametro nulo" : object.getClass().getSimpleName();
            throw new DAOException("Error al intentar eliminar la entidad " + entidad, e);
        }
    }

    @SuppressWarnings("unchecked")
(4) public List<DomainObject> cargar(String query, String[] nombreparametros,
            Object[] valoresparametros) throws DAOException {
        try {
            return this.getHibernateTemplate().findByNamedQueryAndNamedParam(
                    query, nombreparametros, valoresparametros);
        } catch (HibernateException e) {
            throw new DAOException(
                    "Error al intentar cargar una lista de entidades de la BD ", e);
        }
    }

    @SuppressWarnings("unchecked")
(5) public DomainObject cargarPorId(DomainObject object, KeyType id)
            throws DAOException {
        try {
            return (DomainObject) getHibernateTemplate().load(object.getClass(), id);

        } catch (HibernateException e) {
            throw new DAOException("Error al intentar cargar por id la entidad "
                    + object.getClass().getSimpleName(), e);
        }
    }

    @SuppressWarnings("unchecked")
(6) public DomainObject getById(DomainObject object, KeyType id) {
        return (DomainObject) getHibernateTemplate().get(object.getClass(), id);
    }
}
```

Figura 19. Clase BaseDaoImpl.java

(1) Se utiliza para salvar un objeto transitorio en la base de datos.

(2) Se utiliza para salvar o modificar un objeto transitorio o persistente en la base de datos en dependencia del valor del atributo en el XML *usaved-value*.

(3) Se utiliza para eliminar un objeto persistente de la base de datos.

(4) Se utiliza para cargar objetos persistentes de la base de datos en dependencia de los parámetros que se le pasen al método en cuestión.

(5) Se utiliza para cargar un objeto persistente a partir de su identificador devolviendo una excepción en caso de no encontrar el objeto con la clave asociada al valor pasado.

(6) Se utiliza para cargar un objeto persistente a partir de su identificador devolviendo *null* en caso de no encontrar el objeto con la clave asociada al valor pasado.

Como se mencionó anteriormente los DAOs para fijarse necesitan de una SessionFactory que se configuró de la siguiente forma a través de una property en el bean llamada SessionFactory que hace referencia al bean anteriormente mencionado llamado sessionFactory:

```
<bean id="baseDao" class="cu.uci.ccv.common.dao.impl.BaseDaoImpl">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

Figura 20. Configuración del bean baseDao

Dentro de las DAOs que heredan de BaseDaoImpl se encuentra DocumentoDaoImpl que posee operaciones comunes para los DAOs FichaDaoImpl y CronogramaDaoImpl que a su vez heredan de DocumentoDaoImpl. A continuación se muestra la configuración del bean que representa a la clase DocumentoDaoImpl y las clases que heredan de él. (**Ver Anexos XIV y XV**)

```
<bean id="documentoDao"
    class="cu.uci.ccv.common.dao.impl.DocumentoDaoImpl"
    parent="baseDao">
</bean>
```

Figura 21. Configuración del bean documentoDao

```
<bean id="cronogramaDao"
    class="cu.uci.ccv.contratacion.dao.impl.CronogramaDaoImpl"
    parent="baseDao">
</bean>
```

Figura 22. Configuración del bean cronogramaDao

```
<bean id="fichaDao"
    class="cu.uci.ccv.presentacion.dao.impl.FichaDaoImpl"
    parent="documentoDao">
</bean>
```

Figura 23. Configuración del bean fichaDao

Observe que los beans fichaDao y cronogramaDao heredan a través del atributo parent del bean de documentoDao.

2.6 Proceso de Software Personal

A través del PSP se pueden practicar las habilidades y métodos que ingenieros del software profesionales han desarrollado durante muchos años de pruebas y errores. Mediante la experiencia acumulada se aprende rápidamente y se evita repetir errores. El PSP muestra cómo gestionar el tiempo, como planificar y controlar el trabajo y como producir regularmente programas de alta calidad. Puesto que lleva tiempo desarrollar las habilidades y hábitos de forma efectiva, el desarrollador debe practicar los métodos del PSP en cada ejercicio de software, de esta forma se aprende y se perfeccionan las habilidades antes de necesitarlas en el trabajo del ingeniero de software.

La forma de mejorar la calidad de del trabajo comienza por entender lo que realmente se hace, para esto es necesario conocer las tareas que se realiza, como se realizan y los resultados que se obtienen. Para realizar este proceso es de primordial relevancia conocer el tiempo dedicado a cada una de las tareas realizadas, esto permite que los desarrolladores ganen experiencia y rapidez en el trabajo logrando una aproximación entre el tiempo real y el tiempo planificado. Asimismo una buena gestión de los defectos permite dedicar más atención a aquellos aspectos de la calidad que afectan a la utilidad y valor de los programas que se desarrollen. PSP proporciona las

habilidades y prácticas que posibilitan entender los defectos que se introducen y dota al desarrollador de un mecanismo eficiente para encontrar y corregir muchos de los defectos.

Para gestionar el tiempo empleado y controlar los defectos se hace necesario el trabajo con los cuadernos de registro de tiempo y de defectos.

No.	Implementación	Tiempo	LOC	Min/LOC	Funciones
1	BaseDao.java (interface)	20	9	2.2222	Declaración de métodos
2	BaseDaoImpl.java	20	42	0.4761	Implementación de métodos
Modulo: Presentación de proyectos (POJOs, archivos de mapeo y DAOs)					
3	DFicha_Proyecto.hbm.xml	1863	621	3.0000	Configuración y consultas
4	DFicha_Proyecto.java	345	274	1.2591	Implementación de métodos simples
5	DActividad.hbm.xml	44	71	0.6197	Configuración y consultas
6	DActividad.java	437	502	0.8705	Implementación de métodos simples
7	DFinanciamiento.hbm.xml	12	32	0.3750	Configuración y consultas
8	DFinanciamiento.java	25	43	0.5581	Implementación de métodos simples
9	DGasto_Administrativo.hbm.xml	33	31	1.0645	Configuración y consultas
10	DGasto_Administrativo.java	20	97	0.2061	Implementación de métodos simples
11	DOtro_Recurso.hbm.xml	35	33	1.0606	Configuración y consultas
12	DOtro_Recurso.java	32	93	0.3440	Implementación de métodos simples
13	DRecurso_Humano.hbm.xml	49	47	1.0425	Configuración y consultas

14	DRecurso_Humano.java	87	163	0.5337	Implementación de métodos simples
15	DRecurso_Material.hbm.xml	35	35	1.0000	Configuración y consultas
16	DRecurso_Material.java	78	94	0.8297	Implementación de métodos simples
17	NFuente_Financiamiento.hbm.xml	10	27	0.3703	Configuración y consultas
18	NFuente_Financiamiento.java	8	39	0.2051	Implementación de métodos simples
19	NModalidad.hbm.xml	12	24	0,5000	Configuración y consultas
20	NModalidad.java	9	43	0.2093	Implementación de métodos simples
21	FichaDaoImpl.java	4257	413	10.3075	Implementación de métodos complejos
22	FichaDao.java	78	54	1.4444	
Modulo: Contratación de proyectos (POJOs, archivos de mapeo y DAOs)					
23	DContrato.hbm.xml	1070	647	1.6537	Configuración y consultas
24	DContrato.java	150	137	1.0948	Implementación de métodos simples
25	DCronograma.hbm.xml	988	202	4.8910	Configuración y consultas
26	DCronograma.java	29	45	0.6444	Implementación de métodos simples
27	DDesembolsoInv.hbm.xml	17	26	0.6538	Configuración y consultas
28	DDesembolsoInv.java	32	118	0.2711	Implementación de métodos simples
29	DDesembolsoTransf.hbm.xml	19	26	0.7307	Configuración y consultas
30	DDesembolsoTransf.java	45	124	0.2639	Implementación de métodos simples
31	ContratoDaoImpl.java	5289	478	11.0648	Implementación de métodos complejos

32	ContratoDao.java	120	113	0.9022	Declaración de métodos
33	CronogramaDaoImpl.java	1698	145	11.7103	Implementación de métodos complejos
34	CronogramaDao.java	42	53	0.7924	Declaración de métodos
35	DesembolsoInvDaoImpl.java	35	27	1.2972	Implementación de métodos complejos
36	DesembolsoInvDao.java	6	17	0.2539	Declaración de métodos
37	DesembolsoTransfDaoImpl.java	38	29	1.3103	Implementación de métodos complejos
38	DesembolsoTransfDao.java	8	14	0.5714	Declaración de métodos

Figura 24. Cuaderno de registro de tiempo

Tipos de defectos	
10	Documentación
20	Sintaxis
30	Construcción paquetes
40	Asignación
50	Interfaz
60	Comprobación
70	Datos
80	Función
90	Sistema
100	Entorno

Figura 25. Tipos de defectos

Fecha	Número	Tipo	Introducido	Eliminado	Tiempo de corrección(min)	Defecto corregido
22/4/2008	1	20	codificación	compilación	1	X

Descripción: punto y coma omitido						
23/4/2008	2	20	codificación	compilación	1	X
Descripción: etiqueta de XML no cerrada						
	3	20	codificación	compilación	10	X
Descripción: operadores lógicos incorrectamente posicionados en las consultas HQL						
28/4/2008	4	20	codificación	prueba	25	X
Descripción: uso incorrecto de la unión de tablas en las consultas HQL						
01/5/2008	5	80	codificación	prueba	47	X
Descripción: base de datos no sincronizada con la Session de Hibernate						
02/5/2008	6	40	codificación	prueba	1	X
Descripción: intentos de persistir objetos con referencias nulas						
04/5/2008	6	40	codificación	prueba	5	X
Descripción: intentar acceder a objetos *separados* por Hibernate						
05/5/2008	7	80	Codificación	prueba	78	X
Descripción: error al cargar en la Session de Hibernate dos objetos diferentes con el mismo identificador.						
10/5/2008	8	40	codificación	prueba	24	X
Descripción: configuración errónea en los hbm.xml (cascada)						
14/5/2008	9	40	codificación	compilación	3	X
Descripción: asignación incorrecta de parámetros						
17/5/2008	10	40	codificación	prueba	26	X
Descripción: configuración errónea en los hbm.xml (relación many-to-one)						
	11	20	codificación	compilación	5	X
Descripción: operadores lógicos incorrectamente posicionados en las consultas HQL						
22/5/2008	12	40	codificación	compilación	3	X
Descripción: incompatibilidad de tipos de datos						
27/5/2008	13	40	codificación	prueba	21	X
Descripción: configuración incorrecta en los hbm.xml (one-to-one)						
02/06/2008	14	40	codificación	prueba	5	X
Descripción: configuración incorrecta en los hbm.xml (many-to-one)						
08/6/2008	15	80	codificación	prueba	29	X
Descripción: base de datos no sincronizada con la Session de Hibernate						
17/6/2008	16	20	codificación	prueba	12	X
Descripción: uso incorrecto de la unión de tablas en las consultas HQL						
17/6/2008	16	80	codificación	prueba	158	X

Descripción: incoherencia entre la cantidad de objetos reales en el reporte de proyecto y el valor que muestra la cantidad.						
17/6/2008	16	80	codificación	prueba	92	X
Descripción: implementación incorrecta de la concurrencia						

Figura 26. Cuaderno de registro de defectos

2.7 Conclusiones del capítulo

Al término del capítulo han sido implementados los elementos relacionados con la capa de acceso a datos que anteriormente habían sido creados y validados por el diseñador. El uso del patrones DAO permitió incrementar la legibilidad en el código, la reutilización del mismo y de esta forma disminuir el tiempo de implementación de los componentes de la capa de acceso a datos. Se ha logrado la persistencia y lectura de datos desde la base de datos en el lenguaje Java aprovechando la facilidad en la manipulación de objetos y otras potencialidades que ofrece la herramienta ORM Hibernate. De esta forma se ha demostrado la viabilidad de lograr la implementación de los componentes de la capa de persistencia de datos utilizando Hibernate, logrando el vínculo entre el modelo de objetos de la aplicación y el relacional de la base de datos y dejando al implementador de la capa de negocio la responsabilidad de implementar las reglas del negocio.

CAPITULO 3. Validación de la solución propuesta

3.1 Introducción

En este capítulo se abordara la temática relacionada con las pruebas de unidad que permitan validar la solución propuesta. Primeramente, se desarrollan algunos conceptos que deber tener presente el implementador, se explicará la importancia de las pruebas de unidad y se desarrollarán pruebas de unidad a los componentes pertenecientes a la capa de persistencia de los módulos de Presentación y Contratación a través del framework JUnit.

3.2 Pruebas de unidad en Java

Uno de los pasos importantes que debe tener en cuenta el rol de programador es evaluar o probar los componentes resultantes de implementar los elementos de diseño. Las pruebas surgen con la necesidad de prevenir y detectar los errores en la implementación del código. Además las pruebas ayudan al programador a evitar escribir código incorrecto permitiendo implementar componentes con calidad. Una de las pruebas más importantes que puede realizar el programador es la prueba de unidad. Una prueba de unidad examina el comportamiento individual de las unidades de trabajo. En una aplicación Java, una unidad de trabajo individual es a menudo un solo método, aunque no siempre. Una unidad de trabajo es una tarea que no es directamente dependiente del completamiento de otra tarea. (Massol, et al., 2004) Las pruebas de unidad experimentan si un método cumple los términos de su API contrato. Una API contrato es un acuerdo hecho por la interfaz del método. El acuerdo se realiza entre los valores de los objetos entrada del método y los valores de los objetos que retorna el método. De esta forma, si la conducta del método no es la esperada entonces podemos decir que el método rompió el contrato.

Es necesario conocer cómo aplicar las pruebas y que herramientas utilizar debido a que una mala selección puede causar pérdida de tiempo en la realización.

En Java la implementación de las pruebas de unidad puede realizarse utilizando clases de pruebas improvisadas que dado algunos valores de entrada comprueban los

valores esperados por una unidad de trabajo. La dificultad de estas clases está dada porque no cumplen con los siguientes aspectos:

- Cada unidad de trabajo debe correr independientemente de todas las otras unidades de trabajo.
- Los errores deben ser detectados y reportados pruebas por pruebas.
- Debe ser fácil de definir cual prueba de unidad correr en el momento que se desee.

Con el objetivo de solucionar estos problemas han sido creadas herramientas muy potentes para desarrollar pruebas de unidad. Entre ellos el más popular es el framework JUnit.

3.3 Framework JUnit

El framework JUnit posee un grupo de características que facilitan la escritura y ejecución de las pruebas. Estas son:

- Alterna las entradas-salidas, o ejecución de las pruebas, para mostrar el resultado de las pruebas.
- Separa los cargadores de clase (classloaders), por cada prueba de unidad para evitar efectos secundarios
- Recurso de inicialización y reclamación de métodos estándar(*setUp()* y *tearDown()*)
- Una variedad de métodos *assert()* para facilitar el chequeo de los resultados de la pruebas.
- Integración con varios IDEs entre ellos Eclipse.

Las pruebas de unidad utilizando JUnit se realizan a través de casos de pruebas. Los casos de pruebas incluyen una o varias pruebas de unidad.
La siguiente tabla muestra los aspectos que se tuvieron en cuenta para seleccionar y desarrollar las pruebas de unidad a determinadas unidades de trabajo utilizando el framework JUnit:

Caso de prueba:	FichaDaoTest			
Unidad de trabajo:	TestCrearFicha			
Precondiciones:				
Objetivo	Valores de entrada	Resultado esperado	Resultado real	Tipo de resultado
Insertar el objeto de la clase DFicha_Proyecto en la tabla correspondiente en la base de datos.	Objeto de la clase DFicha_Proyecto.	Se espera que el objeto de la clase DFicha_Proyecto sea insertado satisfactoriamente en la tabla correspondiente.	Se inserto el objeto de la clase DFicha_Proyecto satisfactoriamente. Se insertaron correctamente los valores definidos en las propiedades del objeto en las columnas de la tabla correspondiente en la base de datos.	Correcto.
Unidad de trabajo:	TestModificarFicha			
Precondiciones:				
Objetivo	Valores de entrada	Resultado esperado	Resultado real	Tipo de resultado
Modificar el objeto de la clase DFicha_Proyecto en la tabla correspondiente en la base de datos.	Objeto de la clase DFicha_Proyecto.	Se espera que el objeto de la clase DFicha_Proyecto sea modificado satisfactoriamente en la tabla correspondiente.	Se modifico el objeto de la clase DFicha_Proyecto satisfactoriamente, además se modifico la tupla especificada por el objeto en la tabla de la base de datos correspondiente.	Correcto
Unidad de trabajo:	TestEliminarFicha			

Precondiciones:				
Objetivo	Valores de entrada	Resultado esperado	Resultado real	Tipo de resultado
Eliminar el objeto de la clase DFicha_Proyecto en la tabla correspondiente en la base de datos.	Objeto de la clase DFicha_Proyecto.	Se espera que el objeto de la clase DFicha_Proyecto sea eliminado satisfactoriamente en la tabla correspondiente.	Se elimino el objeto de la clase DFicha_Proyecto satisfactoriamente, además se elimino la tupla especificada por el objeto en la tabla de la base de datos correspondiente.	Correcto
Unidad de trabajo:	TestReportePlanInversion			
Precondiciones:				
Objetivo	Valores de entrada	Resultado esperado	Resultado real	Tipo de resultado
Obtener satisfactoriamente el plan de inversión de un objeto de la clase DFicha_Proyecto.	Identificador del objeto de la clase DFicha_Proyecto.	Se espera obtener el objeto de la clase DFicha_Proyecto con los valores necesarios para mostrar el plan de inversión satisfactoriamente.	Se obtuvo el objeto de la clase DFicha_Proyecto satisfactoriamente con los valores necesarios para mostrar el plan de inversión.	Correcto

Caso de prueba:	ContratoDaoTest			
Unidad de trabajo:	TestCrearContrato			
Precondiciones:				
Objetivo	Valores de entrada	Resultado esperado	Resultado real	Tipo de resultado
Insertar el objeto de la	Objeto de la clase	Se espera que el	Se inserto el	Correcto.

clase DContrato en la tabla correspondiente en la base de datos.	DContrato.	objeto de la clase DContrato sea insertado satisfactoriamente en la tabla correspondiente.	objeto de la clase DContrato satisfactoriamente. Se insertaron correctamente los valores definidos en las propiedades del objeto en las columnas de la tabla correspondiente en la base de datos.	
Unidad de trabajo:	TestModificarContrato			
Precondiciones:				
Objetivo	Valores de entrada	Resultado esperado	Resultado real	Tipo de resultado
Modificar el objeto de la clase DContrato en la tabla correspondiente en la base de datos.	Objeto de la clase DContrato.	Se espera que el objeto de la clase DContrato sea modificado satisfactoriamente en la tabla correspondiente.	Se modifico el objeto de la clase DContrato satisfactoriamente, además se modificó la tupla especificada por el objeto en la tabla de la base de datos correspondiente.	Correcto
Unidad de trabajo:	TestEliminarContrato			
Precondiciones:				
Objetivo	Valores de entrada	Resultado esperado	Resultado real	Tipo de resultado

Eliminar el objeto de la clase DContrato en la tabla correspondiente en la base de datos.	Objeto de la clase DContrato.	Se espera que el objeto de la clase DContrato sea eliminado satisfactoriamente en la tabla correspondiente.	Se eliminó el objeto de la clase DContrato satisfactoriamente, además se elimino la tupla especificada por el objeto en la tabla de la base de datos correspondiente.	Correcto
Unidad de trabajo:	TestCargarFichasPorContrato			
Precondiciones:				
Objetivo	Valores de entrada	Resultado esperado	Resultado real	Tipo de resultado
Obtener satisfactoriamente un listado de objetos de la clase DFicha_Proyecto asociadas al objeto de la clase DContrato.	Identificador del objeto de la clase DContrato.	Se espera obtener un listado de la clase DFicha_Proyecto pertenecientes al objeto de la clase DContrato.	Se obtuvo el listado de la clase DFicha_Proyecto pertenecientes al objeto de la clase DContrato satisfactoriamente.	Correcto.

Caso de prueba:	CronogramaDaoTest			
Unidad de trabajo:	TestCrearCronograma			
Precondiciones:				
Objetivo	Valores de entrada	Resultado esperado	Resultado real	Tipo de resultado

Insertar el objeto de la clase DCronograma en la tabla correspondiente en la base de datos.	Objeto de la clase DCronograma.	Se espera que el objeto de la clase DCronograma sea insertado satisfactoriamente en la tabla correspondiente.	Se inserto el objeto de la clase DCronograma satisfactoriamente. Se insertaron correctamente los valores definidos en las propiedades del objeto en las columnas de la tabla correspondiente en la base de datos.	Correcto.
Unidad de trabajo:	TestModificarCronograma			
Precondiciones:				
Objetivo	Valores de entrada	Resultado esperado	Resultado real	Tipo de resultado
Modificar el objeto de la clase DCronograma en la tabla correspondiente en la base de datos.	Objeto de la clase DCronograma.	Se espera que el objeto de la clase DCronograma sea modificado satisfactoriamente en la tabla correspondiente.	Se modifico el objeto de la clase DCronograma satisfactoriamente, además se modifico la tupla especificada por el objeto en la tabla de la base de datos correspondiente.	Correcto

Unidad de trabajo:	TestEliminarCronograma			
Precondiciones:				
Objetivo	Valores de entrada	Resultado esperado	Resultado real	Tipo de resultado
Eliminar el objeto de la clase DCronograma en la tabla correspondiente en la base de datos.	Objeto de la clase DCronograma.	Se espera que el objeto de la clase DCronograma sea eliminado satisfactoriamente en la tabla correspondiente.	Se elimino el objeto de la clase DCronograma satisfactoriamente, además se elimino la tupla especificada por el objeto en la tabla de la base de datos correspondiente.	Correcto
Unidad de trabajo:	TestCargarFichaParaCronograma			
Precondiciones:				
Objetivo	Valores de entrada	Resultado esperado	Resultado real	Tipo de resultado
Obtener satisfactoriamente el objeto de la clase DFicha_Proyecto y su relación con el objeto de la clase DCronograma.	Identificador del objeto de la clase DFicha_Proyecto.	Se espera obtener un objeto de la clase DFicha_Proyecto y su relación con el objeto de la clase DCronograma.	Se obtuvo el objeto de la clase DFicha_Proyecto y su relación.	Correcto.

Figura 27. Aspectos que se tuvieron en cuenta para seleccionar y desarrollar las pruebas de unidad

Para implementar los casos de pruebas, el framework JUnit ofrece la clase TestCase, que es heredada por las clases que contienen las pruebas para validar a través de sus métodos las unidades de trabajo que serán probadas.

A continuación se muestra la implementación del caso de prueba FichaDaoTest, incluyendo la unidad de trabajo testCrearFicha:

```
public class FichaDaoTest extends TestCase {  (1)
    private ApplicationContext factory = null;
    private FichaDao fichaDao = null;

    public FichaDaoTest() {
        super();
        String[] contexts = new String[] { "presentacion-applicationContext-dao.xml" };
        factory = new ClassPathXmlApplicationContext(contexts);
    }

    @Before
    public void setUp() throws Exception {  (2)
        super.setUp();
        fichaDao = (FichaDao) factory.getBean("fichaDao");
    }

    @After
    public void tearDown() throws Exception {  (3)
        super.tearDown();
        fichaDao = null;
    }

    public void testCrearFicha() throws Exception {  (4)
        NTipo_Documento td=new NTipo_Documento();
        td.setIdTipoDocumento(1);
        NEstado estado=new NEstado();
        estado.setIdEstado(1);
        NProceso proceso=new NProceso();
        proceso.setIdProceso(1);
```

```
        DDocumento documento=new DDocumento();
        documento.setNombre("ficha");
        documento.setTipoDocumento(td);
        documento.setEstado(estado);
        documento.setProceso(proceso);
        documento.setActivo(new Boolean(false));

        DFicha_Proyecto fp=new DFicha_Proyecto(); (5)
        NModalidad modalidad=new NModalidad();
        modalidad.setIdModalidad(1);
        NModalidad modalidad1=new NModalidad();
        modalidad1.setIdModalidad(2);
        Set<NModalidad> modalidades=new HashSet<NModalidad>();
        modalidades.add(modalidad);
        modalidades.add(modalidad1);

        fp.setDuracionMeses(1);
        fp.setFundamentacion("fundamentacion");
        fp.setDescripcionBreveProyecto("descripcion");
        fp.setObjetivosProyecto("objetivosP");
        fp.setObjetivosEspecificosProyecto("objetivosEspP");
        fp.setMetasProyecto("metas");
        fp.setImpactoProyecto("impacto");
        fp.setPoblacionBeneficiada("poblacion");
        fp.setModalidad(modalidades);
        fp.setTasaCambio(new NTasa_Cambio(1));
        fp.setMarcoAprobacion(null);
        fp.setDocumento(documento);
        fichaDao.crear(fp); (6)
        assertNotNull(fp.getDocumento().getIdDocumento()); (7)
    }

}
```

Figura 28. Caso de prueba FichaDaoTest, incluyendo la unidad de trabajo testCrearFicha

(1) Muestra la herencia existente entre la clase FichaDaoTest y la clase base estándar de JUnit, junit.framework.TestCase. Esta clase base incluye los métodos necesarios para realizar las pruebas automáticamente.

(2) Este método se encarga de inicializar los recursos o datos necesarios para realizar las pruebas, se ejecuta antes de comenzarlas.

(3) Este método se encarga de liberar los objetos o recursos que hayan sido utilizados en las pruebas una vez después de culminadas estas.

(4) Este método representa la prueba de unidad y es ejecutado automáticamente, para una mejor practica se sigue el patrón de nomenclatura *testXXX()*.

(5) Representa la instancia del objeto que será sometido a prueba.

(6) Representa la operación que será probada dentro de la prueba de unidad.

(7) Es la forma de verificar los resultados de la prueba ejecutada. El método *asserNotNull (Object o)* permite verificar si el objeto sometido a prueba es o no nulo, el resultado es mostrado a través de reportes que nos brinda el framework como se muestra a continuación:

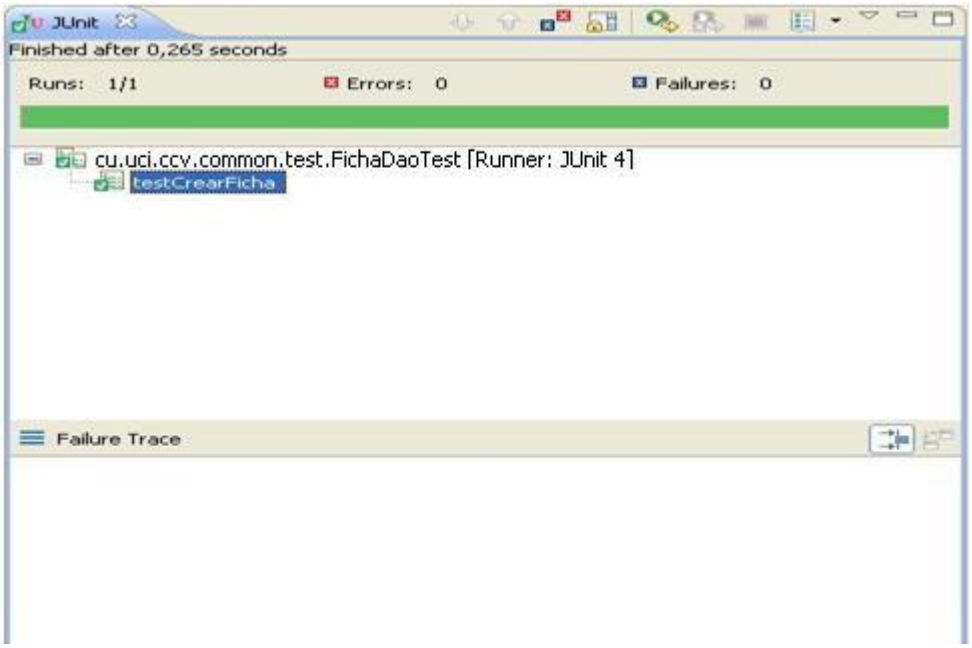


Figura 29. Visualización de una prueba satisfactoria con JUnit

En caso de que falle la prueba, el reporte quedaría mostrado de la siguiente forma:

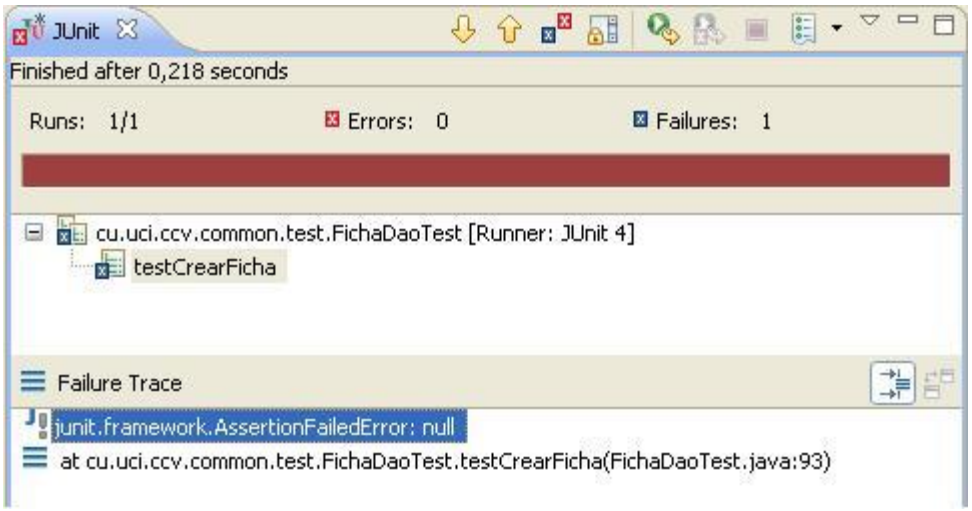


Figura 30. Visualización de una prueba no satisfactoria con JUnit

Además de realizar pruebas de unidad con el framework JUnit se hace necesario verificar la persistencia de los datos en la base de datos. A continuación se muestra

como quedó creado el objeto de tipo DFicha_Proyecto a partir de la ejecución de la unidad de trabajo testCrearFicha.

id	nombre	tipodocumento	estado	proceso	activ	anterior	modificar	fechacreacion
1	fichaProyectoTest1	1	1	1	☐	Null	1	Null
2	fichaProyectoTest2	1	1	1	☐	Null	1	Null
3	fichaProyectoTest3	1	1	1	☐	Null	1	Null
4	fichaProyectoTest4	1	1	1	☐	Null	1	Null

Figura 31. Resultado de la prueba en la tabla DFicha_Proyecto en la base de datos

Otros ejemplos de pruebas de unidad realizadas en la implementación de los módulos de Presentación y Contratación de proyectos y los reportes mostrados por el framework JUnit:

Unidad de trabajo *testCargarFichasPorContrato*:

```
public void testCargarFichasPorContrato() throws Exception {
    List<Object[]> values = contratoDao.cargarFichasPorContrato(2000);
    assertEquals(2, values.size());
}
```

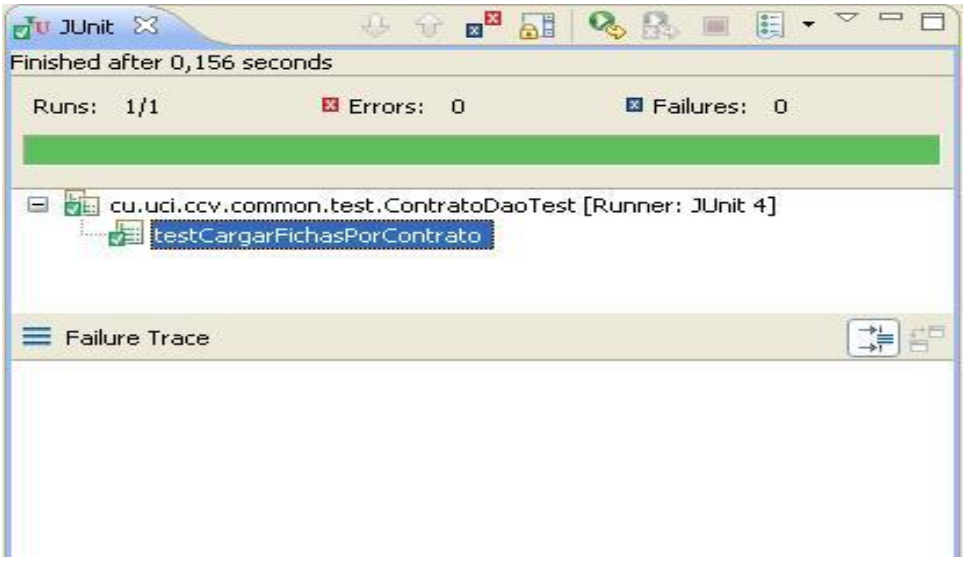


Figura 32. Visualización con JUnit de la prueba de unidad testCargarFichasPorContrato

Unidad de trabajo *testCargarFichaParaCronograma:*

```
public void testCargarFichasParaCronograma() throws Exception {
    DFicha_Proyecto value = cronogramaDao.cargarFichaParaCronograma(4);
    assertNotNull(value);
}
```

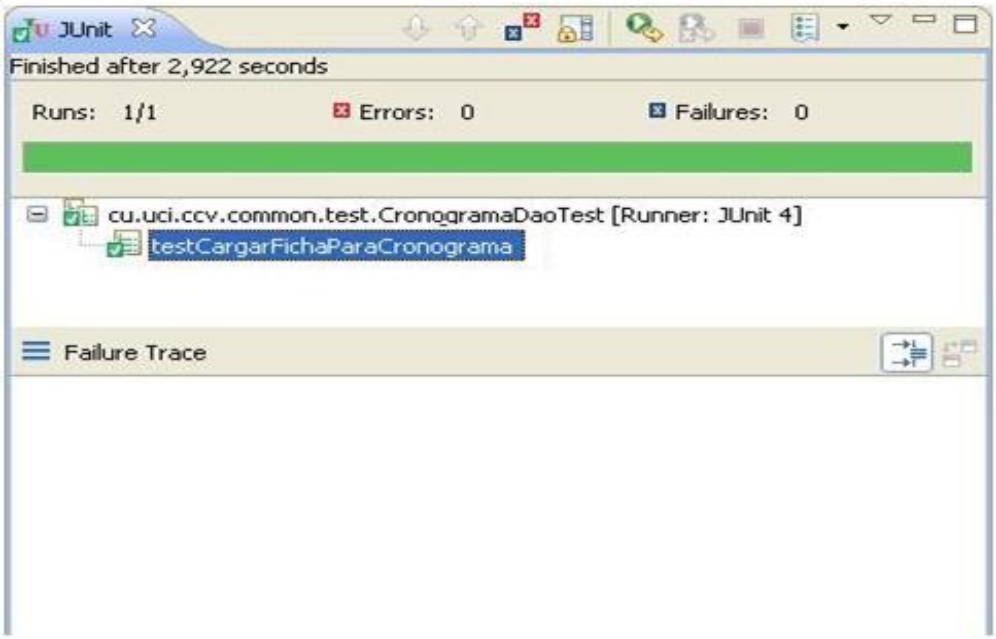


Figura 33. Visualización con JUnit de la prueba de unidad testCargarFichaParaContrato

Unidad de trabajo *testReportePlanInversion:*

```
public void testReportePlanInversion() throws Exception {
    DFicha_Proyecto value = fichaDao.reportePlanInversionFicha(4);
    assertNotNull(value);
}
```

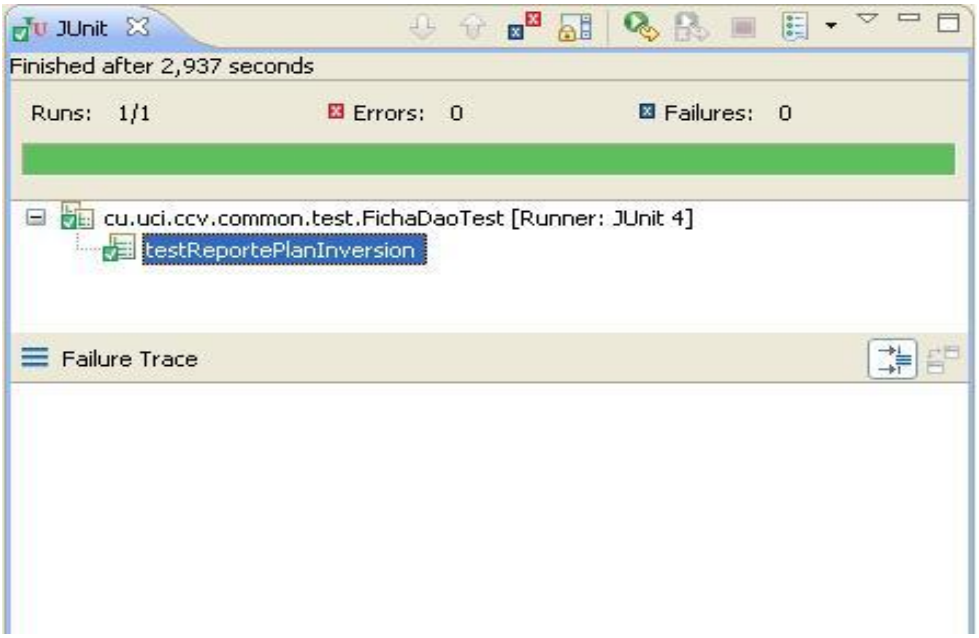


Figura 34. Visualización con JUnit de la prueba de unidad testReportePlanInversion

3.4 Conclusiones

A través de este capítulo se demostró que las pruebas de unidad simples no son difíciles de escribir manualmente, pero al aumentar su complejidad, su escritura y mantenimiento resulta más difícil. Además se pudo constatar que en temas de pruebas de unidad en Java el framework JUnit facilita la creación de pruebas útiles, que disminuyen el costo de la escritura de las pruebas mediante la reutilización de código. Asimismo quedó validado de forma fehaciente la solución propuesta para resolver el problema de la investigación.

Conclusiones

Mediante el presente trabajo de diploma se efectuó un estudio del arte acerca de la metodología de desarrollo RUP y su fase de implementación. Además se analizaron las potencialidades del lenguaje de programación Java y el IDE Eclipse. Conjuntamente se realizó un estudio detallado de los esquemas de persistencia, alternativas para realizar pruebas de unidad y métodos y habilidades para implementar la capa de persistencia de datos de los módulos Presentación y Contratación del proyecto CICCV. Lo anterior concluyó en:

- Se vinculó el mundo objetual y el relacional a través de la Herramienta ORM Hibernate, logrando una mayor calidad en la gestión y persistencia de los datos y garantizó las entradas esperadas por la capa de lógica del negocio.
- Se logró implementar los componentes aplicando patrones de diseño y técnicas adecuadas a la hora programar los atributos, operaciones y asociaciones, que permitieron la legibilidad y rehúso del código.
- Paralelamente a la implementación se aplicaron métodos como el PSP que permitieron disminuir el tiempo de implementación y producir programas de alta calidad.
- Quedó validada la propuesta de solución mediante la aplicación de pruebas de unidad a los componentes implementados, utilizando el framework JUnit.

Recomendaciones

Se recomienda a otros desarrolladores de la capa de persistencia implementar el mapeo relacionado con la herencia utilizando estrategias polimórficas, teniendo en cuenta que de esta manera se hace más legible el código.

Bibliografía

Abián, Miguel Angel. 2003. javahispano. *javahispano.* [En línea] 04 de 10 de 2003. [Citado el: 16 de 02 de 2009.] http://www.javahispano.org/contenidos/es/el_archipielago_eclipse__2_parte/.

Aulbach, Alexander, y otros. 2001. *Manual de PHP.* s.l. : Free Software Foundation, 2001.

Bauer, Christian y King, Gavin. 2005. *Hibernate in Action.* 2005.

Bobrow, D. y Stefik, M. 1986. *Perspectives on Artificial Intelligence Programming.* s.l. : Science, 1986. vol. 23 1, p. 951.

Bonanata, M. 2003. *Programación y algoritmos.* Buenos Aires : MP Ediciones, 2003.

Booch, G. 1996. *Análisis y Diseño Orientado a Objetos con Aplicaciones.* s.l. : 2ª Ed. Addison-Wesley/Díaz de Santos, 1996.

Díaz, Luis Carlos, Carrillo, Angela y Alvarado, Deicy. 2008. http://74.125.95.132/search?q=cache:vpx7LfU6TqoJ:sophia.javeriana.edu.co/~lcdiaz/ADOO2008-1/IngSoftwareEnADOO(IS-RUP-UML).pdf+Ventajas+de+Rup&hl=es&ct=clnk&cd=3&gl=cu. [En línea] 2008.

Dijkstra, Edsger W. 1979. *Programming considered as a human activity.* New York : NY Yourdon Press, 1979.

Fernandez de la Pera, Elian Luis y López Naranjo, Danis. 2007. *Desarrollo de la capa de acceso a datos para los modulos de administracion y nomencladores del sistema de gestion de inventario de almacenes SIGIA.* La Habana : s.n., 2007.

Gonzalez Seco, Jose Antonio. 2001. *El lenguaje de programacion C#.* 2001.

Jacobson, Ivar, Booch, Grady y Rumbaugh, James. 2000. *El Proceso Unificado de Desarrollo de Software.* Madrid : s.n., 2000.

Johnson, Ralph E. y Foote, Brian. June/July 1988. *Journal of Object-Oriented Programming.* Department of Computer Science, University of Illinois at Urbana-Champaign : s.n., June/July 1988.

Leyet Fernández, Osmar y Rodriguez Lorenzo, Iosmel. 2008. *Desarrollo de una herramienta generadora de ficheros de mapeo para la persistencia de esquemas de objetos relacionales basada en NHibernate.* Ciudad de la Habana : s.n., 2008.

Liskov, B. 1988. *Data Abstraction and Hierarchy.* s.l. : SIGPLANNofices, 1988.

Massol, Vincent y Husted, Ted. 2004. *JUnit in action.* 209 Bruce Park Avenue, Geenwich 06830 : Manning Publications Co, 2004. ISBN 1-930 110-99-5.

Mendoza Sanchez, María A. 2004. http://www.rhernando.net/modules/tutorials/doc/ing/met_soft.html. [En línea] 7 de Junio de 2004.

Microsystems, Sun. *El lenguaje de Programación Java™.*

Pacheco, Alberto. 2008. http://expo.itchihuahua.edu.mx/view.php?f=prog_10. [En línea] 26 de Jan de 2008.

Paola Pérez, Yanina y Marina López, Lidia. 2007. *MULTIPARADIGMA EN LA ENSEÑANZA DE LA PROGRAMACION.* 2007.

Paterson, Jim. 2004. onjava. *onjava.* [En línea] 12 de 01 de 2004. [Citado el: 17 de 02 de 2009.] http://www.onjava.com/pub/a/onjava/2004/12/01/db4o.html.

Pizarro, Pablo. 2006. arquitectura-de-software. *arquitectura-de-software.* [En línea] 23 de 05 de 2006. [Citado el: 20 de 02 de 2009.] http://arquitectura-de-software.blogspot.com/2006/05/orm-object-relational-mapping-i-parte.html.

RUP. 2003. *Rational Unified Process (Rational Help).* s.l. : Rational Software Corporation, 2003.

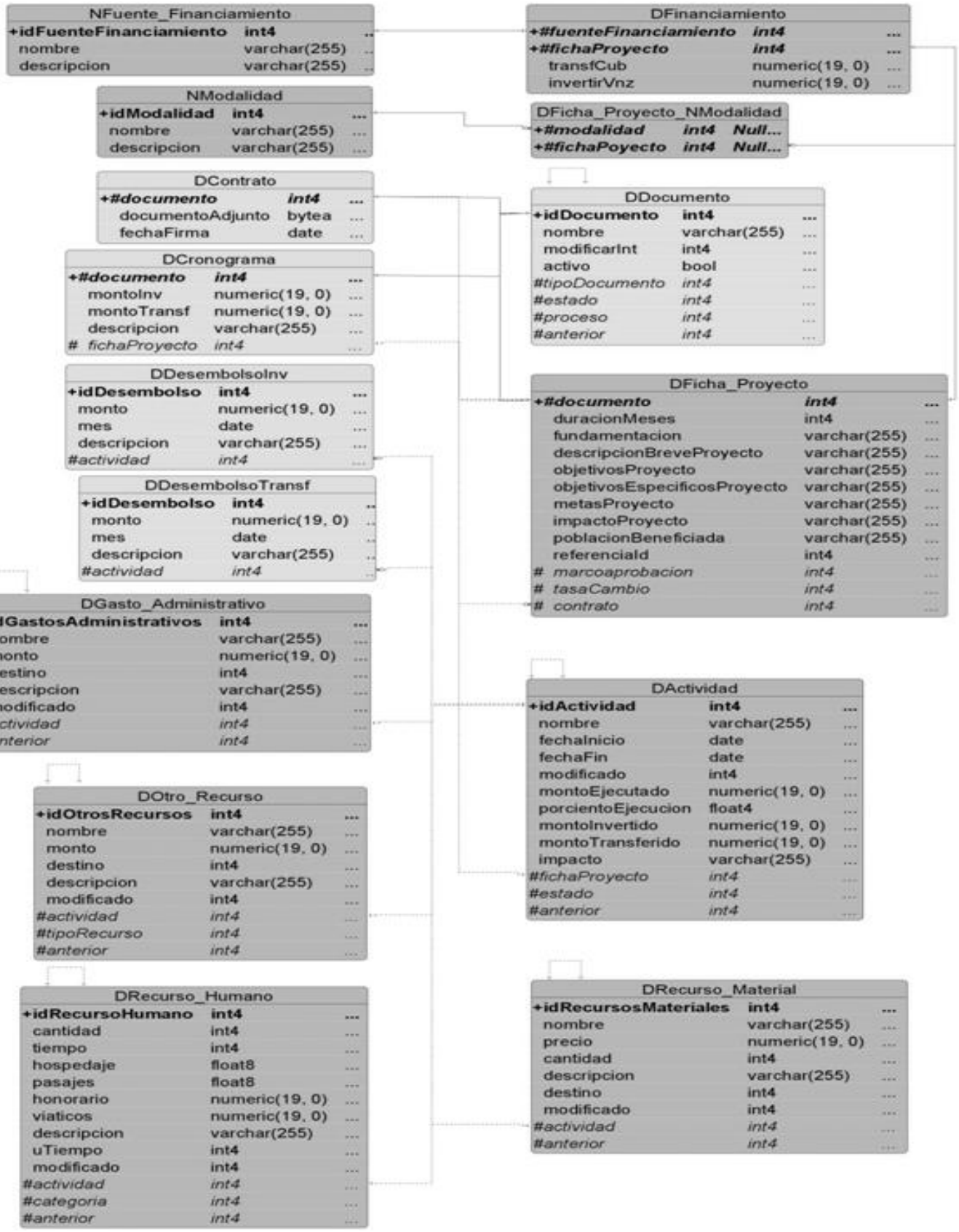
Schildt, Herbert. 2001. *Java 2 Manual de referencia.* Madrid : s.n., 2001.

Turtschi, Adrian, y otros. *C# . N ET Web Developer's Guide.* s.l. : Syngress Publishing, Inc.

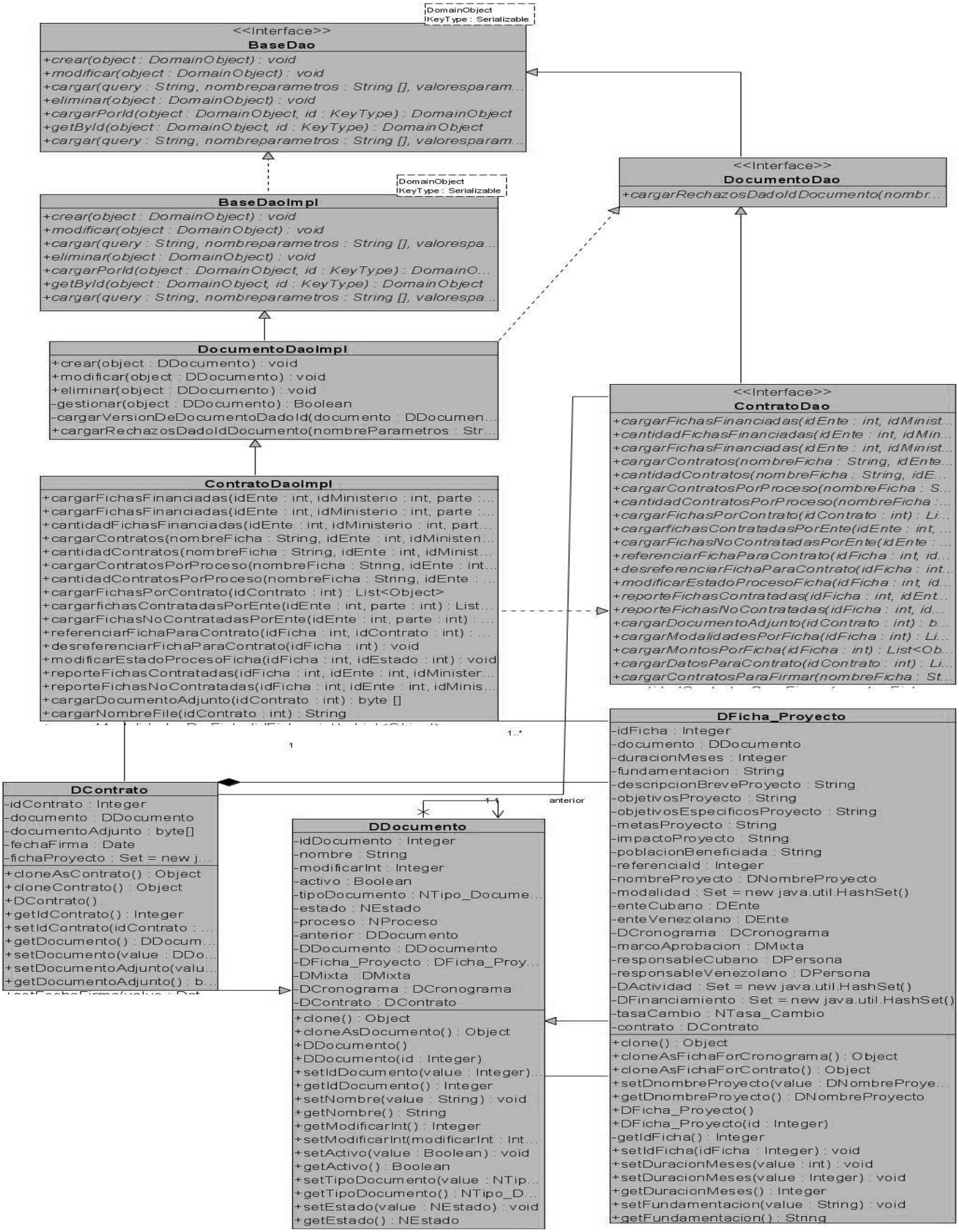
Zárate Rea, Hector. 2008. *Paradigmas de Programacion.* Ciudad Mexico : s.n., 2008.

Anexos

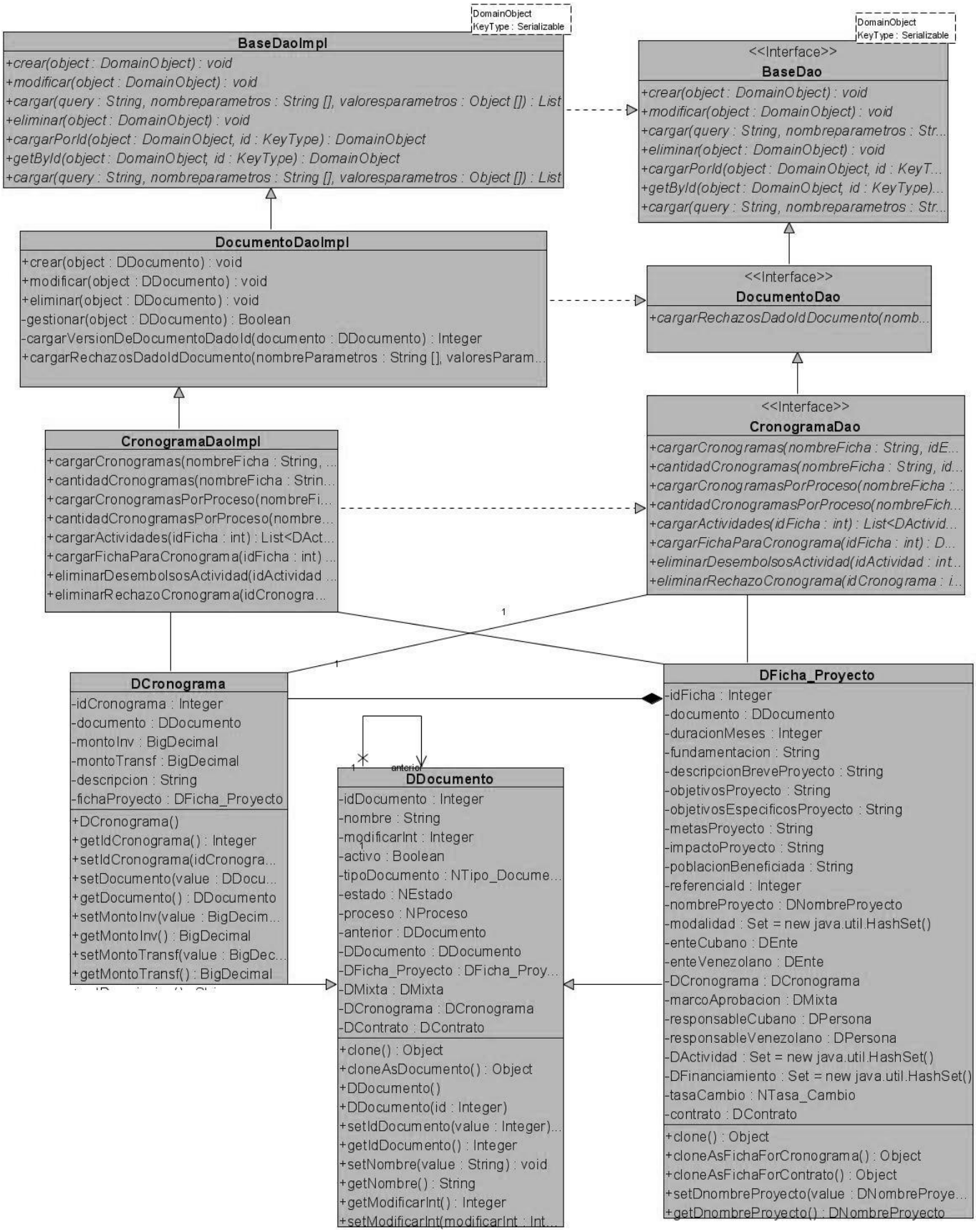
Anexo I. Modelo de datos de los módulos Presentación y Contratación



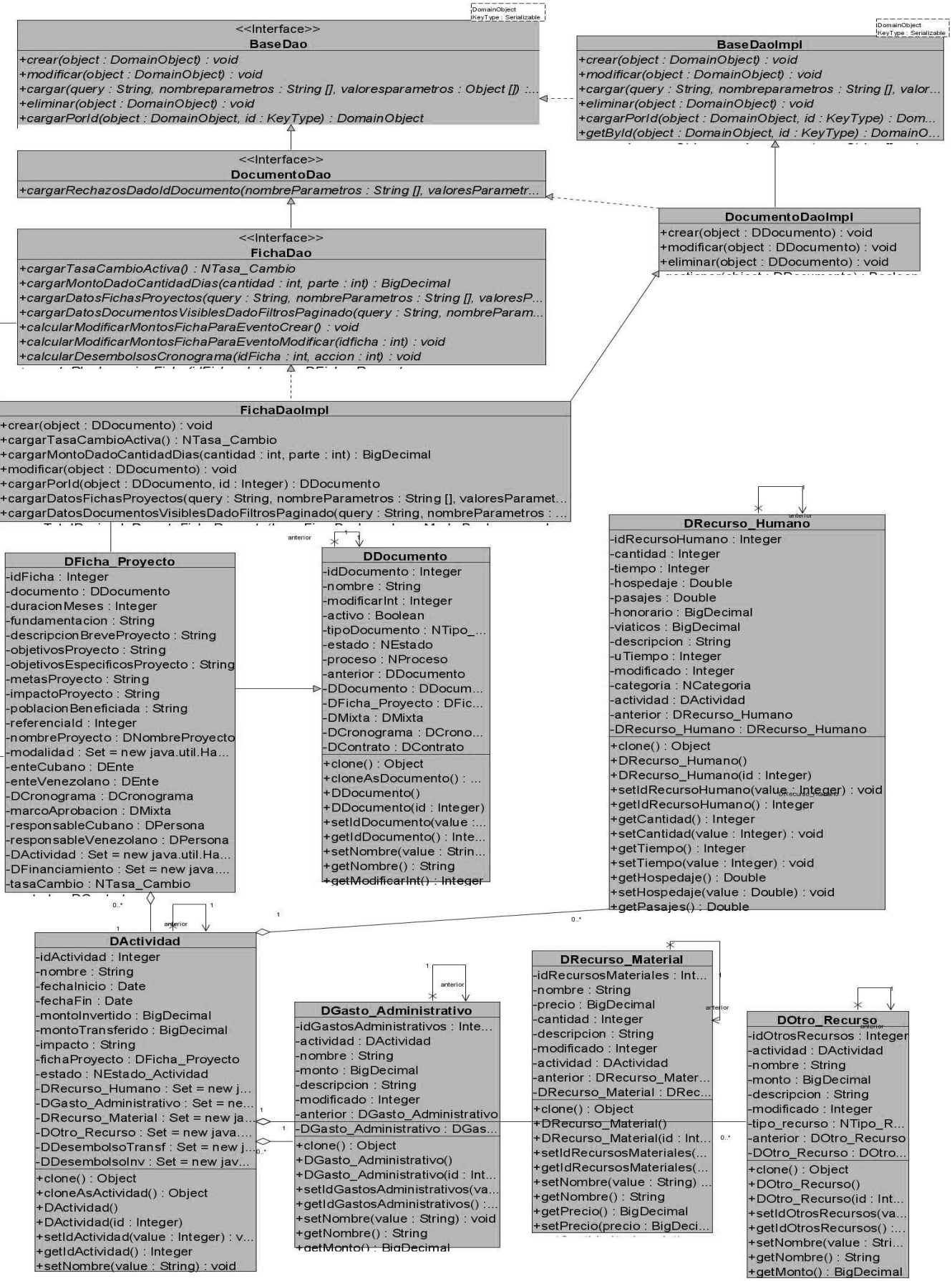
Anexo II. Diagrama de clases del CU Gestionar Contrato



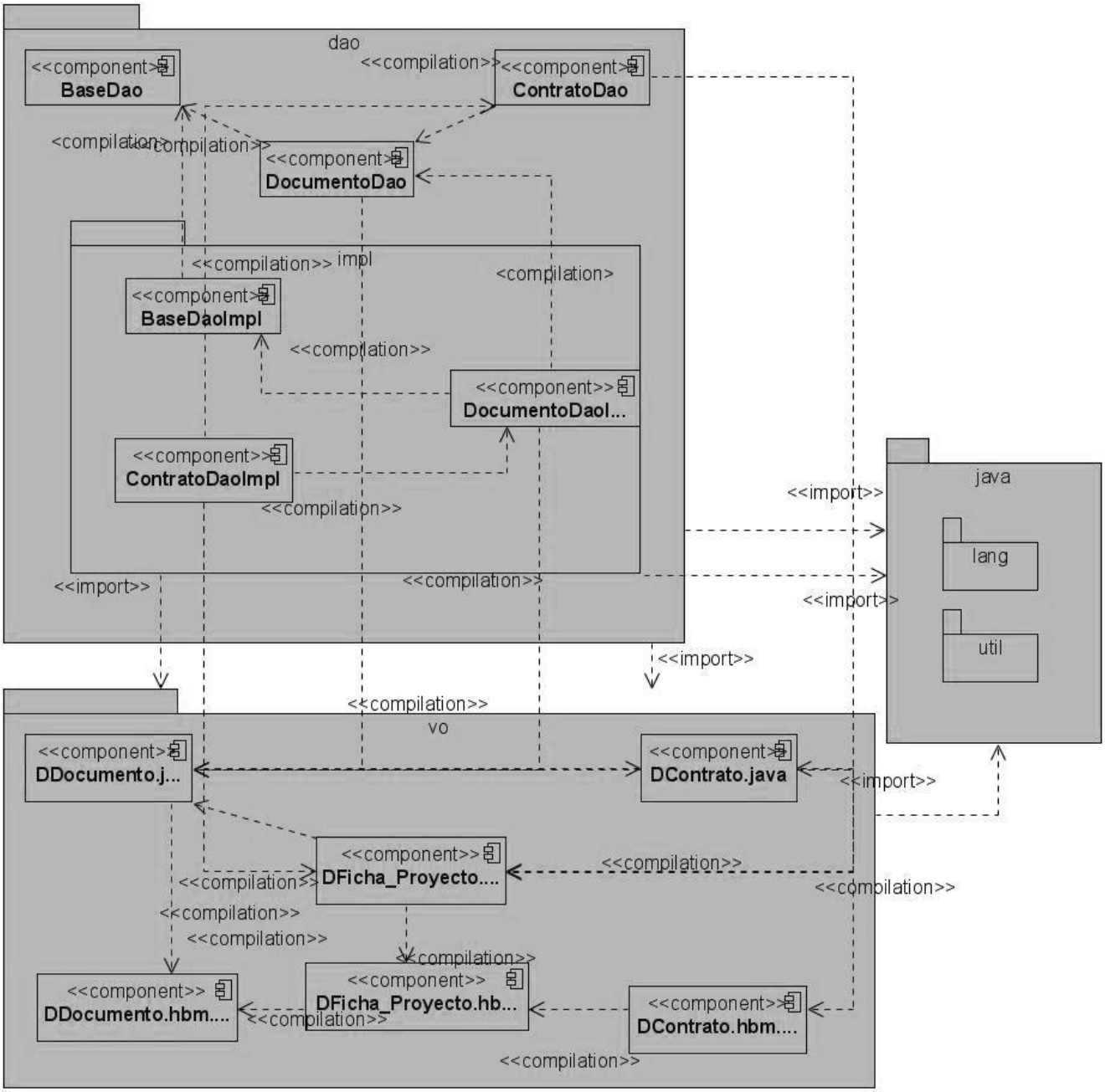
Anexo IIII. Diagrama de clases del CU Gestionar Cronograma



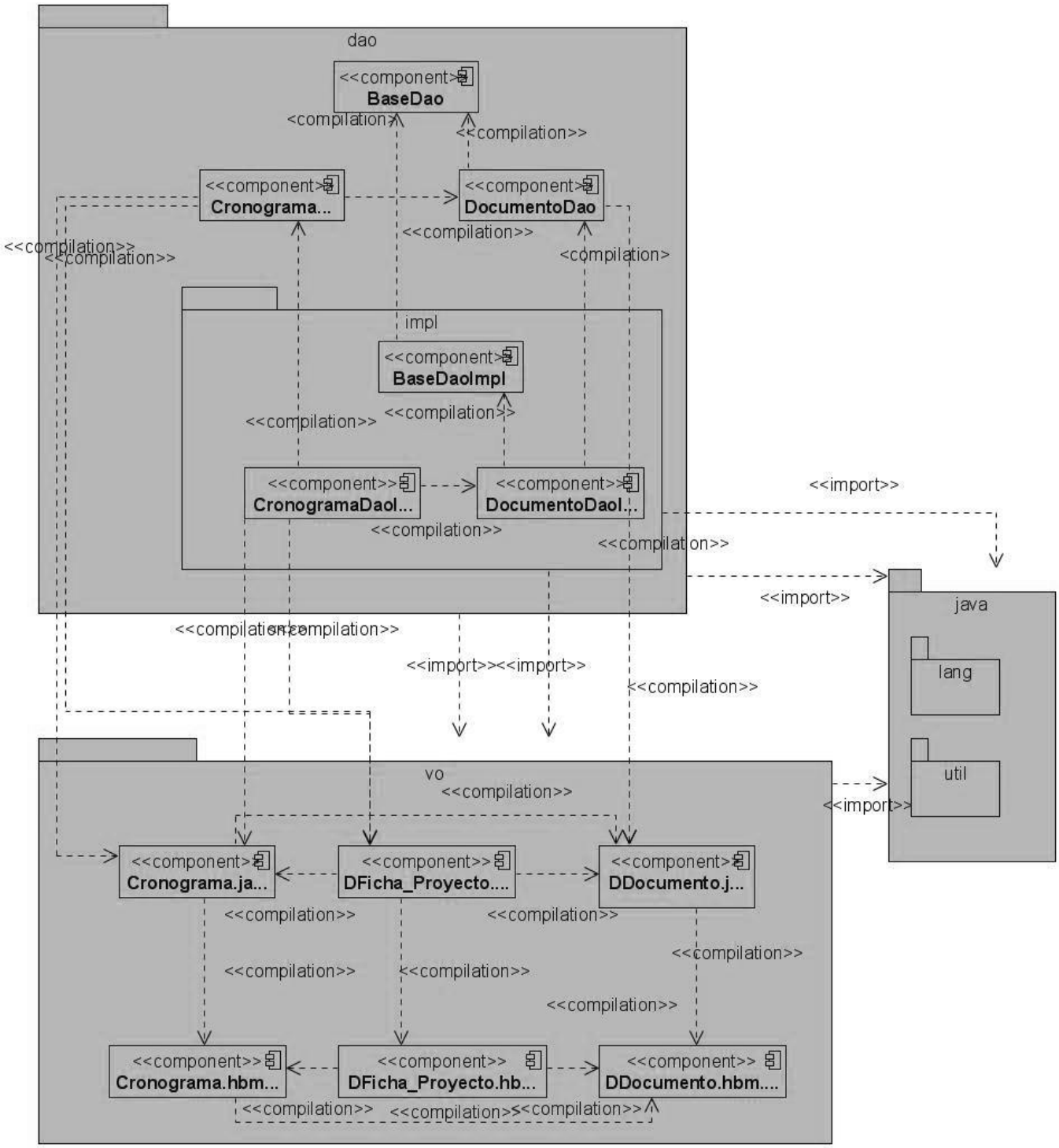
Anexo IVV. Diagrama de clases del CU Gestionar Proyecto



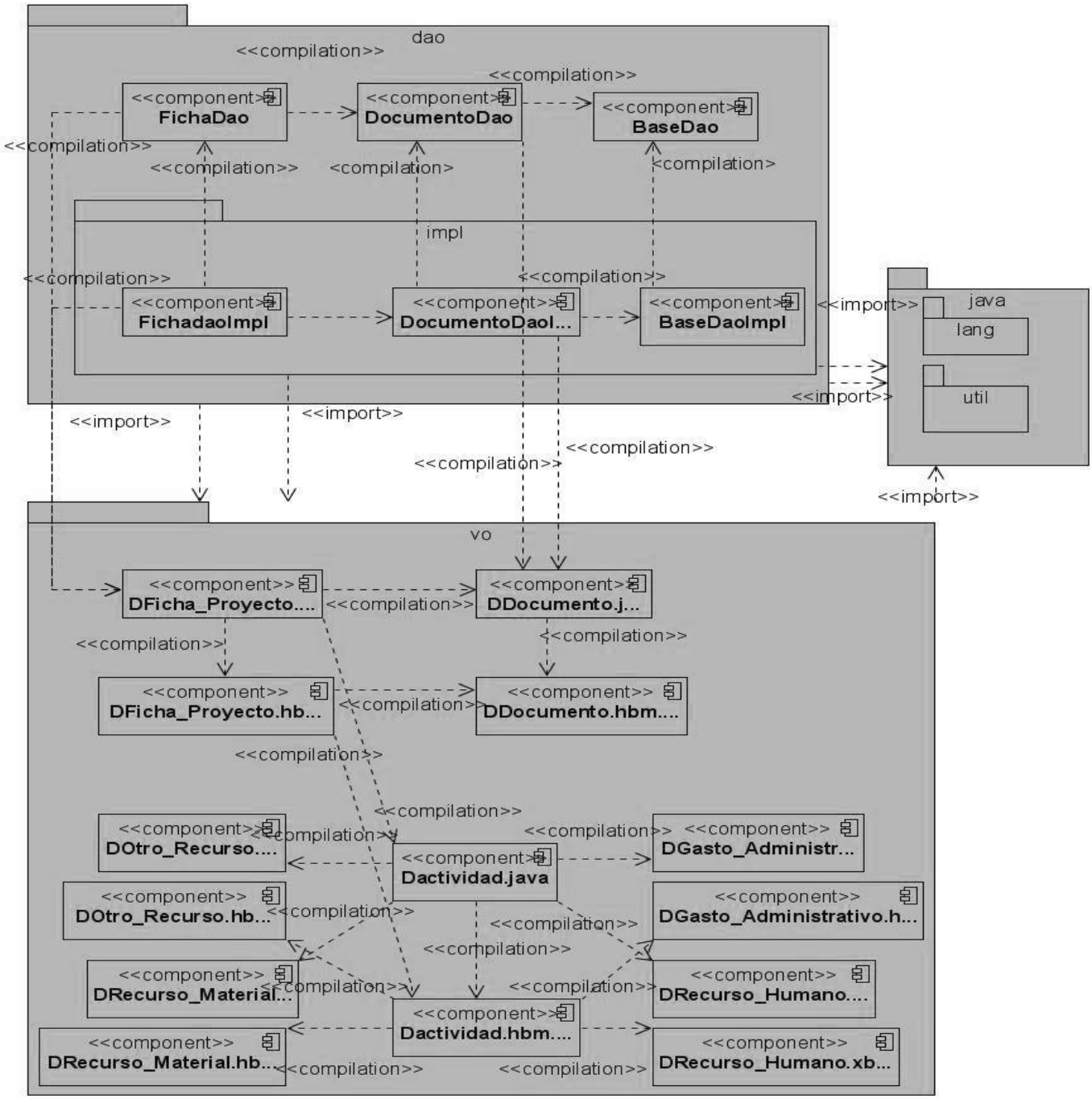
Anexo V. Diagrama de componentes del CU Gestionar Contrato



Anexo VI. Diagrama de componentes del CU Gestionar Contrato



Anexo VI. Diagrama de componentes del CU Gestionar Proyecto



Anexo VII. Configuración del bean sessionFactory

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean"
    lazy-init="true">
    <property name="dataSource">
        <ref bean="dataSource" />
    </property>

    <property name="mappingDirectoryLocations">
        <list>
            <value>classpath:/cu/uci/ccv/common/vo</value>
            <value>classpath:/cu/uci/ccv/presentacion/vo</value>
            <value>classpath:/cu/uci/ccv/contratacion/vo</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">
                ${hibernate.dialect}
            </prop>
            <prop key="hibernate.cglib.use_reflection_optimizer">
                ${hibernate.cglib.use_reflection_optimizer}
            </prop>
            <prop key="show_sql">${hibernate.show_sql}</prop>
        </props>
    </property>
</bean>
```

Anexo VIII. Configuración del bean dataSource

```
<bean id="dataSource"
     class="org.apache.commons.dbcp.BasicDataSource"
     destroy-method="close">
     <property name="url">
          <value>${hibernate.connection.url}</value>
     </property>
     <property name="driverClassName">
          <value>${hibernate.connection.driver_class}</value>
     </property>
     <property name="username">
          <value>${hibernate.connection.username}</value>
     </property>
     <property name="password">
          <value>${hibernate.connection.password}</value>
     </property>
     <property name="initialSize">
          <value>${dbcp.initialSize}</value>
     </property>
     <property name="maxActive">
          <value>${dbcp.maxActive}</value>
     </property>
     <property name="maxIdle">
          <value>${dbcp.maxIdle}</value>
     </property>
     <property name="minIdle">
          <value>${dbcp.minIdle}</value>
     </property>
     <property name="maxWait">
          <value>${dbcp.maxWait}</value>
     </property>
</bean>
```

Anexo IX. Configuración del bean propertyConfigurer

```
<bean id="propertyConfigurer"
    class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
        <list>
            <value>
                classpath:cu/uci/ccv/common/resources/database_connection.properties
            </value>
            <value>
                classpath:cu/uci/ccv/common/resources/ccvmessages.properties
            </value>
        </list>
    </property>
</bean>
```

Anexo X. Configuración del archivo database_connection.properties

```
hibernate.connection.url=jdbc:postgresql://localhost/ccv
hibernate.connection.driver_class=org.postgresql.Driver
hibernate.connection.username=postgres
hibernate.connection.password=postgres
hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
hibernate.cglib.use_reflection_optimizer=false
hibernate.show_sql=false

# La maxima cantidad inicial de conexiones que son creadas cuando el Pool es creado.
dbcp.initialSize=2
#La maxima cantidad de conexiones activas que el pool va a contener al mismo tiempo.
dbcp.maxActive=2
#La maxima cantidad de conexiones activas que permaneceran desocupadas en el pool.
dbcp.maxIdle=1
#La minima cantidad de conexiones activas que permaneceran desocupadas en el pool.
dbcp.minIdle=1
#El tiempo en milisegundos que el pool espera para lanzar una excepcion cuando no hay conexiones disponibles
dbcp.maxWait=-1
```

Anexo XI. Clase persistente DContrato.java

```
@SuppressWarnings("serial")
public class DContrato extends DDocumento implements Serializable, Cloneable {
    public DContrato() {

    private Integer idContrato;

    private cu.uci.ccv.common.vo.DDocumento documento;

    private byte[] documentoAdjunto;

    private java.util.Date fechaFirma;

    private String nombredoc;

    private java.util.Set fichaProyecto = new java.util.HashSet();

    public void setDocumentoAdjunto(byte[] value) {

    public byte[] getDocumentoAdjunto() {

    public void setFechaFirma(java.util.Date value) {

    public java.util.Date getFechaFirma() {

    public String getNombredoc()

    public void setNombredoc(String nombredoc)

    public Integer getORMID() {

    public java.util.Set getFichaProyecto() {

    public void setFichaProyecto(java.util.Set value) {

    public String toString() {

    public cu.uci.ccv.common.vo.DDocumento getDocumento() {

    public void setDocumento(cu.uci.ccv.common.vo.DDocumento value) {

    public Integer getIdContrato() {

    public void setIdContrato(Integer idContrato) {

}
```

Anexo XII. Clase persistente DFicha_Proyecto.java

```
@SuppressWarnings("serial")
public class DFicha_Proyecto extends DDocumento implements Serializable {
    private Integer idFicha;
    private cu.uci.ccv.common.vo.DDocumento documento;
    private Integer referenciaId;
    private java.util.Set modalidad = new java.util.HashSet();
    private DCronograma DCronograma;
    private DMixta marcoAprobacion;
    private java.util.Set DActividad = new java.util.HashSet();
    private java.util.Set DFinanciamiento = new java.util.HashSet();
    private NTasa_Cambio tasaCambio;
    private cu.uci.ccv.contratacion.vo.DContrato contrato;
    public DFicha_Proyecto() {
    public DFicha_Proyecto(Integer id) {
    private Integer getIdFicha() {
    public void setIdFicha(Integer idFicha) {
    public void setDuracionMeses(int value) {
    public java.util.Set getModalidad() {
    public void setModalidad(java.util.Set value) {
    public void setDocumento(cu.uci.ccv.common.vo.DDocumento value) {
    public cu.uci.ccv.common.vo.DDocumento getDocumento() {
    public void setDActividad(java.util.Set value) {
    public java.util.Set getDActividad() {
    public DCronograma getDCronograma() {
    public void setDCronograma(DCronograma cronograma) {
    public DMixta getMarcoAprobacion() {
    public void setMarcoAprobacion(DMixta value) {
    public DContrato getContrato() {
    public void setContrato(DContrato value) {
    public NTasa_Cambio getTasaCambio() {
    public void setTasaCambio(NTasa_Cambio value) {
    public Integer getReferenciaId() {
    public void setReferenciaId(Integer referenciaId) {
}
```

Anexo XIII. Implementación de los métodos equals() y hashCode() para la clase persistente Dfinanciamiento.java

```
public class DFinanciamiento implements Serializable {

    public boolean equals(Object aObj) {
        if (aObj == this)
            return true;
        if (!(aObj instanceof DFinanciamiento))
            return false;
        DFinanciamiento dfinanciamiento = (DFinanciamiento) aObj;
        if (getFuenteFinanciamiento() == null
                && dfinanciamiento.getFuenteFinanciamiento() != null)
            return false;
        if (!getFuenteFinanciamiento().equals(
                dfinanciamiento.getFuenteFinanciamiento()))
            return false;
        if (getFichaProyecto() == null
                && dfinanciamiento.getFichaProyecto() != null)
            return false;
        if (!getFichaProyecto().equals(dfinanciamiento.getFichaProyecto()))
            return false;
        return true;
    }

    public int hashCode() {
        int hashcode = 0;
        if (getFichaProyecto() != null) {
            hashcode = hashcode + (int) getFichaProyecto().getIdDocumento();
        }
        if (getFuenteFinanciamiento() != null) {
            hashcode = hashcode + (int) getFuenteFinanciamiento()
                            .getIdFuenteFinanciamiento();
        }
        return hashcode;
    }
}
```

Anexo XIV. Clase CronogramaDaoImpl.java

```
public class CronogramaDaoImpl extends BaseDaoImpl<DCronograma, Integer>
        implements CronogramaDao {
    public List<Object[]> cargarCronogramas(String nombreFicha, int idEnte,

    public int cantidadCronogramas(String nombreFicha, int idEnte,

    public List<Object[]> cargarCronogramasPorProceso(String nombreFicha,

    public int cantidadCronogramasPorProceso(String nombreFicha, int idEnte,

    public List<DActividad> cargarActividades(int idFicha) throws DAOException {

    public DFicha_Proyecto cargarFichaParaCronograma(int idFicha)

    public void eliminarDesembolsosActividad(int idActividad)

    public void eliminarRechazoCronograma(int idCronograma) throws DAOException {
}
```

Anexo XV. Clase FichaDaoImpl.java

```
public class FichaDaoImpl extends DocumentoDaoImpl implements FichaDao {

    public void crear(DDocumento object) throws DAOException {

    public NTasa_Cambio cargarTasaCambioActiva() throws DAOException {

    public BigDecimal cargarMontoDadoCantidadDias(int cantidad, int parte)

    public void modificar(DDocumento object) throws DAOException {

    public DDocumento cargarPorId(DDocumento object, Integer id)

    public List<Object[]> cargarDatosFichasProyectos(String query,

    public List<Object[]> cargarDatosDocumentosVisiblesDadoFiltrosPaginado(

    private Long cargarTotalPaginadoReporteFichaProyecto(Boolean borroFin,

    public DFicha_Proyecto reportePlanInversionFicha(Integer idFicha)

    public List<Object[]> reporteMontoTotalProyectosPresentados(

    public List<Object[]> cargarDatosDocumentosVisiblesDadoFiltrosPaginadoSeguimiento(

    private Long cargarTotalPaginadoReporteFichaProyectoSeguimiento(

}
```

Glosario de términos y siglas

Java: Java es toda una tecnología orientada al desarrollo de software con el cual podemos realizar cualquier tipo de programa. La tecnología Java está compuesta básicamente por 2 elementos: el lenguaje Java y su plataforma. Con plataforma nos referimos a la máquina virtual de Java (Java Virtual Machine).

Apache Tomcat: Tomcat es un servidor web con soporte de servlets y JSPs. Incluye el compilador Jasper, que compila JSPs convirtiéndolas en servlets. El motor de servlets de Tomcat a menudo se presenta en combinación con el servidor web Apache. Tomcat puede funcionar como servidor web por sí mismo. En sus inicios existió la percepción de que el uso de Tomcat de forma autónoma era sólo recomendable para entornos de desarrollo y entornos con requisitos mínimos de velocidad y gestión de transacciones. Hoy en día ya no existe esa percepción y Tomcat es usado como servidor web autónomo en entornos con alto nivel de tráfico y alta disponibilidad. Dado que Tomcat fue escrito en Java, funciona en cualquier sistema operativo que disponga de la máquina virtual Java.

Herramienta: software que permite automatizar el proceso de desarrollo de software.

Visual Paradigm: es una herramienta UML profesional que soporta el ciclo de vida completo del desarrollo de software: análisis y diseño orientados a objetos, construcción, pruebas y despliegue. El software de modelado UML ayuda a una más rápida construcción de aplicaciones de calidad, mejores y a un menor coste. Permite dibujar todos los tipos de diagramas de clases, código inverso, generar código desde diagramas y generar documentación. La herramienta UML CASE también proporciona abundantes tutoriales de UML, demostraciones interactivas de UML y proyectos UML.

Software colaborativo: Un sistema colaborativo es un conjunto de hardware y de herramientas de software que soportan el trabajo en colaboración de equipos de personas, que se desarrolla sobre una red de telecomunicaciones. Se trata de herramientas informáticas que son especialmente diseñadas para ayudar a los usuarios a trabajar en colaboración de forma más eficaz. Existen diferentes tipos de software colaborativos como son: software para trabajo en grupo, software para gestión de documentos y software para gestión de proyectos

Subsistema de implementación: Una colección de componentes y otros subsistemas de implementación usados para estructurar el modelo de implementación y dividirlos en pequeñas partes que pueden ser integradas y probadas de forma separada

Base de datos relacionales: Permiten almacenar gran cantidad de datos y son ampliamente utilizadas por su flexibilidad y robustez en la gestión de los datos. Al usar tecnología relacional brinda la posibilidad de compartir los datos a diferentes aplicaciones y aun siendo implementadas por tecnologías diferentes. Es usado ampliamente en las aplicaciones que necesiten una gran gestión de los datos.

Persistencia: Es uno de los conceptos más importantes in el desarrollo de aplicaciones. Se refiere a la capacidad de estas de preservar la información de sus datos incluso cuando hayan sido cerradas.

Framework: Es un conjunto de clases que contienen un diseño abstracto para soluciones a familias de problemas y brindan rehúso con una granularidad mayor que las clases. (Johnson, y otros, June/July 1988) Un framework ofrece una estructura común y reusable que puede ser compartida entre aplicaciones. (Massol, y otros, 2004)

Java 2 Enterprise Edition: Es una plataforma diseñada para la computación mainframe, escala típica de las grandes empresas. Diseñada para simplificar el desarrollo de aplicaciones en un cliente ligero en niveles medio ambiente.

SGBD: El sistema de gestión de la base de datos es una aplicación que permite a los usuarios definir, crear y mantener la base de datos, y proporciona acceso controlado a la misma.

JDBC: Java Database Connectivity es una interfaz de programación de aplicaciones (API) para la conexión de especificación en los programas escritos en Java a los datos de la popular base de datos de s.