
VNIVERSIDAD NACIONAL AVTÓNOMA DE MEXICO

**UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO**

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

# Proceso Unificado con el Paradigma de Programación Orientada a Aspectos: Requerimientos, Diseño e Implementación

**T E S I S**

**QUE PARA OBTENER EL GRADO DE:**

**MAESTRO EN INGENIERÍA (COMPUTACIÓN)**

**P R E S E N T A:**

**ELY SCHOENFELD LIBERMAN**

**DIRECTORA DE TESIS:**
**M. en C. Ma. Guadalupe Elena Ibargüengoitia González**

**México, D.F.** **2008.**

*A Debie,*
*por ser, por estar, por existir.*

*A mi madre,*
*Artista, Madre, Corazón.*

Agradecimientos:

Quiero agradecer la valiosísima ayuda recibida de mi directora de tesis Lupita Ibargüengoitia, a quien debo el haber desarrollado este trabajo. Sin su constante apoyo y guía, no me habría sido posible realizar ésta investigación, y de quien recibí la inyección de perseverancia necesaria para terminar este programa de posgrado.

A mi padre, de quien he recibido tanto apoyo, y de quien he podido aprender tantas cosas. A Esther, quien siempre se ha preocupado por mí y me ha impulsado a mejorar en todos los aspectos de la vida.

A mis hermanos, de quienes he recibido tanto amor y apoyo. Especialmente a Ari, quien me indujo hacia diversas áreas de conocimiento que han resultado ser sumamente útiles y me dirigió en mis primeros pasos en el desarrollo de software. También por haber hecho posibles tan perfectas palabras de dedicatoria hacia nuestra madre.

A mis compañeros de trabajo, quienes lograron resistir todo este tiempo.

Asimismo, quiero agradecer la valiosa colaboración de la Dra. Raquel Anaya de la Universidad EAFIT de Medellín, Colombia, por sus valiosos comentarios durante la especificación de las modificaciones necesarias al Proceso Unificado de Desarrollo de Software para incorporar la Programación Orientada a Aspectos. A Claudia Morales Almonte, quien me fue de gran ayuda en el desarrollo de la aplicación para éste trabajo. A mis sinodales, de quienes recibí un gran apoyo y guía.

# Índice de contenido

# Índice de Figuras

# Introducción

Con los años de experiencia en desarrollos Orientados a Objetos, se han detectado algunas carencias de la Programación Orientada a Objetos (POO). La programación Orientada a Aspectos (POA) es una solución a algunas de éstas, y el desarrollo de software con la ayuda de marcos de trabajo MDA (*Arquitectura Dirigida por Modelos*, por sus siglas en inglés) es una solución a otras.

De estas dos nuevas tecnologías, MDA pretende resolver el problema de la rápida evolución de los sistemas de cómputo, es decir está mayormente dirigida hacia la parte de implementación tecnológica: hacer fácil implementar una aplicación para cambiantes plataformas, sistemas operativos, lenguajes, etc. La otra, es decir POA, pretende resolver o facilitar el mantenimiento de aplicaciones y hacer más fácil la reutilización de módulos ya desarrollados.

La programación orientada a aspectos (POA) se basa en la idea de que es mejor desarrollar los sistemas de cómputo por medio de la especificación de varias *necesidades* (preocupaciones, propiedades o áreas de interés) en forma separada, junto con alguna descripción de sus relaciones. De tal forma que posteriormente se utilicen mecanismos en el ambiente de POA para entrelazarlos, tejerlos o componerlos en un programa coherente [Elrad2001].

Por otro lado, MDA es un marco arquitectónico de desarrollo, y está respaldado por una especificación detallada provista por la OMG (http://www.omg.org/, *Grupo de Administración de Objetos*, por sus siglas en inglés).

Por otro lado, para el desarrollo de software se debe seguir un proceso. Por *proceso* entendemos una guía que define quién está haciendo qué, cuándo y cómo para alcanzar una cierta meta. La principal meta en la Ingeniería de Software es construir productos de software satisfactorios o mejorar los ya existentes. Un proceso efectivo es la guía para el desarrollo eficiente de software con calidad. Captura y presenta las mejores prácticas que el actual *estado de arte* permite. En consecuencia, reduce riesgos e incrementa la predictibilidad en el desarrollo de software [Jacobson2000].

En el desarrollo de software Orientado a Objetos es ampliamente utilizado el Proceso Unificado de Desarrollo de Software. Éste proceso es el producto final de tres décadas de desarrollo y uso práctico. Su desarrollo ha recibido influencias de muchas fuentes, como el “Proceso Objectory” desde 1987 y el “Proceso Unificado de Rational” de 1997 [Jacobson2000].

El trabajo de recopilación y publicación del Proceso Unificado es debida, principalmente, a la colaboración de Ivar Jacobson, Grady Boooch y James Rumbaugh [Jacobson2000].

El objetivo de este trabajo es proponer un proceso de desarrollo de software Orientado a Aspectos con el cual desarrollar aplicaciones de software. Un objetivo adicional es integrar la POA con el uso de un marco MDA. Así que al momento de implementar la aplicación de ésta tesis, por un lado, se utilizará un marco de desarrollo MDA para generar la arquitectura básica y, por otro lado, se realizará el desarrollo completo de su funcionalidad utilizando la tecnología de Aspectos.

El objetivo de integrar la POA en el Proceso Unificado de Desarrollo de Software (PUDS) está principalmente basado en el trabajo realizado por Jacobson sobre Desarrollo de Software Orientado a Aspectos, y adecuado a la definición original de dicho proceso.

El objetivo de trabajar con MDA se lleva a cabo incorporando AndroMDA, que es un marco de desarrollo de código abierto y libre, sumamente poderoso.

Cabe resaltar que al momento de solicitar al comité académico la aprobación del tema de esta tesis, no

se había identificado la posibilidad de integrar ambas tecnologías para ofrecer una mejor solución a la necesidad de desarrollo de aplicaciones de una manera más rápida y mejor. Es debido a ésto que el título de este trabajo solamente menciona la Programación Orientada a Aspectos.

Este trabajo está compuesto por:

Capítulo 1. *Fundamentos*

Se hace una recapitulación de los conceptos necesarios para comprender el presente trabajo. Se hace un breve resumen de lo planteado en el Proceso Unificado de Desarrollo de Software (PUDS), de las mejoras obtenidas gracias a la utilización de la Programación Orientada a Objetos (POO), el planteamiento de la Programación Orientada a Aspectos, y la Arquitectura Dirigida por Modelos.

Capítulo 2. *Proceso Unificado Orientado a Aspectos*

En forma más detallada serán presentadas las adecuaciones que fueron consideradas para integrar la POA en el PUDS.

El flujo de trabajo de pruebas (que es parte fundamental del PUDS), aún cuando se pueden obtener grandes beneficios con la utilización de los conceptos de la Orientación a Aspectos, va más allá del alcance de este trabajo.

Capítulo 3. *Programación Orientada a Aspectos (POA) y Arquitectura Dirigida por Modelos (MDA)*

Es en este capítulo que se plantea un primer acercamiento a la creación de un marco de desarrollo MDA que sea capaz de producir aplicaciones Orientadas a Aspectos desde el modelo, pasando por el código fuente generado, hasta llegar a su funcionamiento. En este capítulo se presenta un marco MDA específico: AndroMDA. Es un desarrollo de código abierto sumamente poderoso y versátil. Así mismo, se plantea un primer análisis de las necesidades a cubrir para hacer que este marco MDA sea capaz de generar aplicaciones Orientadas a Aspectos.

Capítulo 4. *Construcción de una aplicación.*

Es en este capítulo se aplican los conocimientos y las propuestas realizadas. Se presenta paso a paso la construcción de una aplicación. No serán mostrados todos los detalles de esta aplicación, puesto que no es el objeto de este trabajo, sino las partes relevantes al tema que nos atañe: Programación Orientada a Aspectos en el Proceso Unificado de Desarrollo de Software, utilizando un marco MDA.

Como todo trabajo de tesis, es necesario llegar a cierta(s) conclusión(es) y esta tesis no es diferente. Consecuentemente, presenta un análisis del aprendizaje obtenido, ventajas y desventajas, así como dificultades encontradas durante el proceso de creación de este trabajo en las conclusiones.

# Capítulo 1. Fundamentos

## 1.1. El Proceso Unificado

### 1.1.1. Introducción al Proceso Unificado de Desarrollo de Software

El software es cada vez más grande, sofisticado y complejo. Este apetito de software aún más sofisticado crece a medida que se ve cómo pueden mejorarse los productos de una versión a otra.

Los usuarios o interesados esperan que esté mejor adaptado a sus necesidades, solicitan un nivel muy alto de complejidad, y al mismo tiempo, esperan que esté listo a tiempo.

Para hablar sobre qué es el Proceso Unificado hay que empezar por definir qué es proceso. Un proceso define *quién* hace *qué*, *cuándo* y *cómo* para alcanzar un determinado objetivo. En el caso de la Ingeniería de Software, el objetivo es construir un producto de software o mejorar uno existente.

Un proceso de desarrollo de software es el conjunto de actividades necesarias para transformar los requisitos de un usuario o interesado en un sistema de software.

El Proceso Unificado es más que un simple proceso; es un marco de trabajo genérico que puede especializarse para una gran variedad de sistemas de software, para diferentes áreas de aplicación, diferentes tipos de organizaciones, diferentes niveles de aptitud y diferentes tamaños de proyecto.

Es llamado "Unificado" porque representa la unión a lo largo de su historia de metodologías antes separadas como "Objectory" de Ivar Jacobson, "El método Booch" de Grady Booch y "Object Modelling Technique" (OMT) de James Rumbaugh. No sólo reúne el trabajo de estos tres autores, sino que incorpora numerosas aportaciones de otras personas y empresas que han contribuido.

El Proceso Unificado utiliza al *Lenguaje Unificado de Modelado* (Unified Modeling Language, UML) para preparar todos los modelos del sistema de software [Jacobson2000].

Sin embargo lo que caracteriza al Proceso Unificado es que está dirigido por casos de uso (CUs), está centrado en la arquitectura y es iterativo e incremental.

#### *Dirigido por casos de uso*

Para construir un sistema con éxito se debe conocer lo que sus futuros usuarios necesitan y desean.

Con usuario, no sólo se hace referencia a personas, sino también a otros sistemas que interactuarán con el sistema en desarrollo.

Un caso de uso es un fragmento de funcionalidad del sistema que proporciona a un usuario un resultado importante. Los casos de uso representan los requerimientos funcionales. Todos los casos de uso juntos constituyen el **modelo de casos de uso**, el cual describe la funcionalidad total del sistema, para todos sus usuarios.

Los casos de uso deben contestar a la pregunta "¿Qué hace el sistema para cada usuario?". Hay que notar que esta visión fuerza al desarrollador a ver al sistema en función de lo que es importante para el usuario, y no solamente en términos de funciones que sería bueno tener.

La importancia de los casos de uso no solamente está en que ayudan a especificar los requerimientos de un sistema sino que se utilizan para guiar su diseño, implementación y prueba; esto es, *guían el proceso de desarrollo.*

Sin embargo, no se desarrollan aisladamente. Se desarrollan a la vez que la arquitectura del sistema. Es decir, los casos de uso guían la arquitectura del sistema y la arquitectura del sistema influye en la selección de los casos de uso. Entonces, tanto la arquitectura del sistema como los casos de uso maduran según avanza el ciclo de desarrollo.

### *Centrado en la arquitectura*

El papel de la arquitectura de software es parecido al papel que juega la arquitectura en la construcción de edificios. El edificio se contempla desde varios puntos de vista: estructura, servicios, fontanería, electricidad, etc. Esto permite a un constructor ver una imagen completa antes de que comience la construcción. Análogamente, la arquitectura de un sistema de software se describe mediante diferentes vistas del sistema en construcción.

La arquitectura es una vista del diseño completo con las características más importantes resaltadas, dejando de lado los detalles.

El concepto de arquitectura de software incluye los aspectos estáticos y dinámicos más significativos del sistema. Se ve influida por muchos otros factores, como: la plataforma en la que tiene que funcionar el software (arquitectura de hardware, sistema operativo, sistema de gestión de base de datos, protocolos para comunicaciones en red), los bloques de construcción reutilizables de que se dispone, consideraciones de implementación, sistemas heredados, y requisitos no funcionales (por ejemplo, rendimiento, confiabilidad).

Así como cada producto tiene una función y una forma, los casos de uso especifican la función, y la arquitectura especifica la forma. Ambos deben evolucionar en paralelo.

El arquitecto debe poseer una comprensión general de los casos de uso antes de comenzar la creación del esquema arquitectónico. Posteriormente, el arquitecto trabaja con aquellos casos de uso que representen las funciones clave del sistema en desarrollo. Cada caso de uso seleccionado es especificado en detalle y se realiza en términos de **subsistemas**, **clases**, y **componentes**. A medida que los casos de uso se especifican y maduran, se descubre más de la arquitectura. Esto, a su vez, lleva a la maduración de más casos de uso. Este proceso continúa hasta que se considera que la arquitectura es estable.

### *Iterativo e Incremental*

El desarrollo de un producto comercial puede durar varios meses o varios años, por lo que puede ser práctico dividir el trabajo en partes más pequeñas o mini proyectos. Cada mini proyecto es una iteración que resulta en un incremento. Las iteraciones hacen referencia a pasos en el flujo de trabajo, y los incrementos, al crecimiento del producto. Las iteraciones deben seleccionarse y ejecutarse de una forma planificada.

Se selecciona lo que se implementará en una iteración en base a dos factores. En primer lugar, la iteración trata un grupo de casos de uso que juntos amplían la utilidad del producto desarrollado hasta ahora. En segundo lugar, la iteración trata los riesgos más importantes. Las iteraciones sucesivas se

construyen sobre los artefactos de desarrollo tal como quedaron al final de la última iteración.

En cada iteración se identifican y especifican los casos de uso relevantes, se crea un diseño utilizando la arquitectura seleccionada como guía, se implementa el diseño mediante componentes, y se verifica que los componentes satisfagan los casos de uso. Si una iteración cumple con sus objetivos, el desarrollo continúa con la siguiente. Si una iteración no cumple con sus objetivos, los desarrolladores deben revisar sus decisiones previas y probar con un nuevo enfoque.

Son muchos los beneficios de un proceso iterativo controlado:

- La iteración controlada reduce el costo del riesgo a los costos de un solo incremento. Si los desarrlladores tienen que repetir la iteración, la organización sólo pierde el esfuerzo mal empleado de la iteración, no el valor del producto entero.
- La iteración controlada reduce el riesgo de no sacar al mercado el producto en el calendario previsto. Mediante la identificación de riesgos en fases tempranas del desarrollo, el tiempo que se gasta en resolverlos se emplea al principio de la planificación, cuando la gente está menos presionada por cumplir los plazos.
- La iteración controlada acelera el ritmo del esfuerzo de desarrollo en su totalidad debido a que los desarrolladores trabajan de manera más eficiente para obtener resultados claros a corto plazo, en lugar de tener un calendario largo, que se prolonga eternamente.
- La iteración controlada reconoce una realidad que a menudo se ignora: que las necesidades del ususario y sus correspondientes requisitos no pueden definirse completamente al principio. Típicamente, se refinan en iteraciones sucesivas. Esta forma de operar hace más fácil la adaptación a los requisitos cambiantes.

### *Ciclo de vida del Proceso Unificado*

El Proceso Unificado se repite a lo largo de una serie de ciclos que constituyen la vida de un sistema. Cada ciclo concluye con una **versión** del producto para los clientes.

Cada uno de estos ciclos consta de cuatro fases: inicio, elaboración, construcción y transición. Cada fase se subdivide a su vez en iteraciones, como se ha dicho anteriormente.

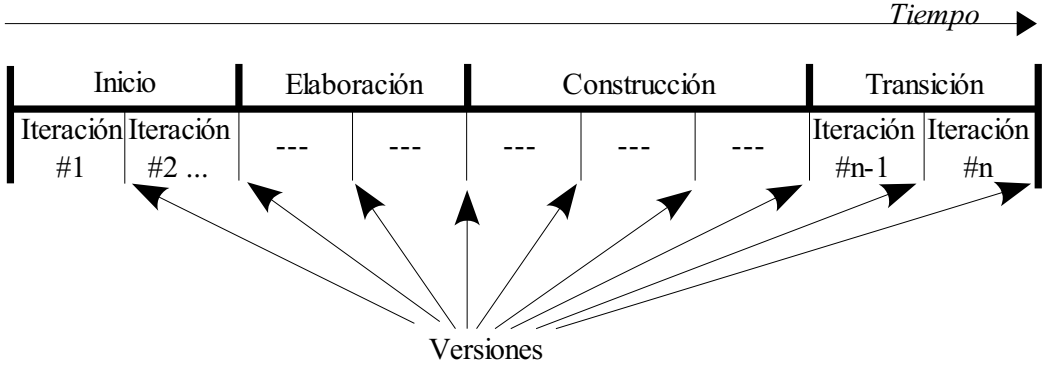



*Figura 1.1.1: Un ciclo con sus fases e iteraciones.*

Para llevar a cabo el siguiente ciclo de desarrollo de manera eficiente, los desarrolladores necesitan todas las representaciones del producto de software:

- Un modelo general de casos de uso, con todos los casos de uso y su relación con los usuarios.
- Un modelo de análisis, con dos propósitos: refinar los casos de uso con más detalle y establecer la asignación inicial de funcionalidad del sistema a un conjunto de objetos que proporcionan el comportamiento.
- Un modelo de diseño que define: a) la estructura estática del sistema en la forma de subsistemas, clases e interfaces y b) los casos de uso reflejados con colaboraciones entre subsistemas, clases, e interfaces.
- Un modelo de implementación, que incluye componentes (que representan al código fuente) y la correspondencia de las clases con los componentes.
- Un modelo de despliegue o de distribución que define los nodos físicos (computadoras) y la correspondencia de los componentes con estos nodos.
- Un modelo de prueba, que describe los casos de prueba que verifican los casos de uso.
- Una representación de la arquitectura.

El sistema también debe tener un modelo del dominio o modelo del negocio que describa el contexto del negocio en el que se halla el sistema.

### *Fases dentro de un ciclo*

Como se mencionó anteriormente, cada ciclo se divide en cuatro fases. Dentro de cada fase, el trabajo se descompone en iteraciones. Cada fase termina con un conjunto de **indicadores** o **hito** (ver Apéndice A. Glosario.) Cada hito se determina por la disponibilidad de un conjunto de artefactos; es decir, ciertos modelos o documentos han sido desarrollados hasta alcanzar un estado predefinido.

Los hitos tienen muchos objetivos. El más crítico es que los desarrolladores deben tomar ciertas decisiones cruciales antes de que el trabajo pueda continuar con la siguiente fase. Los hitos o indicadores también permiten a los desarrolladores, controlar el progreso del trabajo según pasa por esas cuatro fases. Al final, se obtiene un conjunto de datos a partir del seguimiento del tiempo y esfuerzo consumido en cada fase. Estos datos son útiles en la estimación del tiempo y los recursos humanos para otros proyectos, en la asignación de los recursos durante el tiempo que dura el proyecto, y en el control del progreso contrastado con la planificación.

La Figura 1.1.2 muestra en la columna izquierda los flujos de trabajo: requisitos, análisis, diseño, implementación y prueba. Las curvas son una aproximación de hasta dónde se llevan a cabo los flujos de trabajo en cada fase. Una iteración típica pasa por los cinco flujos de trabajo.

Durante la *fase de inicio*, se desarrolla una descripción del

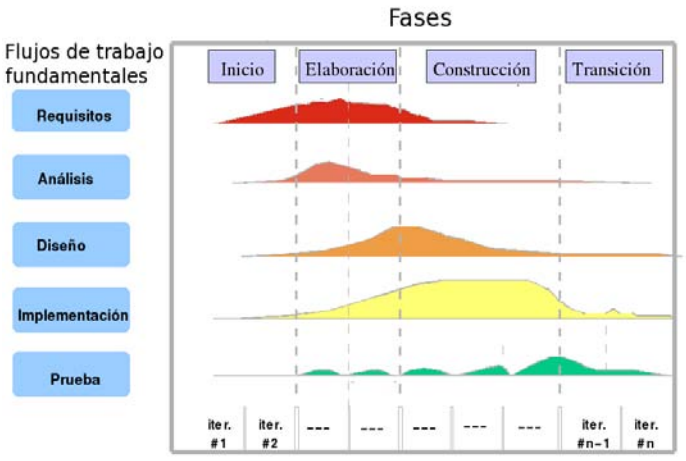



*Figura 1.1.2: Los cinco flujos de trabajo (requisitos, análisis, diseño, implementación y prueba) tienen lugar sobre las cuatro fases: inicio, elaboración, construcción y transición.*

producto final a partir de una idea y se presenta el análisis de negocio para el producto. Esencialmente, esta fase responde a las siguientes preguntas:

- ¿Cuáles son las principales funciones del sistema para sus usuarios más importantes?
- ¿Cómo podría ser la arquitectura del sistema?
- ¿Cuál es el plan de proyecto y cuánto costaría desarrollar el producto?

La respuesta a la primera pregunta se encuentra en un modelo de casos de uso simplificado que contenga los casos de uso más críticos. En esta fase, se identifican y priorizan los riesgos más importantes, se planifica en detalle la fase de elaboración, y se estima el proyecto de manera aproximada.

Durante la *fase de elaboración*, se especifican en detalle la mayoría de los casos de uso del producto y se diseña la arquitectura del sistema.

La arquitectura se expresa en forma de vistas de todos los modelos del sistema, los cuales juntos representan al sistema entero. Esto implica que hay vistas arquitectónicas del modelo de casos de uso, del modelo de análisis, del modelo de diseño, del modelo de implementación y del modelo de despliegue. Durante esta fase del desarrollo, se realizan los casos de uso más críticos que se identificaron en la fase de inicio. El resultado de esta fase es una **línea base** de la arquitectura. (ver Apéndice A. Glosario.)

Durante la *fase de construcción* se crea el producto (se añaden los músculos), es decir el software terminado, sobre la arquitectura (el esqueleto). En esta fase, la línea base de la arquitectura crece hasta convertirse en el sistema completo. La descripción evoluciona hasta convertirse en un producto preparado para ser entregado a la comunidad de usuarios.

La *fase de transición* cubre el período durante el cual el producto se convierte en versión beta. En la versión beta un número reducido de usuarios con experiencia prueba el producto e informa de defectos y deficiencias. Los desarrolladores corrigen los problemas e incorporan algunas de las mejoras sugeridas en una versión general dirigida a la totalidad de la comunidad de usuarios. La fase de transición conlleva actividades como la fabricación, capacitación del cliente, el proporcionar una línea de ayuda y asistencia, y la corrección de los defectos que se encuentren tras la entrega.

### *1.1.2. Personas, Proyecto, Producto y Proceso*

El resultado final de un proyecto de software es un producto que toma forma durante su desarrollo gracias a la intervención de muchos tipos distintos de personas. Un proceso de desarrollo de software guía los esfuerzos de las personas implicadas en el proyecto, a modo de plantilla que explica los pasos necesarios para terminar el proyecto. Típicamente, el proceso está automatizado por medio de una herramienta o de un conjunto de ellas. (*Ver figura 1.1.3)*

Se entiende por:

- *Personas*. Los principales autores de un proyecto de software son los arquitectos, desarrolladores, ingenieros de prueba, y el personal de gestión que les da soporte, además de los usuarios, clientes, y otros interesados. Las personas son realmente seres humanos, a diferencia del término *trabajador* que se utiliza para especificar un puesto que puede ser asignado a una persona o a un equipo.

- *Proyecto*. Elemento organizativo a través del cual se gestiona el desarrollo de software. El resultado de un proyecto es una versión de un producto.
- *Producto*. Artefactos que se crean durante la vida del proyecto, como los modelos, código fuente, ejecutables y documentación. Cada ciclo produce una nueva versión del sistema, y cada versión es un producto preparado para su entrega. El producto de software debe ser algo más que el código de máquina que se ejecuta ya que debe ajustarse a las necesidades de todos los interesados, es decir, toda la gente que trabajará con él, y no solamente ajustarse a las necesidades de los usuarios.
- *Proceso*. Un proceso de ingeniería de software es una definición del conjunto completo de actividades necesarias para transformar los requisitos de usuario en un producto. Un proceso es una plantilla para crear proyectos.
- *Herramientas*. Software que se utiliza para automatizar las actividades definidas en el proceso.

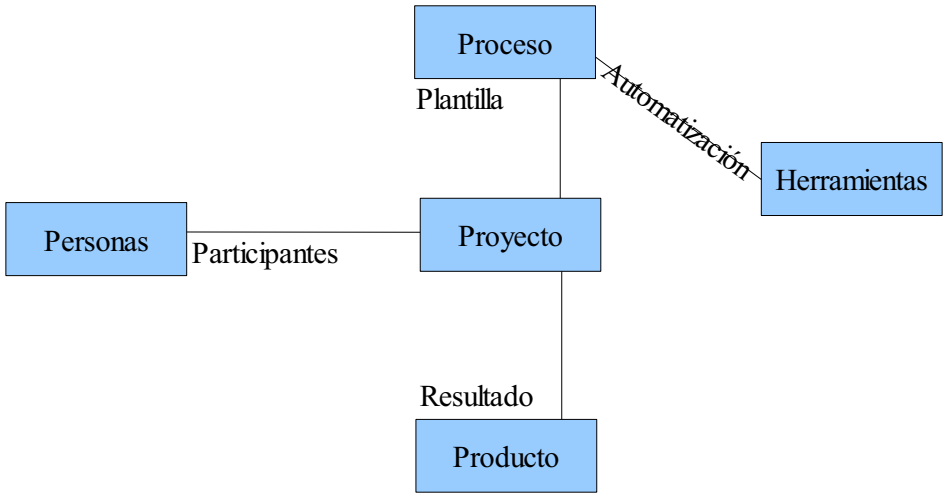



*Figura 1.1.3: Personas, Proyecto, Producto y Proceso en el desarrollo de software*

## 1.2. Aportes de la programación orientada a objetos

### 1.2.1. Fundamentos del paradigma de programación Orientada a Objetos

#### *Introducción*

Los sistemas de software deben ser rápidos, confiables, fáciles de utilizar, entendibles, modulares, estructurados, etc. Pero estos adjetivos describen dos tipos diferentes de cualidades o factores:

a) Factores externos: Aquellas propiedades detectables por usuarios (ya sean humanos o no), como velocidad o facilidad de uso.

b) Factores internos: Aquellos que son perceptibles solamente por profesionales de la computación que tienen acceso al texto mismo de programación del software. Al código fuente.

Desde el punto de vista del usuario final, solamente los factores externos importan. Para que los usuarios disfruten las cualidades visuales de un sistema de software, los diseñadores e implementadores tienen que haber utilizado técnicas que aseguren dichas cualidades. Por otro lado; a futuro, cuando sea necesario dar mantenimiento al sistema (como sucede con la mayoría de los desarrollos), son críticos los factores internos.

Incluso si algunos de estos factores se contraponen entre sí, (por ejemplo, ¿cómo se puede obtener integridad sin introducir diversas protecciones que pueden afectar la facilidad de uso del sistema de software?) un buen sistema debe comprender un buen balance de éstos dependiendo del tipo de sistema de cómputo elaborado y del usuario final del mismo [Clarke2005].

Se pueden identificar cuatro factores externos, como los más importantes:

- Corrección o exactitud (*correctness*) y Robustez.
    - Sigue siendo extremadamente difícil producir software sin defectos, y muy difícil corregirlos una vez presentes en el sistema. Las técnicas utilizadas para mejorar ambos factores incluyen: Un acercamiento más sistemático para la construcción de software; especificaciones más formales; revisiones a lo largo del proceso de construcción de software (no sólo al final por medio de pruebas); mejores mecanismos del lenguaje como tipos estáticos, aserciones, manejo automático de memoria y manejo automatizado de excepciones. Debido a lo cercanos que son la corrección o exactitud y la robustez, un mejor término que incluye ambos factores es **confiabilidad.**
- Extensibilidad y Reusabilidad.
    - El software debería ser fácil de cambiar. Nuevamente se pueden utilizar técnicas similares para mejorar ambas cualidades: Puede ayudar cualquier idea que propicie la producción de arquitecturas mas descentralizadas, en que los componentes sean autocontenidos y solamente se comuniquen por medio de canales restringidos y claramente definidos. El término **modularidad** cubre tanto la reusabilidad como la extensibilidad.

Precisamente la POO mejora significativamente estos cuatro factores de calidad. Es por eso que se hizo tan atractiva su adopción en los años 80s. Sin embargo no solamente ayuda con éstos, sino que también

tiene contribuciones significativas que hacer en otros aspectos.

Un punto muy importante que hay que recalcar es la mantenibilidad del software. Mantenimiento es lo que sucede después de la entrega del software. Normalmente las metodologías de desarrollo de software tienden a enfocarse en la etapa de desarrollo. Sin embargo se estima que un 70% de los costos de software está destinado al mantenimiento, así que no hay que olvidar este factor de calidad [April2008].

Para software, "mantenimiento" significa en realidad dos cosas. Por un lado las modificaciones que son necesarias como consecuencia de cambios en los sistemas de cómputo o necesarias para reflejar cambios en el mundo real. Por otro lado representa los cambios necesarios como consecuencia de la corrección de defectos [Meyer1997].

Más de dos quintas partes del costo de mantenimiento es destinado a modificaciones y extensiones solicitadas por el usuario. Es por esto que el concepto de extensibilidad es tan importante, y algo en lo que la tecnología orientada a objetos ha ayudado extensivamente [Meyer1997].

El siguiente rubro, en orden descendente, en los costos de mantenimiento es el efecto de cambios en formatos de datos. Ejemplos de esto es el cambio de milenio ya que muchos programas utilizaban solamente dos dígitos para especificar años, y ahora serían necesarios cuatro. Otro ejemplo, comparable para aquellos afectados fue cuando el sistema postal de Estados Unidos introdujo los códigos postales de "5+4" dígitos para grandes empresas; muchas compañías tuvieron que invertir grandes cantidades de dinero para corregirlo ya que los programadores no habían previsto posible un cambio así, y utilizaban exactamente 5 dígitos para el código postal.

El problema no es que sea necesario realizar cambios en el formato de datos, es inevitable, lo que no es permisible es que el conocimiento de la longitud de los datos esté especificado por todos lados en el código, y para hacer cualquier cambio sea necesario hacer extensas y complicadas modificaciones.

### *Programación Orientada a Objetos*

El marco de trabajo conceptual para la orientación a objetos es el *modelo de objetos*.

Los cuatro elementos principales de dicho modelo son:

- Abstracción
- Encapsulamiento
- Modularidad
- Jerarquía

Adicionalmente, los tres elementos menores del modelo son:

- Manejo de Tipos
- Concurrencia
- Persistencia

Un modelo al que le falte cualquiera de los elementos principales, no es orientado a objetos. Sin embargo, cada uno de los elementos menores es útil, pero no esencial, en el modelo orientado a objetos.

De no utilizar este marco de trabajo conceptual, se perdería o se abusaría del poder expresivo del lenguaje orientado a objetos en uso. Es más, seguramente no se habría dominado la complejidad del problema en cuestión [Booch1994].

### *Abstracción*

Abstracción es una de las formas fundamentales en que los humanos manejamos la complejidad. Booch define abstracción como:

> *Una abstracción denota las características esenciales de un objeto que se distingue de todos los otros tipos de objetos y, entonces, provee fronteras conceptuales claramente definidas, desde el punto de vista del observador* [Booch1994].

Una abstracción está enfocada a la vista externa de un objeto, por lo que sirve para separar el comportamiento esencial de un objeto de su implementación.

### *Encapsulamiento*

La abstracción de un objeto debe preceder a las decisiones sobre su implementación. Una vez seleccionada una implementación, debe ser tratada como un secreto de la abstracción y oculta para la mayoría de los *clientes*, en donde por *cliente* se entiende cualquier objeto que utiliza recursos de otro objeto (llamado *servidor*). Ninguna parte de un sistema complejo debe depender de los detalles internos de ninguna otra parte. Mientras que la abstracción ayuda a la gente a pensar sobre lo que están haciendo, la encapsulación permite en forma confiable y con un esfuerzo limitado realizar cambios al programa [Booch1994].

Encapsulamiento y abstracción son conceptos complementarios: la abstracción se enfoca en el comportamiento observable de un objeto, mientras que el encapsulamiento se enfoca en la implementación que genera dicho comportamiento. La encapsulación frecuentemente se logra gracias al *ocultamiento de información*, que es el proceso de ocultar todos los secretos de un objeto que no contribuyen con sus características principales; típicamente, la estructura de un objeto es oculta, así como la implementación de sus métodos [Booch1994].

El encapsulamiento provee barreras explícitas entre diferentes abstracciones, por lo que conduce a una separación clara de intereses [Booch1994].

### Modularidad

El acto de partir un programa en componentes individuales puede reducir su complejidad hasta cierto punto. Sin embargo, una justificación más poderosa para dividir un programa es que se crea una serie de fronteras bien definidas y documentadas dentro del programa. Dichas fronteras, o interfaces, son invaluables para la comprensión del programa [Booch1994].

Muchos de los lenguajes que aceptan al módulo como un concepto separado también distinguen entre la interfaz de un modelo y su implementación. Entonces, sería justo decir que la modularidad y el encapsulamiento van de la mano [Booch1994].

Los módulos sirven como contenedores físicos en los que se declaran las clases y objetos del diseño lógico. Una buena solución es agrupar clases y objetos relacionados lógicamente en el mismo módulo,

y exponer solamente aquellos elementos que es indispensable que otros módulos vean. Sin embargo es importante no exagerar, una modularización arbitraria en algunas ocasiones es peor que no modularizar en lo absoluto [Booch1994].

Booch define modularidad como:

> *Modularidad es la propiedad de un sistema que ha sido descompuesto en un conjunto de módulos con alta cohesión (por medio del agrupamiento de abstracciones localmente relacionadas) y muy bajo acoplamiento (minimizando las dependencias entre módulos)* [Booch1994].

### *Jerarquía*

Es posible encontrar más abstracciones diferentes de las que se puede comprender al mismo tiempo. El encapsulamiento ayuda con esta complejidad por medio del ocultamiento de la vista interna de las abstracciones. La modularidad también ayuda, por que permite agrupar abstracciones relacionadas lógicamente. Sin embargo esto no es suficiente. Un conjunto de abstracciones forman una jerarquía frecuentemente, e identificando dichas jerarquías en el diseño, se puede simplificar mucho la comprensión del problema [Booch1994].

Booch define jerarquía como:

> *Jerarquía es una clasificación u ordenamiento de abstracciones* [Booch1994].

Las jerarquías más importantes en un sistema complejo son su estructura de clases (jerarquía "es un") y su estructura de objetos (jerarquía "es parte de")

La jerarquía más importante de tipo "es un" es la herencia simple y es un elemento esencial de los sistemas orientados a objetos. Básicamente, la herencia define una relación entre clases, en que una clase comparte la estructura o comportamiento definido en otra clase. Herencia, entonces, representa una jerarquía de abstracciones, en la que una subclase hereda de una superclase. Típicamente, una subclase aumenta o redefine la estructura y comportamiento existentes en sus superclases.

Semánticamente, herencia denota una relación "es un". Por ejemplo, un oso "es una" clase de mamífero, una casa "es un" tipo de bien tangible, y *quick sort* "es un" tipo particular de algoritmo de ordenamiento. Por tanto, herencia implica una jerarquía de generalización/especialización, en que una subclase especializa la estructura más general o comportamiento de sus superclases.

"Si B *no es* un tipo de A, entonces B no debería heredar de A"

### *Manejo de Tipos (typing)*

El concepto de *tipo* deriva principalmente de las teorías de tipos de datos abstractos. Aún cuando el concepto de tipo y de clase no es exactamente lo mismo, es suficiente con decir que una clase implementa un tipo. Es debido a eso que Booch define el manejo de tipos como:

> *Manejar los tipos es asegurar la clase de un objeto, tal que objetos de diferentes tipos no puedan ser intercambiados o, a lo más, puedan ser intercambiados con muchas restricciones.*

Asegurar tipos permite expresar las abstracciones de tal forma que el lenguaje de programación en que sean implementadas pueda reforzar las decisiones de diseño.

### *Concurrencia*

Mientras que la POO se enfoca en abstracción de datos, encapsulamiento y herencia, la concurrencia se enfoca en abstracción de procesos y sincronización. El concepto de objeto abarca ambos puntos de vista: cada objeto (visto como una abstracción del mundo real) puede representar un hilo de control separado (una abstracción de un proceso). Dichos objetos son llamados *activos*. En un sistema basado en un diseño orientado a objetos, se puede conceptualizar al mundo como un conjunto de objetos que cooperan entre sí, algunos de los cuales son activos por lo que sirven como centros de actividad independiente. Basado en ésto, Booch define concurrencia como:

*Concurrencia es la propiedad que distingue un objeto activo de uno que no lo es.*

### *Persistencia*

Un objeto en software ocupa cierta cantidad de espacio y existe por un tiempo determinado. El espectro de persistencia incluye:

- Datos que existen entre ejecuciones del programa.
- Datos que existen entre varias versiones de un programa.
- Datos que sobreviven al programa.

La persistencia no solamente se encarga de la vida de los datos. En bases de datos orientadas a objetos, no solamente persiste el estado de los objetos, sino que también la clase debe trascender cualquier programa individual, para asegurar que cualquier programa interprete el estado guardado, de la misma forma.

Al hablar de sistemas de procesadores distribuidos, algunas veces hay que preocuparse por la persistencia en el espacio. En estos sistemas, es útil pensar en objetos que se pueden mover de una máquina a otra, y que incluso pueden tener diferentes representaciones en diferentes máquinas.

*Persistencia es la propiedad de un objeto por la cual su existencia trasciende el tiempo (p.e. el objeto continúa existiendo después de que su creador deja de existir) y/o espacio (p.e. la localización del objeto se mueve del espacio de direccionamiento en que fue creado)*

### *Conceptos de POO más importantes*

- Clases
    - El método y el lenguaje orientado a objetos tienen la noción de clase como concepto central. Informalmente una clase es un elemento de software que describe un tipo de datos abstracto y su implementación parcial o total. Un tipo de datos abstracto es un conjunto de objetos definidos por la lista de operaciones aplicables a ese objeto, y las propiedades de dicho objeto.
- Clases como módulos
    - Desde una perspectiva puramente orientada a objetos, las clases deben ser los únicos módulos. En particular, no hay una noción de programa principal y no existen subprogramas como módulos.

- Clases como tipos.
  - Todos los tipos deben estar basados en alguna clase.
- Objeto
  - Un objeto es una instancia de una clase. Los objetos son elementos dinámicos creados en tiempo de ejecución. Un objeto está compuesto por valores llamados campos. Cada campo corresponde a un atributo del generador de objetos (la clase de la cual el objeto es una instancia directa).
- Computación basada en características.
  - Las llamadas a operaciones (o métodos) debe ser el mecanismo principal. Dado un cierto objeto, que siempre es una instancia de alguna clase, se le puede llamar a realizar una operación determinada.
- Ocultamiento de información.
  - Las caracrerísticas deben ser privadas. La única forma de acceder a ellas debe ser por medio de operaciones. Por otro lado, debe ser posible que el autor de una clase pueda especificar que una operación esté disponible para todas, ninguna o algunas llamadas.
- Habilidad de tener parámetros genéricos.
  - Debe ser posible escribir clases con parámetros formales genéricos que representen tipos arbitrarios.
- Herencia sencilla
  - Debe ser posible definir una clase que hereda de otra. Muchas clases son variantes de otras, para lo que se necesita un mecanismo de clasificación que permita manejar la potencial complejidad resultante. Este mecanismo es la herencia. Una clase será heredera de otra si incorpora las características de la otra y adiciona las propias. Un *descendiente* es un heredero directo o indirecto de otra clase, la noción inversa es *ancestro*.
- Polimorfismo
  - Debe ser posible que algunas operaciones tengan un comportamiento diferente dependiendo del objeto (o tipo de dato) sobre el que se aplican. En un ambiente con tipos estáticos, el polimorfismo no debe ser arbitrario, sino controlado por la herencia. Para comprender mejor este concepto, supongamos una clase *FiguraGeométrica* que contiene un método llamado *Area*. De dicha clase heredan una clase *Rectángulo* y una *Círculo*. Es evidente que el cálculo del área de un rectángulo y el de un círculo son diferentes, por lo que la invocación del método *Area* en cada uno de los descendientes se debe comportar diferente.

### *1.2.2. Inconvenientes de la Orientación a Objetos*

No se puede negar la importancia que tiene la POO para el desarrollo de programas complejos. Sin embargo, al menos en dos tipos de situaciones es imposible escribir programas claros y elegantes utilizando solamente POO: Cuando la aplicación tiene funcionalidades distribuidas (*crosscutting concerns*) y cuando tiene código disperso o mezclado (*code scattering*) [Pawlak2005].

#### Un caso de funcionalidades distribuidas *(crosscutting concerns) [Pawlak2005].*

En POO uno analiza cómo organizar una aplicación en clases. El análisis debe estar dirigido por la necesidad de separar y encapsular información y su procesamiento asociado en entidades coherentes.

Aunque las clases son programadas en forma independiente una de la otra, en ocasiones son interdependientes en cuanto a comportamiento. Por ejemplo el objeto `cliente` no debe ser borrado mientras exista una `orden` sin pagar; de otra forma se corre el riesgo de perder los datos de contacto del cliente. Para hacer cumplir esta regla se podría modificar el método `borrar` en `cliente` para determinar previamente si quedan órdenes pendientes. Sin embargo esta solución es deficiente por diferentes razones:

- Determinar si una orden ha sido pagada no pertenece a la administración del cliente, sino a la administración de órdenes. Es por esto que la clase `cliente` no debería administrar dicha funcionalidad.
- La clase `cliente` no debería necesitar saber de todas las reglas de integridad de datos que imponen otras clases de la aplicación.
- Modificar la clase `cliente` para tomar en cuenta estas reglas de integridad restringe la posibilidad de reutilizar la clase en otras situaciones. En otras palabras, una vez que la clase `cliente` implementa cualquier funcionalidad que está relacionada con otra clase, en muchos casos ya no puede utilizarse en forma independiente.

Otra posibilidad podría ser implementar dicha funcionalidad en la clase `orden`. Sin embargo esta solución no es mejor. La clase `orden` no tiene por qué permitir (o no) que se borre un cliente. Estrictamente hablando, esta regla no está asociada específicamente con el cliente, ni con la orden, sino que atraviesa ambos tipos de entidad.

#### Un caso de código disperso o mezclado (*code scattering*) [Pawlak2005].

En POO, la forma en que los objetos interactúan es por medio de la invocación de métodos. En otras palabras, un objeto que requiere llevar a cabo una acción, invoca un método que pertenece a otro objeto (o a uno propio). La POO siempre conlleva dos roles: el de invocar y el de ser invocado.

Cuando uno escribe el código para llamar a un método, no necesita preocuparse por cómo ha sido implementado el servicio ya que la llamada interactúa exclusivamente con la interfaz del objeto invocado. Es suficiente con asegurar que los parámetros especificados correspondan con aquellos definidos por la firma del método.

Ya que los métodos son implementados dentro de clases, éstos son implementados como bloques de código claramente delimitados. Para modificar un método, obviamente se modifica el archivo que

contiene la clase donde está definido el método. Si se modifica solamente el cuerpo del método, la modificación es transparente, porque el método seguirá siendo llamado exactamente de la misma forma.

Sin embargo, si se modifica la firma del método (por ejemplo agregando un parámetro), la modificación es mucho más laboriosa. Ahora será necesario modificar todas y cada una de las llamadas hacia dicho método. Entonces hay que modificar todas aquellas clases que lo invocan. Si estas llamadas aparecen en muchos lugares en el programa, hacer los cambios puede consumir una gran cantidad de tiempo.

En resumen: incluso si la implementación de un método está localizada en una sola clase, las llamadas a dicho método pueden estar dispersas en toda la aplicación. Este fenómeno de código disperso hace más lentas y complicadas las labores de mantenimiento, y dificulta la capacidad de adaptación y evolución de las aplicaciones orientadas a objetos. Cualquier cambio en la forma de utilización de un servicio requiere muchas otras modificaciones; un proceso costoso que además introduce errores.

## 1.3. Programación Orientada a Aspectos (POA)

La orientación a aspectos se ha investigado desde hace relativamente mucho tiempo, sin embargo comenzó a obtener reconocimiento en 1997 cuando Gregor Kiczales de Xerox Parc presentó su ponencia principal sobre programación orientada a aspectos (POA) en OOPSLA '97 [Jacobson2004].

Un "aspecto" es un tipo particular de interés. Un interés es cualquier código relacionado con una meta, característica, concepto o tipo de funcionalidad. Un Aspecto es un interés cuya funcionalidad es disparada por otros intereses y en múltiples situaciones. Aunque el desarrollo de software orientado a aspectos no es un cambio de paradigma como lo fue la orientación a objetos (OO), ayuda a tener una mejor modularidad en las aplicaciones, permitiendo desarrollar sistemas más comprensibles que sean fáciles de mantener y extender según las necesidades de los involucrados [Jacobson2004].

En POA los diferentes aspectos del comportamiento del sistema son programados por separado en su forma más natural, y posteriormente son "tejidos" entre sí, para producir código ejecutable [Kiczales1997].

POA resuelve el problema [de *características distribuidas*] por medio de un mecanismo de composición para agregar comportamientos adicionales desde afuera de una clase, hacia la clase misma. Es una forma de separar la implementación de diferentes *intereses* en módulos separados. La composición puede ocurrir durante la compilación o en tiempo de ejecución [Jacobson2004].

La mayoría de los ambientes de desarrollo POA están construidos en Java (incluidos AspectJ/AspectWerkz, JAC, JBoss AOP y Spring). Sin embargo ninguno de los conceptos de POA son específicos a este lenguaje. De la misma forma que el concepto de objeto se puede aplicar con éxito a varios lenguajes, el concepto de aspecto se puede implementar en C++ (AspectC++), C# (AcpectC#), Smalltalk (Apostle), e incluso en lenguajes procedurales como C (AspectC).

Incluso, los conceptos de POA son importantes no solamente durante la etapa de desarrollo, sino antes. Para la etapa de diseño conceptual (análisis) hay herramientas que proponen notaciones para UML para modelar los conceptos de POA como Theme/UML y JAC UML Aspectual Factory (UMLAF) [Pawlak2005].

### 1.3.1. Conceptos de POA

#### *El concepto "Aspecto"*

Para comprender y manejar un programa complejo, generalmente es mejor dividirlo en sub-programas.

Cuando un programa está escrito con un lenguaje procedural, la aplicación es modularizada de acuerdo con procedimientos a ser llevados a cabo. Por otro lado, cuando se utiliza un lenguaje orientado a objetos, la modularización está basada en la encapsulación de datos en las clases.

En ambos casos, algunas funcionalidades son más difíciles de modularizar. En esos casos, se dice que el código relacionado con dichas funcionalidades está mezclado o disperso.

#### *Código Mezclado.*

El código mezclado o disperso en una aplicación hace que el desarrollo, mantenimiento, y evolución

del programa sean más lentos.

La razón principal de que existan *características distribuidas* tiene que ver con las diferencias en la forma de hacer disponibles los servicios, y la forma en que se utilizan. Una clase proporciona acceso a uno o más servicios por medio de sus métodos. Es relativamente sencillo agrupar los servicios disponibles en un solo lugar, en otras palabras, en la misma clase. Sin embargo, como fue mencionado con anterioridad, una vez que estos servicios se han usado desde múltiples clases es difícil hacer re-ingeniería en la aplicación para agrupar las llamadas a esos métodos.

El problema con el código mezclado no está relacionado con un lenguaje en particular. El código mezclado o disperso es un efecto que se presenta en cualquier programa complejo. Sin embargo, ya que depende fuertemente del problema concreto atacado por la aplicación, es muy difícil de eliminar.

### *Dos dimensiones en cuanto a Modularidad.*

La contribución principal de POA es que provee una forma de agrupar (en un aspecto) código que de otra forma tendría que estar disperso o mezclado en la aplicación.

> **Definición-** ***Aspecto***: Unidad de programación diseñada para capturar una funcionalidad o característica dispersa en la aplicación [Pawlak2005].

Juntar clases y aspectos en la misma aplicación, significa que la modularidad puede funcionar en dos dimensiones: las funcionalidades básicas que pueden implementarse en clases (podría llamarse "*estructural*") y las funcionalidades dispersas o mezcladas, implementadas con aspectos (podría llamarse "*operacional*")

Una aplicación diseñada con ambas dimensiones de funcionalidad por separado, es más fácil de escribir, mantener y adaptar [Pawlak2005].

*Servicios no funcionales y aspectos.*

En la mayoría de los casos, los servicios no funcionales son llamados a lo largo del código de la capa de negocio. Por lo tanto, éstos servicios son fundamentalmente características que están distribuidas (*crosscutting concerns*). En consecuencia, los servicios no funcionales se implementan como aspectos en POA, mientras que los intereses funcionales se implementan como clases. Sin embargo, en algunos casos, hay intereses de negocio distribuidos (*crosscutting*) que es mejor que sean implementados como aspectos.

*Inversión de dependencias.*

Utilizando POA, el servicio no funcional es integrado a la aplicación por medio de un aspecto. Contrariamente a la POO, la aplicación ya no depende del servicio, sino que ahora el aspecto es lo que depende del servicio. Este cambio en la dirección de las dependencias entre el servicio y la aplicación, no es específico para POA. En la utilización de algunos marcos de trabajo (*frameworks*) también es el caso. La ventaja principal de este cambio en las dependencias, es facilitar el trabajo del desarrollador.

Utilizando POA, todo el código necesario para utilizar un servicio por medio de un API (Interfaz de Programación de Aplicaciones, por sus siglas en inglés) se encuentra concentrado en un solo lugar. El desarrollador de aspectos encargado de desarrollar dicho servicio, también se encarga de manejar la

integración del servicio en la aplicación. La ventaja de hacerlo así es que se reduce el riesgo de usar inadecuadamente el servicio, ya que el desarrollador especializado en aspectos tiene un mejor entendimiento del servicio que el resto de los desarrolladores, quienes solamente utilizan el API.

*Aspectos y marcos de trabajo (frameworks)*

Un marco de trabajo es un conjunto de clases que proveen una estructura reusable para escribir aplicaciones.

Desarrollar una aplicación con un marco de trabajo consiste en escribir código que es manejado por el marco de trabajo. Este código no es ejecutado directamente; en cambio es el marco de trabajo quien lo invoca de acuerdo al contexto. En otras palabras el marco de trabajo forma un conjunto de servicios que extienden el código que uno escribe.

La situación es similar a POA. Los servicios son provistos por aspectos que extienden la capa de negocio en la aplicación. La diferencia es que los marcos de trabajo proporcionan un conjunto fijo de servicios, mientras que aquellos proporcionados por la POA son enteramente programables con aspectos. Por lo tanto la POA proporciona un mecanismo mucho más general para la inversión de dependencias que los marcos de trabajo, que están limitados al dominio de la aplicación

### *Tejido de aspectos*

Una aplicación orientada a aspectos contiene clases y aspectos. La operación que toma estas clases y aspectos como entrada y produce una aplicación que integra ambas funcionalidades, es conocida como *tejido de aspectos.* El programa que realiza dicha operación, es conocido como *tejedor de aspectos* o solamente *tejedor*.

**Definición-** ***Tejedor de Aspectos***: Programa que integra clases y aspectos. El tejido se puede llevar a cabo en tiempo de compilación o en tiempo de ejecución [Pawlak2005].

*Tejido en tiempo de compilación. (Utilizado por AspectJ y opcionalemente en JBoss)*

Con el tejido en tiempo de compilación, el tejedor es un programa que produce el código de la aplicación en que las clases son extendidas por los aspectos, antes de cualquier ejecución.

Un tejedor en tiempo de compilación es muy similar a un compilador, y comúnmente es llamado *compilador de aspectos*.

El tejedor en tiempo de compilación más conocido y popular actualmente es ***AspectJ***.

Con lenguajes modernos OO, como Java o C#, las aplicaciones son compiladas a código intermedio. Al tejer aspectos con estas aplicaciones, puede haber dos soluciones: tejer los aspectos con el código fuente, o tejerlos con el código intermedio.

Es más comúnmente utilizada la segunda opción. Un tejedor de código intermedio puede tejer aplicaciones comerciales y de terceros que no proporcionan su código fuente. Aparte, es más sencillo realizar el análisis del código intermedio. Una consecuencia directa de realizar el tejido en código intermedio, es que dicho tejedor tiene un mejor desempeño que el tejedor de código fuente.

El resultado de un tejedor en tiempo de compilación puede ser código fuente o código intermedio (al

menos cuando se utiliza lenguaje Java). Evidentemente, si se realiza el tejido de código intermedio, solamente se puede obtener código intermedio. La ventaja de generar código fuente, es que es fácilmente manejable por un programador, quien puede estudiar el proceso de tejido y entender lo que el tejedor ha hecho. La desventaja es que este código ahora debe ser compilado, lo que reduce la velocidad de producción de código ejecutable.

Con el tejido en tiempo de compilación, los aspectos son añadidos al código de la aplicación. Al ser ejecutada, este nuevo código no es diferente en lo absoluto del código original. Entonces se dice que este código es estático. Para agregar o remover un aspecto, es necesario volver a tejer la aplicación.

*Tejido en tiempo de ejecución (Utilizado opcionalmente en JBoss, por ejemplo)*

Con el tejido en tiempo de ejecución, la distinción entre los objetos de la aplicación y los aspectos es clara durante la ejecución. Un tejedor en tiempo de ejecución es un programa capaz de orquestar la ejecución de estos dos tipos de entidades. En otras palabras el tejedor ejecuta ya sea el código de la aplicación o el código del aspecto, dependiendo de las directivas de tejido definidas.

El proceso de tejer aspectos en tiempo de ejecución puede ser comparado con mantener la relación entre un conjunto de objetos de la aplicación y un conjunto de instancias de aspectos. Un objeto de la aplicación que está vinculado con una instancia de un aspecto es modificado por dicho aspecto. Una instancia de un aspecto puede ser vinculada con varios objetos de la aplicación (el aspecto está distribuido o *crosscuts* varios lugares en la aplicación) así como un objeto de la aplicación puede estar vinculado con varias instancias de aspectos (más de un aspecto aplica para la misma ubicación).

La ventaja del tejido en tiempo de ejecución es que las relaciones entre objetos y aspectos pueden ser administradas dinámicamente. Agregando o removiendo un vínculo se puede tejer o destejer un interés, mientras que la aplicación sigue ejecutándose. Esta cualidad dinámica es particularmente útil al tratarse de aplicaciones de alta disponibilidad, como páginas de Internet.

Los tejedores en tiempo de ejecución pueden trabajar de dos formas. Agregando “ganchos” en tiempo de carga de la aplicación, o por medio de ejecutar la aplicación en modo de supervisión.

La técnica de “ganchos” consiste en transformar la aplicación introduciendo “lugares” donde podría ser ejecutado código de aspectos. Un “gancho” es una porción de código que redirige el flujo de ejecución de la aplicación hacia un aspecto. Los tipos de lugares donde se puede agregar “ganchos” depende del tejedor.

La técnica de ejecutar la aplicación en modo de supervisión, es similar a ejecutar la aplicación en un depurador, en el que cada vez que la aplicación llega a un punto en que aplica un aspecto, el “supervisor” interrumpe la ejecución normal de la aplicación, y ejecuta el aspecto. La ventaja de esta técnica es que la aplicación permanece libre de cualquier transformación. La desventaja de esta técnica es que el modo de supervisión representa un costo computacional que reduce el desempeño de ejecución de la aplicación.

### *Puntos de unión (Joinpoints)*

La definición de aspecto o estructura de una característica distribuida (*crosscutting structure*) está basada en la noción de *punto de unión*.

**Definición-** ***Punto de unión***: Un punto en el control de flujo de un programa en donde uno o varios aspectos aplican [Pawlak2005].

Aunque la noción de punto de unión es muy general (potencialmente, todas las instrucciones de un programa podrían ser un punto de unión), no todos los puntos en el control de flujo se consideran útiles para la POA. Los puntos de unión son agrupados de acuerdo a su tipo, y solamente un subconjunto de todos los posibles tipos de puntos de unión son admitidos por los lenguajes orientados a aspectos.

Incluso, la noción de punto de unión está fuertemente relacionada con una ejecución en particular de un programa (un flujo de control). Se pueden obtener diferentes conjuntos de puntos de unión desde diferentes ejecuciones del mismo programa.

### *Diferentes tipos de puntos de unión.*

Aún cuando la definición de un punto de unión ocurre en tiempo de ejecución, dicha definición está basada en la estructura de un programa (sus clases, métodos, atributos, etc). Independientemente de alguna implementación específica, los siguientes son tipos de puntos de unión:

- *Métodos*. Con lenguajes orientados a objetos, la ejecución de un programa se puede ver como una secuencia de llamadas y ejecución de métodos. Los diferentes escenarios de ejecución de una aplicación se pueden expresar en términos de secuencias de mensajes que disparan ejecuciones de métodos. Por lo tanto, llamadas a métodos y ejecuciones de métodos son dos tipos de puntos de unión comúnmente utilizados. Hay que notar que las ejecuciones de métodos no son "puntos" estrictamente hablando, pero aún así se consideran puntos de unión.
- *Constructores*. Los constructores son la entidad principal utilizada para crear objetos de una aplicación. Así como en los métodos, las llamadas y las ejecuciones de un constructor representan tipos de puntos de unión.
- *Excepciones*. Una excepción es lanzada para señalizar una situación anormal en tiempo de ejecución, y es "cachada" para ejecutar un tratamiento en particular a dicha situación. Estos dos eventos son puntos muy importantes en la ejecución de una aplicación. Ambos pueden ser considerados tipos de puntos de unión.
- *Atributos*: Muchos aspectos, como por ejemplo el aspecto encargado de la persistencia, necesitan manejar los datos de la aplicación. Los atributos son el elemento principal de código que implementa estos datos. Por lo tanto, los lenguajes orientados a aspectos consideran la lectura y escritura de dichos atributos como tipos de puntos de unión.

### *Puntos de corte (Pointcuts)*

Un aspecto en sí está compuesto por dos partes: el punto de corte y el código del *advice.* El código del *advice* contiene el código a ser ejecutado, mientras que el punto de corte define los puntos en el programa donde este código debe ser implementado.

La noción de punto de unión no es suficiente por sí sola para definir cuáles puntos de unión son pertinentes para un aspecto dado. Es necesaria una entidad encargada de describirlos específicamente. Esta entidad está definida por la noción de punto de corte.

**Definición-** ***Punto de corte***: El conjunto de puntos de unión en donde se aplica un aspecto [Pawlak2005].

Por encima de todo, la naturaleza distribuida de un aspecto es expresada con un punto de corte, porque éste agrupa puntos de unión que pueden estar en diferentes archivos de código fuente. Los elementos de dicho conjunto de puntos de unión comparten algo en común: representan solamente aquellos lugares del programa en los que aplica un aspecto dado. Por lo tanto, los puntos de corte son una forma de "hablar al respecto" de la aplicación. Comúnmente se dice que un punto de corte representa un "dónde". Por otro lado, el "qué" es representado por un *advice*.

Es importante tomar en cuenta que la mayoría de los puntos de corte dependen de la aplicación. Cuando es necesario reutilizar un aspecto para una aplicación diferente, comúnmente va a ser necesario adaptar la definición de los puntos de corte para los lugares existentes en la nueva aplicación.

Incluso cuando la noción de punto de unión y la noción de punto de corte están estrechamente relacionadas, su naturaleza es diferente. Los puntos de unión son *entidades de tiempo de ejecución* bien definidas. En cambio, se debe ver a los puntos de corte como *elementos estructurales de código* que participan en la definición de un aspecto.

Como es de esperarse, cada herramienta de POA implementa su propio lenguaje para definir los puntos de corte.

### *Código advice*

Como fue visto previamente, el código del *advice* define "qué" instrucciones debe realizar el aspecto:

**Definición- *Código advice***: La definición del comportamiento del aspecto [Pawlak2005].

El código de un *advice* está asociado con un punto de corte para implementar la funcionalidad de una característica distribuida (*crosscutting concern*). Las sentencias definidas en el código del *advice* son realizadas para todos los puntos de unión que conforman al punto de corte definido. Es parecido a un método, porque posee un cuerpo que contiene instrucciones. Pero es diferente a un método, porque su código nunca es llamado directamente, sino tejido en los puntos de unión especificados en su punto de corte asociado.

### *Diferentes tipos de código advice*

- *before*: Es el código del *advice* que se ejecuta antes de los puntos de unión.
- *after*: Es el código del *advice* que se ejecuta después de los puntos de unión.
- *around*: Es el código del *advice* que se ejecuta antes y después de los puntos de unión.

Para el código del *advice* de tipo "around", es necesario separar las instrucciones que deben ser ejecutadas antes de los puntos de unión, de las que deben ser ejecutadas después. Para esto se utiliza una instrucción especial, que en la mayoría de los casos es llamada `proceed`. La instrucción `proceed` provee una forma de retomar la ejecución normal del programa y ejecutar el código contenido en el punto de unión.

Se podría resumir el comportamiento de un código *advice* de tipo "*around*" de la siguiente manera:

1. El programa se ejecuta normalmente
2. Justo antes del punto de unión que está incluido en el punto de corte, se ejecutan las

instrucciones de la parte "*before*" definidas en el *advice* de tipo "*around*"

3. Se realiza una llamada a la instrucción `proceed`
4. El código que está definido en el punto de unión es ejecutado.
5. Es ejecutada la parte "*after*" definida en el *advice* de tipo "*around*".
6. La ejecución del programa es retomada justo después del punto de unión.

En caso de ser omitida la llamada a la instrucción `proceed`, entonces no se ejecuta el código del punto de unión. Es una forma de reemplazar dicho código por el definido en el *advice*.

Además de los tres tipos de código *advice* mencionados previamente, algunos lenguajes orientados a aspectos, definen dos tipos más: "*after returning*" y "*after throwing*". La idea es que un método puede retomar el control de flujo normalemente, o levantar una excepción. En el primer caso, el código del *advice* de tipo "*after returning*" será ejecutado después de regresar al flujo normal de ejecución, y en el segundo caso, el *advice* de tipo "*after throwing*" será ejecutado en caso de que hubiera sido levantada una excepción.

### *El mecanismo "Introduction"*

Las nociones de punto de corte y de código *advice* permiten implementar características distribuidas por medio de modificaciones de *comportamiento*. Es decir, el flujo de control del programa se desvía hacia el código del *advice*, que agrega, remueve o modifica comportamiento. Sin embargo, es necesario un mecanismo complementario que permita extender la estructura estática de un programa. Este mecanismo es la *Introduction*.

**Definición-** ***Introduction***: Un mecanismo de extensión para agregar elementos de código estructural nuevos en una aplicación [Pawlak2005].

De esta manera, se pueden "introducir" nuevos elementos estructurales de código previamente inexistentes. Los dos elementos de código más comúnmente "introducidos" por aspectos son atributos y métodos. En relación con los métodos, también se puede "introducir" interfaces; la idea es hacer que una clase específica previamente existente, ahora implemente una nueva interfaz adicionalmente.

Así como las relaciones de herencia en la POO, el mecanismo "*introduction*" permite la extensión de clases existentes. Sin embargo, al contrario que la herencia, el mecanismo "*introduction*" no permite la redefinición de elementos ya existentes; solamente puede agregar nuevos. Esta limitación se debe al objetivo de conservar la integridad del programa, especialemnte cuando son compuestos varios aspectos simultáneamente.

### *Composición de aspectos*

Los conceptos definidos hasta el momento, se ocupan de la definición de aspectos y su integración en aplicaciones. Ahora es necesario tratar el problema de la interacción entre diferentes aspectos. Este problema es conocido como *composición de aspectos*.

El estudio de la composición de aspectos puede llevarse a cabo desde dos puntos de vista complementarios: en tiempo de diseño (al elegir los aspectos para una aplicación), y en tiempo de

tejido (al momento de aplicar varios aspectos en un mismo punto de unión)

### *Diseño*

Así como sucede con las clases, los aspectos pueden ser programados en forma independiente, y después reutilizados e integrados en una aplicación. El programador de aspectos, entonces, debe verificar que no haya conflictos ni se introduzcan inconsistencias en la ejecución de la aplicación.

Pueden ocurrir varios tipos de conflictos:

- Incompatibilidad: Dos aspectos pueden introducir funcionalidades incompatibles. Por ejemplo, un aspecto encargado de manejar transacciones, y un aspecto encargado de manejar persistencia. Ambos comparten el uso de una base de datos. Al ser desarrollados por separado, cada uno puede no tomar en cuenta las operaciones de base de datos realizadas por el otro aspecto.
- Dependencia: Dos aspectos pueden estar relacionados. En este caso, si un aspecto se usa, el otro debe ser usado también.
- Redundancia: Dos aspectos podrían implementar la misma funcionalidad en forma diferente. La utilización de dichos aspectos, podría ser inútil y debe ser evitado.

Estos tipos de conflictos ocurren a nivel semántico y no pueden ser abordados en forma sencilla a nivel de programación con las herramientas existentes en la actualidad.

### *Tejido*

Adicionalmente a los problemas semánticos mencionados, la composición de varios aspectos que pueden utilizar puntos de unión en común requiere la definición de su orden de ejecución. No hay una solución automática a dicho problema. Por ejemplo, la utilización de un aspecto de rastreo y un aspecto de seguridad. Una persona puede querer rastrear todas las peticiones, incluso aquellas que fueron rechazadas por no tener autorización. En ese caso primero debe ejecutarse el aspecto de rastreo y después el de seguridad. En cambio, si solamente se quieren rastrear las peticiones realizadas por usuarios autorizados, el aspecto de seguridad debe ejecutarse antes del aspecto de rastreo. Por lo tanto, es necesario que alguien decida el orden de ejecución para cada aspecto que se ejecute en un mismo punto de unión.

## 1.4. Arquitectura Dirigida por Modelos (MDA)

Lanzado por la OMG (Object Management Group) en el 2001, Arquitectura Dirigida por Modelos (MDA, por sus siglas en inglés) define un enfoque para la especificación de sistemas de tecnologías de información. Su objetivo es separar la especificación de la funcionalidad del sistema, de la especificación de la implementación de dicha funcionalidad en una plataforma tecnológica específica. MDA define una arquitectura de modelos que provee un conjunto de directivas o lineamientos para estructurar especificaciones expresadas como modelos [Miller2001].

El enfoque de MDA y los estándares que le dan sustento, permiten que el mismo modelo de especificación de funcionalidad, se pueda realizar en múltiples plataformas [Miller2001].

La promesa de la arquitectura dirigida por modelos es: brindar la posibilidad de definir aplicaciones y modelos de datos de tal manera que puedan ser leídos por computadoras, y así proveer flexibilidad a largo plazo en cuanto a [Miller2003]:

- Implementación: Una nueva infraestructura de implementación puede ser integrada, utilizando diseños existentes [Miller2003].
- Integración: Siendo que el diseño también existe al momento de la integración, y que no solamente existe la implementación, se puede automatizar la producción de puentes de integración de datos y la conexión a nuevas infraestructuras de integración [Miller2003].
- Mantenimiento: La disponibilidad del diseño en forma que pueda ser leído por computadoras proporciona a los desarrolladores acceso directo a la especificación del sistema, haciendo mucho más sencillas las labores de mantenimiento [Miller2003].
- Pruebas y simulación: Ya que los modelos desarrollados pueden ser utilizados para generar código, de igual forma pueden ser validados contra los requerimientos, pueden ser probados en diferentes infraestructuras y pueden ser utilizados para simular el comportamiento del sistema que está siendo diseñado [Miller2003].

En realidad la Arquitectura Dirigida por Modelos es solamente otro paso evolutivo en el campo de desarrollo de software. La magia de la automatización de software desde modelos es realmente sólo otro nivel de compilación [Miller2003].

MDA es un marco arquitectónico de desarrollo y está respaldado por una especificación detallada provista por la OMG. En la especificación de MDA, desde 1995, la OMG adoptó el uso extensivo de UML, ya que actualmente es el lenguaje de modelado más popular en la industria.

Sin embargo, debido a que diferentes herramientas de modelado UML utilizan su propia forma de guardar archivos, se convirtió en un problema poder compartirlos o procesarlos. Es por esto que la OMG también desarrolló la especificación MOF (*MetaObject Facility Specification*). MOF es actualmente un estándar en la industria con el cual se pueden exportar modelos de una aplicación y ser importados en otra, pueden ser almacenados en un repositorio para ser recuperados en el futuro con una cada vez mayor variedad de programas, e incluso ser presentados con diferentes formatos, transformados y utilizados para generar código.

La mayoría de los marcos MDA permiten la utilización de modelos generados con diversas herramientas UML ya sea con un formato propietario o por medio del formato MOF.

### 1.4.1. Conceptos de MDA

Entre las especificaciones que la OMG ha definido se encuentra una definición detallada de lo que significan ciertos conceptos básicos con respecto a MDA, para asegurar que todos hablan de lo mismo. Los conceptos principales sobre MDA son:

- **Sistema**

  Se presentan los conceptos de MDA en términos de un sistema existente o planeado. Ese sistema puede contener cualquier cosa: un programa, un único sistema de cómputo, alguna combinación de partes de diferentes sistemas, un conjunto de sistemas cada uno bajo un control independiente, gente, una empresa, un corporativo de empresas... [Miller2003]

- **Modelo**

  Un modelo de un sistema es una descripción de la especificación de dicho sistema junto con su ambiente para algún propósito específico. Frecuentemente, un modelo es presentado como la combinación de dibujos y texto. El texto puede estar en un lenguaje de modelado o en lenguaje natural [Miller2003].

- **Dirigido por modelos**

  MDA es un enfoque respecto al desarrollo de sistemas que incrementa el poder de los modelos para este trabajo. Es dirigido por modelos ya que provee una forma de utilizar modelos para dirigir el curso del entendimiento, diseño, construcción, despliegue, operación, mantenimiento y modificación [Miller2003].

- **Arquitectura**

  La *arquitectura* de un sistema es una especificación de las partes y conexiones del sistema así como de las reglas para la interacción de las partes que utilizan dichas conexiones. La Arquitectura Dirigida por Modelos prescribe ciertos tipos de modelos a ser utilizados, cómo deben ser preparados y las relaciones entre los diferentes tipos de modelos [Miller2003].

- **Punto de vista**

  Un *punto de vista* en un sistema es una técnica de abstracción, utilizando un conjunto de conceptos arquitectónicos y reglas estructurales, con el objetivo de enfocarse en intereses o preocupaciones específicas dentro de ese sistema. Abstracción se refiere al proceso de suprimir ciertos detalles para establecer un modelo simplificado [Miller2003].

  MDA especifica tres puntos de vista de un sistema: Independiente de la computadora, independiente de la plataforma y específico para una plataforma [Miller2003].

- **Vista**

  Un modelo de punto de vista o una *vista* de un sistema es una representación de dicho sistema desde la perspectiva de un punto de vista determinado [Miller2003].

- **Plataforma**

  Por *plataforma* se entiende el conjunto de subsistemas y tecnologías que proveen un conjunto de funcionalidades coherente por medio de interfaces y patrones específicos de uso, que cualquier aplicación admitida por dicha plataforma puede utilizar sin preocuparse de los detalles

de cómo fue implementada [Miller2003].

- **Aplicación**

  Aplicación se refiere a una funcionalidad en desarrollo. Un sistema es descrito en términos de una o más aplicaciones admitidas por una o más plataformas.

- **Independencia de la plataforma**

  Es una cualidad que un modelo puede exhibir. Es la cualidad de que el modelo es independiente de las características de una plataforma de cualquier tipo.

  Es cuestión de grado. Un modelo puede asumir la disponibilidad de características de un tipo muy general de plataforma, o podría comprometerse 100% con un tipo de plataforma determinado.

- **Puntos de vista de MDA**

  - Punto de vista independiente de la computadora.

    Se enfoca en el ambiente del sistema, y los requerimientos para el sistema: los detalles de la estructura y procesamiento del sistema están ocultos, o indeterminados aún.

  - Punto de vista independiente de la plataforma.

    Se enfoca en la operación del sistema, mientras que oculta los detalles para una plataforma específica. Muestra aquella parte de la especificación completa del sistema que no cambia de una plataforma a otra.

  - Punto de vista específico para una plataforma.

    Combina el punto de vista independiente de la plataforma con un enfoque adicional en el detalle del uso de una plataforma específica.

- **Modelo Independiente de la Computadora (*Computational-Independent model, CIM*)**

  Un CIM es una vista del sistema desde un punto de vista independiente de la computadora. El CIM no muestra detalles de la estructura de los sistemas. En algunas ocaciones, el CIM es llamado modelo del dominio, y se desarrolla utilizado un lenguaje familiar para los profesionales en el dominio de la aplicación.

  El CIM juega un papel importante al establecer un puente entre aquellos que son expertos entre el dominio del problema y sus requerimientos, y aquellos que son expertos en el diseño y construcción de los artefactos que satisfacen los requerimientos del dominio [Miller2003].

- **Modelo Independiente de la Plataforma (*Platform Independent Model, PIM*)**

  El PIM representa una vista del sistema desde un punto de vista independiente de la plataforma.

  Una técnica muy común para lograr independencia de la plataforma es dirigir el modelo del sistema hacia una máquina virtual neutral en cuanto a tecnología. Una máquina virtual está definida como un conjunto de partes y servicios (comunicaciones, calendarización, nombrado, etc.) que están definidos independientemente de alguna plataforma y que es llevada en forma específica a múltiples plataformas. Una máquina virtual es una plataforma, y dicho modelo es específico para esa plataforma. Sin embargo, ese modelo es independiente con relación a las plataformas en que la máquina virtual ha sido implementada. Entonces, se ajusta perfectamente

al criterio de independencia de la plataforma, mencionado con anterioridad [Miller2003].

- **Modelo Específico de Plataforma (*Platform Specific Model*, PSM)**

  El PSM es una vista del sistema desde un punto de vista específico para una plataforma. Combina especificaciones del PIM con los detalles de cómo dicho sistema utiliza un tipo de plataforma en particular [Miller2003].

- **Transformación de Modelos**

  La transformación de modelos es el proceso por el cual un modelo es convertido en otro modelo del mismo sistema, Un ejemplo de transformación de modelos es que la combinación de un PIM junto con información extra, se convierte en un PSM.

  Hay que notar que en el caso de PIMs basados en máquinas virtuales, no son necesarias transformaciones. En cambio, lo que se transforma es el PIM de la máquina virtual en sí en un PSM de una plataforma específica [Miller2003].

- **Implementación**

  Una implementación es una especificación que provee toda la información necesaria para construir un sistema y ponerlo en operación [Miller2003].

### *1.4.2. Desarrollo de software con MDA*

Como se dijo previamente, el desarrollo de software con MDA consiste en la definición y transforamción de diferentes modelos.

A continuación se presenta una breve explicación del proceso involucrado en el desarrollo de software utilizando este marco de desarrollo.

1. **CIM (Modelo Independiente de la Computadora)**

   Los requerimientos del sistema son modelados en un modelo independiente de la computadora, CIM, describiendo la situación en que el sistema será utilizado. En ocasiones dicho modelo es llamado modelo del dominio o modelo del negocio [Miller2003].

   Un CIM es un modelo del sistema que muestra el sistema en el entorno en que operará, por lo tanto, ayuda a presentar exactamente lo que se espera que haga. Es útil, no solamente como ayuda para comprender el problema, sino como fuente de un vocabulario compartido para ser utilizado en otros modelos. En una especificación MDA de un sistema, los requerimientos CIM, deben ser rastreables en el PIM y en el PSM que lo implementan y viceversa [Miller2003].

2. **PIM (Modelo Independiente de la Plataforma)**

   Un modelo independiente de la plataforma, un PIM, es construido. Se describe el sistema sin mostrar detalles del uso de la plataforma [Miller2003].

3. **Modelo de la plataforma**

   El arquitecto seleccionará entonces una o varias plataformas que permitan la implementación del sistema con las características arquitectónicas deseadas. Frecuentemente este modelo está presentado en forma de manuales de software y equipo (o simplemente en la cabeza del arquitecto) [Miller2003].

4. **Mapeo**

   En MDA, el mapeo provee especificaciones para la transformación de un PIM en un PSM para una plataforma en particular. El modelo de la plataforma determinará la naturaleza del mapeo [Miller2003].

   Hay diferentes tipos de mapeo, como *mapeo del tipo de modelo*, *mapeo de instancias de* modelos, una combinación del mapeo de *tipos de modelos* con *instancias de modelos*, *modelos de marcas*, *plantillas* y un *lenguaje de mapeo*, sin embargo éstos están fuera del alcance de este trabajo.

5. **Marcado del modelo**

   En el *mapeo con instancias de modelos* el arquitecto marca elementos del PIM para indicar los elementos a ser utilizados en el mapeo para transformarlo de PIM a PSM [Miller2003].

6. **Transformación**

   El siguiente paso es tomar el PIM marcado y transformarlo en un PSM. Esto puede realizarse en forma manual, con ayuda de la computadora o en forma automática [Miller2003].

7. **Transformación directa a Código**

   Una herramienta puede transformar un PIM directamente a código desplegable, sin producir un PSM [Miller2003].

8. **Registro de la transformación**

   Los resultados de transformar un PIM utilizando una técnica en particular son el PSM junto con un registro de la transformación. Dicho registro incluye un mapa desde los elementos del PIM hacia los elementos correspondientes en el PSM, y muestra qué elementos del mapeo fueron utilizadas para cada parte de la transformación [Miller2003].

9. **PSM (Modelo Específico de Plataforma)**

   El modelo específico para una plataforma producido con la transformación es un modelo del mismo sistema; también especifica cómo utiliza la plataforma elegida [Miller2003].

   Un PSM puede proveer más o menos información dependiendo del objetivo. Un PSM va a ser una implementación, en el caso de contener toda la información necesaria para construir el sistema y ponerlo en operación; o puede actuar como un PIM utilizado para lograr un mayor refinamiento en un PSM que sí pueda ser implementado [Miller2003].

### *1.4.3. Limitaciones de los Marcos MDA*

Los marcos de generación de código fuente desde un modelo (marcos MDA) tienen sus limitaciones.

Como fue presentado anteriormente, se lleva a cabo una transformación del modelo al código. Evidentemente, dicho modelo no puede contener toda la implementación de algoritmos y métodos. Sería inoperante. El modelo se haría tan complejo que no sería fácil de comprender. Estos marcos generan solamente el esqueleto, no toda la aplicación. No pretenden hacerlo.

Entonces, el código generado desde un modelo contiene métodos vacíos a ser completados posteriormente con la implementación de su funcionalidad.

¿Qué pasa si el modelo sufre modificaciones? ¿Cómo se puede generar nuevamente la arquitectura sin que ésto represente un dolor de cabeza para adecuar el código previamente modificado manualmente?

Diferentes marcos MDA abordan esta problemática de formas diferentes. Sin embargo, a fin de cuentas, incluso si fuera necesario, sigue siendo mucho más rápido y eficiente volver a generar el código fuente de la arquitectura en forma automática, y después agregar nuevamente la implementación de métodos y algoritmos.

# Capítulo 2. Proceso Unificado Orientado a Aspectos

## *2.1. Flujo de trabajo de Requerimientos orientado a aspectos*

El objetivo del flujo de trabajo de requerimientos es la comprensión de las necesidades de los involucrados en el sistema (sean usuarios, patrocinadores y demás interesados). Es indispensable encontrar la forma de asegurarse de que ambas partes (equipo de desarrollo y cliente) hablan de lo mismo y llegar a un acuerdo en la definición de las funcionalidades que debe tener el sistema.

Para cumplir estas metas, el Proceso Unificado describe el siguiente flujo de trabajo, artefactos y trabajadores [Jacobson2000]:

Flujo de trabajo:

- Encontrar trabajadores y casos de uso
- Priorizar casos de uso
- Detallar un caso de uso
- Realizar un prototipo de la interfaz de usuario
- Estructurar el modelo de casos de uso

Artefactos:

- Modelo de casos de uso
- Trabajador
- Caso de uso
- Descripción de la arquitectura
- Glosario
- Prototipo de interfaz

Trabajadores:

- Analista del sistema
- Especificador de casos de uso
- Diseñador de interfaces de usuario
- Arquitecto

### *2.1.1. Modificaciones (Requerimientos)*

Ya que el objetivo de la programación orientada a aspectos es mantener las preocupaciones o necesidades separadas, es importante identificarlas y estructurarlas lo antes posible en el ciclo de vida del proyecto, en vez de que sea un "descubrimiento" posterior. Dicho de otra manera, es necesario comenzar con la identificación y separación de preocupaciones o necesidades desde el flujo de trabajo de requerimientos. Esto no significa capturar a detalle absolutamente cada requerimiento para cada

necesidad o preocupación al principio del proyecto, sino organizar sistemáticamente las necesidades y preocupaciones de los interesados para tener una visión de futuros aspectos en el sistema [Jacobson2004].

La técnica de casos de uso, que plantea el Proceso Unificado, provee los medios para modelar sistemáticamente las necesidades o preocupaciones de los interesados por medio de relaciones como inclusión, extensión y generalización entre casos de uso.

**Actividades agregadas al flujo de trabajo:**

- Definición de casos de uso de Infraestructura.

  En primera instancia, es necesario estructurar los casos de uso identificando los de aplicación y de infraestructura para capturar los aspectos del software. Los primeros se refieren a casos de uso que corresponden a los requerimientos funcionales y los segundos a los requerimientos no funcionales y de infraestructura.

  No hay que olvidar que durante el flujo de trabajo de requerimientos, el objetivo es comprender las necesidades a satisfacer con el sistema. Es por esto que a este nivel se espera un esfuerzo por identificar aquellos requerimientos no funcionales que son independientes de la tecnología específica en que será implementado el sistema.

- Identificar casos de uso de extensión y casos de uso de inclusión.

  Desde el flujo de trabajo de requerimientos, es necesario hacer lo posible por identificar casos de uso de extensión, y casos de uso de inclusión. Los casos de uso de extensión, representan funcionalidad agregada a una funcionalidad base. Permiten agregar comportamiento sin modificar el caso de uso base [Jacobson2004]. Los casos de uso de inclusión, permiten factorizar comportamiento común entre casos de uso [Jacobson2004].

- Identificación de conceptos comunes entre los casos de uso

  El objetivo a este nivel, es un primer acercamiento en cuanto a identificar los elementos que se repiten de un caso de uso a otro. Para realizar esta actividad es necesario producir una matriz de Casos de Uso contra conceptos que originan superposición de funciones en entidades o clases. Su utilidad inmediata es identificar posibles rebanadas (que serán definidas en el flujo de trabajo del análisis). Sin embargo, más allá de esto, puede ayudar a identificar desde una etapa temprana en el desarrollo de la aplicación, elementos que pueden formar rebanadas de casos de uso no específicas a un caso de uso, así como elementos de infraestructura.

- Identificación y clasificación de casos de uso "*pares"*

  Los casos de uso *pares,* son casos de uso diferentes entre sí y aparentemente independientes. Sin embargo, dichos casos de uso utilizan o afectan elementos en común [Jacobson2004].

  El hecho de que no haya relaciones entre casos de uso "*pares*" significa que es posible que personas diferentes detallen dichos casos de uso por separado, y en paralelo. Cierta coordinación es requerida definitivamente, ya que los autores deben usar el mismo vocabulario [Jacobson2004].

**Artefactos agregados al flujo de trabajo:**

- Modelo de Casos de Uso (CUs) Enriquecido con CUs de infraestructura, extensiones e inclusiones
- Matriz de Casos de Uso contra Conceptos que originan superposición de funciones (en entidades o clases, por ejemplo)

  En un sistema de bolsa de trabajo, un ejemplo de dicha matriz es:

| Caso de Uso | Empresa | Postulante | Vacante |
|---|---|---|---|
| Consultar vacantes | √ | | √ |
| Administrar vacantes | | | √ |
| Postularse a vacante | | √ | √ |

  Son marcados aquellos conceptos que son utilizados en el caso de uso. En este ejemplo es fácil confundir los conceptos con los actores que pueden utilizar el caso de uso. Por ejemplo, *consultar vacantes* muestra la lista de vacantes según su tipo, o muestra la lista de vacantes de una empresa. Y no fue marcado al postulante aún cuando tanto el actor empresa, como el actor postulante pueden hacer uso de este caso de uso. El objetivo de este caso de uso es obtener una lista de las vacantes existentes, no aquellas personas que se han postulado.

- Clasificación de casos de uso "pares", por concepto que origina superposición.

  Con base en la Matriz de Casos de Uso contra Conceptos que originan superposición, se obtiene la clasificación de casos de uso "pares" por concepto. Esta lista será utilizada para la identificación de aspectos en fases de desarrollo posteriores.

  En el mismo sistema de bolsa de trabajo se obtiene, por ejemplo:

  Comparten el concepto *Vacante*
  - Caso de uso *Consultar Vacantes*
  - Caso de uso *Administrar Vacantes*
  - Caso de uso *Postularse a vacante*

**Trabajadores agregados al flujo de trabajo:**

- Analista de Aspectos.

  El Analista, junto con el Analista de Aspectos, debe identificar y organizar los casos de uso en casos de uso de Aplicación y casos de uso de infraestructura, los casos de uso "*pares*" y los casos de uso de extensión e inclusión. Los casos de uso de Aplicación son aquellos que proporcionan funcionalidades específicas solicitadas para el sistema, y los casos de uso de infraestructura son aquellos que son necesarios para implementar dichas funcionalidades, sin embargo no proporcionan funcionalidades que son directamente de utilidad para los actores y demás interesados, por lo que nunca son creadas instancias de éstos por sí mismos. Estos casos de uso son también llamados Casos de Uso Utilitarios [Jacobson2004].

Como se dijo anteriormente, no se tiene que hacer un trabajo exhaustivo desde el principio, ya que es muy difícil identificar todos los posibles casos de uso que entran en cada una de estas clasificaciones. Para eso nos apoyamos en que el PU (Proceso Unificado) es iterativo e incremental.

## 2.2. Flujo de trabajo Análisis orientado a aspectos

En este flujo se analizan los requisitos que se describieron en la captura de requisitos, refinándolos y estructurándolos. El objetivo de hacerlo es conseguir una comprensión más precisa de los requisitos y una descripción de los mismos que sea fácil de mantener y que ayude a estructurar el sistema.

En la descripción de los casos de uso, durante el flujo de trabajo de requerimientos, es utilizado el lenguaje del cliente ya que lo importante es que los comprenda y se pueda llegar a un acuerdo. Se utilizan muchas descripciones que pueden ser algo imprecisas a la hora de ser implementados. Los casos de uso deben mantenerse independientes uno de otro lo más posible. Debido a todo ésto, es probable que aún queden aspectos sin resolver relativos a los requisitos del sistema.

En consecuencia, en el análisis se puede razonar más sobre los aspectos internos del sistema, y por tanto resolver aspectos relativos a la interferencia que pueda existir entre casos de uso.

También se puede utilizar un lenguaje más formal para apuntar detalles relativos a los requerimientos del sistema, es decir, "refinar los requerimientos".

El modelo del análisis es:

- Descrito con el lenguaje del desarrollador.
- Vista interna del sistema.
- Estructurado por clases y paquetes estereotipados; proporciona la estructura a la vista interna.
- Utilizado fundamentalemente por los desarrolladores para comprender la estructura que debería tener el sistema.
- No debería contener redundancias e inconsistencias, entre requisitos.
- Esboza cómo llevar a cabo la funcionalidad dentro del sistema, incluida la funcionalidad significativa para la arquitectura; sirve como una primera aproximación al diseño.
- Define realizaciones de casos de uso, y cada una ellas representa el análisis de un caso de uso del modelo de casos de uso.

Una realización de caso de uso es una colaboración dentro del modelo de análisis que describe cómo se lleva a cabo y se ejecutará un caso de uso determinado en términos de las clases del análisis y de sus objetos en interacción [Jacobson2000].

Para cumplir estas metas, el Proceso Unificado describe el siguiente flujo de trabajo, artefactos y trabajadores [Jacobson2000]:

Flujo de trabajo:

- Análisis de la arquitectura
- Analizar un caso de uso
- Analizar una clase
- Analizar un paquete

Artefactos:

- Modelo del Análisis
- Clase del análisis
- Realización de caso de uso-análisis
- Paquete del análisis
- Descripción de la arquitectura (Vista del modelo de análisis)

Trabajadores:

- Arquitecto
- Ingeniero de casos de uso
- Ingeniero de componentes

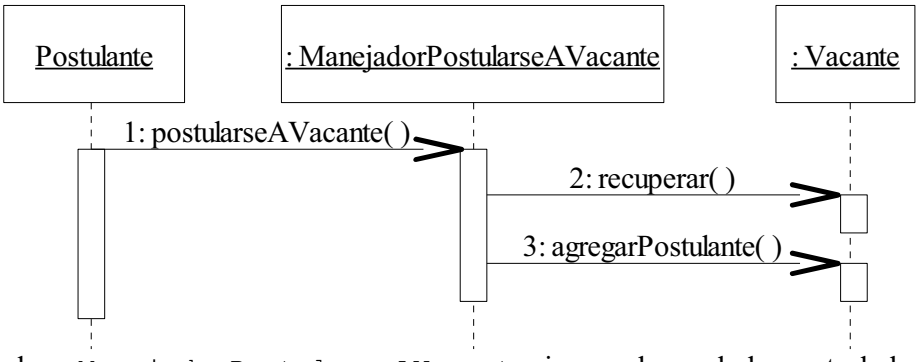
### 2.2.1. Modificaciones (Análisis)

Cuando se realiza un caso de uso, son identificadas las clases requeridas y sus características (atributos, métodos y relaciones). Algunas de estas clases y sus características son específicas para la realización del caso de uso, mientras que otras son necesarias para otras realizaciones de otros casos de uso [Jacobson2004].

A diferencia de las realizaciones de casos de uso definidas en el PUDS (Proceso Unificado de Desarrollo de Software) original, ahora es muy importante mantener separados los elementos que cada una de éstas aporta a una clase. Dicho de otra forma, cada extensión a una clase representada en el diagrama de colaboración de una realización de un caso de uso, identifica solamente aquel subconjunto de características necesarias para ese caso de uso.

Las características requeridas en cada clase son identificadas analizando cada paso en la descripción del caso de uso. Se utilizan los diagramas de interacción. Un diagrama de interacción representa, en orden cronológico, cómo interactúa una instancia de una clase con otra.

Un ejemplo de cómo se hace una realización de caso de uso integrando la POA y tomando el caso de uso "Postularse a Vacante" en un sistema de bolsa de trabajo, es mostrado a continuación.



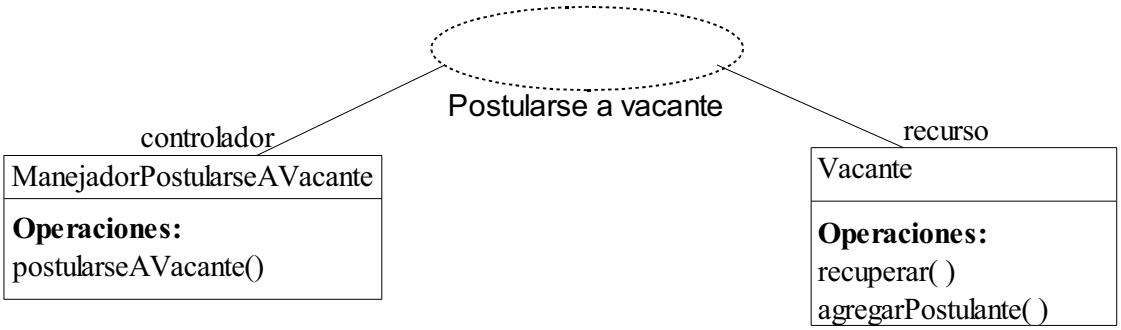

- La clase `ManejadorPostularseAVacante` juega el papel de controlador. Coordina otras clases en la realización del caso de uso Postularse a Vacante. En particular tiene un método `postularseAVacante()` para coordinar las acciones necesarias para postularse a una

vacante.

- La clase `Vacante` juega el papel de recurso al cual es posible postularse. Es responsable de obtener y actualizar la información sobre las vacantes.

Posteriormente, una vez identificadas las características de las clases involucradas en un caso de uso, se presentan agrupadas en una colaboración. Los roles jugados por las clases `ManejadorPostularseAVacante` y por `Vacante` son representadas por etiquetas en la punta de las asociaciones que conectan la colaboración y las extensiones a las clases [Jacobson2004].

Por ejemplo:




Muestra solamente aquellas características que tienen relación con el caso de uso postularse a vacante, mientras que en realidad puede tener muchas más; al momento de la composición/tejido de los diferentes aspectos.

En el desarrollo de software orientado a aspectos (DSOA) se diseñan los casos de uso en términos de rebanadas de casos de uso y módulos de casos de uso. Posteriormente son sobrepuestos unos sobre otros (como capas de acetatos) para formar el modelo completo del sistema.

En base a la realización de los casos de uso, son definidas rebanadas de cada caso de uso. Cada rebanada de caso de uso mantiene los elementos específicos de la realización de caso de uso en un modelo. Aquellas partes genéricas y reutilizables son mantenidas en rebanadas no específicas de un caso de uso [Jacobson2004].

Una rebanada de caso de uso contiene las especificaciones del modelo en un solo paquete. Una rebanada de caso de uso en el modelo de análisis contiene clases y aspectos de clases específicos a un caso de uso. También contiene la colaboración que describe la realización del caso de uso en términos de interacción, comunicación y diagramas de clases, entre otros [Jacobson2004].

**Actividades agregadas al flujo de trabajo:**

- Diseñar la *estructura de elementos* en capas, paquetes y subsistemas a nivel de análisis, con lo que se define una primera aproximación a la arquitectura.
- Diseñar la *estructura de los casos de uso* a nivel de análisis.
- Identificar nuevamente y diseñar elementos comunes entre casos de uso refinando lo encontrado durante los requerimientos.
- Realizar la composición de casos de uso en la estructura de análisis de elementos para asegurar

su concordancia, ya que todos los elementos definidos en la *estructura de elementos* debe tener su contraparte en la *estructura de casos de uso* y viceversa.

- Definición de casos de uso de Infraestructura y sus rebanadas no específicas de un caso de uso.

**Artefactos agregados al flujo de trabajo:**

- Análisis de la *estructura de elementos*. Está compuesta por clases del análisis organizadas en paquetes y capas. Estas clases del análisis están vacías.
- Análisis de la *estructura de casos de uso*. Está compuesto por rebanadas de casos de uso que añaden contenido a los elementos definidos en la *estructura de análisis de elementos*.

**Trabajadores agregados al flujo de trabajo:**

- Ingeniero de Aspectos.

  Junto con el ingeniero de casos de uso, es responsable de la integridad de una o más realizaciones de casos de uso y debe garantizar que cumplen los requisitos que se esperan de ellos, pero enfocado hacia las rebanadas de casos de uso.

  De la misma forma como establece el PUDS original, suele ser adecuado hacer que el ingeniero de aspectos sea el responsable por la integridad de una o más rebanadas de casos de uso. De esta forma es posible garantizar que sus contenidos sean correctos.

- También los trabajadores definidos en el PUDS deben tener nociones del trabajo con Aspectos. De otra forma sería muy difícil su coordinación.

## 2.3. Flujo de trabajo Diseño orientado a aspectos

En el diseño se modela el sistema y se encuentra su forma (incluida la arquitectura) para que sea capaz de admitir todos los requisitos, tanto funcionales como no funcionales y otras restricciones analizados hasta el momento. Es esencial el modelo del análisis para realizar este flujo de trabajo. Dicho modelo representa una comprensión detallada de los requisitos. Y lo que es más importante, impone una estructura del sistema que es necesario hacer el mejor esfuerzo por conservar lo más fielmente posible al dar forma al sistema [Jacobson2000].

Propósitos:

- Adquirir una comprensión en profundidad de lo relacionado con los requerimientos no funcionales y restricciones relacionadas con los lenguajes de programación, componentes reutilizables, sistemas operativos, tecnologías de distribución y concurrencia, tecnologías de interfaz de usuario, tecnologías de gestión de transacciones, etc. [Jacobson2000].
- Crear un punto de partida para actividades de implementación subsiguientes capturando los requisitos o subsistemas individuales, interfaces y clases [Jacobson2000].
- Ser capaces de descomponer los trabajos de implementación en partes más manejables que puedan ser llevadas a cabo por diferentes personas, teniendo en cuenta la posible concurrencia [Jacobson2000].
    - Ser capaces de visualizar y reflexionar sobre el diseño utilizando una notación común [Jacobson2000].
    - Crear una abstracción sin costuras de la implementación del sistema, en el sentido de que la implementación es un refinamiento directo del diseño que rellena lo existente sin cambiar la estructura. Esto permite la utilización de tecnologías como la generación de código y la ingeniería de ida y vuelta entre el diseño y la implementación [Jacobson2000].
    - Capturar las interfaces entre subsistemas pronto en el ciclo de vida del software [Jacobson2000].

Para cumplir estas metas, el Proceso Unificado describe el siguiente flujo de trabajo, artefactos y trabajadores [Jacobson2000]:

Flujo de trabajo:

- Diseño de la arquitectura
- Diseñar un caso de uso
- Diseñar una clase
- Diseñar un subsistema

Artefactos:

- Modelo de diseño
- Clase del diseño
- Realización de caso de uso-diseño
- Subsistema del diseño
- Interfaz
- Descripción de la arquitectura (vista del modelo de diseño)
- Modelo de despliegue
- Descripción de la arquitectura (vista del modelo de despliegue)

Trabajadores:

- Arquitecto
- Ingeniero de casos de uso
- Ingeniero de componentes

### 2.3.1. Modificaciones (Diseño)

El modelo de diseño, así como el del análisis, están compuestos por dos estructuras: La *estructura de elementos* y la *estructura de casos de uso*. La primera define los elementos, mientras que la segunda define el contenido de dichos elementos. Si se hiciera una comparación con la industria de la construcción, la *estructura de los elementos* sería equivalente a los planos estructurales de una edificación, mientras que la *estructura de casos de uso* sería equivalente al conjunto de trabajos que realiza cada especialista sobre los planos estructurales de una edificación para agregar servicios, como la instalación eléctrica; la instalación de cableado de voz y datos; las tuberías de agua caliente, fría y desagüe; etc.

El modelo del análisis es utilizado para definir la estructura a un alto nivel, y el modelo de diseño para refinar esta estructura e incorporar detalles [Jacobson2004].

Se toma cada rebanada de casos de uso en el análisis y se refina para el diseño. Se incorporan elementos de la interfaz de usuario y elementos para soportar la distribución, persistencia, y demás necesidades de infraestructura [Jacobson2004].

Se mantienen separados los elementos de las clases de interfaz, control y entidad tanto en la estructura de elementos como en la estructura de casos de uso. En la estructura de elementos se tienen subpaquetes para estos elementos. De forma similar, en la estructura de casos de uso se tienen sub rebanadas de casos de uso [Jacobson2004].

**Actividades agregadas al flujo de trabajo:**

- Diseñar la *estructura de los elementos* del diseño en capas, paquetes y subsistemas de acuerdo a la arquitectura.

- Diseñar la *estructura de casos de uso.*
- Identificar nuevamente y diseñar elementos comunes.

  Durante la realización de un caso de uso, se obtiene una descripción del flujo de eventos textual, diagramas de clases y diagramas de interacción entre objetos del diseño. Debido a esto, es el lugar perfecto para identificar elementos comunes entre realizaciones de diferentes casos de uso [Jacobson2004].

  - Definir nuevas rebanadas de casos de uso

    Identificación de elementos comunes entre realizaciones de diferentes casos de uso para poder definir nuevas rebanadas, casos de uso de extensión, y clases de extensión.

    Estas nuevas rebanadas de casos de uso entran en la categoría de las rebanadas no específicas de un caso de uso.

  - Definir interfaces y puntos de extensión entre componentes y los casos de uso de extensión
- Realizar la composición de casos de uso en la estructura de elementos del diseño

  El objetivo de este flujo de trabajo es asegurarse que todos los elementos del modelo de casos de uso tienen su contraparte estructural en el modelo de elementos del diseño y no hay inconsistencias.

**Artefactos agregados al flujo de trabajo:**

Para describir los elementos específicos de la plataforma, el modelo de diseño debe contener más conceptos y, por lo tanto, más estructuras que el modelo del análisis. Este modelo contiene las siguientes estructuras:

- Estructura de distribución.

  Ahora es indispensable que la estructura de distribución contenga nodos de procesamiento que se utilizan en el sistema, los nodos con los que el sistema necesita una interfaz, y las ligas entre estos nodos [Jacobson2004].
- Estructura de proceso.

  La estructura de proceso está compuesta por elementos activos como procesos e hilos. Estos procesos e hilos se ejecutan dentro de nodos en la estructura de distribución [Jacobson2004].

  Este modelo es detallado solamente en el caso de ser requerido.
- Estructura de Elementos del Diseño.

  La estructura de Elementos del Diseño está compuesta por clases de diseño organizadas en capas, subsistemas y paquetes [Jacobson2004].
- Estructura de Casos de Uso del Diseño

  Rebanadas de casos de uso, aspectos, clases de extensión, etc. que corre ortogonalmente respecto a la estructura de elementos del diseño [Jacobson2004].
- Diagrama de composición de paquetes

  Concordancia de las dos estructuras en cada paquete.

**Trabajadores agregados al flujo de trabajo:**

- Ingeniero de Aspectos.

  Junto con el ingeniero de casos de uso, es responsable de la integridad de una o más realizaciones de casos de uso en el diseño, y debe garantizar que cumplen los requisitos que se esperan de ellos, pero enfocado hacia las rebanadas de casos de uso.

  De la misma forma como establece el PUDS original, suele ser adecuado hacer que el ingeniero de aspectos sea el responsable por la integridad de uno o más rebanadas de casos de uso. De esta forma es posible garantizar que sus contenidos sean correctos. De igual forma, es recomendable hacer que el ingeniero de aspectos, que es responsable de ciertos aspectos en el modelo, también sea responsable de sus elementos al momento de implementarlos. De esta forma se garantiza un desarrollo uniforme y sin discontinuidades.

- También los trabajadores definidos en el PUDS deben tener nociones del trabajo con Aspectos. De otra forma sería muy difícil su coordinación.

## 2.4. Flujo de trabajo de Implementación orientado a aspectos

En la implementación se empieza con el resultado del diseño y se implementa el sistema en términos de componentes, es decir, archivos de código fuente, *scripts*, archivos de código binario, ejecutables y similares [Jacobson2000].

Propósitos:

- Planificar las integraciones de sistema necesarias en cada iteración. Se sigue un enfoque incremental, lo que da lugar a un sistema que se implementa en una sucesión de pasos pequeños y manejables [Jacobson2000].
- Se distribuye el sistema asignando componentes ejecutables a nodos en el diagrama de distribución. Esto se basa fundamentalmente en las clases activas encontradas durante el diseño [Jacobson2000].
- Se implementan las clases y subsistemas encontrados durante el diseño [Jacobson2000].
- Se prueban los componentes individualmente, y a continuación se integran compilándolos y enlazándolos en uno o más ejecutables, antes de ser enviados para ser integrados y llevar a cabo las comprobaciones de sistema [Jacobson2000].

Para cumplir estas metas, el Proceso Unificado describe el siguiente flujo de trabajo, artefactos y trabajadores [Jacobson2000]:

Flujo de trabajo:

- Implementación de la arquitectura
- Integrar el sistema
- Implementar un subsistema
- Implementar una clase
- Realizar prueba de unidad

Artefactos:

- Modelo de implementación
- Componente
- Subsistema de la implementación
- Interfaz
- Descripción de la arquitectura (vista del modelo de implementación)
- Plan de integración de construcciones

Trabajadores:

- Arquitecto
- Ingeniero de componentes
- Integrador de sistemas.

### 2.4.1. Modificaciones (Implementación)

En realidad el trabajo de implementación no es diferente cuando se trata de la POA en cuanto al flujo de trabajo y artefactos diferentes. En cada implementación, dependiendo de la plataforma seleccionada, hay diferentes necesidades de desarrollo, y el PUDS no hace una mención específica de cada uno de éstos, sino que en cada caso es necesario hacerle adecuaciones.

La forma de desarrollar un sistema es por medio del refinamiento gradual de las rebanadas de casos de uso a través de los diferentes modelos. La estructura de un modelo de casos de uso es esencialmente preservada a lo largo de todos los modelos posteriores por medio de las rebanadas de casos de uso de dichos modelos [Jacobson2004].

Generalmente se trabaja en el modelo de diseño y en el de implementación en conjunto. Incluso tratándose de modelos diferentes, se realizan cambios en ambos en una misma actividad. Se realizan pruebas de manera conjunta con el diseño y la implementación. El diseño no está completo hasta que se conoce exactamente cómo probar los elementos del diseño, y la implementación no está completa hasta que se pasan correctamente dichas pruebas [Jacobson2004].

Sin embargo, siendo meticulosos se puede ver que es necesario agregar lo siguiente:

**Actividades agregadas al flujo de trabajo:**

- Implementar un Aspecto
- Especificar punto(s) de corte de un Aspecto
- En caso de haber optado por realizar el tejido en tiempo de compilación, es necesario realizar el tejido.
- Composición y configuración de Módulos de caso de uso.

  Ya que se trabaja en rebanadas de casos de uso para varios modelos en forma conjunta, tiene sentido agrupar todas estas rebanadas en un mismo paquete. Este paquete es llamado “módulo de caso de uso”. Un módulo de caso de uso contiene una rebanada de especificación de caso de uso (del modelo de casos de uso), una rebanada de análisis (del modelo de análisis), una rebanada de diseño (del modelo de diseño), y una rebanada de implementación (del modelo de implementación), como se muestra en la figura 2.4.1.

  Adicionalmente, hay rebanadas del modelo de diseño y del modelo de implementación con el propósito de realizar las pruebas. Cada rebanada de prueba contiene clases y clases de extensión necesarias para conducir las pruebas en una rebanada en particular, en un escenario de pruebas en particular.

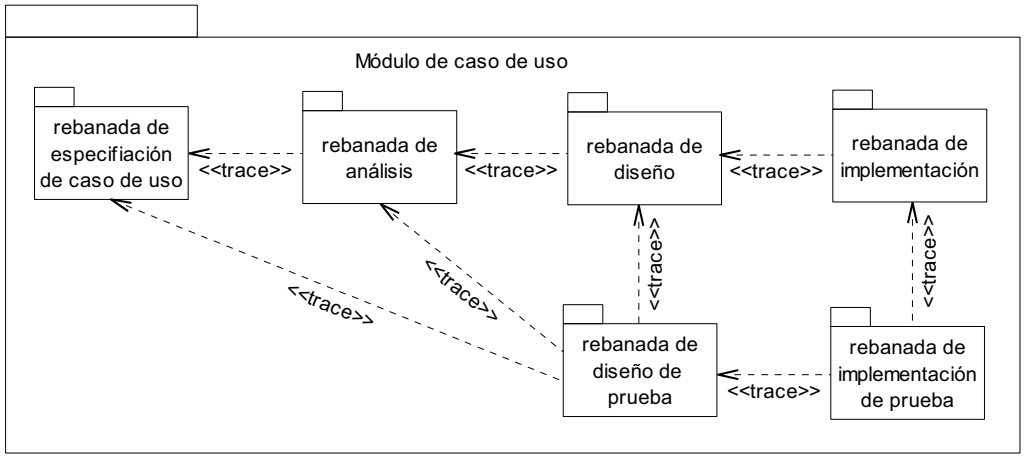



*Figura 2.4.1: Rebanadas contenidas en un módulo de caso de uso*

Las dependencias `<<trace>>`, en la figura 2.4.1, indican que hay ciertas reglas para derivar elementos del modelo previo a un elemento del modelo siguiente. Frecuentemente, dichas reglas se describen informalmente por el arquitecto como lineamientos de desarrollo y principios a ser acatados por el equipo de desarrollo. Sin embargo, es recomendable formalizar dichos lineamientos [Jacobson2004].

**Artefactos agregados al flujo de trabajo:**

- Módulos de casos de uso

**Trabajadores agregados al flujo de trabajo:**

- Implementador de Aspectos.

  El implementador de Aspectos define y mantiene el código fuente de una o varias rebanadas de casos de uso, garantizando que cada una implementa la funcionalidad correcta.

- También a los actores definidos en el PUDS ahora les compete estar al tanto del trabajo con Aspectos, ya que todo el equipo de implementación debe compartir los conceptos y manejo de este paradigma.

# Capítulo 3. Programación Orientada a Aspectos (POA) y Arquitectura Dirigida por Modelos (MDA)

## *3.1. Introducción*

La especificación existente respecto a MDA proporciona la posibilidad de crear herramientas muy poderosas (ver la sección 1.4 Arquitectura Dirigida por Modelos (MDA)) que permitan la generación de código fuente directo desde modelos. Estas herramientas son llamadas *marcos MDA*.

A pesar de que POA y MDA fueron desarrollados desde puntos de vista muy diferentes: programación contra modelado, hay una gran base en común entre ambas consistente en que se complementan y pueden ser utilizadas juntas exitosamente [AOP_y_MDA_1].

Respecto a la separación de intereses, ambas tecnologías tienen un objetivo similar: tratan de atacar la complejidad de los sistemas de TI por medio de la separación de intereses. Por ejemplo, en POA el enfoque está basado en la separación de las *características distribuidas* de la lógica básica del sistema. En MDA el enfoque está basado en separar *características tecnológicas* de la lógica de la aplicación [AOP_y_MDA_1].

Actualmente, hasta donde fue posible investigar, los desarrollos en el área de la Arquitectura Dirigida por Modelos están enfocados a la generación de código orientado a objetos. Y los desarrollos en el área de la tecnología orientada a aspectos ni siquiera tocan el tema de generación de código en forma automática desde un modelo.

En la utilización de marcos MDA, todavía está el problema del código mezclado y las características dispersas en el código generado. Tanto en el modelo, como en el código fuente generado, se tienen clases y paquetes con funcionalidades mezcladas y hay funcionalidades dispersas a lo largo de la aplicación.

La separación de intereses tiene múltiples ventajas, como se ha mencionado anteriormente, por lo que reunir las ventajas de la POA con la utilización de marcos MDA representa un avance importante hacia el rápido desarrollo de mejores aplicaciones que sean fácilmente mantenibles y extensibles.

No se ha encontrado evidencia clara de la utilización de algún MDA para generar aplicaciones POA. Por lo que el aporte de este trabajo ha sido entender a fondo lo que es el desarrollo con MDA y el desarrollo OA para mostrar los pasos que son necesarios para aprovechar las potencialidades del MDA que simplifica la parte de construcción tecnológica, pero generando una aplicación con todas las ventajas y características del DSOA.

No se pretende abarcar la problemática en su totalidad, sino realizar un primer acercamiento. Identificar hasta qué punto es posible aprovechar las herramientas actualmente existentes y obtener una idea de cambios necesarios.

A simple vista, no debe ser difícil extender un marco MDA para que maneje POA. Como fue mencionado previamente, es ámpliamente utilizado UML en este tipo de marcos arquitectónicos. Una de las razones por las cuales es utilizado UML, es debido a sus mecanismos de extensibilidad.

Los mecanismos de extensibilidad de UML incluyen los estereotipos, los valores etiquetados y las restricciones. (ver Apéndice A. Glosario.)

Los marcos MDA hacen un uso extensivo de paquetes, estereotipos y valores etiquetados, presentes en el modelo, para dirigir la generación de código fuente. Por lo tanto, lo que se requiere es hacer que el procesador del marco MDA pueda comprender nuevos estereotipos, valores etiquetados y una nueva organización de paquetes.

Tal como fue establecido en la sección 2.3.1 (Modificaciones (Diseño)), en el desarrollo de aplicaciones Orientadas a Aspectos es necesario utilizar dos estructuras relacionadas entre sí: la *estructura de elementos* y la *estructura de casos de uso*.

La propuesta de este trabajo es que por medio de la utilización de un marco MDA sin adecuar para POA, se genere la *estructura de elementos*. Dicha estructura fungirá como base para agregar funcionalidades distribuidas contenidas en la *estructura de casos de uso*. Esta estructura, por lo pronto, se debería implementar en forma manual. Este procedimiento, resulta de gran utilidad para poder, como se dijo unos párrafos antes, identificar claramente las modificaciones que sería necesario hacer a dicho marco MDA (o a otro) para que directamente se pudieran generar aplicaciones orientadas a aspectos desde uno o varios modelos.

### 3.1.1. Problemas de integración

A simple vista, se pueden identificar algunos problemas que será necesario resolver al momento de utilizar un marco MDA junto con POA. Va más allá del alcance de este trabajo encontrar y comprobar alguna solución a éstos y otros problemas que se descubran sobre la marcha.

Por ejemplo:

- Interfaz de usuario.

  Es difícil lograr que diferentes *aspectos* se puedan presentar en una misma interfaz de usuario en forma automática. Generalmente la interfaz requiere una especificación explícita de cada uno de los elementos que la conforman. Sin embargo, al agregar o modificar comportamiento por medio de la tecnología de POA es posible que las interfaces existentes ya no se adecuen a la nueva realidad.

- Precedencia

  Sería necesario encontrar la manera de especificar, desde el modelo, la precedencia de diferentes rebanadas de casos de uso al momento de realizar la composición.

- Persistencia

  Los diferentes atributos agregados a cada clase por medio de cada rebanada de caso de uso debe ser *mapeada* a un atributo de una entidad de la base de datos. Si se utiliza un marco de trabajo encargado de la persistencia, será necesario encontrar la manera de ajustar su configuración de acuerdo a una composición de rebanadas de casos de uso determinada. Al momento de utilizar un marco MDA Orientado a Aspectos, una posible solución sería simplemente aprovechar su capacidad de generar código, para crear adecuadamente la configuración del marco de trabajo de persistencia.

## 3.2. AndroMDA es un marco MDA

AndroMDA (pronunciado "andromeda") es un marco generador que cumple con el paradigma establecido por la Arquitectura Dirigida por Modelos. Modelos generados con herramientas UML serán transformados en componentes desplegables para una plataforma como J2EE, Spring o .NET. A diferencia de otros juegos de herramientas para MDA, AndroMDA ya viene con un conjunto prefabricado de cartuchos (*cartridges, en inglés*) que apuntan a tecnologías y herramientas de desarrollo actuales como Axis, JBPM, Struts, JSF, Spring y Hibernate. AndroMDA también contiene un juego de herramientas para construir cartuchos a la medida, o adaptar los existentes [AndroMDA_1].

### 3.2.1. Características principales

Actualmente viene con las siguientes características [AndroMDA_1]:

- Diseño modular. Todos los bloques de construcción de AndroMDA se pueden incluir o no, así como pueden ser modificados para cumplir las necesidades del desarrollador.
- Soporte de múltiples herramientas UML. Por ejemplo: ArgoUML, MagicDraw, Poseidon, Enterprise Architect, entre otras.
- Viene con el metamodelo completo para UML 1.4. El manejo de UML2 todavía está en desarrollo. También es posible definir un metamodelo propio en MOF XMI, y generar código a partir de modelos basados en él.
- Valida los modelos de entrada. Utilizando restricciones OCL relacionadas con las clases del metamodelo. Viene con restricciones pre-configuradas que protegen de los errores de modelado más comunes. También es posible agregar restricciones según sea necesario.
- Transformación de modelo a modelo que ayuda a aumentar el nivel de abstracción. Actualmente se pueden definir transformaciones en lenguaje Java, en la siguiente liberación grande del producto, en cualquier lenguaje de transformación similar a QVT (QVT es un estándar de la OMG para la definición de transformaciones de modelos).
- Puede generar cualquier tipo de texto de salida utilizando plantillas (código fuente, *scripts* de base de datos, páginas web, archivos de configuración, etc.).
- Basado en motores de plantillas bien conocidos. Actualmente admite Velocity y FreeMaker.
- Cartuchos listos para usar, como: Spring, EJB 2 / 3, Webservices, Hibernate, Struts, JSF, Java, XSD
- Soporte Técnico prácticamente inmediato a cargo de miembros del equipo en todo el mundo.

Es una muy buena opción de marco MDA para desarrollar una integración con POA.

### 3.2.2. Visión general del sistema AndroMDA

Hay dos componentes principales que se utilizan en el sistema:

- El motor de generación de código.
- El sistema de administración y constructor de proyectos llamado Maven, de Apache.

El generador de código AndroMDA es, en realidad, un motor genérico de generación de código. Es una plataforma que contiene módulos de codificación (llamados cartuchos) que realizan la generación de código en sí. Un cartucho en AndroMDA es una colección de plantillas de código fuente y clases de apoyo (llamadas "*Metafacades*") empacadas en un archivo .JAR.

En realidad Maven es un componente opcional. El motor AndroMDA podría ser ejecutado directo desde la línea de comando, desde un ambiente de desarrollo o desde un script (como Ant). Sin embargo ya existen múltiples *plug-ins* que facilitan de gran manera el desarrollo con AndroMDA.[1]

### 3.2.3. Ciclo de desarrollo con AndroMDA

A continuación se presenta un ciclo típico de desarrollo (Java) utilizando AndroMDA:

1. Se crea un proyecto con un comando.
2. Se configura el proyecto generado para que se ajuste a las necesidades específicas del desarrollador.
3. Editar un archivo generado en forma automática, en donde se desarrolla el modelo en UML de la aplicación.
4. Generar el código fuente. Existen diferentes comandos para instalar la aplicación en el servidor de aplicaciones y para generar el esquema de la base de datos en forma automática, por nombrar algunos.
5. Codificar manualmente aquellos métodos que lo requieren.
6. Compilar y probar la aplicación
7. Conforme evoluciona la aplicación, repetir desde el paso 3, refinando el modelo UML.

1 Es posible encontrar una visión general de lo que es Maven y de lo que es capaz de hacer en http://www.devx.com/java/Article/17204

### 3.2.4. Arquitectura de las aplicaciones

La arquitectura de sistemas utilizada en la actualidad está basada en el modelo de capas. Los componentes ubicados en capas superiores en la pila, hacen uso de componentes ubicados en capas inferiores. Los componentes en cierta capa, generalmente utilizan la funcionalidad de otros componentes en esa misma capa, o de las capas inferiores [AndroMDA_2]. (Ver figura 3.2.1) Las capas más comunes son:

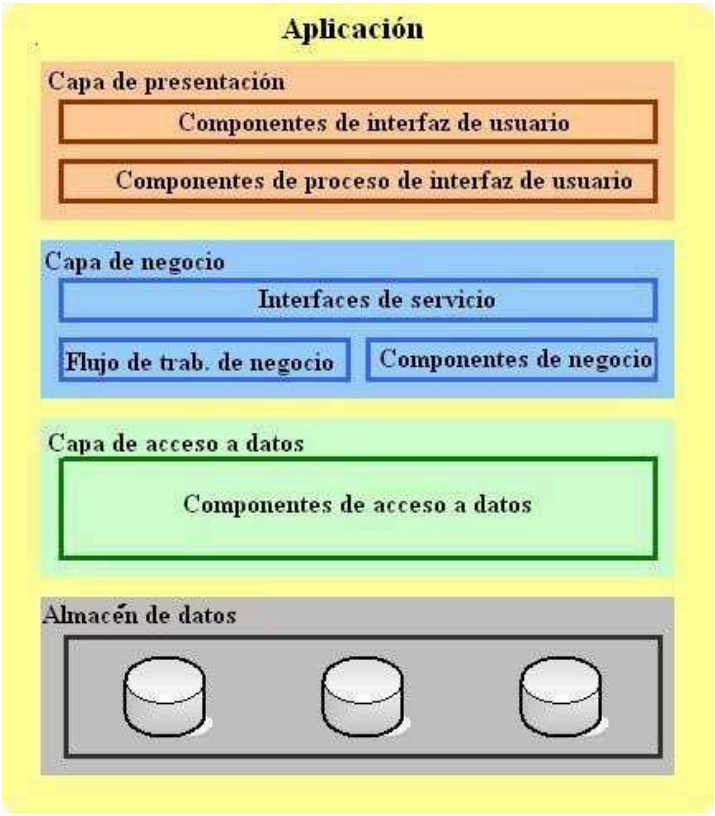



*Figura 3.2.1: Arquitectura recomendada*

**Capa de presentación:**

La capa de presentación contiene componentes necesarios para interactuar con el usuario de la aplicación

**Capa de negocio:**

La capa de negocio encapsula el corazón de la funcionalidad de negocio de la aplicación. Funciones simples de negocio pueden ser implementadas con componentes que no almacenan el estado (*stateless*), mientras que elementos complicados, o transacciones que duran mucho tiempo, pueden ser implementados con flujos de trabajo que si almacenan el estado (stateful). Los componentes de negocio, generalmente son conectados con la capa de presentación por medio de interfaces de servicio, que actúan como fachada para ocultar la complejidad de la lógica de negocio. Esto generalmente se conoce como Arquitectura Orientada a Servicios (SOA por sus siglas en inglés)

**Capa de acceso a datos:**

La capa de acceso a datos provee un API simple para manipular y acceder a datos. Los componentes en esta capa, abstraen la semántica de la tecnología de acceso a datos subyacente, con lo que se permite a la capa de negocios enfocarse a la lógica de negocio. Cada componente típicamente provee métodos para realizar las operaciones de crear, leer, modificar y borrar entidades de negocios.

**Almacén de Datos:**

Las aplicaciones de negocios guardan su información en uno o varios almacenes de información. Bases de datos y sistemas de archivos son dos tipos muy comunes de almacenes de datos [AndroMDA_2].

### *Arquitectura de aplicaciones generadas utilizando AndroMDA*

Específicamente la arquitectura generada por AndroMDA es presentada a continuación [AndroMDA_2]. (Ver figura 3.2.2)

AndroMDA toma como entrada un modelo de negocio especificado en UML y genera porciones significativas de las capas necesarias para construir una aplicación en Java.


*Figura 3.2.2: Arquitectura generada con AndroMDA*

**Capa de presentación:**

AndroMDA proporciona dos opciones tecnológicas para construir aplicaciones con capa de presentación WEB: Struts (http://struts.apache.org/) y JSF (http://java.sun.com/javaee/javaserverfaces/). Acepta diagramas de Actividad como entrada para especificar flujos de navegabilidad, y genera componentes conforme al marco de trabajo Struts o conforme al marco de trabajo JSF, según se haya elegido.

**Capa de negocio:**

La capa de negocio generada por AndroMDA consiste principalmente en servicios configurados utilizando el marco de trabajo Spring (http://www.springframework.org/). Estos servicios son implementados en forma manual en métodos vacíos generados por AndroMDA, en donde la lógica de negocio puede ser definida. A estos servicios generados, opcionalmente se les pueden agregar interfaces EJB (Enterprise Java Beans). En caso de agregar EJB's al modelo de negocio, será necesario desplegar la aplicación en un contenedor de EJB's, como JBoss (http://labs.jboss.com/portal/jbossas/), por ejemplo. Los servicios también pueden ser expuestos como "Servicios WEB" con lo que se provee una forma independiente de la plataforma para que los clientes accedan a su funcionalidad. Incluso, AndroMDA es capaz de generar procesos de negocio y flujos de trabajo para el motor de flujos de trabajo jBPM (http://www.jboss.com/products/jbpm), que es parte de la línea de productos JBoss.

**Capa de acceso a datos:**

AndroMDA ofrece la posibilidad de generar la capa de acceso a datos utilizando Hibernate. Hibernate es la herramienta de mapeo objeto-relacional más utilizada actualmente. (http://www.hibernate.org/) AndroMDA hace esto generando Objetos de Acceso a Datos (DAOs, por sus siglas en inglés) para las entidades definidas en el modelo UML. Estos objetos de acceso a datos utilizan el API de Hibernate para convertir registros de la base de datos en objetos y viceversa. AndroMDA también permite utilizar

EJB3/Seam (http://www.jboss.com/products/seam) para la capa de acceso a datos.

**Almacén de Datos:**

Ya que las aplicaciones generadas con AndroMDA utilizan Hibernate para acceder a datos, es posible utilizar cualquiera de las bases de datos admitidas por este marco de trabajo, como hypersonic, mysql, oracle, db2, informix, mssql, pointbase, postgres, sybase, sabdb, progress o derby.

**Propagación de datos entre las diferentes Capas**

Es importante comprender cómo se propagan los datos entre las diferentes capas de una aplicación. A continuación, se va a explicar este proceso de abajo hacia arriba.

Las bases de datos relacionales guardan datos como registros en tablas. Las capas de acceso a datos recuperan estos registros de la base de datos y los transforman en objetos que representan entidades en el dominio de negocio. Debido a esto, los objetos mencionados son llamados "*entidades de negocios*".

La capa de acceso a datos pasa las entidades de negocios a la capa de negocios, la cual lleva a cabo la lógica de negocios utilizando estas entidades.

El último aspecto a discutir es la propagación de datos entre la capa de negocios y la capa de presentación. Existen en realidad dos escuelas de pensamiento sobre este asunto. Algunas personas recomiendan que la capa de presentación debe tener acceso directo a las entidades de negocios. Otros recomiendan lo contrario, por ejemplo, que las entidades de negocios no deben ser accesibles desde la capa de presentación y que la capa de negocios debe empacar la información necesaria dentro de los llamados "*Value Objects*" y transferirlos a la capa de presentación. Estos dos enfoques presentan ventajas y desventajas.

El primer enfoque (solamente entidades, sin "*Value Objects*") es más sencillo de implementar. No es necesario crear "*Value Objects*" o escribir ningún código para transferir la información entre las entidades y éstos. De hecho, para las aplicaciones sencillas y pequeñas donde la capa de presentación y la capa de servicio corren en la misma máquina, esta aplicación probablemente trabaje suficientemente bien. Sin embargo, para aplicaciones más grandes y complejas este enfoque no es el adecuado. Debido a las siguientes causas:

- La lógica de negocios no está contenida en la capa de negocios. Es tentador manipular libremente las entidades en la capa de presentación y así llevar la lógica de negocios a diversos lugares lo cual creará una pesadilla para el mantenimiento. En caso de que existan múltiples modos de que los usuarios accesen un servicio, la lógica de negocios deberá ser duplicada en todos estos modos de acceso. Además, no existe una protección para evitar que la capa de presentación corrompa las entidades, tanto intencional como accidentalmente.
- Cuando la capa de presentación corre en una máquina diferente (como en el caso de un "*cliente rico*" (*rich client*), es muy ineficiente serializar todo un conjunto de entidades y mandarlo a través de la red. Por ejemplo, si se quiere mostrar una lista de órdenes al usuario, en realidad no se necesita transferir a la aplicación del cliente todos los detalles de cada orden. Probablemente todo lo que necesita es el número de orden, la fecha y la cantidad total de cada orden. Más tarde, si el usuario desea ver los detalles de una orden específica, siempre se puede serializar esa

orden completa y mandarla a través de la red.

- Pasar entidades reales al cliente puede presentar un riesgo de seguridad. No es conveniente que la aplicación del cliente tenga acceso a información de los salarios dentro del objeto Empleado o al margen de ganancias dentro del objeto Orden.

Los "V*alue Objects"* proveen una solución para todos estos problemas. Si van a requerir que se escriba un poco de código extra, pero va a recibir una capa de negocios muy robusta que se comunica con la capa de presentación en forma efectiva. Se puede pensar que un "*Value Object*" es una vista controlada de una o más entidades que es relevante para la aplicación del cliente. AndroMDA provee una funcionalidad básica para la traducción automática entre entidades y "*Value Objects*".

## *3.3. AndroMDA y POA*

AndroMDA genera código utilizando Spring, y viene preparado para el despliegue automático al contenedor de EJBs JBoss. Ambos tienen implementación de ambientes de programación orientada a Aspectos [Pawlak2005].

El servidor de aplicaciones JBoss, en conjunto con el marco de trabajo JBoss-AOP ofrecen la posibilidad de realizar el tejido de aspectos durante un pre-proceso, durante la carga así como en tiempo de ejecución (*Dynamic AOP*) [JBoss_AOP_1].

Spring, por su parte, contiene Spring AOP, que pretende simplificar el desarrollo de aplicaciones J2EE. El manejo de POA que provee, es limitado. Spring AOP utiliza un API desarrollado por el "AOP Alliance" que es un esfuerzo de estandarización de intercepción genérica para Java. Está diseñado para ser utilizado fácilmente en muchos ambientes de J2EE y servidores de aplicaciones, como WebLogic, Tomcat, JBoss, Resin y Jetty [Pawlak2005].

JBoss AOP se considera la mejor opción. Este marco permite crear puntos de corte para cualquier clase de la aplicación generada con AndroMDA, por el hecho de estar integrada en el servidor de Aplicaciones necesario para "publicar" dicha aplicación.

Cuando se utilizan los cartuchos para generar aplicaciones J2EE, el código generado por AndroMDA está compuesto por diferentes módulos. Es suficiente con agregar un módulo nuevo que contenga la implementación de las rebanadas de casos de uso.

También es necesario empaquetar la aplicación con una estructura diferente a la que genera automáticamente AndroMDA. Sin embargo es fácilmente configurable gracias a que este marco utiliza Maven (el sistema de administración y constructor de proyectos).

Por otro lado, no es necesario modificar gran cosa del ciclo de desarrollo estándar de AndroMDA. Tal como es establecido en la sección 3.2.3 (Ciclo de desarrollo con AndroMDA), es necesario generar código desde el modelo UML, que en el caso de este trabajo es la *estructura de elementos*, y hay que realizar implementación manual, que en el caso de esta tesis son las rebanadas de casos de uso, es decir, la *estructura de casos de uso*.

Debido a que AndroMDA utiliza diferentes cartuchos (*cartridges)* encargados de diversas funcionalidades y contiene herramientas para construir nuevos, como fue mencionado en la sección 3.2, se considera relativamente fácil agregar elementos al modelo UML que sean interpretados por un nuevo cartucho capaz de generar código orientado a aspectos. El trabajo de implementar en forma manual la *estructura de casos de uso*, actualmente, permitirá más fácilmente saber qué modificaciones

hay que hacerle al modelo. Es cierto, también existe la posibilidad de que sea necesario modificar alguno o todos los cartuchos existentes, ya que deberán ignorar algunos elementos que hoy procesan, y/ o generar código en forma diferente. Al final sería posible lograr producir un MDA-OA en donde ambas estructuras, así como el resto de los conceptos para POA, sean mantenidos desde el modelo y a lo largo de todo el proceso de desarrollo.

# Capítulo 4. Construcción de una aplicación.

## *4.1. Introducción*

A continuación se presenta el desarrollo de una aplicación siguiendo las modificaciones al Proceso Unificado y con los conceptos, las nuevas actividades y artefactos propuestos en capítulos anteriores.

El objetivo del desarrollo de esta aplicación es comprobar la utilidad y factibilidad de las propuestas realizadas con respecto al Proceso Unificado, así como la evaluación de las dificultades que es necesario sobrellevar para lograr la integración de las dos nuevas tecnologías descritas anteriormente: AOP y MDA.

No se pretende desarrollar una aplicación completa. Lo novedoso de las propuestas planteadas en cuanto al manejo de un marco MDA, junto con la aplicación de la teoría de Desarrollo de Software Orientado a Aspectos representa un desafío en sí mismo. Dicho planteamiento es un aporte nuevo e importante. Es por esto que es suficiente con demostrar si es posible o no llevarlo a cabo.

En el desarrollo de esta aplicación será evaluado AndroMDA, un marco de trabajo MDA que no está adecuado específicamente para orientación a aspectos, sino para el desarrollo de aplicaciones Orientadas a Objetos. La estrategia es procurar definir la Estructura de Elementos en un modelo UML de forma que AndroMDA sea capaz de recibirlo como entrada para generar la Arquitectura básica de la aplicación, y posteriormente agregar en forma manual la funcionalidad de un caso de uso utilizando la tecnología de desarrollo Orientada a Aspectos. En un futuro, sería posible utilizar lo aprendido durante este proceso para desarrollar modificaciones a cartuchos existentes, para que sea posible definir todo el modelo Orientado a Aspectos en UML y en forma automática se genere código fuente Orientado a Aspectos tan solo para “rellenar” la implementación específica del contenido de los métodos definidos en el modelo.

Se procurará llevar a cabo este desarrollo de manera formal, es decir con toda la documentación que se presentaría al desarrollar una aplicación de la vida real con el Proceso Unificado. Sin embargo, no será incluida toda esta documentación en esta tesis, debido a que puede distraer del objetivo principal.

En el cuerpo de este documento solamente se presenta un caso de uso a modo de muestra.

## 4.2. Requerimientos

### Objetivo

Desarrollar una bolsa de trabajo que le permita a un usuario que busca empleo consultar y postularse a las vacantes disponibles. El sistema apoyará a las Empresas facilitándoles el registro y la modificación de sus vacantes.

### Definición del Problema

Se requiere desarrollar un sistema para una bolsa de trabajo. Con éste, se pretende apoyar a las personas (Postulantes) en su búsqueda de empleo y facilitar a las empresas que solicitan empleados la publicación de sus vacantes. El acceso a esta bolsa de trabajo será únicamente por Internet, a través de una interfaz gráfica fácil de usar.

### Requerimientos No funcionales

#### Software

- Desarrollado con base en el Proceso Unificado.
- Integrar la Orientación a Aspectos y la utilización de un macro MDA

### Actores del Sistema

Un **usuario** (una persona cualquiera) puede:

- Entrar al sistema y consultar las vacantes publicadas, sin importar que no tenga una cuenta.
- Darse de alta (crear su cuenta) como empresa o como postulante, cuando lo ha hecho se convierte en un **postulante** o **empresa**, según sea el caso.

Una **empresa** o **postulante** puede:

- Darse de baja o modificar los datos de su cuenta.
- Autentificarse en el sistema (ingresar con su cuenta).

Una **empresa** podrá dar de alta, dar de baja y modificar sus vacantes. Podrá consultar los datos de los postulantes que han solicitado sus vacantes en la bolsa de trabajo.

Un **postulante** puede consultar las vacantes publicadas y postularse a una vacante.

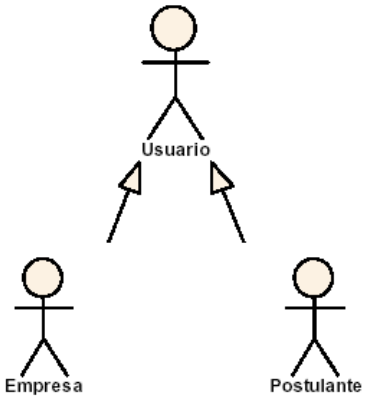



*Figura 4.2.1: Actores*

## *Diagrama general de casos de uso*

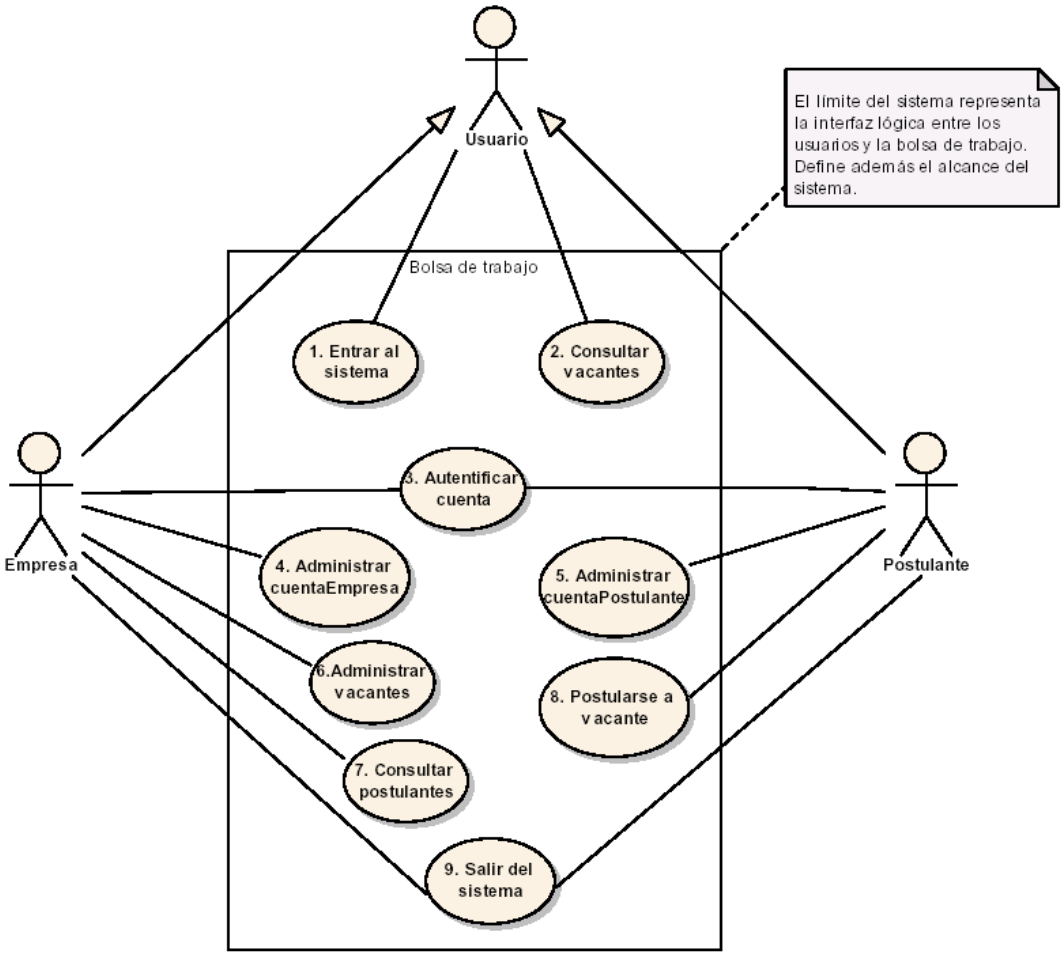



*Figura 4.2.2: Diagrama general de casos de uso*

## *Casos de uso del sistema*

1. Entrar al sistema
2. Consultar vacantes
3. Autentificar cuenta
4. Administrar cuenta Empresa
5. Administrar cuenta Postulante
6. Administrar vacantes
7. Consultar Postulantes
8. Postularse a vacante
9. Salir del sistema.

## *Glosario de términos*

**Bolsa de trabajo**

Es un sistema que facilita la interacción entre las personas que solicitan empleo (*Postulante*) y las *Empresas* que lo ofrecen.

**Cuenta**

Un *Usuario* que se da de alta obtiene una cuenta: clave y contraseña, para ingresar al sistema (para autentificarse).

**Empresa**

Es una organización comercial o industrial que ofrece servicios o productos específicos. Una *Empresa* es un *Usuario* del sistema que tiene una cuenta, puede dar de alta, modificar o dar de baja vacantes. Además puede consultar los postulantes a las vacantes.

**Postulante**

Es una persona que cuenta con habilidades, conocimientos y experiencia en un área específica y que busca un puesto de trabajo. Un *Postulante* es un *Usuario* que se dio de alta, tiene una cuenta y puede administrarla (darse de baja, modificar sus datos), además puede consultar las vacantes disponibles y puede postularse a ellas.

**Usuarios**

Son todas las personas que entran al sistema, pueden o no tener una cuenta, se les permite consultar las vacantes. Si se dan de alta y obtienen una cuenta tienen acceso a otras funcionalidades del sistema, según el tipo de usuario: *Postulante* o *Empresa*.

**Vacantes**

Son las plazas disponibles ofrecidas y administradas por una *Empresa*. Se publican con los siguientes datos: nombres, requisitos, descripción, sueldo, horario de trabajo (por horas, tiempo completo, etc…), solicitud contacto (la persona que está dando de alta la vacante).

**Sesión**

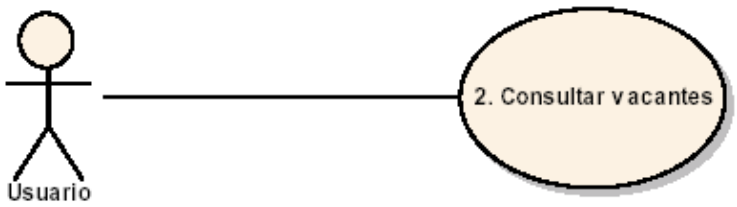
Cuando un usuario se autentifica, una sesión es iniciada. En el transcurso de dicha sesión, el usuario está plenamente identificado para todas sus peticiones y consultas. La sesión termina cuando el usuario expresamente lo solicita, o en forma automática después de un tiempo determinado sin movimientos.

## *Detalle de los casos de uso*

*A continuación se presenta solamente uno de los casos de uso a manera de ejemplo.*

### *2. Consultar vacantes*

*Actor*: Usuario




*Descripción:*

Un *Usuario* (*Postulante* o *Empresa)* puede consultar las vacantes publicadas por todas las empresas. Puede haber varios criterios para mostrar la lista, por empresa, por fecha de publicación, por el tipo de vacante.

*Precondiciones:*

- *Entrar al sistema* y seleccionar la opción "Consultar vacantes".

*Flujo:*

| ACTOR | | SISTEMA | | |
|---|---|---|---|---|
| *Paso* | *Acción* | *Paso* | *Acción* | *Excepción* |
| 1 | Seleccionar la opción "consultar vacantes" | 2 | Se busca la información | E1 |
| | | 3 | Despliega las plazas disponibles según el criterio elegido | E2 |

*Excepciones:*

| ID | TIPO | ACCIÓN |
|---|---|---|
| E1 | Error buscando la información | El sistema muestra un mensaje indicando dicha condición de la cual no se puede recuperar y que se debe informar al encargado. |
| E2 | No fue encontrada información para desplegar con los criterios elegidos. | El sistema muestra un mensaje indicando dicha condición y permite corregir criterios. |

### *Matriz de Casos de Uso contra Conceptos que originan superposición de funciones*

Para identificar elementos que se repiten entre dos o más casos de uso es que se genera este artefacto, como fue establecido en el capítulo *2*, sección *2.1* (*Flujo de trabajo de Requerimientos orientado a aspectos*). Esto será de gran ayuda al momento de identificar rebanadas. Sin embargo, ésta no es su única utilidad. En donde se puede ver un gran apoyo, es al momento de identificar rebanadas no específicas a un caso de uso y/o de infraestructura. Estas rebanadas suelen "descubrirse" mucho más avanzado el desarrollo de la aplicación, y por medio de esta matriz se hace un especial esfuerzo por adelantar este suceso.

| Caso de Uso | Empresa | Postulante | Cuenta | Vacante |
|---|---|---|---|---|
| 1.Entrar al sistema | | | | |
| 2.Consultar vacantes | √ | | | √ |
| 3.Autentificar cuenta | √ | √ | √ | |
| 4.Administrar cuenta Empresa | √ | | √ | |
| 5.Administrar cuenta Postulante | | √ | √ | |
| 6.Administrar vacantes | | | | √ |
| 7.Consultar Postulantes | | √ | | |
| 8.Postularse a vacante | | √ | | √ |
| 9.Salir del sistema. | | | √ | |

### *Clasificación de casos de uso "*pares*"*

- Comparten el concepto Empresa
  - 2.Consultar vacantes
  - 3.Autentificar cuenta
  - 4.Administrar cuenta Empresa

- Comparten el concepto Postulante
  - 3.Autentificar cuenta
  - 5.Administrar cuenta Postulante
  - 7.Consultar Postulantes
  - 8.Postularse a vacante

- Comparten el concepto Cuenta
  - 3.Autentificar cuenta
  - 4.Administrar cuenta Empresa
  - 5.Administrar cuenta Postulante
  - 9.Salir del sistema.

- Comparten el concepto Vacante
  - 2.Consultar vacantes
  - 6.Administrar vacantes
  - 8.Postularse a vacante

## 4.3. Análisis

### Diagramas de clases del análisis

Los diagramas de clases ayudan a representar los elementos involucrados en cada capa del sistema y la relación entre ellos. Para la arquitectura de esta aplicación, según lo establecido en la sección *3.2.4 (Arquitectura de las aplicaciones),* se tiene:

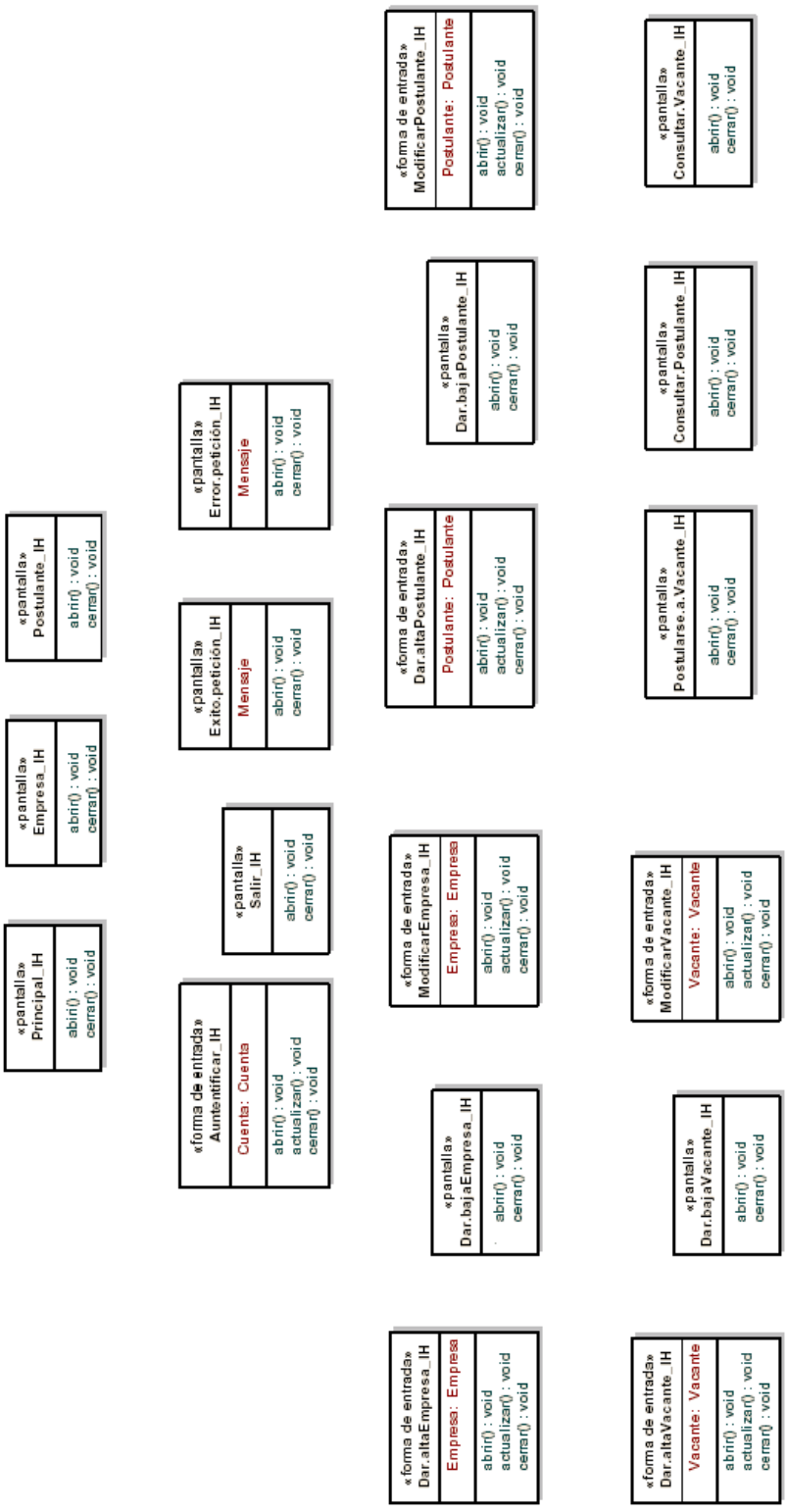
- Diagramas de clases de interfaz, con los componentes de la capa de presentación.
- Diagramas de clases de control, con los componentes de la capa de negocio.
- Diagramas de clases de entidad, con los componentes de la capa de acceso a datos.

No son agregados diagramas de clases para la capa de almacén de datos, puesto que dicha implementación se puede llevar a cabo con diferentes sistemas de almacenamiento de información y lo importante, en realidad, es modelar los elementos que es necesario almacenar, sin importar exactamente cómo.

#### Clases de interfaz (capa de presentación)

Una clase de interfaz representa una <<pantalla>> o una <<forma de entrada>> que solicita información al usuario y permite la interacción del usuario con el sistema.

A continuación se incluyen las clases, una por cada pantalla, por cada funcionalidad importante (las definidas en los casos de uso) y por cada actor o sistema externo. Los métodos de las pantallas son: abrir, cerrar y actualizar, sus atributos son iguales a los de las clases correspondientes en la capa de control.




Hay pantallas para el usuario, otras para las empresas y otras para los postulantes.

*Figura 4.3.1: Diagrama de Clases de Interfaz*

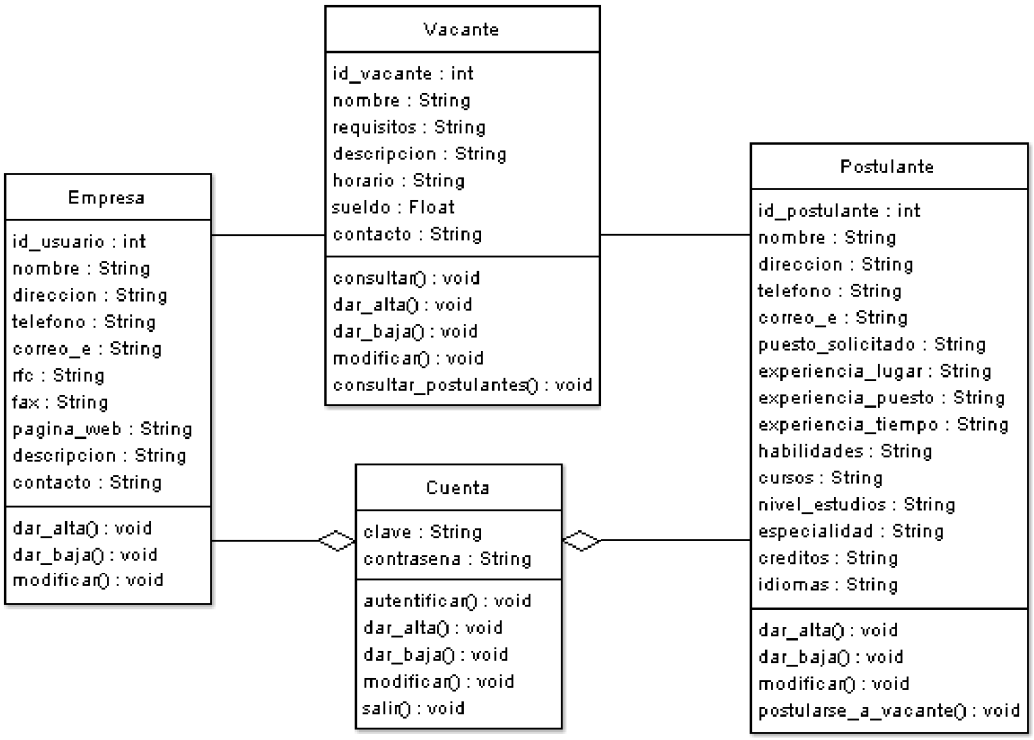
### *Clases de control (capa de negocio)*

Las clases de control encapsulan el comportamiento y manejan el control del flujo del sistema.

Un criterio para determinar las clases de control es identificar:

- Participantes.
  - Usuario: Empresa o Postulante.
- Transacciones.
  - Ingresar al sistema.
  - Salir del sistema.
  - Administrar vacantes (altas, bajas, cambios).
  - Dar de alta usuario.
  - Consultar vacantes.
  - Consultar postulantes.
- Elementos involucrados.
  - Vacantes, Cuenta.

El diagrama de clases de control, si ésta fuera una una aplicación orientada a objetos, sería:

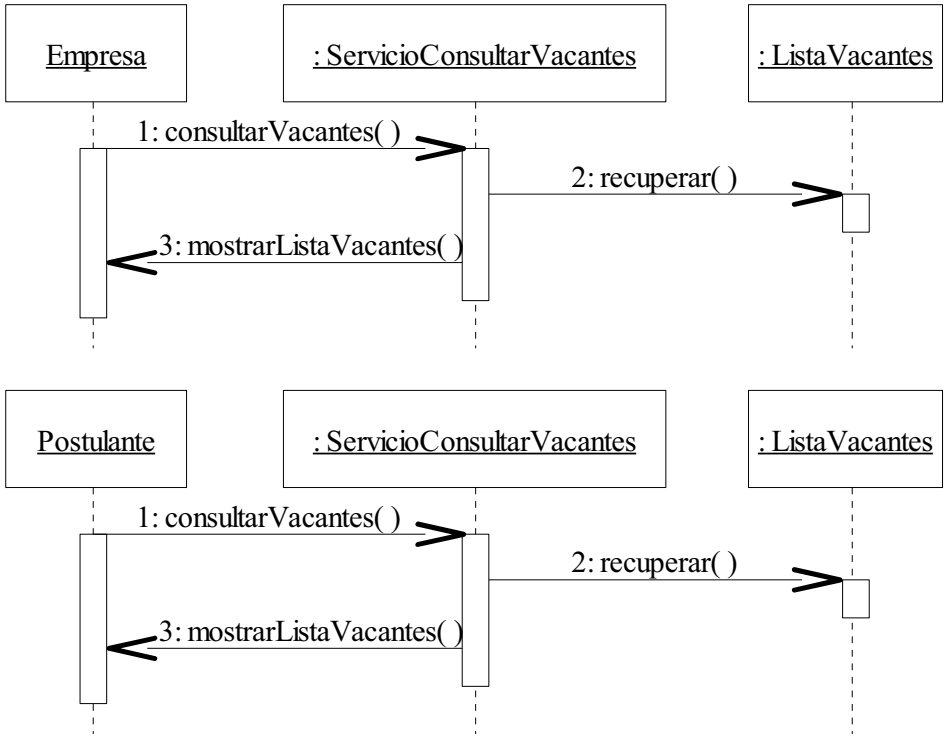



Sin embargo, es necesario analizar los conceptos comunes y comenzar con la definición de rebanadas de casos de uso desde el flujo de trabajo del Análisis. Por otro lado, también en el análisis es necesario definir la *Estructura de Elementos del análisis* y la *Estructura de Casos de Uso del análisis.*

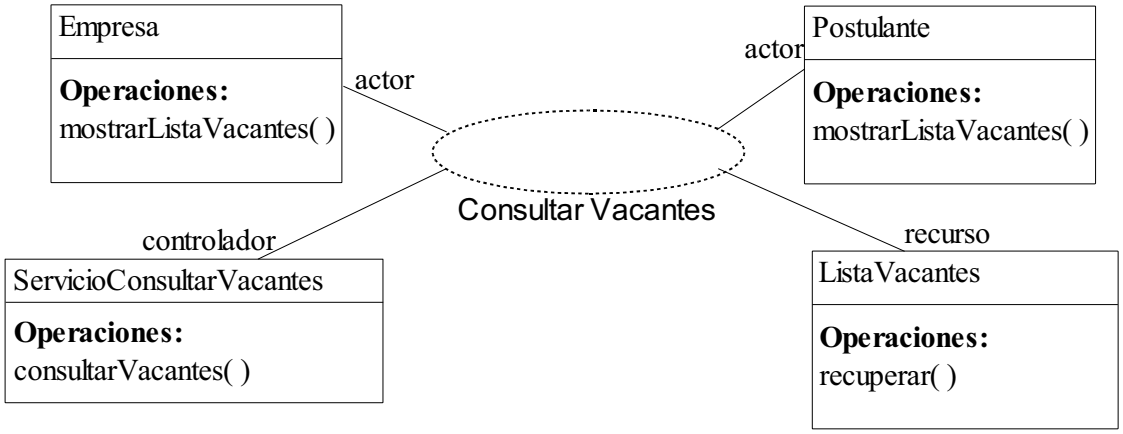
**Realización de algunos casos de uso *pares***

- 2.Consultar vacantes

  a) Diagramas de Interacción

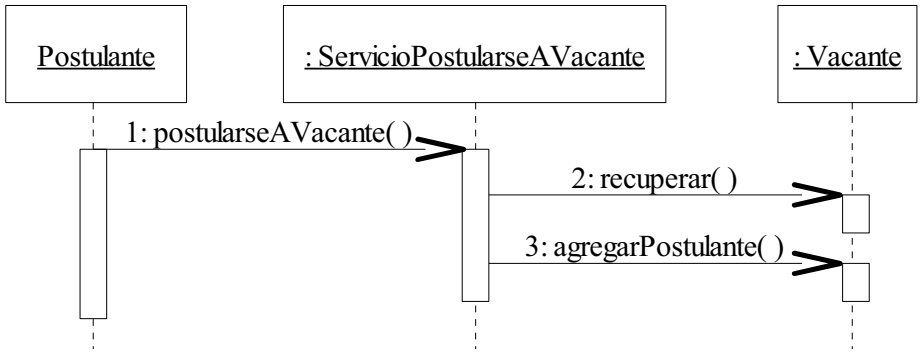



  b) Diagrama de Colaboración o de comunicación.

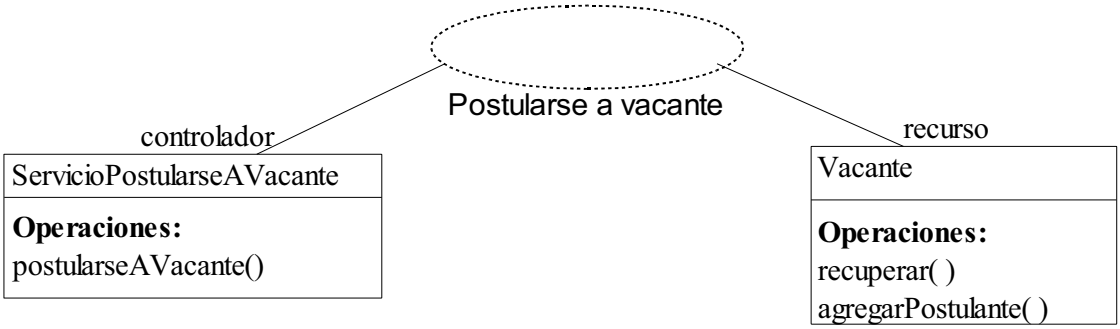


- 8.Postularse a vacante

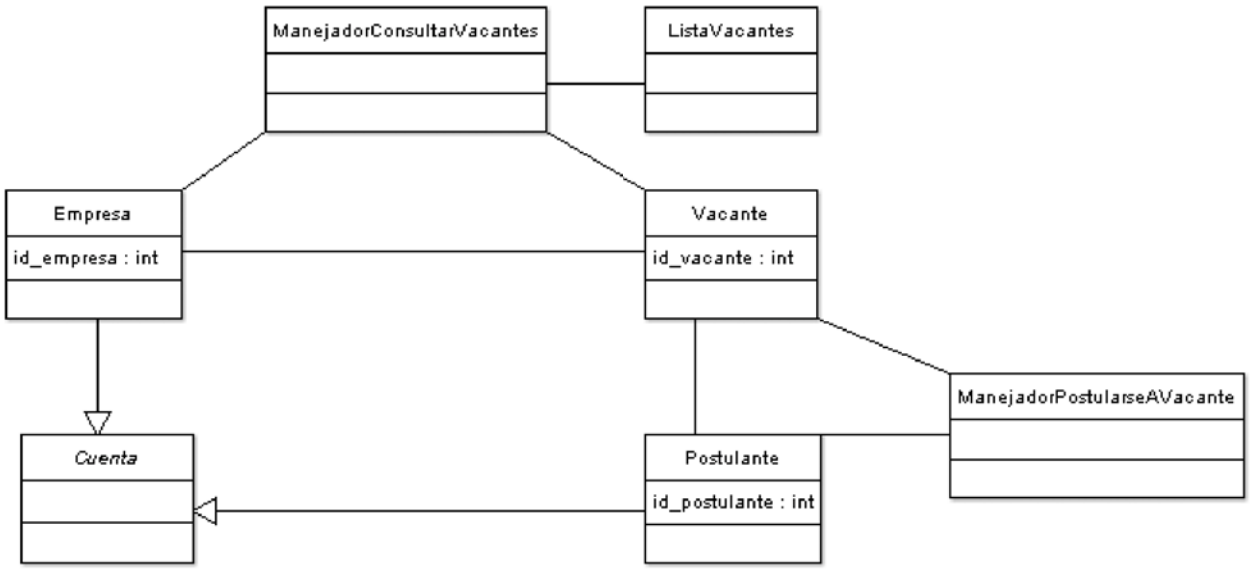
  a) Diagramas de Interacción




  b) Diagrama de Colaboración




**Estructura de análisis de elementos para la capa de control solamente para estos dos casos de uso**

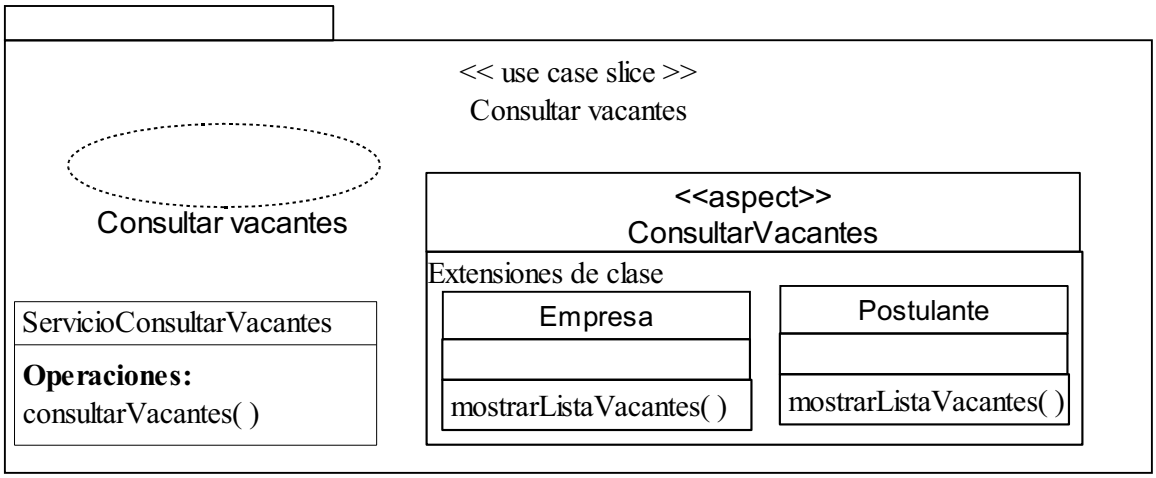


**Estructura de análisis de casos de uso para la capa de control**

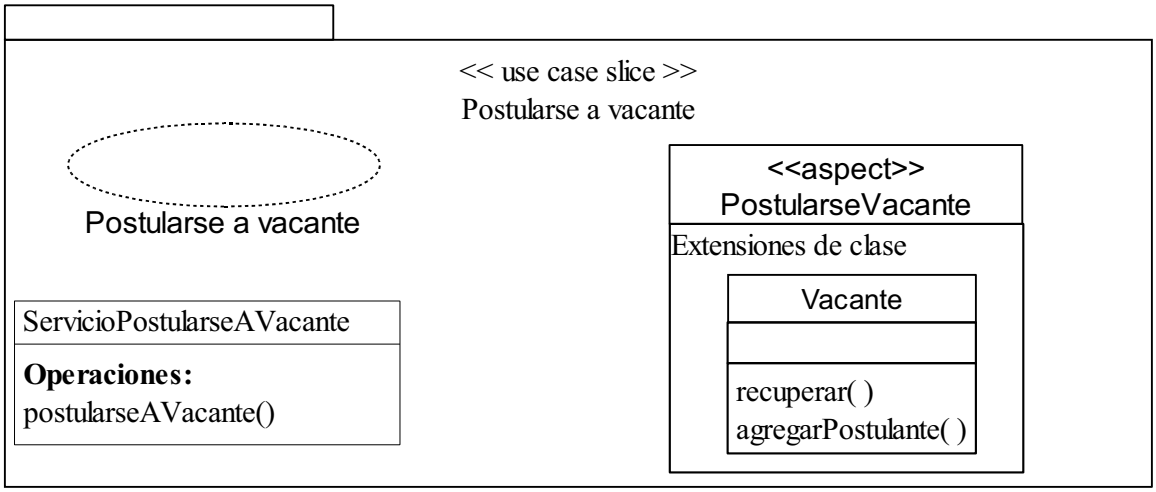
Se comienza por definir una rebanada por cada caso de uso. Dicha rebanada contiene exclusivamente aquellas clases necesarias para realizar la funcionalidad del caso de uso que representa. Asimismo, estas clases solamente contienen los atributos y métodos necesarios para implementar dicha funcionalidad.

Las rebanadas en el análisis para estos dos casos de uso son:

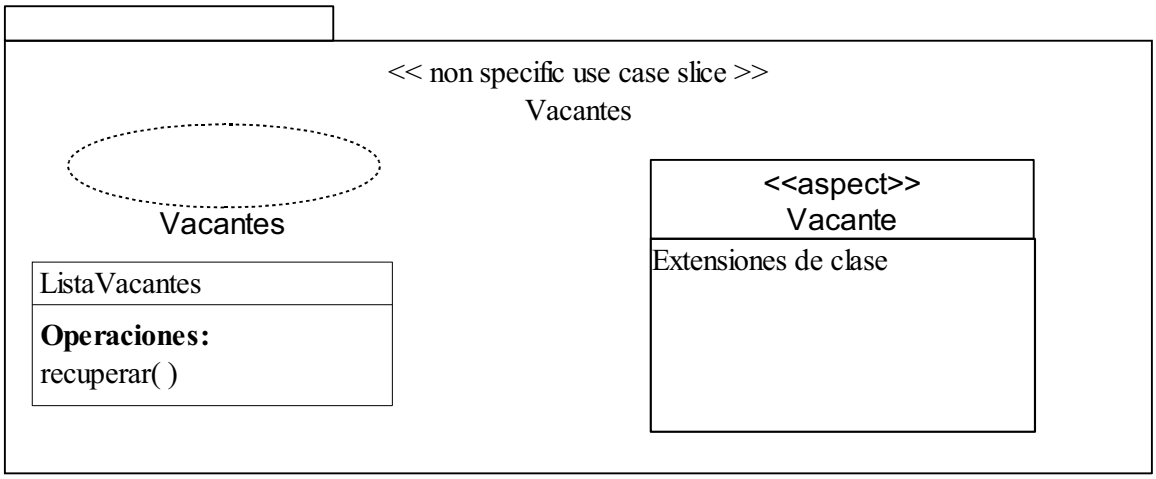
- 2. Consultar Vacantes.




- 6. Postularse a Vacante



- Rebanada no específica de un caso de uso




Es necesaria esta rebanada no específica de un caso de uso, debido a que la lista de vacantes es utilizada tanto en *Consultar vacantes*, como en *Consultar Postulantes*, por ejemplo.

## 4.4. Diseño

Tal como fue planteado en la sección 3.1 (Introducción) del capítulo "Programación Orientada a Aspectos (POA) y Arquitectura Dirigida por Modelos (MDA)", para crear la *estructura de elementos*, se utilizará AndroMDA. Utilizando "Value Objects" para la propagación de datos entre las diferentes capas, como fue planteado en la subsección "Arquitectura de aplicaciones generadas utilizando AndroMDA" dentro de la sección 3.2.4. Para el desarrollo del modelo en UML, será utilizado ArgoUML debido a que su licencia de uso permite su utilización sin restricciones y sin necesidad de pago alguno.

La aplicación será desplegada hacia el contenedor de EJBs JBoss v4.2.1 debido a que es un servidor de aplicaciones libre sumamente versátil y también porque AndroMDA está listo para desplegar sus aplicaciones hacia éste, sin necesidad de modificar su configuración.

Para el desarrollo de la *estructura de casos de uso*, será utilizado el marco de trabajo Orientado a Aspectos JBoss AOP v1.5.6. Haciendo las modificaciones necesarias a la configuración de la aplicación generada con AndroMDA en forma manual.

Jacobson plantea la necesidad de utilizar un Manejador (*Handler*) por cada caso de uso [Jacobson2004]. En el caso específico de esta aplicación, será omitido aún cuando proporciona una mayor extensibilidad y reusabilidad, en pos de la simplificación del ejemplo desarrollado. Aparte, aún cuando al utilizar manejadores queda más claro el funcionamiento arquitectónico de la aplicación para cualquiera que vea el código fuente, salvo en implementaciones muy complejas, se obtienen muchas clases prácticamente vacías.

En el desarrollo de aplicaciones orientadas a servicios, tendríamos un manejador que hace uso de una serie de servicios genéricos, para que éstos puedan ser reutilizables más fácilmente. Sin embargo, en esta aplicación, debido a su sencillez, será utilizado un servicio por caso de uso, que hará las veces de Manejador y de Servicio, a menos que el caso específico amerite una arquitectura más compleja.

En este capítulo será descrito solamente el caso de uso *Consultar vacantes* a manera de demostración.

El diseño, como fue presentado en capítulos anteriores, debe especificar una plataforma en concreto. La parte fundamental del diseño, como también fue mencionado previamente, es la arquitectura.

### *Arquitectura Utilizada*

Este trabajo está basado en buena medida en AndroMDA, un marco generador MDA. Tal como fue expuesto en la sección 3.2.4 (Arquitectura de las aplicaciones).

Como consecuencia, esta aplicación será desarrollada utilizando una arquitectura de cuatro capas:

*Capa de presentación:*

Esta capa, cuyo objetivo es manejar la interacción de la aplicación con el usuario final, será desarrollada utilizando Struts (Un marco para aplicaciones web compuesto por servlets y páginas JSP)

*Capa de negocio:*

Como fue expuesto anteriormente, el objetivo de esta capa es encapsular el corazón de la funcionalidad requerida por el negocio. Esta capa será desarrollada utilizando Servicios, cuyas transacciones y persistencia estarán controlados por Spring y EJB's (Enterprise Java Beans).

*Capa de acceso a datos:*

La razón de ser de esta capa, es proveer un intermediario entre las capas ya descritas, y el almacenamiento mismo de la información. De esta forma hay una menor dependencia hacia un manejador específico de base de datos.

Será llevada a cabo utilizando Hibernate. Hibernate es una herramienta de mapeo objeto-relacional muy poderosa.

*Capa de almacén de datos:*

Será utilizado MySQL. Aún cuando sería posible utilizar algun otro manejador de bases de datos más robusto, el objetivo de la aplicación a desarrollar no lo requiere. El objeto de desarrollar esta aplicación está enfocado a la demostración de las propuestas planteadas en esta tesis, y no a desarrollar una aplicación con gran capacidad de peticiones.

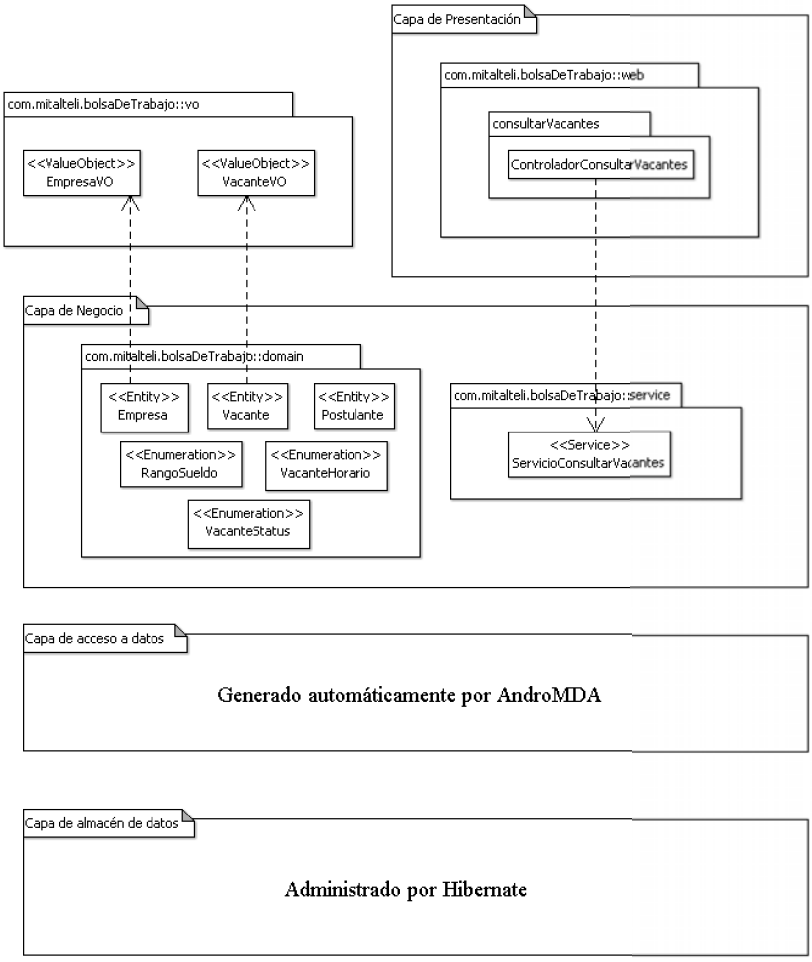



*Figura 4.4.1: Diagrama de Capas (Diseño)*
*Estructura de elementos*

### *Detalle de capa de presentación.*

Una consecuencia de utilizar AndroMDA es que no es necesario definir y especificar todas las clases que son necesarias para manejar la presentacion. Apoyándose en Struts, solamente es necesario definir la clase de control, encargada del control de la navegabilidad, y un diagrama de actividades que especifica la secuencia y orden de navegación. Por medio de estereotipos y "value tags" es como se especifica qué estados requieren de alguna pantalla para que el usuario introduzca información o si es necesario un botón que realice alguna acción determinada. Los campos que una forma específica debe presentar o los botones necesarios, son definidos por medio de transiciones que contienen disparadores con diferentes características [AndroMDA_4].

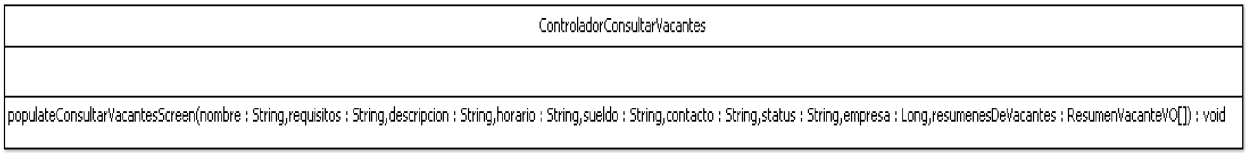



*Figura 4.4.2: Clases de Presentación (Diseño)*

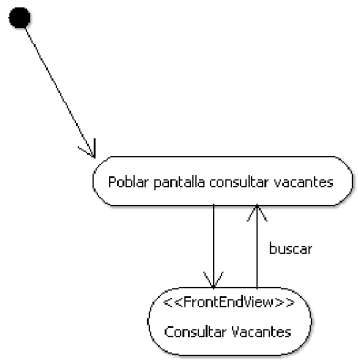



*Figura 4.4.3: Diagrama de Actividades Presentación (Diseño)*

### *Detalle de capa de negocio*

En un desarrollo completo, hay que definir por separado la *estructura de elementos* y la *estructura de casos de uso* a todo lo largo de la aplicación; para todas las capas de su arquitectura. Sin embargo, todavía no se cuenta con un marco MDA capaz de interpretar un modelo Orientado a Aspectos, abarcar toda la aplicación de esta forma es muy ambicioso para ser la primera vez que se intenta hacer.

La capa de negocio es el lugar idóneo para hacer la división y experimentar con esta nueva unión de tecnologías.

En medida de lo posible, se definieron clases vacías en la *estructura de elementos*, y toda su funcionalidad y atributos se concretan en la *estructura de casos de uso.*

Debido a que toda la configuración de Hibernate para manejar la persistencia de la aplicación que genera AndroMDA está basado en las clases de Entidad, dichas clases contienen todos sus atributos desde la *estructura de elementos*. Sin embargo, en cuanto a la implementación de los métodos, ésta si se lleva a cabo por medio de la tecnología de Aspectos en la *estructura de casos de uso*.

### *Estructura de Elementos (Diseño)*

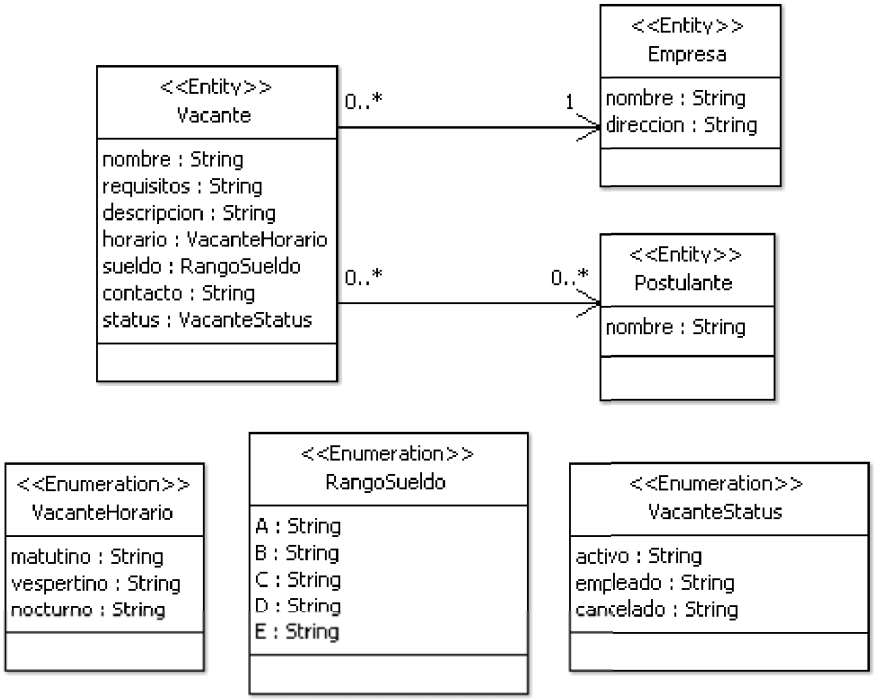



*Figura 4.4.4: Estructura de elementos del paquete Dominio (Diseño)*


*Figura 4.4.5: Estructura de elementos del paquete Servicio (Diseño)*




*Figura 4.4.6: Estructura de elementos del paquete ValueObjects (Diseño)*

### *Estructura de Casos de Uso (Diseño)*

Estas clases y sus métodos son generados por AndroMDA, es por eso que algunas no habían sido nombradas previamente.

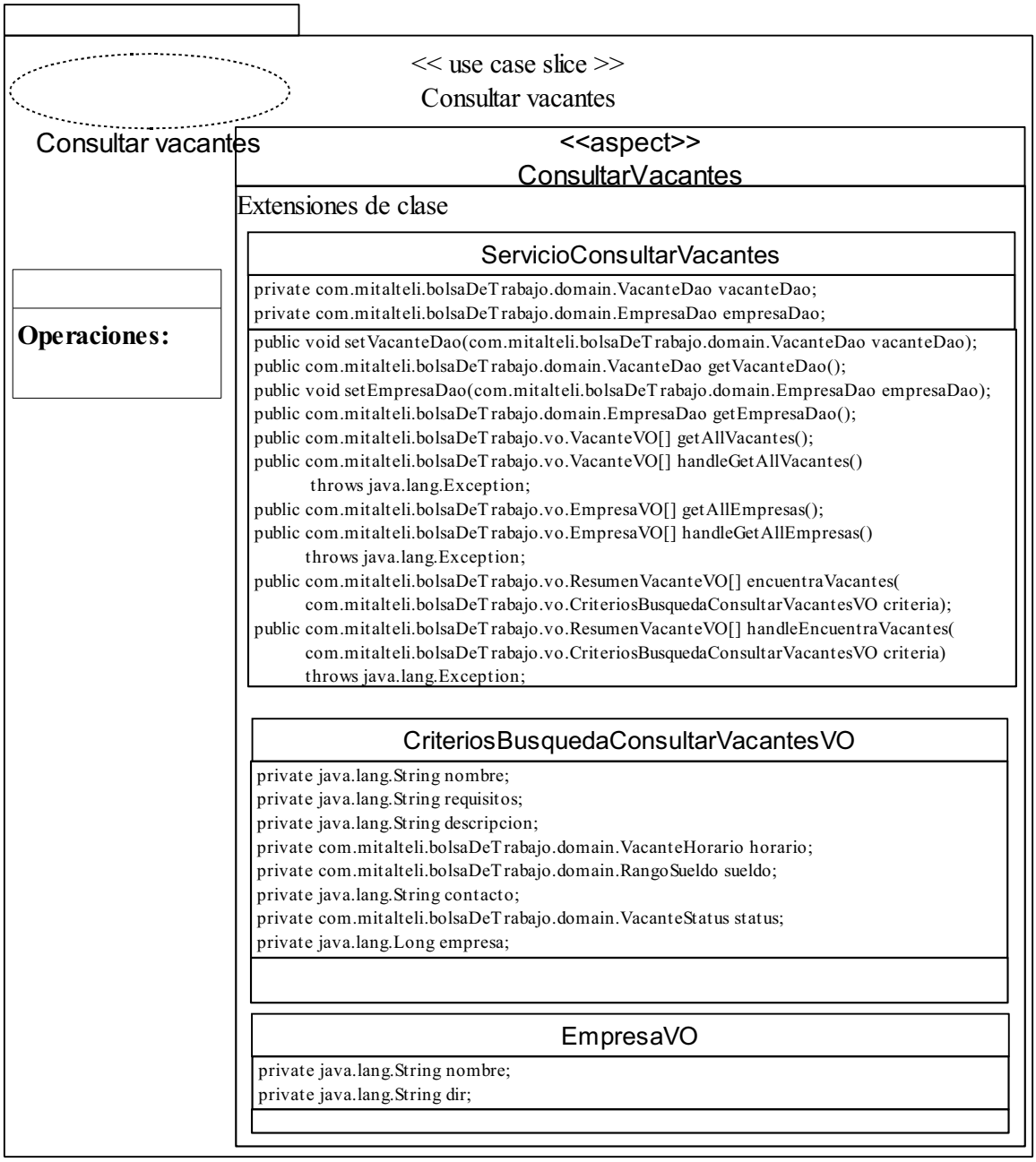



*Figura 4.4.7: Estructura de Casos de Uso (Diseño) (parte A)*

### *Estructura de Casos de Uso (Diseño)*

Continuación...

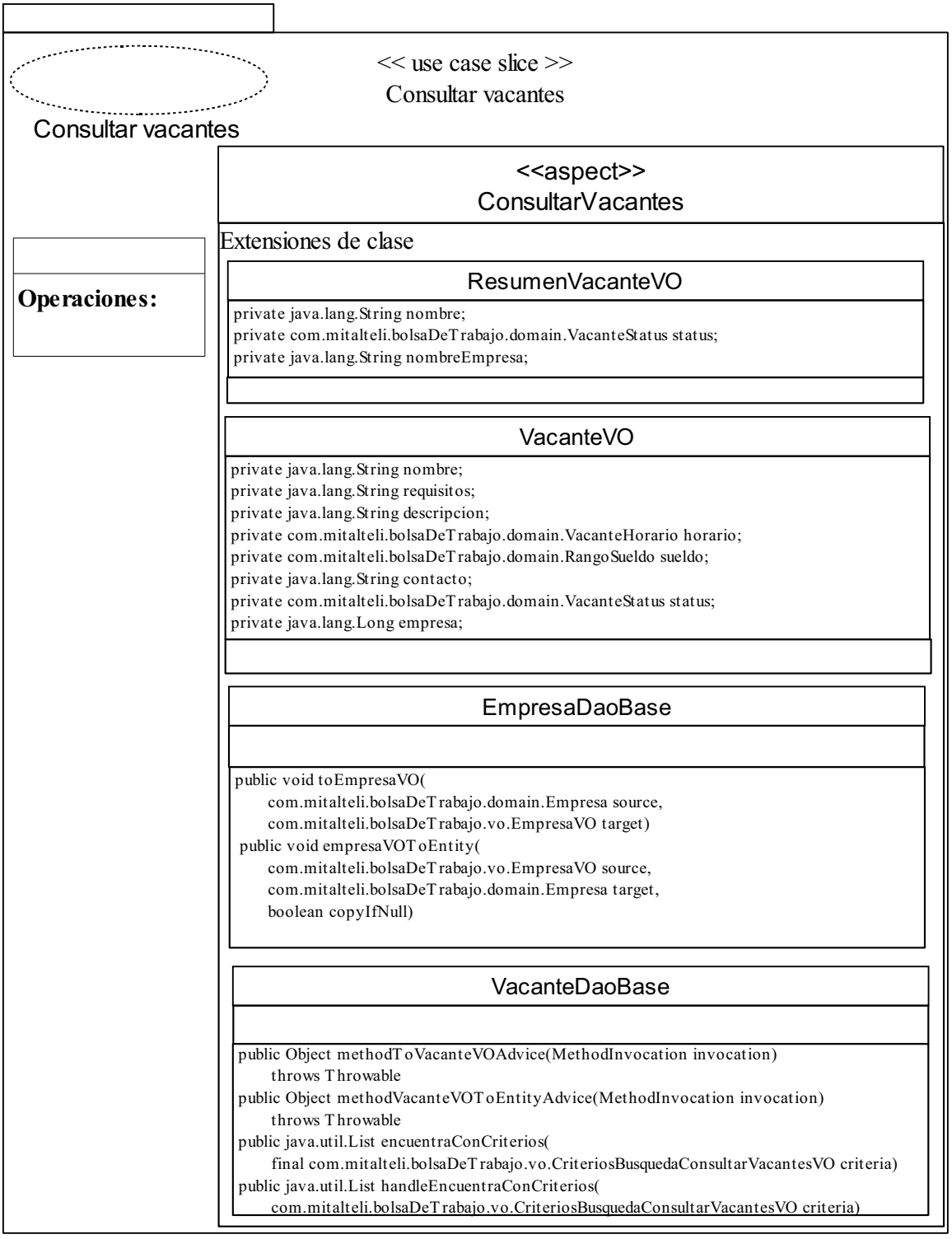



*Figura 4.4.8: Estructura de Casos de Uso (Diseño) (parte B)*

### *Diagrama de distribución*

El modelo de distribución describe la forma en que están distribuidos los componentes de software, físicamente, es decir, los diferentes equipos que conforman el sistema.

Este sistema consiste en una aplicación Web conectada a una base de datos, el modelo está formado por un servidor en el que se encuentra la aplicación y la base de datos y varios usuarios que ejecutan un navegador Web, como Internet Explorer, Mozilla o Netscape Navigator.

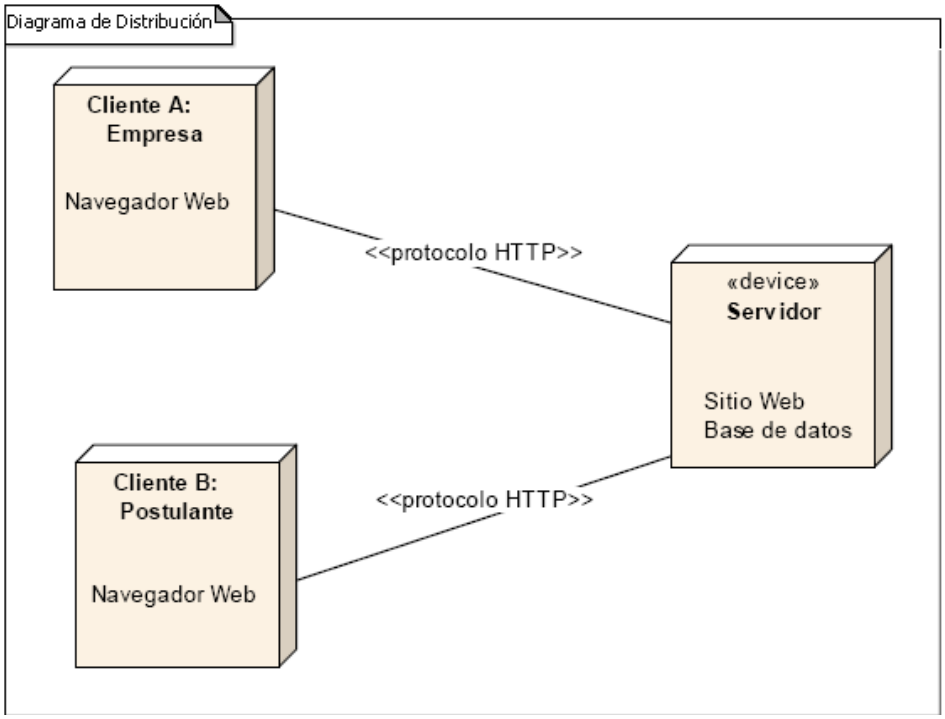



*Figura 4.4.9: Diagrama de Distribución (Diseño)*

### *Estructura de proceso.*

Debido a que se trata de una aplicación WEB, no es necesario definir esta estructura. El procesamiento se lleva a cabo en el servidor y el cliente solamente realiza funciones sumamente sencillas de validación. (No se utilizará AJAX)

***Diagrama de composición de paquetes***

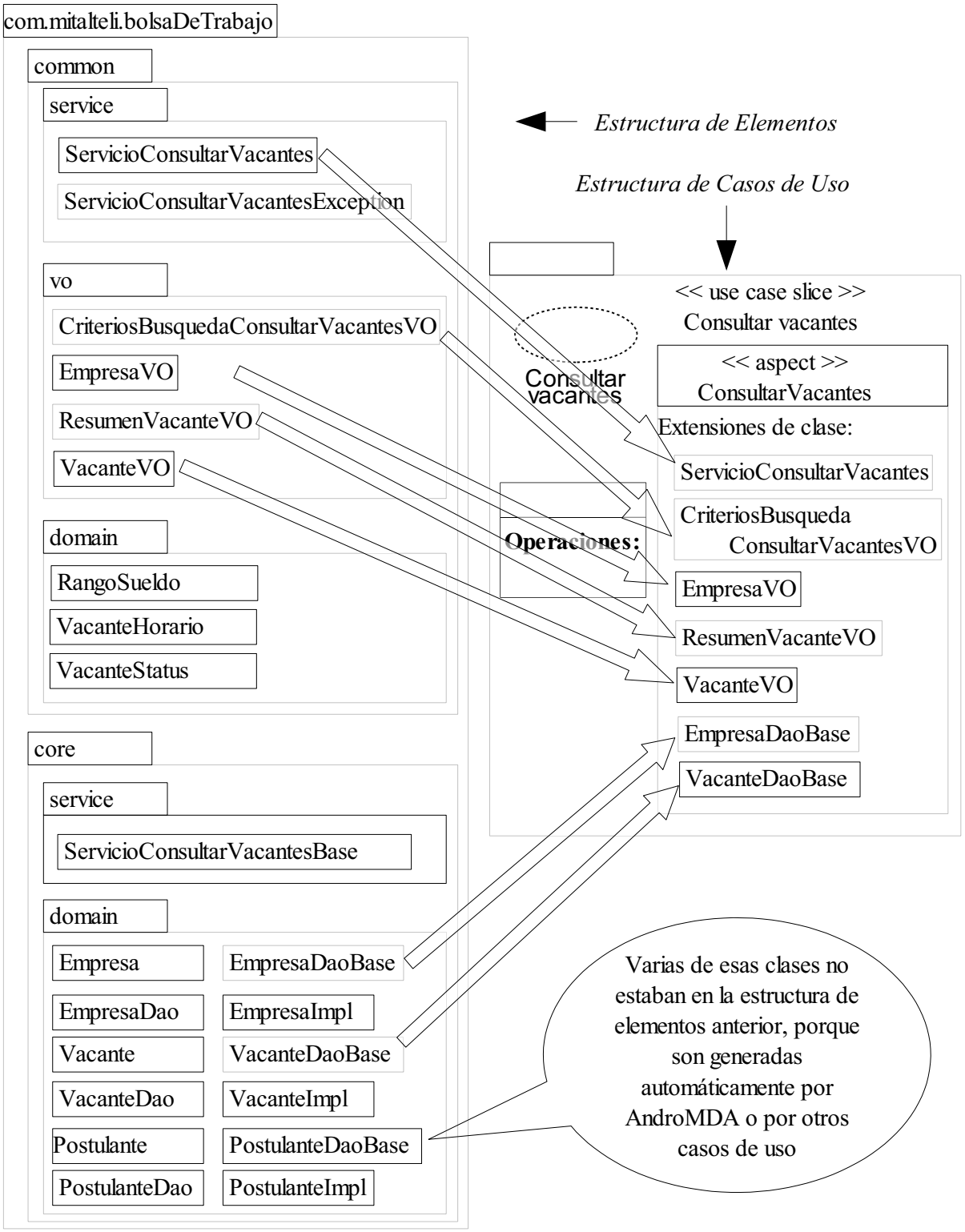



*Figura 4.4.10: Diagrama de composición de paquetes*

## 4.5. Implementación

A continuación se presentarán los detalles más importantes sobre la implementación de la aplicación, así como una descripción somera de dicho proceso. Una descripción mucho más detallada, de los primeros pasos, se presenta en el Apéndice B y en el Apéndice C.

Cabe señalar que en esta ocasión sería innecesario el artefacto de "Módulos de casos de uso" tal como fue definido en la sección 2.4.1 del Flujo de trabajo de Implementación orientado a aspectos debido a que la concordancia entre otros flujos de trabajo y el de implementación se lleva a cabo con base en el MDA. Por otro lado, la aplicación en cuestión es sumamente sencilla.

Hay que hacer notar que el trabajo de implementación estuvo basado en el tutorial presentado en la sección C.1 del Apéndice C. Este tutorial presenta en detalle todo lo necesario para desarrollar aplicaciones con AndroMDA.

Debido a que es la primera vez que se trabaja tanto con AndroMDA, como con JBoss-AOP, se decidió comenzar por desarrollar la funcionalidad deseada exclusivamente utilizando AndroMDA, en una primera instancia. De esta manera fue posible identificar las necesidades de funcionalidad a satisfacer por medio de Aspectos en una etapa posterior.

Este procedimiento no fue carente de complicaciones. El proceso de generar la aplicación requiere de una gran cantidad de pasos a seguir, para lograr la funcionalidad deseada, así como el modelo en UML requiere una gran cantidad de pequeños ajustes. Al mismo tiempo, fue necesario acostumbrarse a las peculiaridades de las diferentes herramientas utilizadas.

Posteriormente, se creó un nuevo proyecto que serviría para especificar la *estructura de elementos*. Con las clases necesarias, pero sin atributos y sin métodos, con la intención de que toda su funcionalidad fuera implementada por separado, utilizando la tecnología de Aspectos. Esto es, la *estructura de casos de uso*. El resto del capítulo presenta el trabajo realizado con este segundo proyecto.

### *Modelo de implementación.*

AndroMDA utiliza Maven2 para construir la aplicación (ver la sección 3.2.2 llamada "Visión general del sistema AndroMDA"). Maven2 es una herramienta para manejar y comprender mejor proyectos. Está basado en archivos de configuración llamados, cada uno, "modelo de objetos de proyecto" (POM, por sus siglas en inglés). Maven permite administrar la construcción, reportes y documentación de aplicaciones desde una pieza central de información [Maven2_1]. Maven ofrece una forma estándar para construir proyectos, una definición clara de lo que consiste el proyecto, una forma sencilla de publicar información sobre el proyecto y una forma de compartir archivos JAR entre varios proyectos [Maven2_2].

Por sus características, es posible establecer una jerarquía de configuraciones, y así dividir el proyecto en sub-proyectos, cada uno con su configuración particular.

Tal como fue expuesto en la sección 3.2.3 (Ciclo de desarrollo con AndroMDA), es suficiente con ejecutar un comando para que AndroMDA genere la estructura de directorios y lo necesario para el desarrollo de la aplicación.

La estructura de directorios obtenida fue:

```
bolsaDeTrabajo
        |
        |-- mda
        |
        |-- common
        |
        |-- core
        |
        |-- web
        |
        +-- app
```

- **bolsaDeTrabajo**: Es el proyecto principal, que controla el proceso global de construcción de la aplicación, y las propiedades generales.
- **mda**: Es el sub-proyecto o módulo de Maven que alberga el modelo en UML de la aplicación, dentro del directorio src/main/uml/. Asimismo, es el proyecto en donde AndroMDA es configurado para generar los archivos necesarios para ensamblar la aplicación.
- **common**: Es el sub-proyecto o módulo de Maven con recursos y clases que se deben compartir entre otros sub-proyectos. Incluye los *Value Objects*, por ejemplo.
- **core**: Alberga recursos y clases que utilizan el marco de trabajo Spring y Hibernate/EJB's. Incluye clases de entidad, objetos de acceso a datos (DAO), archivos de mapeo de Hibernate, y servicios.
- **web**: Alberga los recursos y clases que contiene la capa de presentación.
- **app**: Alberga los recursos y clases necesarios para construir el paquete ".ear" que es el archivo que se utiliza para desplegar la aplicación al servidor de aplicaciones.

También se genera automáticamente un archivo "readme.txt" que contiene información muy importante sobre el proyecto, así como los comandos más útiles. El contenido de dicho archivo se presenta en el apartado README.TXT de la sección C.2 del Apéndice C.

Lo primero que hay que hacer, después de inicializar el proyecto, es tomar el modelo en UML de la *estructura de elementos* creado en el análisis, y adecuarlo para su utilización con AndroMDA, por medio de la utilización de estereotipos y etiquetas de valor (*Value Tags*).

### *Modelo de la estructura de elementos*

Para que ArgoUML pueda procesar un modelo, requiere de una estructura de paquetes específica. El modelo de paquetes de la estructura de elementos de la aplicación desarrollada para este trabajo de Bolsa de Trabajo es:

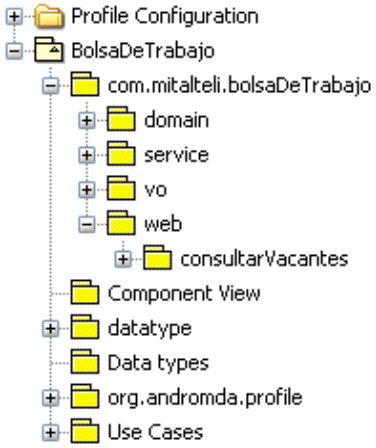



*Figura 4.5.1: Diagrama de Paquetes (Implementación)*

La estructura de paquetes requerida en el modelo, no corresponde directamente con la estructura de directorios generada por AndoMDA. Simplemente sirve para que AndroMDA "sepa" cómo generar el código necesario.

Por ejemplo, el paquete "Profile Configuration" contiene la definición de los estereotipos que UML1.4 soporta. "org.andromda.profile" contiene todos los estereotipos y definiciones de etiquetas de valor que se pueden asignar a diferentes elementos del modelo, con el fin de controlar la forma en que el código es generado por AndroMDA. Uno de estos estereotipos es "<<ValueObject>>", por ejemplo.

Cada paquete contenido en "com.mitalteli.bolsaDeTrabajo" contiene diagramas, clases, y demás elementos requeridos para implementar la funcionalidad de la aplicación. En esta ocasión estarán prácticamente vacíos debido a que la funcionalidad será implementada por medio de POA.

**Detalle del paquete de los ValueObjects**

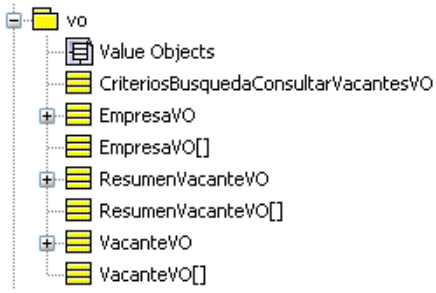



*Figura 4.5.2: Detalle Paquete ValueObjects (implementación)*

**Capa de Presentación**

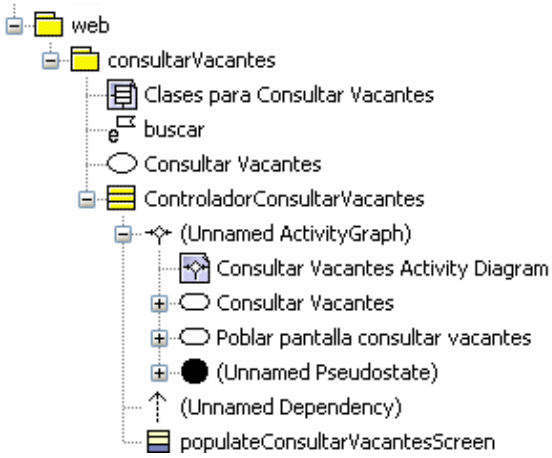



*Figura 4.5.3: Detalle paquete web (implementación)*

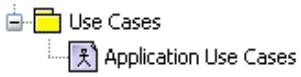



*Figura 4.5.4: Detalle paquete de casos de uso (implementación)*

Es importante el diagrama de casos de uso, debido a que es en este diagrama en donde se define cuál es el caso de uso con que la aplicación comienza su ejecución por medio de los estereotipos <<FrontEndUseCase>> y <<FrontEndApplication>>

**Capa de Negocio**:

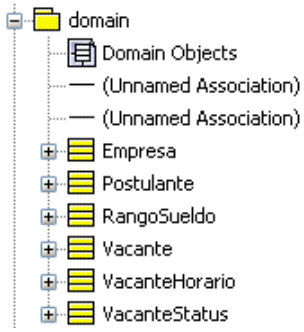



*Figura 4.5.5: Detalle Paquete de dominio (implementación)*


*Figura 4.5.6: Detalle Paquete de servicio (implementación)*

A diferencia del modelo del diseño, AndroMDA requiere especificar ciertas relaciones entre clases de diferentes capas/paquetes.

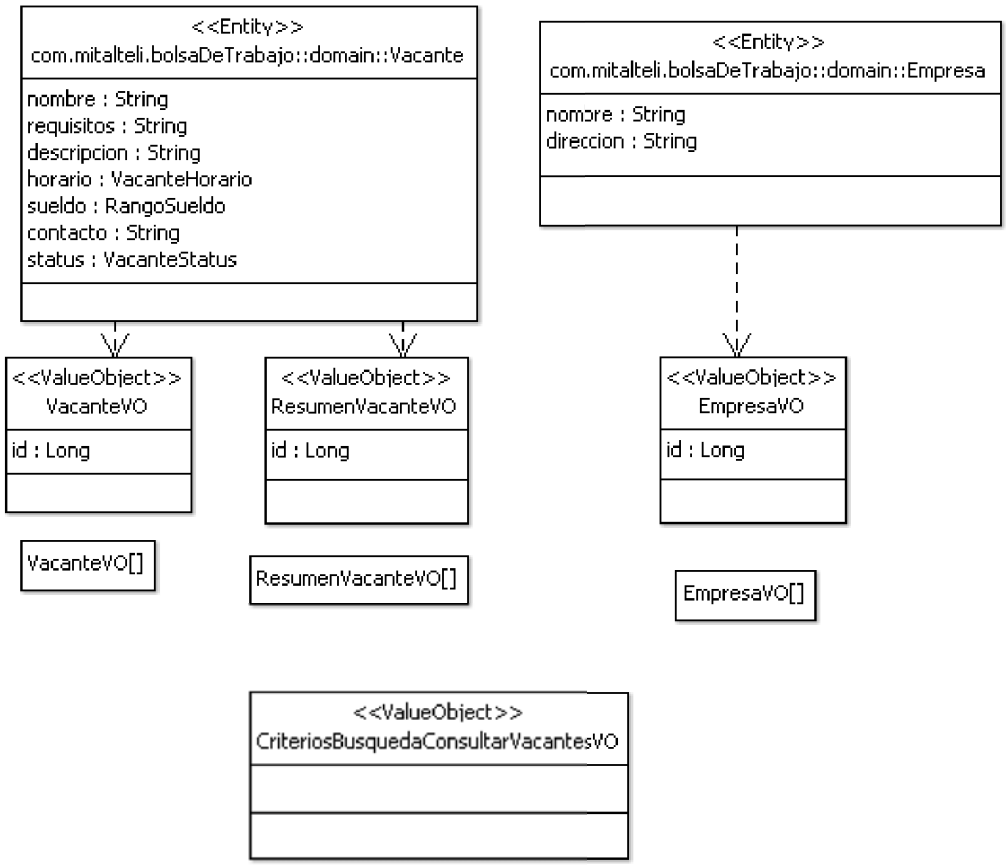



*Figura 4.5.7: Dependencias entre clases de dominio y ValueObjects*

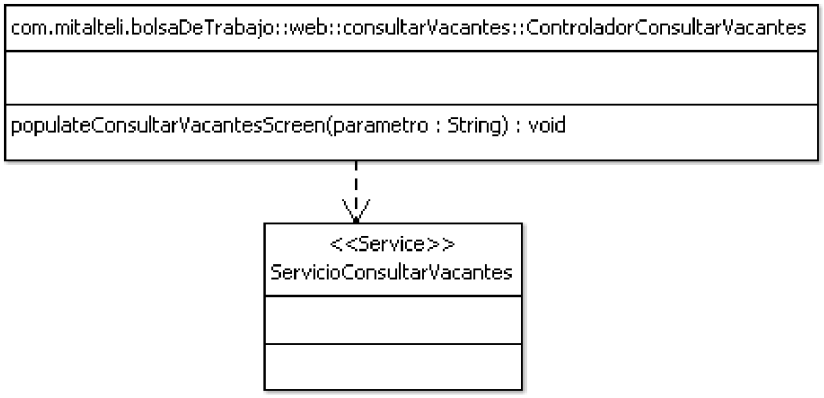



*Figura 4.5.8: Dependencias entre clases de control de presentación y servicio*

### *Estructura de Casos de Uso*

Una vez especificado el modelo de la *estructura de elementos* en UML, es generado el código fuente gracias a un comando de Maven, formado por clases y métodos prácticamente vacíos. De esta forma se obtiene la estructura en la cual serán inyectadas las funcionalidades definidas en la *estructura de casos de uso* por medio de POA. Estas funcionalidades serán implementadas en forma manual debido a que aún no existe un marco MDA capaz de interpretar modelos Orientados a Aspectos y generar aplicaciones Orientadas a Aspectos, y es justamente lo que se quiere comprobar con éste trabajo de tesis.

**¿Cómo se agregarán funcionalidades a la *estructura de elementos*?**

Hay que hacer notar que hay diferentes tipos de funcionalidades a ser agregadas en la *estructura de elementos* desde la de casos de uso. Por un lado, en algunos casos es necesario agregar nuevos atributos o métodos a las clases de la primera estructura, y por otro lado en algunos casos es necesario modificar el comportamiento de métodos ya generados automáticamente.

Para agregar atributos o métodos nuevos a clases, es necesario utilizar *Introducciones* ([JBoss_AOP_4]). Las *Introducciones* consisten en agregar interfaces Java a clases ya existentes. También se debe agregar la implementación de dichas interfaces. Esta implementación es llamada *Mixin* ([JBoss_AOP_5]). Es posible agregar diferentes combinaciones *Introducción-Mixin* a una misma clase.

Para modificar el comportamiento de métodos ya existentes, se decidió utilizar la definición de *advices* en un archivo que agrupe todas estas modificaciones para una clase así como la definición de aquellos puntos de corte (ver su definición en la página 19) para los que aplican dichos *advices*. Este archivo es llamado “aspect file” por la documentación de JBoss-AOP.

**¿Cuál es la mejor manera de definir las rebanadas de casos de uso?**

Lo primero que fue necesario resolver, fue encontrar la mejor manera de definir las rebanadas de casos de uso en forma totalmente independiente una de otra y por separado de la aplicación base, es decir, la *estructura de elementos*.

Se decidió crear una estructura de directorios de la siguiente forma: Un directorio para las rebanadas; dentro de éste, un directorio para las rebanadas que son específicas para un caso de uso y otro directorio para las rebanadas que no lo son. Posteriormente, un directorio por rebanada dentro de estos dos directorios según corresponda.

**¿En dónde es el mejor lugar para ubicar la implementación de las rebanadas de casos de uso y cómo generar el archivo (`.ear`) a desplegar en el servidor de aplicaciones con una estructura adecuada?**

Debido a que la estructura generada por AndroMDA está compuesta por diferentes “sub-proyectos” gobernados por una jerarquía de archivos de configuración de Maven2, se tomó la decisión de agregar un nuevo “sub-proyecto”. Dicho sub-proyecto fue llamado “aop”

Según la documentación (sección 4.2.2 del manual de referencia [JBoss_AOP_1]) para que el servidor de aplicaciones encuentre la definición de *introducciones* y *aspectos* (puntos de corte y especificación de archivos que contienen el código fuente del *advice*), es necesario un archivo JAR que contenga un archivo de configuración en `/META-INF/jboss-aop.xml` y que dicho archivo JAR tenga como extensión `.aop`.

Aún cuando es posible definir los puntos de corte por medio de anotaciones directamente en el código del *advice*, se decidió definir esta información junto con la definición de las *Introducciones-Mixins* en un mismo lugar: el archivo `jboss-aop.xml`.

Utilizando un tutorial sobre diferentes métodos de despliegue de aplicaciones JBoss-AOP en servidores de aplicaciones JBoss (ver [JBoss_AOP_3]), fueron realizadas configuraciones equivalentes hasta encontrar una que generara la estructura interna del archivo `.ear` deseada, es decir, debe contener el JAR con extensión `.aop` nombrado unos párrafos atrás.

La estructura de directorios creada para el "sub-proyecto" aop capaz de generar la estructura interna deseada del archivo `.ear` es la siguiente:

```
C:\AplicacionTesis\bolsaDeTrabajo>tree
Listado de rutas de carpetas
C:.
├───aop
│   ├───.settings
│   ├───src
│   │   └───main
│   │       ├───java
│   │       │   └───com
│   │       │       └───mitalteli
│   │       │           └───bolsaDeTrabajo
│   │       │               └───useCaseSlices
│   │       │                   ├───nonSpecific
│   │       │                   └───specific
│   │       │                       └───consultarVacantes
│   │       └───resources
│   │           └───META-INF
│   └───target
│       └───classes
│           ├───com
│           │   └───mitalteli
│           │       └───bolsaDeTrabajo
│           │           └───useCaseSlices
│           │               └───specific
│           │                   └───consultarVacantes
│           └───META-INF
```

El archivo `jboss-aop.xml` será ubicado en: `./aop/src/main/resources/META-INF/`

El directorio "`target`" es generado automáticamente al momento de compilar la aplicación de la misma forma como se hacía antes de agregar este nuevo sub-proyecto, después de realizar las modificaciones necesarias a los archivos de configuración de Maven2.

**¿Cuál es el mejor momento para realizar el tejido?**

Como fue planteado en la sección 1.3.1 (Tejido de aspectos) AOP define la posibilidad de llevar a cabo el tejido *en tiempo de compilación* o *en tiempo de ejecución*. Adicionalmente, JBoss-AOP permite llevar a cabo el tejido durante la carga de la aplicación (ver la sección 3.3, AndroMDA y POA).

En un comienzo se decidió que lo mejor sería hacerlo en tiempo de compilación, ya que implica menos carga para el servidor en producción. Para realizar este tipo de tejido es necesario modificar la forma en que la aplicación es compilada, ya que después de la compilación normal, es necesario agregar el

proceso de tejido. Esta fue la primera verdadera dificultad encontrada. La curva de aprendizaje de Maven es larga. Su configuración es complicada a primera vista, debido a que es muy sofisticada.

El primer intento fue con un "*plugin*" de JBoss-AOP para Maven2 que está en desarrollo. Sin embargo, están utilizando versiones alpha de JBoss-AOP 2.0.0, y planean liberarlo, junto con documentación, cuando JBoss-AOP 2 sea liberado. Todavía no hay una fecha específica para tal evento [JBoss_AOP_2].

No fue posible utilizarlo. Esa versión del plugin es sumamente complicada de utilizar debido a que no hay documentación. Así mismo, los foros de ayuda mostraron ser de poca ayuda respecto a ese tema.

Para el segundo intento fue utilizada una configuración agregando una "tarea" ANT. ANT es una herramienta multiplataforma (anterior a Maven) para construir aplicaciones Java [ANT_1]. Es posible agregar tareas ANT en configuraciones Maven2 ([Maven2_3]).

En las pruebas no funcionaba nada de la aplicación al momento de desplegarla. Se tomó la decisión de utilizar nuevamente los ejemplos mostrados en [JBoss_AOP_3], y se realizaron configuraciones más pequeñas para demostrar que si era posible modificar un bean de java utilizando Aspectos y desplegarlo en el servidor de aplicaciones Jboss. Las pruebas fueron exitosas. Sin embargo se encontraron muchas dificultades al tratar de reproducir esa configuración en la aplicación generada con AndroMDA [Foros_1]. Aparecía un error al momento de tratar de crear el bean `beanRefFactory` debido a que no era posible registrar un bean con nombre vacío y que no se podía encontrar la clase modificada por el aspecto. Después de múltiples intentos, incluso con la ayuda de colaboradores del proyecto AndroMDA, se decidió comenzar nuevamente desde cero con la esperanza de sobrepasar este punto.

Todavía existía cierta duda de si el problema estaba en el proceso de tejido, o en la estructura interna del archivo `.ear`. Debido a estas dificultades encontradas, y por sugerencia de un colaborador del proyecto AndroMDA, se decidió que la mejor forma de atacar el problema sería cambiar la estrategia. Ahora sería realizado el tejido en tiempo de carga. De esa forma, se reducen los posibles problemas causados por un mal pre-proceso de tejido, en combinación con una posiblemente errónea estructura de empaquetamiento.

En la nueva aplicación, se decidió comenzar con la atención centrada en lograr la estructura correcta del archivo `.ear`. Se hicieron muchas pruebas modificando su contenido en forma manual. Poniendo y quitando un archivo jar con extensión `.aop` del archivo `.ear` así como cambiando su estructura interna. Todavía quedaba la duda de si era correcta la sintaxis del archivo jboss-aop.xml, la forma de especificar la ubicación de las clases a modificar por medio de aspectos, la forma de especificar la ubicación del código del *advice* y/o de las "*Introducciones-mixins*", entre otras cosas.

Una vez encontrada una estructura (y configuración) que mostraba en los archivos de bitácora que el servidor de aplicaciones era capaz de encontrar los aspectos, así como llevar a cabo su tejido, llegó el momento de buscar la manera de automatizar el proceso de construcción del archivo `.ear` por medio de Maven. Una tarea nada fácil.

Las modificaciones realizadas a la configuración de Maven2 para generar la estructura interna adecuada del archivo `.ear` se presentan en el Apéndice C.

Se tomó la decisión de comenzar con aspectos solamente en la capa de negocio, es por esto que una serie de configuraciones necesarias en la capa de presentación (para que se mostraran todos los

atributos de un ValueObject modificado con aspectos, por ejemplo) se realizara en forma manual y por fuera. Asi que durante el proceso de compilación de la aplicación es necesario copiar dichos archivos sobreescribiendo los generados en forma automática por AndroMDA. Todos esos archivos fueron guardados en un subdirectorio a nivel de raiz del proyecto llamado:

"000-Extra\capaPresentación"

Por último, se habilitaron todas las opciones de depuración en la configuración de JBoss-AOP en el servidor de aplicaciones JBoss para verificar el proceso de carga de la aplicación, una vez desplegada. Se pudo comprobar que las clases a ser modificadas con los aspectos si estaban siendo encontradas por JBoss-AOP, y que también estaban siendo tejidas ya que se pueden encontrar mensajes como los siguientes en los archivos de bitácora:

```
2008-05-22 09:39:22,968 INFO  [STDOUT] [trying to transform]
      com.mitalteli.bolsaDeTrabajo.domain.VacanteDaoBase

...

2008-05-22 09:39:23,015 INFO  [STDOUT] [debug] javassist.CtMethod@ce8af663[public toVacanteVO
      (Lcom/mitalteli/bolsaDeTrabajo/domain/Vacante;Lcom/mitalteli/bolsaDeTrabajo/vo/VacanteVO;)V]
      matches pointcut: execution(* com.mitalteli.bolsaDeTrabajo.domain.VacanteDaoBase-
      >toVacanteVO(com.mitalteli.bolsaDeTrabajo.domain.Vacante,
      com.mitalteli.bolsaDeTrabajo.vo.VacanteVO))

...

2008-05-22 09:39:23,484 INFO  [STDOUT] [debug] was com.mitalteli.bolsaDeTrabajo.domain.VacanteDaoBase
      converted: true
```

Lamentablemente al momento de escribir este documento todavía no era posible hacer funcionar la aplicación adecuadamente. Se logró que no aparecieran errores en las bitácoras durante el despliegue y carga de la aplicación en el servidor de aplicaciones. Sin embargo, al momento de ejecutar la aplicación desde el navegador se obtenía la siguiente excepción debida a un constructor con parámetros incorrectos, que no se pudo resolver:

```
2008-05-27 08:08:23,671 DEBUG [org.springframework.beans.factory.support.DefaultListableBeanFactory]
      Ignoring constructor [public
      org.springframework.context.support.ClassPathXmlApplicationContext(java.lang.String[],boolean,org.
      springframework.context.ApplicationContext) throws org.springframework.beans.BeansException] of
      bean 'beanRefFactory': Error creating bean with name 'beanRefFactory' defined in URL
      [jar:file:/C:/jboss-4.2.1.GA/server/default/tmp/deploy/tmp23672bolsaDeTrabajo-1.0-SNAPSHOT.ear-
      contents/bolsaDeTrabajo-core-1.0-SNAPSHOT.jar!/beanRefFactory.xml]: Unsatisfied dependency
      expressed through constructor argument with index 1 of type [boolean]: Ambiguous constructor
      argument types - did you specify the correct bean references as constructor arguments?

2008-05-27 08:08:23,687 DEBUG [org.springframework.beans.factory.support.DefaultListableBeanFactory]
      Ignoring constructor [public
      org.springframework.context.support.ClassPathXmlApplicationContext(java.lang.String[],org.springfr
      amework.context.ApplicationContext) throws org.springframework.beans.BeansException] of bean
      'beanRefFactory': Error creating bean with name 'beanRefFactory' defined in URL
      [jar:file:/C:/jboss-4.2.1.GA/server/default/tmp/deploy/tmp23672bolsaDeTrabajo-1.0-SNAPSHOT.ear-
      contents/bolsaDeTrabajo-core-1.0-SNAPSHOT.jar!/beanRefFactory.xml]: Unsatisfied dependency
      expressed through constructor argument with index 1 of type
      [org.springframework.context.ApplicationContext]: Ambiguous constructor argument types - did you
      specify the correct bean references as constructor arguments?
```

De contar con más tiempo, me parece razonablemente posible sobrellevar esta dificultad y llevar a buen término la aplicación.

## 4.6. Conclusiones de la implementación con AndroMDA y AOP

Se considera exitoso el intento de armar una aplicación basada en AndroMDA con funcionalidades añadidas gracias a la tecnología de Programación Orientada a Aspectos, incluso sin terminarse en su totalidad. Después de una larga lista de dificultades, intentos por resolverlas y avances considerables, se llegó a un punto en el que ya no era práctico continuar aplazando la entrega de este trabajo de tesis.

Se considera que fue razonablemente comprobado que es posible realizar una aplicación utilizando AndroMDA en conjunto con JBoss-AOP. La lógica de implementación es correcta. Esto fue demostrado gracias a que se pudieron combinar ambas tecnologías de forma que las bitácoras del servidor de aplicaciones demostraron llevar a cabo el tejido de clases.

Sin embargo, el aspecto técnico representa un desafío en sí mismo. Se encontró una gran cantidad de obstáculos con la especificación y estructura de *Aspectos* e *Introducciones-Mixins* así como para encontrar la forma de hacer que Maven2 generara el archivo de despliegue (`.ear`) con la estructura adecuada para el servidor de aplicaciones (con el módulo JBoss-AOP).

Es posible que las dificultades que se presentan actualmente en el desarrollo de la aplicación sean consecuencia de lo novedoso de ambas tecnologías, o una consecuencia de la complejidad añadida debido a la unión de tantas tecnologías, como lo son Struts, Spring, Hibernate, AndroMDA y JBoss-AOP. Hacer aplicaciones convencionales con cada una de las herramientas por separado, que son tan sofisticadas y poderosas sería mucho más fácil. Pero como esta aplicación incluye conceptos de AOP, que no son convencionales, implica un alto nivel de dificultad.

Adicionalmente a la dificultad inherente al crisol de tecnologías utilizadas en este proyecto, también hay que nombrar las dificultades que las herramientas de desarrollo agregaron. Un ejemplo de esto fue el modelado en UML. En un comienzo se hizo la especificación del modelo en UML con MagicDraw. Sin embargo, debido a que se trataba de una versión de prueba, rápidamente se alcanzaron sus limitaciones en cuanto al número de clases permitidas por diagrama, así que fue necesario volver a especificar el modelo utilizando ArgoUML. ArgoUML es una poderosísima herramienta libre y de código abierto para hacer modelos UML, sin embargo todavía le falta mucho trabajo para llegar a ser una herramienta realmente estable.

# Conclusiones

Tal como se planteó en la introducción, esta tesis tiene dos objetivos: por un lado la propuesta de un Proceso Unificado de Desarrollo de Software que sea Orientado a Aspectos (PUDSOA); por otro lado la utilización de un marco MDA con la finalidad de combinar las fortalezas de ambas tecnologías y así ofrecer una mejor solución a la necesidad de desarrollo de aplicaciones de una manera más rápida y mejor.

Al Proceso Unificado de Desarrollo de Software Orientado a Aspectos le fueron agregadas actividades, roles y productos de POA para cada flujo de trabajo, menos para el de pruebas[2], de tal forma que se pudiera utilizar como guía para el desarrollo de aplicaciones Orientadas a Aspectos. Siguiendo las propuestas aquí realizadas, es posible llevar a cabo la identificación de elementos que podrían significar la generación de código mezclado o código disperso, desde el principio del proceso de desarrollo, y así poder evitarlo. Sin la tecnología de POA sería imposible especificar completamente por separado las diversas funcionalidades necesarias para la aplicación.

Se estudió la tecnología MDA y, lo más importante, se comprendió cómo es que se puede pasar desde una especificación en UML del modelo de una aplicación, a obtener prácticamente todo el código fuente de dicha aplicación lista para recibir la implementación de los algoritmos que la harían útil, sin necesidad de preocuparse del resto de la codificación necesaria simplemente para lograr que funcione.

Se entendió cómo ligar ambas tecnologías: AOP y MDA. Cuáles son y cómo se cumplen los requerimientos de cada una de estas dos tecnologías, es decir, su codificación y configuraciones necesarias. Definitivamente no se trata de dos tecnologías mutuamente excluyentes. Sin embargo ambas tecnologías llevan de la mano una larga curva de aprendizaje.

Fue llevado a cabo el desarrollo de la aplicación utilizando al PUDSOA para "planearla" de tal forma que fuera Orientada a Aspectos, es decir, con sus diversas funcionalidades especificadas, analizadas, diseñadas e implementadas por separado. Así mismo, durante los procesos de diseño e implementación, fue utilizado AndroMDA para generar la *estructura de elementos* de la aplicación que serviría como el esqueleto necesario para recibir todas las funcionalidades especificadas en la *estructura de casos de uso*, utilizando la tecnología de Orientación a Aspectos.

La utilización de AndroMDA en conjunto con POA, tal como está actualmente, resulta ser sumamente complicado al tratar de separar estructura de funcionalidad en el código generado debido a que este código es sumamente inter-dependiente internamente. Sin embargo, si se modificara lo necesario desde la interpretación del modelo, es altamente probable que se pueda generar código Orientado a Aspectos muy eficientemente.

En el desarrollo de la aplicación fue necesario enfrentarse a dificultades más allá del tema mismo a resolver. Enfrentarse a tantas tecnologías, marcos de trabajo y herramientas nuevas vino de la mano con una gran cantidad de dificultades. Así mismo, algunas herramientas libres representaron un desafío por sí solas al encontrarse en una etapa poco madura de su desarrollo.

2 Debido a la complejidad que representa el proceso de pruebas de software, podría hacerse una tesis completa sobre la utilización de POA para llevarlas a cabo.

## *Investigación Futura*

Es necesario incursionar en la definición y/o modificación de componentes de AndroMDA para facilitar la generación de aplicaciones Orientadas a Aspectos directamente especificando modelos Orientados a Aspectos en UML. Es del parecer del autor de este trabajo que lo descubierto en el mismo proporciona un gran avance en la dirección adecuada para lograr este fin.

# Referencias bibliográficas

[April2008]: April, Alain; Abran, Alain, "*Software Maintenance Management: Evaluationand Continuous Improvement*", John Wiley & Sons, Inc, 2008, 312 pag. ISBN 978-0470-14707-8. (Libro)

[Booch1994]: Booch, Grady, "*Object-Oriented Analisis and Design with Applications*", Series in Object-Oriented Software Engineering, 2nd ed,The Benjamin/Cummings Publishing Company, Inc, 1994, 587 pag. ISBN 0-8053-5340-2. (Libro)

[Clarke2005]: Clarke, Siobhán; Banissad, Elisa, "*Aspect-Oriented Analisis and Design. The Theme Approach*", Addison-Wesley, 2005, 438 pag. ISBN 0-321-24674-8. (Libro)

[Elrad2001]: Elrad, Tzilla; Filman, Robert E.; Bader, Atef, "Aspect-Oriented Programming", Communications of the ACM, Vol 44, Num 10, Octubre 2001, pag. 29-32. (Artículo)

[Jacobson2000]: Jacobson, I.; Booch, G.; Rumbaugh, J., "*El proceso unificado de desarrollo de software*", Pearson Educación, S.A., Madrid, 2000, 464 pag. ISBN 84-7829-036-2. (Libro)

[Jacobson2004]: Jacobson, Ivar; Ng, Pan-Wei, "*Aspect-Oriented Software Development with Use Cases*", Addison Wesley, 2004, 418 pag. ISBN 0-321-26888-1. (Libro)

[Kiczales1997]: Kiczales, Gregor; Lamping, John; Mendhekar, Anurag; Maeda, Chris; Lopes Videira, Cristina; Loingtier, Jean-Marc; Irwin, John, "Aspect-Oriented Programming", Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland, Vol 1241, Num N/A, Junio 1997, pag. 220-242. (Artículo)

[Meyer1997]: Meyer, Bertrand, "*Object-Oriented Software Construction*", 2nd ed,Prentice Hall PTR, 1997, 1254 pag. ISBN 0-13-629155-4. (Libro)

[Miller2001]: Miller, Joaquin; Mukerji, Jishnu, "Model Driven Architecture, Document number ormsc/2001-07-01", 2001, 31 pag. (Documentación técnica)

[Miller2003]: Miller, Joaquin; Mukerji, Jishnu, "MDA Guide Version 1.0.1", 2003, 62 pag. (Documentación técnica)

[Pawlak2005]: Pawlak, Renaud; Retaillé, Jean-Philippe; Seinturier, Lionel, "*Foundations of AOP for J2EE Development*", Apress, 2005, 474 pag. ISBN 1-59059-507-6. (Libro)

## Ligas a Internet

[AndroMDA_1]: “What is AndroMDA?”
http://galaxy.andromda.org/index.php?option=com_content&task=blogcategory&id=0&Itemid=42

[AndroMDA_2]: -Naresh Bhatia; “Application Architecture”
http://galaxy.andromda.org/index.php?option=com_content&task=view&id=108&Itemid=89

[AndroMDA_3]: http://galaxy.andromda.org/docs-3.1/starting.html

[AndroMDA_4]: -Naresh Bhatia; “Search Criteria Panel”
http://galaxy.andromda.org/index.php?option=com_content&task=view&id=141&Itemid=89

[Hibernate_1]: http://www.hibernate.org/

[Struts_1]: http://struts.apache.org/

[JSF_1]: http://java.sun.com/javaee/javaserverfaces/

[Spring_1]: http://www.springframework.org/

[JBoss_1]: http://labs.jboss.com/portal/jbossas/

[jBPM_1]: (http://www.jboss.com/products/jbpm

[EJB3/Seam_1]: http://www.jboss.com/products/seam

[AOP_y_MDA_1]: http://www.devx.com/enterprise/Article/27703/1763

[JBoss_AOP_1]: http://labs.jboss.com/jbossaop/docs/1.5.0.GA/docs/aspect-framework/reference/en/html_single/index.html

[JBoss_AOP_2]: http://www.jboss.com/index.html?module=bb&op=viewtopic&t=104619

[JBoss_AOP_3]: http://labs.jboss.com/jbossaop/docs/1.5.0.GA/docs/aspect-framework/examples/injboss/aopInJbossPackaging.html

[JBoss_AOP_4]: http://labs.jboss.com/jbossaop/docs/1.5.0.GA/docs/aspect-framework/userguide/en/html_single/index.html#introductions

[JBoss_AOP_5]: http://labs.jboss.com/jbossaop/docs/1.5.0.GA/docs/aspect-framework/quickref/en/html_single/index.html#mixins

[JBoss_AOP_6]: http://www.jboss.org/jbossaop/docs/1.5.0.GA/docs/aspect-framework/reference/en/html_single/index.html#pointcuts-pointcut

[Foros_1]: http://www.jboss.com/index.html?module=bb&op=viewtopic&p=4111457

[Foros_2]: http://www.jboss.com/index.html?module=bb&op=viewtopic&p=4110773#4109864

[Foros_3]: http://galaxy.andromda.org/forum/viewtopic.php?p=24217#24217

[Foros_4]: http://galaxy.andromda.org/forum/viewtopic.php?p=24220#24220

[Foros_5]: http://argouml.tigris.org/issues/show_bug.cgi?id=5111

[Maven2_1]: http://maven.apache.org/index.html

[Maven2_2]: http://maven.apache.org/what-is-maven.html

[Maven2_3]: http://maven.apache.org/ant-tasks.html

[Maven2_4]: http://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html#Dependency_Scope

[Maven2_5]: http://maven.apache.org/plugins/maven-ear-plugin/examples/including-a-third-party-library-in-application-xml.html

[ANT_1]: http://ant.apache.org/manual/index.html

# Apéndices

## *Apéndice A. Glosario.*

**actividad** Unidad tangible de trabajo realizada por un trabajador en un flujo de trabajo, de forma que (1) implica una responsabilidad bien definida para el trabajador, (2) produce un resultado bien definido (un conjunto de artefactos) basado en una entrada bien definida (otro conjunto de artefactos), y (3) representa una unidad de trabajo con limites bien definidos a la que, probablemente se refiera el plan del proyecto al asignar tareas a los individuos. También puede verse como la ejecución de una operación por un trabajador [Jacobson2000]. Véase *artefacto, trabajador.*

**actor** Un conjunto coherente de *roles* que los usuarios de casos de uso desempeñan cuando interaccionan con estos casos de uso [Jacobson2000].

**arquitectura** Conjunto de decisiones significativas acerca de la organización de un sistema de software, la selección de los elementos estructurales a partir de los cuales se compone el sistema, y las interfaces entre ellos, junto con su comportamiento, tal y como se especifica en las colaboraciones entre esos elementos, la composición de estos elementos estructurales y de comportamiento en subsistemas progresivamente mayores, y el estilo arquitectónico que guía esa organización: estos elementos y sus interfaces, sus colaboraciones y su composición. La arquitectura del software se interesa no sólo por la estructura y el comportamiento, sino también por las restricciones y compromisos de uso, funcionalidad, funcionamiento, flexibilidad al cambio, reutilización, comprensión, economía y tecnología, así como por aspectos estéticos [Jacobson2000].

**artefacto** Pieza de información tangible que (1) es creada, modificada y usada por los trabajadores al realizar actividades; (2) representa un área de responsabilidad, y (3) es candidata a ser tenida en cuenta para el control de la configuración. Un artefacto puede ser un modelo, un elemento de un modelo, o un documento [Jacobson2000]. Véase *trabajador, actividad.*

**componente** Una parte física y reemplazable de un sistema que se ajusta a, y proporciona la realización de, un conjunto de interfaces [Jacobson2000].

**construcción** Versión ejecutable del sistema, por lo general, de una parte específica del mismo. El desarrollo transcurre a través de una sucesión de construcciones [Jacobson2000].

**estereotipo** Una extensión del vocabulario de UML, que permite la creación de nuevos tipos de bloques de construcción que se derivan de otros existentes pero que son específicos a un problema particular [Jacobson2000].

**flujo de trabajo** Realización de un caso de uso de negocio o parte de él. Puede describirse en términos de diagramas de actividad, que incluyen a los trabajadores participantes, las actividades que realizan y los artefactos que producen [Jacobson2000]. Véase *flujo de trabajo de una iteración*.

**flujo de trabajo de una iteración** Flujo de trabajo que representa una integración de los flujos de trabajo fundamentales: captura de requerimientos, análisis, diseño, implementación y pruebas. Descripción de una iteración que incluye los trabajadores participantes, las actividades que éstos

realizan y los artefactos que producen [Jacobson2000]. Véase *flujo de trabajo*.

**gestión de la configuración** Tarea de definir y mantener las configuraciones y versiones de los artefactos. Esto incluye la definición de líneas base, el control de versiones, el control de estado y el control del almacenamiento de los artefactos [Jacobson2000]. Véase *artefacto*, *línea base*.

**hito principal** Punto en el que se han de tomar importantes decisiones de negocio. Cada fase termina con un hito principal en el cual los gestores han de tomar decisiones cruciales de continuar o no el proyecto, y decidir sobre la planificación, presupuesto y requisitos del mismo. Se considera a los hitos principales como puntos de sincronización en los que coinciden una serie de objetivos bien definidos, se completan artefactos, se toman decisiones de pasar o no a la siguiente fase, y en los que las esferas técnicas y de gestión entran en conjunción [Jacobson2000].

**hito secundario** hito intermedio entre dos hitos principales. Puede existir, por ejemplo, al terminar una iteración, o cuando se finaliza una construcción en una iteración [Jacobson2000]. Véase *hito principal*, *iteración*, *construcción.*

**interfaz** Una colección de operaciones que son utilizadas para especificar un servicio de una clase o de un componente [Jacobson2000].

**interfaz de usuario** Interfaz a través de la cual un usuario interactúa con un sistema [Jacobson2000].

**iteración** Conjunto de actividades llevadas a cabo de acuerdo a un plan (de iteración) y unos criterios de evaluación, que lleva a producir una versión, ya sea interna o externa [Jacobson2000]. Véase *plan de iteración, versión, versión interna, versión externa.*

**línea base** Conjunto de artefactos revisados y aprobados que (1) representa un punto de acuerdo para la posterior evolución y desarrollo, y (2) solamente puede ser modificado a través de un procedimiento formal, como la gestión de cambios y configuraciones [Jacobson2000]. Véase *línea base de la arquitectura*, *gestión de la configuración.*

**línea base de la arquitectura** Línea base resultado de la fase de elaboración, centrada en la arquitectura del sistema [Jacobson2000]. Véase *arquitectura*, *línea base*.

**plan de iteración** Plan detallado para una iteración. Plan que determina, para una iteración, los costos previstos, en términos de dinero y recursos, y los resultados previstos, en términos de artefactos. Plan que establece quién debe hacer qué en la iteración y en qué orden. Esto se lleva a cabo asignando trabajadores y describiendo un flujo de trabajo detallado para la iteración [Jacobson2000]. Véase *iteración, artefacto, trabajador, flujo de trabajo de una iteración.*

**prototipo de interfaz de usuario** Fundamentalmente, prototipo ejecutable de una interfaz de usuario, pero que puede, en los momentos iniciales del desarrollo, consistir únicamente en dibujos en papel, diseños de pantallas, etc. [Jacobson2000]. Véase *interfaz de usuario.*

**restricción** Extensión de la semántica de un elemento de UML, que permite añadir nuevas reglas o modificar las existentes [Jacobson2000].

**rol** El comportamiento específico de una entidad que participa en un contexto particular [Jacobson2000].

**trabajador** Puesto que puede ser asignado a una persona o equipo, y que requiere responsabilidades y habilidades como realizar determinadas actividades o desarrollar determinados

artefactos [Jacobson2000]. Véase *actividad, artefacto*.

**valor etiquetado** Extensión de las propiedades de un elemento de UML, que permite crear nueva información en la especificación de ese elemento [Jacobson2000].

**versión** Conjunto de artefactos relativamente completo y consistente (que incluye posiblemente una construcción) entregado a un usuario interno o externo; entrega de tal conjunto [Jacobson2000].

**versión externa** Versión expuesta a los clientes y usuarios, externos al proyecto y sus miembros [Jacobson2000]. Véase *versión*.

**versión interna** Una versión no expuesta a los clientes y usuarios, sino sólo de forma interna al proyecto y sus miembros [Jacobson2000]. Véase *versión*.

# *Apéndice B. Configuración del ambiente de desarrollo*

## *B.1) Ambiente de desarrollo para AndroMDA*

[AndroMDA_2] y [AndroMDA_3]

1 Requerimientos básicos

- JAVA
  Es necesario utilizar por lo menos la versión 1.4 de la máquina virtual para desarrollo de Java (Java 2 SDK) (http://java.sun.com)
- Maven o ANT
  En realidad se recomienda más utilizar Maven, puesto que la mayoría de las herramientas utilizadas por AndroMDA tienen plugin para Maven.
  - http://maven.apache.org
  - http://ant.apache.org

  Al momento de escribir esta tésis, la versión recomendada era Maven 2.0.5, ya que se han detectado algunos problemas con la versión Maven 2.0.7. Con la versión 2.0.8, también fueron encontrados algunos problemas.
- JBoss o Tomcat
  En ninguna manera AndroMDA depende de JBoss o Tomcat para funcionar adecuadamente. Simplemente es que el código generado sin modificaciones, funciona en alguno de estos dos servidores de aplicaciones. Es posible sin mucho problema utilizar otros servidores de aplicaciones.
  - http://www.jboss.org
  - http://jakarta.apache.org/tomcat
- MagicDraw, Poseidon o ArgoUML
  Ninguna de estas herramientas son indispensables para utilizar AndroMDA. Lo que pasa es que estas herramientas permiten importar y exportar modelos UML en una versión XMI que es compatible con AndroMDA, y proveen un conjunto de características adecuadas para ser utilizados en conjunto con AndroMDA
  - MagicDraw: http://www.magicdraw.com
  - Poseidon: http://www.gentleware.com/
  - ArgoUML: http://argouml.tigris.org/
- Eclipse
  - www.eclipse.org
- Una base de datos, por ejemplo MySQL
  - http://www.mysql.org/

2 Paso a paso

**JAVA**

Instalar J2SDK dependiendo de la plataforma. En el proceso de esta tesis fueron utilizados el Java 2 SDK v1.5 y v1.6

Establecer la variable de entorno JAVA_HOME
Es posible que dicha variable ya esté definida. Hay que asegurarse que así sea.

**Maven**

Después de instalar Maven, hay que configurarlo para que baje dependencias desde el servidor de AndroMDA cuando sea necesario.

1. Bajar Maven desde: http://maven.apache.org/start/download.html
2. Instalar la versión binaria descargada (Supondremos la ruta c:\apps\maven)
3. Ir a su directorio home. (Si su usuario es foo, esto significa ir a C:\Documents and Settings\foo)
4. Crear un archivo llamado *build.properties*, y agregar la siguiente línea en él:

   maven.repo.remote=http://www.ibiblio.org/maven,http://team.andromda.org/maven

**JBoss**

(Este paso es opcional, pero permitirá desplegar automáticamente las aplicaciones generadas a JBoss)

El servidor de aplicaciones JBoss permite desplegar aplicaciones EJB generadas:

1. Descargar JBoss desde http://www.jboss.org/downloads/index . Version 4.0.x o superior.
2. Descomprimir la versión binaria descargada a un directorio, por ejemplo, C:\jboss-4.2.1.GA\

**MagicDraw/Poseidon/ArgoUML**

Es necesario descargar e instalar alguno de estos programas para generar el modelo a ser procesado por AndroMDA.

- MagicDraw: http://www.magicdraw.com
- Poseidon: http://www.gentleware.com/
- ArgoUML: http://argouml.tigris.org/

**Variables de entorno**

| Propiedad | Valor | ¿Requerido? |
|---|---|---|
| JAVA_HOME | Directorio de instalación de java, por ejemplo C:\j2sdk1.4.2_08 | SI |
| JBOSS_HOME | Directorio de instalación de JBoss, por ejemplo C:\jboss-4.2.2.GA\ | No, a menos que se decida utilizar JBoss |
| CATALINA_HOME | Directorio de instalación de Tomcat, por ejemplo C:\apps\jakarta-tomcat-5.5.9 | No, a menos que se decida utilizar Tomcat |
| M2_HOME | Directorio de instalación de Maven, por ejemplo C:\apps\maven<br>(NOTA: en versiones de Maven anteriores a 2.0.9 es indispensable que NO contenga una "\" al final) | Si |
| M2 | %M2_HOME%\bin | Si |
| M2_REPO | Repositorio local de Maven, por ejemplo C:\Documents and Settings\NOMBRE DE USUARIO\.m2\repository | Si |
| MAVEN_OPTS | Parámetros pasados a la máquina virtual de java al ejecutar Maven, por ejemplo:<br>`-XX:MaxPermSize=128m -Xmx512m` | No |
| PATH | Esta variable debe contener: %JAVA_HOME%\bin;%M2% | Si |

**Verificar si la configuración funciona**

1. Crear un directorio temporal, por ejemplo, C:\andromda-temp
2. Crear un archivo llamado pom.xml con el contenido:

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>samples.test</groupId>
    <artifactId>test</artifactId>
    <version>1.0</version>
    <packaging>jar</packaging>
    <name>test</name>
    <build>
        <defaultGoal>compile</defaultGoal>
            <plugins>
                <plugin>
                   <groupId>org.andromda.maven.plugins</groupId>
                   <artifactId>andromdapp-maven-plugin</artifactId>
```

```
                <version>3.2</version>
            </plugin>
        </plugins>
    </build>
    <repositories>
        <repository>
            <id>andromda</id>
            <name>AndroMDA Repository</name>
            <url>http://team.andromda.org/maven2</url>
        </repository>
    </repositories>
    <pluginRepositories>
        <pluginRepository>
            <id>andromda</id>
            <name>AndroMDA Repository</name>
            <url>http://team.andromda.org/maven2</url>
        </pluginRepository>
    </pluginRepositories>
</project>
```

3. Abrir una línea de comando y cambiar de directorio al directorio temporal
4. Ejecutar *mvn* sin argumentos. Hay que asegurarse que al final se reciba un mensaje:

   `BUILD SUCCESSFUL`
5. Eliminar el directorio temporal.

**Instalar la base de datos**

Esta base de datos debe ser soportada por Hibernate.

1. Descargar la base de datos MySQL desde http://dev.mysql.com/downloads/

   Es necesario descargar la versión completa, por ejemplo mysql-5.0.45-win32.zip, y no solamente los "essentials".
2. Descargar las herramientas de interfaz gráfica de usuario, para que sea más sencilla su administración
3. Descargar el MySQL Connector/J version 5.0 y descomprimirlo en C:\Archivos de Programa\MySQL.
4. Copiar el controlador de MySQL desde C:\Archivos de Programa\MySQL\mysql-connector-java-5.0.4\mysql-connector-java-5.0.8-bin.jar al directorio lib de JBoss, por ejemplo C:\jboss-4.2.1.GA\server\default\lib. Renombrar el archivo para que tenga el formato estandar: mysql-connector-java-5.0.8.jar. removiendo "-bin" del final del archivo.

**Instalar Eclipse**

AndroMDA no requiere Eclipse para trabajar, sin embargo es recomendable utilizarlo para aumentar la productividad. AndroMDA genera archivos *.project* y *.classpath* perfectamente utilizables por Eclipse. Es suficiente con importar el archivo *.project* para comenzar a editar el proyecto utilizando Eclipse.

1. Descargar eclipse desde http://www.eclipse.org/downloads/ en donde dice "Eclipse IDE for Java EE Developers"
2. Descomprimir el archivo descargado. Se recomienda utilizar la ruta c:\eclipse
3. Modificar el archivo eclipse.ini localizado en c:\eclipse para incrementar la memoria disponible para este ambiente de desarrollo. Se sugiere poner los parámetros siguientes:
    - `-vmargs`
    - `-Xms512m`
    - `-Xmx512m`
    - `-XX:PermSize=64m`
    - `-XX:MaxPermSize=128m`
4. Iniciar Eclipse. Indicar el directorio del espacio de trabajo, por ejemplo C:\proyectos
5. Abrir las preferencias dentro del menú "*Window*"
6. En el árbol de opciones de la izquierada, seleccionar: Java > Build Path > ClasspathVariables
7. Hacer Click en "new"
8. Agregar el repositorio M2_REPO en el campo "Name" y la ruta del repositorio en el campo "Path". (Por ejemplo C:\Documents and Settings\SU NOMBRE DE USUARIO\.m2\repository). Este valor es indispensable para importar configuraciones de proyectos generadas por Maven.
9. Aceptar los cambios.

## *B.2) Configuración del marco de trabajo JBoss-AOP en un servidor de aplicaciones JBoss:*

(Se supondrá que se está trabajando en una máquina Windows XP profesional.)

a) Descargar un servidor de aplicaciones JBoss vacío. http://labs.jboss.com/jbossas/downloads/. En este caso versión 4.2.1.

b) Descomprimirlo en raiz (c:\)

c) Remover la carpeta C:\jboss-4.2.1.GA\server\default\deploy\jboss-aop-jdk50.deployer

d) Descargar la versión más nueva de JBoss-AOP http://labs.jboss.com/jbossaop/downloads/ . En el caso de esta tesis: versión 1.5.6.

e) Descomprimirlo en un lugar temporal.

f) Copiar el directorio jboss-aop-jdk50.deployer, ubicado dentro de la carpeta jboss-40-install, a la carpeta C:\jboss-4.2.1.GA\server\default\deploy\

g) Editar el archivo C:\jboss-4.2.1.GA\server\default\deploy\jboss-aop-jdk50.deployer\META-INF\jboss-service.xml para habilitar el tejido en tiempo de ejecución. Hay que modificar el atributo EnableLoadtimeWeaving para que tenga un valor verdadero. La línea modificada debe quedar similar a:

<attribute name="EnableLoadtimeWeaving">true</attribute>

h) Continuar con las instrucciones contenidas en el archivo:

C:\jboss-4.2.1.GA\server\default\deploy\jboss-aop-jdk50.deployer\ReadMe.txt

En el caso de esta tesis:

```
*If installing in jboss 4.0.4 or later, the classes contained in javassist.jar and common-
softvaluehashmap.jar will already be available,

and you should do the following to avoid versioning conflicts:

-delete common-softvaluehashmap.jar

-move javassist.jar to ../../lib/javassist.jar
```

i) Copiar el archivo pluggable-instrumentor.jar (ubicado en el directorio lib-50 del JBoss-AOP descargado y descomprimido en una carpeta temporal) al directorio bin, es decir, C:\jboss-4.2.1.GA\bin

j) Editar el archivo C:\jboss-4.2.1.GA\bin\run.bat para incluir (sin las comillas)

" -javaagent:pluggable-instrumentor.jar" en la variable JAVA_OPTS

Para la definición de rebanadas de casos de uso, fue creado un módulo nuevo en los fuentes generados por AndroMDA. Este módulo (llamdo aop) es el encargado de contener todas las rebanadas de casos de uso, tanto específicas como no-específicas.

# *Apéndice C. Descripción paso a paso para una aplicación AndroMDA*

## *C.1) Tutorial para desarrollo con AndroMDA*

Un gran trabajo de Naresh Bhatia y Cédric Jeanneret.

Se puede encontrar en:

http://galaxy.andromda.org/index.php?option=com_content&task=view&id=104&Itemid=89

## *C.2) Instrucciones específicas para la aplicación de esta tesis.*

Establecer el ambiente de desarrollo.
Los programas utilizados para desarrollar esta aplicación fueron:

- Java jdk1.6.0_03 (http://java.sun.com/javase/downloads/?intcmp=1281)

  En realidad la versión de Java recomendada para utilizar con AndroMDA es la 1.5.0_xx. Sin embargo no se han detectado problemas a causa de utilizar la versión 1.6.x durante el desarrollo de este trabajo.

- Maven-2.0.5. (http://maven.apache.org/)

  Es una herramienta desarrollada por la fundación Apache para construir aplicaciones.

  (Maven 2.0.7 tiene una modificación que dificulta su utilización)

- JBoss 4.2.1.GA (http://labs.jboss.com/)

  Servidor de aplicaciones

- MySql 14.12 Distrib 5.0.45, para Win32 (ia32) (http://www.mysql.org/)

  Manejador de Bases de Datos.

- Eclipse 3.3 (http://www.eclipse.org/)

  De igual forma, la versión recomendada es la 3.2

- ArgoUML 0.24 y 0.25.4-2 (http://argouml.tigris.org/)

  También es posible utilizar otras herramientas de UML, como MagicDraw, o Rational Software Modeler/Architect. Los desarrolladores de AndroMDA recomiendan MagicDraw. Sin embargo, la versión libre está limitada en número de elementos por diagrama.

Las rutas en donde fueron instalados los programas fueron:

- `C:\Documents and Settings\Ely\Mis Programas\`
  `eclipse3.3-jee-europa-fall-win32\eclipse\maven\repository`
- `C:\Archivos de programa\Apache Software Foundation\`
  `maven-2.0.5`
- `C:\Archivos de programa\Java\jdk1.6.0_03`
- `C:\Archivos de programa\Java\jre1.6.0_03`
- `C:\Archivos de programa\MySQL\MySQL Server 5.0`
- `C:\jboss-4.2.2.GA`

Las variables de entorno definidas fueron:

- JAVA_HOME

  `C:\Archivos de programa\Java\jdk1.6.0_03`

- JBOSS_HOME

  `C:\jboss-4.2.1.GA`

- M2_HOME

  `C:\Archivos de programa\Apache Software Foundation\`
  `maven-2.0.5`

- M2

  `%M2_HOME%\bin`

- M2_REPO

  `C:\Documents and Settings\Ely\Mis Programas\`
  `eclipse3.3-jee-europa-fall-win32\eclipse\maven\repository`

  - OJO: Fue modificado el archivo:

    `C:\Archivos de programa\Apache Software Foundation\`
    `maven-2.0.5\conf\settings.xml`

    Para definir (en una sola línea):

    ```
    <localRepository>
            C:\Documents and Settings\Ely\
            Mis Programas\eclipse3.3-jee-europa-fall-win32\
            eclipse\maven\repository
    </localRepository>
    ```

- MAVEN_OPTS

  `-Xms512m -Xmx512m -XX:PermSize=128m -XX:MaxPermSize=128m`

- PATH (Debe contener, entre otras cosas, lo siguiente)

  `%JAVA_HOME%\bin;%M2%`

### *Creación de la aplicación*

Fue creado el directorio encargado de contener la carpeta de la aplicación, y fue abierta una ventana de línea de comando en ese lugar. Como es muy importante que la ruta de dicho directorio no contenga espacios, de lo contrario es posible encontrarse con problemas al construir la aplicación, fue seleccionada la ruta: `C:\AplicacionTesis>`
Posteriormente, fue ejecutado lo siguiente:

```
C:\AplicacionTesis>mvn org.andromda.maven.plugins:andromdapp-maven-plugin:3.2:generate

[INFO] Scanning for projects...

[INFO] ------------------------------------------------------------------------

[INFO] Building Maven Default Project

[INFO]    task-segment: [org.andromda.maven.plugins:andromdapp-maven-plugin:3.2:generate] (aggregator-
style)
```

```
[INFO] ------------------------------------------------------------------------
[INFO] [andromdapp:generate]
log4j:WARN No appenders could be found for logger (org.apache.commons.digester.Digester).
log4j:WARN Please initialize the log4j system properly.
INFO  [AndroMDA] discovered andromdapp type --> 'richclient'
INFO  [AndroMDA] discovered andromdapp type --> 'j2ee'

Please choose the type of application to generate [richclient, j2ee]
j2ee

Please enter the location in which your new application will be created (i.e. f:/java/development):
C:\AplicacionTesis

Please enter your first and last name (i.e. Chad Brandon):
Ely Schoenfeld

Which kind of modeling tool will you use? [uml1.4, uml2, emf-uml2]:
uml1.4

Please enter the name of your J2EE project (i.e. Animal Quiz):
Bolsa de Trabajo

Please enter an id for your J2EE project (i.e. animalquiz):
bolsaDeTrabajo

Please enter a version for your project (i.e. 1.0-SNAPSHOT):
1.0-SNAPSHOT

Please enter the root package name for your J2EE project (i.e. org.andromda.samples.animalquiz):
com.mitalteli.bolsaDeTrabajo

Would you like an EAR or standalone WAR? [ear, war]:
ear

Please enter the type of transactional/persistence cartridge to use [hibernate, ejb, ejb3, spring,
none]:
spring

Please enter the database backend for the persistence layer [hypersonic, mysql, oracle, db2, informix,
mssql, pointbase, postgres, sybase, sabdb,
 progress, derby]:
mysql
```

```
Will your project need workflow engine capabilities? (it uses jBPM and Hibernate3)? [yes, no]:
no
Please enter the hibernate version number (enter '2' for 2.x or '3' for 3.x) [2, 3]:
3

Will your project have a web user interface? [yes, no]:
yes

Would you like your web user interface to use JSF or Struts? [jsf, struts]:
struts

Would you like to be able to expose your services as web services? [yes, no]:
no
------------------------------------------------------------------------------
    G e n e r a t i n g   A n d r o M D A   P o w e r e d   A p p l i c a t i o n
------------------------------------------------------------------------------
    Output: 'file:/C:/AplicacionTesis/bolsaDeTrabajo/app/src/main/application/META-INF/jboss-app.xml'
    Output: 'file:/C:/AplicacionTesis/bolsaDeTrabajo/app/src/main/config/bolsaDeTrabajo-ds.xml'
    Output: 'file:/C:/AplicacionTesis/bolsaDeTrabajo/app/pom.xml'
    Output: 'file:/C:/AplicacionTesis/bolsaDeTrabajo/core/target/classes/META-INF/ejb-jar.xml'
    Output: 'file:/C:/AplicacionTesis/bolsaDeTrabajo/core/pom.xml'
    Output: 'file:/C:/AplicacionTesis/bolsaDeTrabajo/pom.xml'
    Output: 'file:/C:/AplicacionTesis/bolsaDeTrabajo/readme.txt'
    Output: 'file:/C:/AplicacionTesis/bolsaDeTrabajo/common/pom.xml'
    Output: 'file:/C:/AplicacionTesis/bolsaDeTrabajo/mda/src/main/config/mappings/WebMergeMappings.xml'
    Output: 'file:/C:/AplicacionTesis/bolsaDeTrabajo/mda/src/main/config/andromda.xml'
    Output: 'file:/C:/AplicacionTesis/bolsaDeTrabajo/mda/src/main/uml/bolsaDeTrabajo.xmi'
    Output: 'file:/C:/AplicacionTesis/bolsaDeTrabajo/mda/pom.xml'
    Output: 'file:/C:/AplicacionTesis/bolsaDeTrabajo/web/src/main/properties/messages.properties'
    Output: 'file:/C:/AplicacionTesis/bolsaDeTrabajo/web/pom.xml'
    Output: 'file:/C:/AplicacionTesis/bolsaDeTrabajo/web/target/bolsaDeTrabajo-web-1.0-SNAPSHOT/WEB-
INF/web.xml'
------------------------------------------------------------------------------
    New application generated to --> 'file:/C:/AplicacionTesis/bolsaDeTrabajo/'
    Instructions for your new application --> 'file:/C:/AplicacionTesis/bolsaDeTrabajo/readme.txt'
------------------------------------------------------------------------------
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1 minute 52 seconds
[INFO] Finished at: Fri Jan 11 09:37:19 CST 2008
[INFO] Final Memory: 12M/508M
[INFO] ------------------------------------------------------------------------
```

```
C:\AplicacionTesis>
```

Archivo README.TXT generado por AndroMDA

El archivo Readme es sumamente importante, ya que contiene una explicación sencilla sobre el proyecto y su estructura, así como una gran lista de posibles comandos a ejecutar durante el trabajo con AndroMDA.

```
A n d r o M D A  -  3.2


AndroMDA is an open-source MDA framework distributed under the BSD license.
Go to http://www.andromda.org/ for more information.

The project located in this directory has been generated by Ely Schoenfeld
using the andromdapp:generate Maven plugin.

You should at least be running Maven 2.0.1 if you want to build your
project without any Maven-related problems, below is a summary of what has
been generated and a list of example goals to call from the command line.

The generated project structure is well-tailored for use in the development
of J2EE projects. The build process itself makes use of Maven, dependencies
and often-used goals have been added for your convenience.

Custom configuration can be done by updating the root pom.xml file.

/bolsaDeTrabajo J2EE project
     |
     |    The root of the project contains a few files that control the overall
     |    build process and common properties (in the pom.xml).
     |
     |-- pom.xml
     |        contains information about this project, you may add more information
     |        as long as you do not violate the Maven POM schema, see
     |        http://maven.apache.org/reference/project-descriptor.htmlt
     |
     +-- /mda
     |     |
     |     |    The MDA module is the heart of this project, this is where
     |     |    AndroMDA is configured to generate the files needed to
     |     |    assemble the application
     |     |
     |     +-- pom.xml
```

```
|     |        contains the AndroMDA dependencies and configuration (cartridges, translation-
libraries, etc)
|     +-- /src
|     |        additional sources such as merge-mappings can be
|     |        placed here, check out the /main/uml directory, it contains
|     |        the UML model from which AndroMDA will generate code
|     +-- /src/main/config/andromda.xml
|              configures AndroMDA and its components, most
|              importantly the cartridges which are listed in
|              their own namespace; global settings are done in the
|              'default' namespace
|
+-- /common
|     |
|     |    The COMMON module collects those resources and classes
|     |    that are shared between the other modules.
|     |
|     +-- pom.xml
|     |        lists common dependencies
|     +-- /target
|              shared resources and java classes are generated here,
|              such as value objects and exceptions
|
+-- /core
|     |
|     |    The CORE module collects those resources and classes
|     |    that use the Spring framework, optionally making
|     |    use of Hibernate and/or EJB under the hood.
|     |
|     +-- pom.xml
|     |        lists Spring dependencies
|     +-- /src/main/java
|     |        Spring classes that need manual implementation are
|     |        generated here, they will not be overwritten upon
|     |        regeneration; this includes the service, DAO and
|     |        entity implementations
|     +-- /target
|              the Spring resources and classes here will be
|              overwriten each time AndroMDA generates new code
|              using the Spring cartridge; this includes both
|              the Hibernate entities and the correponding
|              *.hbm.xml mapping files as well as the service
|              and DAO base classes. You'll also find the DDL
```

```
|              for creating and dropping your schema within this
|              directory.
|
+-- /web
|     |
|     |    The WEB module collects those resources and classes
|     |    that make up the presentation layer.
|     |
|     +-- pom.xml
|     |        lists WebApp dependencies
|     +-- /src/main/java
|     |        controller implementations and editable resource bundles
|     |        will be generated here,
|     |        you might consider putting your own JSPs here to
|     |        be copied over the generated ones when bundling the
|     |        .war file
|     +-- /target
|              any Struts classes, JSPs, resource bundles and
|              configuration files are generated here
|
+-- /app
      |
      |    The APP module collects those resources and classes
      |    that are required to actually build the .ear bundle
      |
      +-- pom.xml
      |        lists dependencies to be bundles into the
      |        root of the .ear
      +-- /src
      |        any custom resources that should be deployed
      |        together with the application
      |        (eg. server deployment descriptors)
      +-- /target
               a deployable application is bundled here
```

In order to succesfully build your project you will need to know how to invoke the build process for the existing modules, here's a list of examples:

```
%> mvn install

   simply builds all modules
```

```
%> mvn -f app/pom.xml -Ddeploy

    collects all artifacts and builds a deployable .ear which is then
    deployed

%> mvn clean

    cleans all generated files from each target directory

%> mvn install -Ddeploy

    rebuilds the entire application and deploys

%> mvn install -Denv=prod

    builds the entire application for the production environment, possible 'env' property
values
    are 'prod' for production, 'val' for validation and 'dev' for development; not specifying
    any value for this property will build the application for the local configuration
    (more info at http://maven.apache.org/guides/introduction/introduction-to-profiles.html)

%> mvn -f web/pom.xml andromdapp:undeploy -o

    undeploys the ear from your app server

%> mvn -N andromdapp:build -Dmodules=core (or mvn -f core/pom.xml)

    only build the core module

%> mvn -N andromdapp:build -Dmodules=web (or mvn -f web/pom.xml)

    only build the web module

%> mvn nuke

    cleans out all /target directories and removes all Java classes with names
    ending with 'Impl' from the source directories (from the common, core and
    web modules); this goal asks for confirmation, but be careful calling it
    anyway as you will lose your manually edited files

%>  mvn -f core/pom.xml andromdapp:schema -Dtasks=create
```

```
            generates the DDL create code and subsequently tells the database
            to create the schema for the entities

        %> mvn -f core/pom.xml andromdapp:schema -Dtasks=drop

            generates the DDL drop code and subsequently tells the database
            to drop the schema for the entities

        %> mvn -f core/pom.xml andromdapp:schema -Dtasks=drop,create

            generates the drop and create DDL code and subsequently tells the database
            to drop and then create the schema for the entities

        %> mvn -f core/pom.xml andromdapp:schema -Dtasks=update

            attempts to update the database schema with incremental changes, might not work with all
JDBC drivers
            see http://www.hibernate.org/hib_docs/v3/reference/en/html_single/#toolsetguide-s1-6

        %> mvn -f core/pom.xml andromdapp:schema -Dtasks=validate

            validates the current Hibernate mappings against the database, you can also use this to
simply
            check whether or not the mappings have been properly generated (in case of customizations)

        %> mvn -N andromdapp:build -Dmodules=mda:[andromda:start-server] (or mvn -f mda/pom.xml
andromda:start-server)

            starts the AndroMDA server, with this server running you will be able
            to significantly speedup the generation process although it will require
            you to use another console while it is running

        %> mvn -N andromdapp:build -Dmodules=mda:[andromda:stop-server] (or mvn -f mda/pom.xml
andromda:stop-server)

            stops the AndroMDA server

        %> mvn -N andromdapp:build -Dmodules=mda (or mvn -f mda/pom.xml)

            runs AndroMDA on your model and thereby generating files in
            the subdirectories of the existing modules

        %> mvn -N andromdapp:build -Dmodules=mda -Dfilter=java,hibernate (or mvn -f mda/pom.xml
-Dfilter=java,hibernate)
```

```
            runs AndroMDA, but this time only using the Java and Hibernate
            cartridges (in that order)

        %> mvn -N andromdapp:build -Dmodules=mda -Dfilter=~java,hibernate (or mvn -f mda/pom.xml
-Dfilter=~java,hibernate)

            runs AndroMDA, but this time using all cartridges *except*
            the Java and Hibernate cartridges

        %> mvn -N andromdapp:build -Dmodules=mda,core,web,app -Dfilter=java -Ddeploy

            runs AndroMDA using only the Java cartridge, rebuilds the core
            module and web module and deploys afterwards

        %> mvn -N andromdapp:build -Dconsole

            runs the AndroMDApp build goal in 'console' mode, once running, this allows you to execute
any goal or subproject
            goal in the manner described above without having to specify the andromdapp:build goal or
            restarting maven (i.e. you can run the mda goal by typing 'mda', build the core module by
type 'core', etc).


    Please note that you may use the '-o' flag at any time to avoid
    having Maven downloading any SNAPSHOT dependencies.

    Each module will install its artifact into the local Maven repository,
    this might be a directory looking like this:

        (Windows)
        C:\Documents and Settings\Ely Schoenfeld\.m2\repository\bolsaDeTrabajo\

        (*nix)
        /home/Ely Schoenfeld/.maven/repository/bolsaDeTrabajo/

    For questions or feature requests please use our forum:
    http://forum.andromda.org/


    Good luck!
```

```
-- The AndroMDA Team
```

**Configuración de la aplicación.**

Abrir el archivo de configuración localizado en:

`bolsaDeTrabajo\mda\src\main\config\andromda.xml`

Realizar los siguientes cambios:

1. Buscar la propiedad llamada `enableSpringTransactionsWhenEjbsEnabled` y cambiarle el valor actual, de `true` a `false`. La razón de hacer esto, es que se va a utilizar EJBs, entonces hay que evitar que Spring controle las transacciones de la base de datos.
2. Buscar el espacio de nombres llamado `bpm4struts`. En ese espacio de nombres, agregar una propiedad para asegurarse que las fechas tengan el formato correcto. La propiedad debe verse de la siguiente manera:

   ```
   <property name="defaultDateFormat">dd/MM/yyyy</property>
   ```
3. Agregar otra propiedad en el espacio de nombres `bpm4struts`.

   ```
   <property name="normalizeMessages">true</property>
   ```

   Esta propiedad permite la generación de mensajes que agrupan varios mensajes iguales. Por compatibilidad con versiones anteriores, tiene el valor `false`, por omisión.
4. Guardar y cerrar el archivo `andromda.xml`.

Abrir el archivo pom.xml ubicado directo en el directorio `bolsaDeTrabajo` y realizar los siguientes cambios.

1. Buscar la linea de especificación del controlador JDBC (una sola línea)

   ```
   <jdbc.driver.jar>
       ${jboss.home}/server/default/lib/hsqldb.jar
   </jdbc.driver.jar>
   ```

   Para utilizar MySql, especificando la versión correcta, poner lo siguiente:

   ```
   <jdbc.driver.jar>
       ${jboss.home}/server/default/lib/mysql-connector-java-5.0.8.jar
   </jdbc.driver.jar>
   ```
2. Buscar las siguientes dos líneas en donde se especifica el usuario y contraseña

   ```
   <jdbc.username>sa</jdbc.username>
   <jdbc.password></jdbc.password>
   ```

   y modificar los datos de acuerdo a la creación de la base de datos

   ```
   <jdbc.username>bolsa</jdbc.username>
   <jdbc.password>bolsa</jdbc.password>
   ```

**Utilización de ArgoUML**

Para poder utilizar ArgoUML con AndroMDA, es indispensable comenzar con un *perfil* para AndroMDA que no tenga extensiones, ya que ArgoUML no comprende las extensiones de XML.

Dicho *perfil*, se puede bajar de (una sola linea):

http://argouml.tigris.org/source/browse/*checkout*/argouml/trunk/src/model-mdr/src/org/argouml/model/mdr/profiles/andromda-profile-32-noextensions.xmi

Las instrucciones de cómo hacerlo, son presentadas de manera muy sencilla en el tutorial mencionado en la sección C.1 del Apéndice C en la página 99. Sin embargo, más adelante dichas instrucciones ya no son muy claras en cuanto a cómo configurar estructura generada con AndroMDA para que reconozca dicho modelo al momento de ser interpretado.

En resumen, hay que comenzar por abrir el perfil descargado y guardarlo con el nombre: `C:\AplicacionTesis\bolsaDeTrabajo\mda\src\main\uml\bolsaDeTrabajo.zargo` en ArgoUML v0.24. Si se trata de hacer esto utilizando ArgoUML v0.25.4-2 es levantada una excepción a la hora de compilar el proyecto.

Es necesario modificar `model.uri` en el archivo mda\pom.xml. (En el tutorial, no es muy claro, y hace creer que se refiere al archivo pom.xml ubicado en raiz)

`model.uri` debe contener el valor (en una sola línea y sin espacios):

```
<model.uri>jar:file:${project.build.sourceDirectory}/
      bolsaDeTrabajo.zargo!/bolsaDeTrabajo.xmi</model.uri>
```

(La sintaxis mostrada en el tutorial no es correcta).

Esto es, solamente agregar en lo que ya estaba en el archivo de configuración, al principio "jar:file:" en vez de "file:" y al final "/bolsaDeTrabajo.zargo!/bolsaDeTrabajo.xmi" en vez de "/bolsaDeTrabajo.xmi".

En realidad ArgoUML es sumamente poderoso, sin embargo, hay algunos problemas con la interfaz que pueden sacar de quicio a cualquiera, por lo menos con la versión v0.24. Es por esto que fue necesario alternar su uso con ArgoUML v0.25.4-2. Hasta el momento, si fue posible sortear todos y cada uno de estos problemas.

Entre los problemas encontrados al utilizar ArgoUML están:

1. Cuando se recorre la lista de tipos en un atributo de una clase, hay que tener mucho cuidado en hacer todo con el *mouse*, sin parar en ningún otro tipo que el deseado. Cuando es seleccionado cualquier tipo, aunque sea de manera temporal, dicho tipo es agregado a la estructura del modelo y solamente se deben agregar aquellos tipos de datos que AndroMDA maneja.

   Dejar dichos tipos tal como son agregados en forma automática al “descansar” en la lista de tipos disponibles mientras se busca el tipo deseado, produce clases sin implementación que se sobreponen a los tipos estándar, por ejemplo de Java.

   Para corregir el problema, es suficiente con borrar toda la estructura de paquetes de tipos que se haya generado en forma automática (y que no venga con el *perfil* de AndroMDA, evidentemente), como por ejemplo el paquete `java.*` [Foros_3].

2. En los diagramas de actividad, al crear un nuevo “*signal event*” en una transición no aparecía el

tipo `Long:datatype`. Este tipo es crucial para pasar el ID de un estado a otro. Dicho problema fue resuelto en la version 0.25.4 [Foros_4].

3. En ocasiones, al tratar de abrir nuevamente un archivo .zargo ya modificado, aparece un mensaje de error que dice:

```
XMI format error : org.argouml.model.XmiException:
javax.jmi.xmi.MalformedXMIException:
org.netbeans.lib.jmi.util.DebugException:
The same value of xmi.idref used second time:
-64--88--109-42-6c18a589:118dcc9d48a:-8000:0000000000000CB7,
file:/C:/DOCUME~1/Ely/CONFIG~1/Temp/zargo_model_11647.xmi
If this file was produced by a tool other than ArgoUML, please check to
make sure that the file is in a supported format, including both UML and
XMI versions.
```

   Ese error ya fue solucionado para versiones de desarrollo posteriores a la 0.25.4 Es posible recuperar el archivo dañado eliminando una sección del archivo xmi contenido dentro del archivo .zargo [Foros_5].

**Modificaciones a la configuracion de Maven2 para generar la estructura del archivo `.ear` con el sub-proyecto `aop [Foros_2]`.**

Resumiendo, las modificaciones realizadas a la configuración de Maven2 son:

- En el archivo pom.xml, ubicado en el directorio raíz del proyecto, fue necesario indicar que también existe un nuevo módulo llamado "aop", por lo que fue modificada la sección que contenía originalmente:

```
<project ...
...
    <modules>
        <module>mda</module>
        <module>common</module>
        <module>core</module>
        <module>web</module>
        <module>app</module>
    </modules>
...
```

  por:

```
<project ...
...
    <modules>
        <module>mda</module>
        <module>common</module>
        <module>core</module>
        <module>aop</module> <!-- ESL!!! -->
        <module>web</module>
        <module>app</module>
    </modules>
...
```

- En el archivo app\pom.xml, que es el encargado de reunir todos los sub-proyectos, fue agregada una nueva dependencia:

```
<project ...
...
    <dependencies>
        <!-- ESL!!! AOP --> <dependency>
            <groupId>${pom.groupId}</groupId>
            <artifactId>bolsaDeTrabajo-aop</artifactId>
            <version>${pom.version}</version>
        </dependency>
...
```

- Para que sea generado el jar con extensión `.aop`, fue necesario agregar lo siguiente en app\pom.xml:

```
<project ...
...
 <build>
...
 <plugins>
...
            <plugin>
```

```
...
                <configuration>
...
                    <modules>
                        <!-- ESL!!! AOP --> <javaModule>
                            <groupId>${pom.groupId}</groupId>
                            <artifactId>bolsaDeTrabajo-aop</artifactId>
                            <bundleFileName>
                                ${application.id}-aop-${pom.version}.aop
                            </bundleFileName>
                            <includeInApplicationXml>
                                true
                            </includeInApplicationXml>
                        </javaModule>
...
```

Es sumamente importante que sea agregada una referencia de este nuevo jar en el archivo application.xml, y que sea el primero en la lista. De otra forma el servidor JBoss no será capaz de aplicar el código AOP al resto del código. Es por esto que este nuevo módulo fue definido en la primera posición de la lista de módulos y contiene la directiva `includeInApplicationXml` [Maven2_5].

- En el archivo web\pom.xml:

```
<project ...
...
    <dependencies>
...
        <dependency> <!-- ESL!!! -->
            <groupId>${pom.groupId}</groupId>
            <artifactId>bolsaDeTrabajo-aop</artifactId>
            <version>${pom.version}</version>
            <scope>provided</scope>
        </dependency>
...
```

El ámbito (*scope*) "provided" significa que este sub-proyecto espera recibir dicha dependencia en tiempo de ejecución. De no tener esto especificado podría generarse un bucle infinito, ya que antes de pasar al módulo "web" ya se compiló el módulo "aop". La definición de los diferentes ámbitos permitidos puede ser encontrada en [Maven2_4].

- Y por supuesto que hay que crear el archivo aop\pom.xml. Dicho archivo no se muestra ya que es muy parecido a los otros pom.xml, generados por AndroMDA desde la creación del proyecto.

  Algo que si es importante resaltar es que al definir aspectos, son necesarias algunas bibliotecas de jboss-aop, las cuales obviamente no son parte del proyecto estándar generado por AndroMDA. Como consecuencia, fue necesario agregar una dependencia hacia un jar que se encuentra en el servidor de aplicaciones JBoss configurado con JBoss-AOP:

```
<project ...
...
    <dependencies>
...
        <dependency>
```

```
            <groupId>${pom.groupId}</groupId>
            <artifactId>aop-jar</artifactId>
            <version>${pom.version}</version>
          <scope>system</scope>
          <systemPath>
            ${jboss.home}/server/default/deploy/
                        jboss-aop-jdk50.deployer/jboss-aop-jdk50.jar
          </systemPath>
        </dependency>
...
```

JBoss permite arrancar con diferentes configuraciones de servidor. Ya que es fija la ruta en esta definición de `<systemPath>`, en caso de querer utilizar alguna configuración diferente de la estándar, será necesario corregir esta ruta.

El ámbito (*scope*) debería ser "provided" ya que el servidor de aplicaciones es quien proveerá esa biblioteca, pero con ése no se puede especificar “systemPath” [Maven2_4].

Según algunos mensajes en los foros de discusión, al parecer para Maven 2.1 van a descontinuar el uso de “systemPath”. En el futuro será necesario buscar cómo agregar este jar.

**Definición de *Introducciones*, *Mixins*, *Advices*, *Pointcuts* y *Aspectos***

La implementación de los elementos de AOP requiere de una definición en el archivo jboss-aop.xml. Si se trata de un código *advice*, debe estar contenido en un archivo en el que se especifiquen también todos los *advices* relacionados con una misma clase. Si se trata de una introducción-mixin, son necesarios dos archivos: el que define la interfaz, y el que define la implementación de dicha interfaz.

Para agregar la nueva funcionalidad en la clase `ServicioConsultarVacantes`, por ejemplo, se especifica lo siguiente en el archivo jboss-aop.xml:

```
<!-- ********************************************************************** -->
<!-- INICIO: com.mitalteli.bolsaDeTrabajo.service.ServicioConsultarVacantesBase -->

   <introduction class="com.mitalteli.bolsaDeTrabajo.service.
                           ServicioConsultarVacantesBase">
      <mixin>
         <interfaces>
             com.mitalteli.bolsaDeTrabajo.useCaseSlices.specific.consultarVacantes.
                 IAspectoServicioConsultarVacantesBase
         </interfaces>
         <class>
             com.mitalteli.bolsaDeTrabajo.useCaseSlices.specific.consultarVacantes.
                 MixinAspectoServicioConsultarVacantesBase
         </class>
         <construction>
             new com.mitalteli.bolsaDeTrabajo.useCaseSlices.specific.
                  consultarVacantes.MixinAspectoServicioConsultarVacantesBase(this)
         </construction>
      </mixin>
   </introduction>
```

Como se puede ver, es utilizada la etiqueta `<interfaces>` para especificar el nombre del archivo que contiene la interfaz a ser agregada a la clase que se quiere modificar y es utilizada la etiqueta `<class>` para especificar el nombre del archivo que contiene la implementación de dicha interfaz.

En el caso de la clase "`EmpresaDaoBase`", que es generada en forma automática por AndroMDA, es necesario modificar el comportamiento de dos de sus métodos. Para lograr esto es utilizada la siguiente definición en el archivo jboss-aop.xml:

```
<!-- ********************************************************************** -->
<!-- INICIO: com.mitalteli.bolsaDeTrabajo.domain.EmpresaDaoBase -->

   <aspect class="com.mitalteli.bolsaDeTrabajo.useCaseSlices.specific.
                   consultarVacantes.AspectoEmpresaDaoBase"
           scope="PER_VM"/>

   <bind pointcut="execution(* com.mitalteli.bolsaDeTrabajo.domain.
                   EmpresaDaoBase->toEmpresaVO(com.mitalteli.bolsaDeTrabajo.
                   domain.Empresa, com.mitalteli.bolsaDeTrabajo.vo.EmpresaVO))">
      <advice name="methodToEmpresaVOAdvice" aspect="com.mitalteli.bolsaDeTrabajo.
                   useCaseSlices.specific.consultarVacantes.AspectoEmpresaDaoBase"
                   />

   </bind>
   <bind pointcut="execution(* com.mitalteli.bolsaDeTrabajo.domain.
                   EmpresaDaoBase->empresaVOToEntity(com.mitalteli.bolsaDeTrabajo.
```

```
                    vo.EmpresaVO, com.mitalteli.bolsaDeTrabajo.domain.Empresa,
                    boolean))">
      <advice name="methodEmpresaVOToEntityAdvice" aspect="com.mitalteli.
                    bolsaDeTrabajo.useCaseSlices.specific.consultarVacantes.
                    AspectoEmpresaDaoBase"/>
   </bind>
```

Como se puede ver, antes que nada hay que especificar el archivo que contiene el código de los *advices* utilizando la etiqueta `<aspect>`.

En este caso el primer `<bind>` define un pointcut de tipo "execution" es decir, que la intercepción se llevará a cabo cuando un método o constructor sea llamado. El "*" significa que el método a ser interceptado puede devolver cualquier tipo de objeto o no devolver nada. El método a ser interceptado está dentro de la clase `com.mitalteli.bolsaDeTrabajo.domain.EmpresaDaoBase`. Se llama `toEmpresaVO`. Tiene definidos exactamente dos parámetros: `(com.mitalteli.bolsaDeTrabajo.domain.Empresa, com.mitalteli.bolsaDeTrabajo.vo.EmpresaVO)`. El código del *advice* está definido en `methodToEmpresaVOAdvice`. El *advice* se encuentra en el aspecto `com.mitalteli.bolsaDeTrabajo.useCaseSlices.specific.consultarVacantes.AspectoEmpresaDaoBase`.

La sintaxis de los puntos de corte se puede encontrar en `[JBoss_AOP_6]`.

Sí es posible especificar tanto introducción-mixin, como múltiples *advices* para una misma clase.

El archivo `.ear` generado en la aplicación desarrollada para esta tesis, se llama `bolsaDeTrabajo-1.0-SNAPSHOT.ear`, y contiene la siguiente estructura:

```
bolsaDeTrabajo-1.0-SNAPSHOT.ear
├──META-INF
│   └──application.xml
│  ...
├──bolsaDeTrabajo-aop-1.0-SNAPSHOT.aop
│   └──META-INF
│       └──jboss-aop.xml
│  ...
├──bolsaDeTrabajo-common-1.0-SNAPSHOT.jar
│  ...
├──bolsaDeTrabajo-core-1.0-SNAPSHOT.jar
│  ...
├──bolsaDeTrabajo-web-1.0-SNAPSHOT.war
│  ...
```