INTRODUCCIÓN

La ingeniería del software ciertamente ha evolucionado desde sus comienzos. Al principio, se tenía un código en el que no existía la separación de conceptos, datos y funcionalidad, se mezclaban sin una estrategia que los dividiera claramente.

La Programación Orientada a Objeto, donde se fuerza el encapsulamiento y la abstracción, a través de una unidad que captura tanto funcionalidad como comportamiento y estructura interna, a esta entidad se la conoce como clase, sin embargo existen conceptos que no pueden encapsularse dentro de una unidad funcional debido a que atraviesan todo el sistema o varias partes de él. Algunos de estos conceptos son: sincronización, manejo de memoria, distribución, chequeo de errores, seguridad o redes, entre otros.

Las descomposiciones en la mayoría de software, implementado con Programación Orientada a Objetos no soportan una completa separación de conceptos, característica clave para manejar un software entendible y evolucionadle. Se puede afirmar entonces que las técnicas tradicionales no soportan de una manera adecuada la separación de las propiedades de aspectos distintos a la funcionalidad básica y que esta situación tiene un impacto negativo en la calidad del software.

Para este problema nace la Programación Orientada a Aspectos(POA), permite a los programadores escribir, ver y editar un aspecto diseminado por todo el sistema como una entidad por separado, de una manera inteligente, eficiente e intuitiva; es una nueva metodología de programación que aspira a soportar la separación de las propiedades para los conceptos antes mencionados.

Esto implica separar la funcionalidad básica y los aspectos a través de mecanismos que permitan abstraerlos y componerlos para formar todo el sistema,

finalmente la Programación Orientada a Aspectos es un desarrollo que sigue a la Programación Orientada a Objetos.
El desarrollo de un software prototipo de ventas para la aplicación del paradigma de Programación Orientado a Aspectos, que servirá como ejemplo para los estudiantes de la Carrera de Ingeniería en Informática y Sistemas Computacionales de CIYA de la UTC, para conocer sobre el paradigma de POA.

Toda la información antes descrita impulsó y enmarcó la construcción del software VENTSOFT 1.0 para aportar con el estudio de la Programación Orientada a Aspectos en la carrera, dando lugar a que la Universidad esté al margen de las nuevas tecnologías en lo que respecta a desarrollo e ingeniería de software. Unido a esto la investigación aplicada, la investigación bibliográfica, la investigación de campo y la investigación experimental, permitieron un caminar eficaz de este proyecto investigativo. Métodos investigativos como el científico, deductivo, inductivo, e inductivo-deductivo, técnicas como la entrevista y encuesta; permitieron analizar datos y llegar a concluir esta tesis científica-técnica.

La población participe para la investigación lo cual se aplico a través de la encuesta y entrevista utilizada para la adquisición de información promotora del software prototipo VENTSOFT 1.0; estaba planteada de 102 personas, pero en la práctica fue de 100 personas exactamente, distribuidas entre estudiantes de sexto, séptimo, octavo ciclo y los respectivos docentes de la materia de POO, que es la que abarca el paradigma de Programación Orientado a Aspectos.

El ***VENTSOFT 1.0*** es el nombre del software desarrollado por esta tesis de investigación; y está formado por la unión de las cuatro primeras letras de las palabras SOFTWARE y VENTAS. Es un nombre adecuado para la herramienta, ya que esta aplicación está destina a la automatización del proceso de una venta, pero inicialmente consignada para los estudiantes de la especialidad de Informática y Sistemas Computacionales de la UTC como aplicación del Paradigma de Programación Orientada a Aspectos

Como el VENTSOFT 1.0 está implementado en el Laboratorio de Informática de la Carrera de Ingeniería en Informática y Sistemas Computacionales de la UTC; será aplicativo para el estudio de los alumnos que cursen esta carrera de pregrado, enfocado en la cátedra de Programación Orientada a Objetos.

Software similares han sido desarrollados en varios centros de educación superior, en Cotopaxi es el segundo proyecto que se crea con el paradigma de Programación Orientada a Aspectos, en la Universidad Técnica de Cotopaxi; el primero.

Múltiples actividades tanto investigativas, en complicidad con lo aprendido durante la malla curricular de la carrera de Informática y Sistemas Computacionales fueron de imperante necesidad en el transcurso del desarrollo del VENTSOFT 1.0. El primer paso realizado fue la búsqueda de un problema y por ende una solución que satisfaga tanto al problema como al requerimiento científico que implica culminar una ingeniería.

Para concluir se describe esta tesis de la siguiente manera: en el capítulo uno se cita la información teórica sobre: Programación Orientada a Aspectos (POA), Importancia de POA en la actualidad, Lenguajes Orientada Aspectos, Java, AspectJ, conceptos, etc. El capítulo dos describe los análisis de resultados de las encuestas realizadas a los estudiantes, y entrevista a los respectivos docentes de la materia de Programación Orientada a Objetos, que es la que abarca el paradigma de Programación Orientado a Aspectos. De la carrera de Informática y Sistemas Computacionales. Descripción de esta especialización, entre otras cosas irrelevantes. El capítulo tres establece el diseño, desarrollo, y todo lo referente a la implementación del software prototipo VENTSOFT 1.0, guía de Usuario Al final de este capítulo están presentes las conclusiones del trabajo, recomendaciones, bibliografía y Anexos.

CAPÍTULO I

“PARADIGMA DE LA PROGRAMACIÓN ORIENTADA A ASPECTOS”

1.1 Introducción

Generalmente el desarrollo de una aplicación involucra varias tareas que se deben realizar; tareas que pueden considerarse “principales”, y que son típicamente detalladas por los usuarios como parte del análisis funcional de requerimientos, adicionalmente existen tareas que pueden considerarse “servicios comunes”, generalmente no detalladas en el análisis funcional. Es habitual que esta clase de “servicios comunes” deba ser realizada en forma similar pero independiente en diversas partes del código de programación. Ejemplos de estos “servicios comunes” pueden ser la necesidad de generar registros de auditoría (logging), accesos a bases de datos, temas relacionados con las seguridad, temas relacionados con la concurrencia del acceso a cierta información, etc.

Cada una de estas tareas es considerada una “incumbencia” (*“concern”*); la cual debe incumbir únicamente el código que implementa esta tarea. La “separación de incumbencias” es, por lo tanto, deseable en cualquier desarrollo de software. Sin embargo, en los lenguajes de programación típicos ya sea procedurales u orientados a objetos, es difícil, o imposible, separar claramente las incumbencias de los “servicios comunes” de las incumbencias “principales”, teniendo como consecuencia, que las tareas de los “servicios comunes” queden dispersas dentro del código de las incumbencias principales. Esto es conocido como

"incumbencias transversales" (*"crosscutting concerns"*).

Las incumbencias son los diferentes temas, asuntos o *aspectos* de los que es necesario ocuparse para resolver un problema determinado; separando las incumbencias, se disminuye la complejidad a la hora de tratarlas, y se gana en claridad, adaptabilidad, mantenibilidad, extensibilidad y reusabilidad.
Los conceptos y tecnologías conocidas con el nombre de "Programación Orientada a Aspectos" (POA) buscan resolver el problema de la "separación de incumbencias", de una manera sistemática, clara y eficiente. (Ing. José Joskowicz – Universidad de Vigo).

POA es conocida también como AOP, "Aspect-Oriented Programming" o AOSD, (Aspect-Oriented Software Development).

La Programación Orientada a Aspectos busca la separación de incumbencias o aspectos. Al trabajar con aspectos separados e independientes en el desarrollo del software y después unir todos estos módulos y formar la aplicación requerida dando origen a un software más eficiente y sobre todo facilitando las tareas de ingeniería de software al programador o informático.

1.1.1. Reseña Histórica

El término "separación de incumbencias" fue introducido en la década de 1970, por Edsger W. Dijkstra. Significa simplemente, que la resolución de un problema dado involucra varios aspectos o incumbencias, los que deben ser identificados y analizados en forma independiente.

La "Programación Adaptativa" fue el concepto predecesor de la "Programación Orientada a Aspectos". Las ideas originales de la "Programación Adaptativa" fueron introducidas a principio de la década de 1990 por el grupo Demeter, siendo Karl Lieberherr uno de sus ideólogos. Los conceptos de POA fueron introducidos en 1997 por Gregor Kiezales y su grupo.

Actualmente, hay ya varias implementaciones de lenguajes orientados a aspectos, el nuevo paradigma aún continúa en desarrollo, y todavía está abierta la investigación hacia varias áreas que recién están siendo exploradas, como por ejemplo llevar los "aspectos" a las etapas de diseño.

En épocas pasadas el desarrollo de un software se complicaba demasiado a la hora de programar en herramientas específicamente de lenguajes orientados a objetos, tanto que el diseño del código se convertía en ocasiones en algo repetitivo para cierto módulo o tarea en la que se aplicaba el software realizado. Tal es el caso de un sistema que trabaja con múltiples usuarios y requiera una contraseña de ingreso, actualizar y eliminar datos, etc.

1.1.2 Importancia de POA en la actualidad

El uso de la POA se extiende día a día, aunque a nivel empresarial todavía no se considera como una opción a tener en cuenta, quizás porque se desconocen las grandes posibilidades que ofrece y por su falta de estandarización, aunque a este respecto parece que comienzan a surgir movimientos de unificación como es promovido por los creadores de AspectJ y AspectWerkz (su competidor directo) que van a unir sus fuerzas para crear un único framework Java para POA. Podríamos comparar el estado actual de la POA como el de la POO hace quince años.

Parece que esta década puede representar la madurez definitiva de la POA. Ya empiezan a surgir guías de diseño orientado a aspectos que incluyen mas los usos de la POA, soluciones elegantes orientadas a aspectos para determinados problemas y son muchos los proyectos que están en marcha y que se basan en ideas orientadas a aspectos. Como ejemplo de este fenómeno en auge tenemos la aparición del framework *Spring* que se basa en ideas orientadas aspectos, y que está realizando una importante contribución a la popularización de este joven paradigma de programación.

A continuación se listan los beneficios más importantes que ofrece su uso:

Un diseño más modular. Cada competencia se implementa de forma independiente con un acoplamiento mínimo, eliminando el código duplicado y dando lugar un código más ordenado, fácil de entender y mantener.

Mejora la trazabilidad puesto que cada módulo tiene sus responsabilidades bien definidas.

Fácil evolución del sistema. Si se quiere añadir un nuevo aspecto, no será necesario modificar el código que implementa la funcionalidad básica del sistema, reduciendo el tiempo de respuesta a nuevos requisitos.

Aumenta la reutilización. Cada competencia transversal está encapsulada en un aspecto y los módulos principales del sistema no conocen su existencia, por lo que el acoplamiento es muy bajo, lo que facilita la reutilización.

Reduce el tiempo de mercado. La clara separación de intereses permite a los desarrolladores trabajar con mayor destreza y rapidez, mejorando la productividad. La reutilización de código también reducirá el tiempo de desarrollo. La fácil evolución permite satisfacer rápidamente nuevos requisitos. Todo esto hace que el desarrollo y salida al mercado del producto se realice en un periodo de tiempo más reducido.

Reduce costes de futuras implementaciones. Para añadir un nuevo aspecto ya no hay que modificar el código base con lo que se reduce el coste de implementar nuevas competencias transversales. Esto permite que los programadores estén más centrados en implementar la funcionalidad básica del sistema por lo que aumentará la productividad y la implementación de la funcionalidad básica también disminuirá.

Retrasar decisiones de diseño, primero implementa la funcionalidad básica y en un futuro añade la funcionalidad secundaria del sistema.

Se puede decir que la programación orientada a aspectos es un proceso de programación; que proporcionan en conjunto con la utilidad de una herramienta base y la herramienta de programación orientada a aspectos, el mejor camino para la construcción y creación de sistemas y aplicaciones informáticas; optimizando la etapa de programación de sistemas, mejorando la calidad, evitando errores y trabajando en menor tiempo que con la programación tradicional.

1.2 Consideraciones generales de la POA

La idea central que persigue la POA es permitir que un programa sea construido describiendo cada concepto separadamente.
El soporte para este nuevo paradigma se logra a través de una clase especial de lenguajes, llamados lenguajes orientados a aspectos (LOA), los cuales brindan mecanismos y constructores para capturar aquellos elementos que se diseminan por todo el sistema, a estos elementos se les da el nombre de aspectos. Una definición para tales lenguajes sería: Los LOA son aquellos lenguajes que permiten separar la definición de la funcionalidad pura de la definición de los diferentes aspectos.

Los LOA deben satisfacer varias propiedades deseables, entre ellas:

- Cada aspecto debe ser claramente identificable.
- Cada aspecto debe auto-contenerse.
- Los aspectos deben ser fácilmente intercambiables.
- Los aspectos no deben interferir entre ellos.
- Los aspectos no deben interferir con los mecanismos usados para definir y evolucionar la funcionalidad, como la herencia.

1.2.1 ¿Qué es un aspecto?

La definición formal de "Aspecto" ha evolucionado desde su concepción hasta el momento. Una definición inicial, aunque todavía no se manejaba semánticamente el término "aspecto", fue introducida por Karl Lieberherr: "Un aspecto es una unidad que se define en términos de información parcial de otras unidades".

Gregor Kiezales y su grupo brindan una primera definición de "aspecto" en, Una propiedad que debe ser implementada.

1. ***Como un componente***: si puede ser claramente encapsulada dentro de un procedimiento generalizado. Un elemento es claramente encapsulado si está bien localizado, es fácilmente accesible y resulta sencillo componerlo.

2. ***Como un aspecto***: si no puede ser claramente encapsulado en un procedimiento generalizado. Los aspectos tienden a ser propiedades que afectan la performance o la semántica de los componentes en forma sistemática. (Ejemplo: manejo de memoria, distribución, sincronización, etc.)

La definición más comprendida de aspecto, es la de Gregor Kiezales descrita en mayo de 1999.

"Un ***aspecto*** es una unidad modular que se disemina por la estructura de otras unidades funcionales. Los aspectos existen tanto en la etapa de diseño como en la de implementación. Un aspecto de diseño es una unidad modular del diseño que se entremezcla en la estructura de otras partes del diseño. Un aspecto de implementación es una unidad modular del programa que aparece en otras unidades modulares del programa".

Se desprende de esta definición, que los aspectos de una aplicación son aquellos módulos que generan "incumbencias transversales", es decir, los módulos que están diseminados por el resto de las unidades funcionales.

Identificando los aspectos, y aplicando las técnicas desarrolladas en la POA es posible, realizar adecuadamente la "separación de incumbencias".
Es de resaltar que la definición de aspecto no hace referencia al tipo de programación en la que se implemente (orientada a objetos o procedural), por lo que el concepto, como tal, aplica a ambos.

En una primera instancia puede ser fácil asociar a aspectos los "servicios comunes", incluyendo *aspectos* de auditoría (logging), *aspectos* de seguridad, *aspectos* de concurrencia, etc. Sin embargo, el concepto es mucho más genérico. Las herramientas que soportan POA manejan tanto las "clases" como los "aspectos" como unidades funcionales naturales, lo que permite total generalidad en el uso de los aspectos.

La definición proporcionada por el científico informático Gregor Kiezales es la más popular y la más acorde en lo referente a un aspecto en programación y desarrollo de software, primero por ser una concepción del gestor y principal mentor del paradigma POA y segundo por abarcar en su totalidad el significado de aspecto.

1.2.2 Fundamentos del POA

Los tres principales requerimientos de la POA son:

- Un lenguaje para definir la funcionalidad básica, conocido como lenguaje base o componente. Podría ser un lenguaje de propósito general, tal como es C++, Java.

- Uno o varios lenguajes de aspectos, para especificar el comportamiento de los aspectos. (COOL, para sincronización, RIDL para distribución, AspectJ, de propósito general.)

- Un tejedor de aspectos (Weaver), que se encargará de combinar los lenguajes. Tal proceso se puede retrasar para hacerse en tiempo de ejecución o en tiempo de compilación.

Los lenguajes orientados a aspectos definen una nueva unidad de programación de software para encapsular aquellos conceptos que cruzan todo el código.

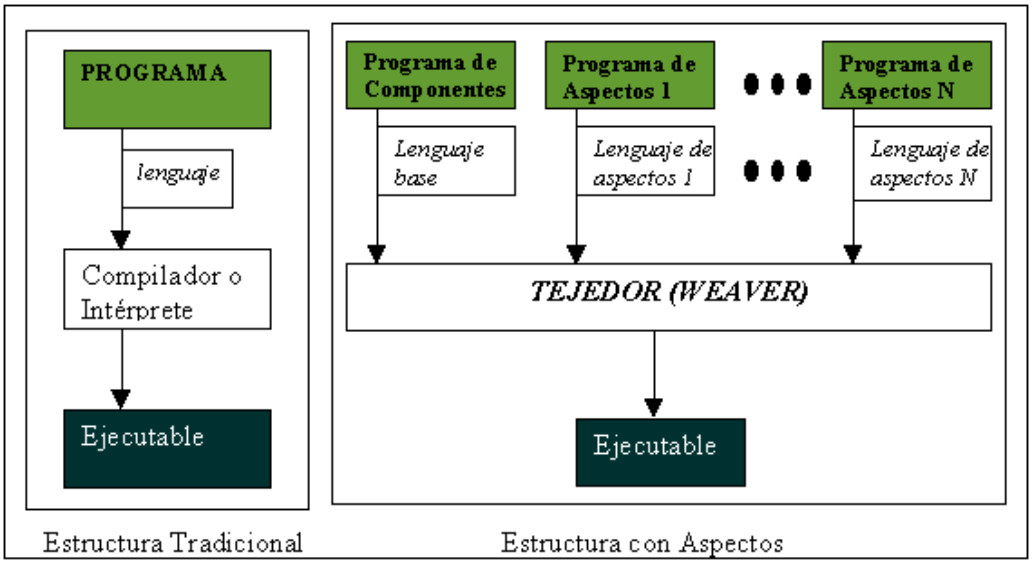
A la hora de "tejer" los componentes y los aspectos para formar el sistema final, es claro que se necesita una interacción entre el código de los componentes y el código de los aspectos. También es claro que esta interacción no es la misma interacción que ocurre entre los módulos del lenguaje base, donde la comunicación está basada en declaraciones de tipo y llamadas a procedimientos y funciones. La POA define entonces una nueva forma de interacción, atreves de los puntos de enlace (join points).

Los puntos de enlace brindan la interfaz entre aspectos y componentes; son lugares dentro del código donde es posible agregar el comportamiento adicional que destaca a la POA. Aún falta introducir el encargado principal en el proceso de la POA. Este encargado principal conocido como tejedor debe realizar la parte final y más importante: "tejer" los diferentes mecanismos de abstracción y composición que aparecen tanto en los lenguajes de aspectos como en los lenguajes de componentes, guiado por los puntos de enlace.

La estructura general de una implementación basada en aspectos difiere de la estructura de una implementación tradicional. Una implementación tradicional consiste de un lenguaje, un compilador, o intérprete para ese lenguaje, y finalmente, un programa escrito en ese lenguaje que implemente la aplicación.

En cambio, una implementación basada en la POA consiste en:
Un lenguaje base que permite implementar la funcionalidad básica. Luego, uno o más lenguajes de aspectos para la implementación de los mismos, y un *tejedor* de aspectos encargado de la combinación de los lenguajes. Por último, se requiere el programa escrito en el lenguaje base, que implementa los componentes funcionales, y uno o más programas de aspectos que implementan a estos.
Gráficamente, se pueden comparar ambas estructuras, como queda reflejado en la Figura Nº 1.1

FIGURA Nº 1.1 ESTRUCTURA TRADICIONAL Y ESTRUCTURA APLICANDO POA



1.2.3 Lenguajes Orientados a Aspectos

La idea central de la POA es permitir que un programa sea construido describiendo cada concepto (incumbencia) separadamente. El soporte para este nuevo paradigma se logra a través de una nueva clase de lenguajes, llamados ***lenguajes orientados a aspectos*** (LOA), los cuales brindan mecanismos para capturar y declarar aquellos elementos que se diseminan por todo el sistema (***aspectos***).

Una definición para tales lenguajes sería: Los **LOA** son aquellos lenguajes que permiten separar la definición de la funcionalidad "principal" de la definición de los diferentes aspectos. Los LOA deben satisfacer varias propiedades deseables:

Debe ser claramente identificable, auto contenerse, ser fácilmente modificables, no deben interferir entre ellos, y tampoco interferir con los mecanismos usados para definir y evolucionar la funcionalidad principal, como la herencia.
Los LOA distinguen dos enfoques diferentes en el diseño de los lenguajes orientados a aspectos: los lenguajes orientados a aspectos de dominio específico y los lenguajes orientados a aspectos de propósito general.

Los ***LOA de dominio específico*** han sido diseñados para soportar algún tipo particular de aspectos, como por ejemplo la concurrencia, sincronización o distribución. Este tipo de lenguajes suelen tener un nivel de abstracción mayor que el lenguaje base y permiten representar los conceptos específicos del aspecto a un nivel de representación más elevado.

Algunos de estos lenguajes necesitan imponer restricciones en el lenguaje base, de manera de poder garantizar que las incumbencias que son tratadas en los aspectos no pueden ser programadas en los componentes, evitando de esta manera inconsistencias o funcionamientos no deseados. Por ejemplo, si el lenguaje de aspectos se especializa en la concurrencia o sincronización, puede requerir que sean deshabilitadas las primitivas del lenguaje base que puedan ser utilizadas para estas funciones (un ejemplo de este tipo de LOA es COOL, el cual se describirá más adelante).

Los **LOA de propósitos generales** han sido diseñados para soportar cualquier tipo de aspectos. Este tipo de lenguajes no pueden imponer restricciones en el lenguaje base. Generalmente tienen el mismo nivel de abstracción que el lenguaje base, y soportan las mismas instrucciones o primitivas, ya que, en principio, cualquier código debería poderse escribir en los aspectos desarrollados con estos

lenguajes (un ejemplo de este tipo de LOA es AspectJ, el cual se describirá más adelante).

Los LOA de propósitos general tienen la clara ventaja de ser, tal como su nombre indica, generales, capaces de ser utilizados para desarrollar con ellos cualquier tipo de aspecto. Sin embargo, tienen también una desventaja de que no garantizan la separación de funcionalidades. Al no poder restringir las instrucciones o primitivas en la programación de los componentes, no puede garantizarse que las tareas que deberían programarse como aspectos no sean programados dentro de los componentes. Esto queda librado al programador.

Los LOA de dominio específico fuerzan a programar las tareas de aspectos dentro de éstos, ya que en el lenguaje base se restringe el uso de las instrucciones relacionadas con las funcionalidades de aspectos.

De esta manera se ha resaltado los criterios y contenidos científicos que proporcionan estos autores en su documento, en especial sobre los lenguajes orientados a aspectos; son de tal interés e importancia que los he ubicado en la tesis porque sirven de fuente de conocimiento y guía para el desarrollo de la aplicación. El saber diferenciar los distintos lenguajes de Programación Orientada a Aspectos que existen y que papel desempeñan, facilitan el trabajo de un investigador del POA.

Los principales lenguajes orientados a aspectos son:

1.2.3.1 JPAL

La principal característica de esta herramienta es que los puntos de enlace son especificados independientemente del lenguaje base. Estos puntos de enlace independientes reciben el nombre de Junction Point (JP). La herramienta se llama JPAL que significa Junction Point Aspect Language, esto es, Lenguaje de Aspectos basados en JP.

Usar programas escritos en JPAL para describir nuevos lenguajes de aspectos facilita para ese lenguaje el desarrollo de tejedores. El tejedor JPAL genera un esquema del tejedor de aspectos llamado Esquema del Tejedor. Este esquema tiene un mecanismo que automáticamente conecta el código base con los programas de aspectos en puntos de control llamados acciones.

Resumiendo, JPAL, un descendiente de AspectJ, tiene la particularidad de separar los puntos de enlace, que son independientes del lenguaje base, de sus acciones asociadas que dependen de decisiones de implementación. Esta separación permite generar un Esquema de Tejedor para cualquier lenguaje de aspectos. Este esquema ofrece un puente entre el control de la ejecución y la ejecución de la acción. Su principal aplicación es para la implementación de sistemas distribuidos.

1.2.3.2 D

D es un ambiente de lenguajes de aspectos para la programación distribuida. Se llama ambiente de lenguajes, en vez de lenguaje, porque consiste en realidad de dos lenguajes: COOL, para controlar la sincronización de hilos (threads) y RIDL para programar la interacción entre componentes remotos.

Estos dos lenguajes se diseñaron de manera independiente de un lenguaje componente y establece un número de condiciones sobre el lenguaje componente.

El diseño de D es semi-independiente del lenguaje componente, ya que D impone requerimientos sobre el lenguaje componente que satisfacen naturalmente los lenguajes orientados a objetos. Luego el lenguaje componente puede ser cualquiera mientras sea orientado a objetos. En teoría podría ser implementado con C++, Smalltalk, CLOS, Java o Eiffel.

La sincronización es un ítem que D captura como un aspecto separado. En donde las clases no pueden tener código para el control de concurrencia. Un programa puede tener varios hilos concurrentes.

La estrategia de sincronización por defecto sin la intervención de COOL, es que no hay estrategia: en la presencia de múltiples hilos todos los métodos de todos los objetos pueden ejecutarse concurrentemente, la integración remota es el otro ítem que D captura como un aspecto separado, luego las clases tampoco pueden tener código para la comunicación remota. Un programa sin la intervención de RIDL no es un programa distribuido, la estrategia de comunicación por defecto es que no hay ninguna estrategia de comunicación, entonces un programa es distribuido sólo si utiliza RIDL.

1.2.3.3 COOL

COOL (COOrdination Language) es un lenguaje de dominio específico desarrollado por Xerox, cuya finalidad es tratar los aspectos de sincronismo entre hilos concurrentes, el lenguaje base que utiliza es Java, pero en una versión modificada en la que se eliminan los métodos *"wait"*, *"notify"* y *"notifyAll"* y la palabra clave *"synchronized"* para evitar que se produzcan inconsistencias al intentar sincronizar los hilos en el aspecto y en los componentes (clases en este caso). En COOL la sincronización de los hilos se especifica de forma declarativa y, más abstracta que la correspondiente codificación en Java.

1.2.3.4 RIDL

RIDL (Remote Interaction and Data transfers aspect Language) es un LOA de dominio específico que maneja la transferencia de datos entre diferentes espacios de ejecución.

Un programa RIDL consiste de un conjunto de módulos de "portales". Un portal es el encargado de manejar la interacción remota y la transferencia de datos de la clase asociada a él, y puede asociarse como máximo a una clase.

1.2.3.5 AspectC

AspectC es un LOA de propósito general que extiende C. Es similar a AspectJ, pero sin soporte para la programación orientada a objetos.

El código de aspectos interactúa con la funcionalidad básica en los límites de una llamada a una función y puede ejecutarse antes, después o durante dicha llamada. Como el lenguaje C es estático, el tejedor de AspectC es también estático.

1.2.3.6 AspectS

AspectS extiende el ambiente Squeak/Smalltalk para permitir un sistema de desarrollo orientado a aspectos. Squeak es una implementación abierta y portable de Smalltalk-80 cuya máquina virtual está completamente escrita en Smalltalk. Principalmente AspectS está basado en dos proyectos anteriores: AspectJ de Xerox Parc y el MethodWrappers de John Brant, que es un mecanismo poderoso para agregar comportamiento a un método compilado en Squeak.

AspectS, un lenguaje de aspectos de propósito general, utiliza el modelo de lenguaje de AspectJ y ayuda a descubrir la relación que hay entre los aspectos y los ambientes dinámicos. Soporta programación en meta-nivel manejando el fenómeno de *Código Mezclado* a través de módulos de aspectos relacionados. Está implementado en Squeak sin cambiar la sintaxis ni la máquina virtual.

1.2.3.7 AspectC++

AspectC++ es un LOA de propósito general que extiende el lenguaje C++ para soportar el manejo de aspectos. Sintácticamente un aspecto en este lenguaje es

muy similar a una clase en C++. Además de funciones un aspecto puede definir "avisos" (*"advice"*); luego de la palabra clave "advice", una expresión de corte (*"pointcut expresion"*) define el punto donde el aspecto modificará al programa (es decir, los "puntos de enlace" o *"join points"*).

Pueden utilizarse expresiones de corte para identificar un conjunto de puntos de enlaces. Se componen a partir de expresiones de corte y un conjunto de operadores algebraicos. La declaración de los avisos es utilizada para especificar código que debe ejecutarse en los puntos de enlace determinados por la expresión de corte. Los avisos pueden insertarse antes, después o durante la ejecución de los métodos donde se insertan.
El tejedor de AspectC++, ac++ transforma programas escritos en AspectC++ a código de C++, por lo que puede ser utilizado con cualquier compilador de C++ como g++ o Microsoft C++ de VisualStudio.NET.

1.2.3.8 MALAJ

MALAJ (Multi Aspect Language for Java) es un LOA de dominio especifico focalizado en la sincronización y reubicación. MALAJ sigue la misma filosofía que COOL y RIDL, indicando que la flexibilidad ganada con los LOA de propósito general pueden potencialmente llevar a conflictos con los principios básicos de la POO. Por esta razón MALAJ propone un LOA de dominio específico donde varios aspectos puedan ser resueltos cada uno especializado en su propia "incumbencia".

Tal como lo indica su nombre, el lenguaje base es Java, pero en una versión restringida en la que se ha eliminado la palabra reservadas "synchronized" así como los métodos *"wait"*, *"notify"*, y *"notifyall"*, en forma similar a lo que sucede con COOL. El propósito final de los creadores de MALAJ es cubrir un gran espectro de aspectos específicos.

1.2.3.9 HYPERJ

La aproximación por Ossher y Tarr sobre la separación multidimensional de conceptos (MDSOC) es llamada hyperspaces, y como soporte se construyó la herramienta HyperJ en Java. Para analizar con mayor profundidad HyperJ es necesario introducir primero cierta terminología relativa a MDSOC:

Un *espacio de concepto* concentra todas las unidades, es decir todos los constructores sintácticos del lenguaje, en un cuerpo de software, como una librería. Organiza las unidades en ese cuerpo de software para separar todos los conceptos importantes, describe las interrelaciones entre los conceptos e indica cómo los componentes del software y el resto del sistema pueden construirse a partir de las unidades que especifican los conceptos.

En HyperJ un *hiperespacio* (hyperspace) es un *espacio de concepto* especialmente estructurado para soportar la múltiple separación de conceptos. Su principal característica es que sus unidades se organizan en una matriz multidimensional donde cada eje representa una dimensión de concepto y cada punto en el eje es un concepto en esa dimensión.

Los *hiperslices* son bloques constructores pueden integrarse para formar un bloque constructor más grande y eventualmente un sistema completo.

Un *hipermódulo* consiste de un conjunto de *hiperslices* y conjunto de reglas de integración, las cuales especifican cómo los *hiperslices* se relacionan entre ellos y cómo deben integrarse.

Una vez introducida la terminología se puede continuar con el análisis de HyperJ. Esta herramienta permite componer un conjunto de modelos separados donde cada uno encapsula un concepto definiendo e implementando una jerarquía de

clases apropiada para ese concepto. Generalmente los modelos se superponen y pueden o no referenciarse entre ellos. Cada modelo debe entenderse por sí solo.
En la siguiente tabla se encuentran resumidas las principales características de las herramientas orientadas a aspectos descriptas en los párrafos anteriores:

TABLA Nº 1.1 COMPARACIÓN ENTRE HERRAMIENTAS ORIENTADAS A ASPECTOS

Leng. OA	Leng. Base	Tejido	Propósito	Características salientes
JPAL	Independiente	Dinámico	General	Meta-Tejedor. Independiente del leng. base. Totalmente dinámico.
D	Cualquier lenguaje OO	Estático	Específico	Usado en la programación distribuida. Provee interacción entre aspectos y herencia.
COOL	Cualquier lenguaje OO	Estático	Específico	Describe la sincronización de hilos concurrentes. Visibilidad limitada del aspecto.
RIDL	Cualquier lenguaje OO	Estático	Específico	Modulariza la interacción remota. Visibilidad limitada del aspecto.
AspectC	C	Estático	General	Usado en la implementación orientada a aspectos de sistemas operativos.
AspectS	Squeak	Dinámico	General	Basado en el paradigma de pasaje de mensajes. Todo aspecto es un objeto.
AspectC++	C++	Estático	General	Aspectos son extensiones del concepto de clase. Descripción más natural de los puntos de enlace.
MALAJ	Java	Dinámico	Específico	Modulariza los aspectos de sincronización y relocación. Su objetivo es eliminar los conflictos entre POA y POO.
HyperJ	Java	Dinámico	General	Basado en "*hyperslices*". Permite la composición de aspectos. Ambiente visual de trabajo.

1.3 Herramientas de programación para el desarrollo del prototipo

1.3.1 Introducción

Hace poco tiempo, la orientación a aspectos se centró principalmente en la implementación y codificación (desarrollo), pero en los últimos tiempos cada vez surgen más trabajos para llevar la separación de incumbencias a nivel de diseño.
Varios trabajos proponen utilizar UML (Unified Modeling Language) como lenguaje de modelado, ampliando su semántica con los mecanismos que el propio lenguaje unificado tiene para tales efectos y consiguiendo así representar la funcionalidad básica separada de los otros aspectos.

Desarrollar un sistema basado en aspectos requiere entender qué se debe incluir en el lenguaje base, también incluir dentro de los lenguajes de aspectos y qué debe compartirse entre ambos lenguajes. El lenguaje de los componentes debe proveer la forma de implementar la funcionalidad principal y asegurar que los programas escritos en ese lenguaje no interfieran con los aspectos. Los lenguajes de aspectos tienen que proveer los medios para implementar los aspectos deseados de una manera intuitiva, natural y concisa.

1.3.2 Diseño y desarrollo de aplicaciones Orientadas a Aspectos

El desarrollo de una aplicación basada en aspectos se suele dividir en tres etapas:

Etapa 1: Identificar competencias/intereses (*concern*)

Consiste en descomponer los requisitos del sistema en competencias y clasificarlas en:

Competencias básicas (*core-concern*): Las que están relacionadas con la funcionalidad básica del sistema, de carácter funcional. Son las que Gregor Kiczales denomina *componentes*.

Competencias transversales (*crosscutting-concern*): Las que afectan a varias partes del sistema, relacionadas con requerimientos no funcionales del sistema, normalmente de carácter no funcional. Son las que Gregor Kiczales denomina *aspectos*.

Etapa 2: Implementar competencias/intereses

Consiste en implementar cada interés independientemente: Para implementar las competencias básicas usaremos el paradigma que mejor se ajuste a ellas (POA), y dentro del paradigma, el lenguaje que mejor satisfaga las necesidades del sistema y de los desarrolladores (Java). A este lenguaje lo denominaremos **lenguaje base**.

Para implementar las competencias transversales se usa **lenguajes orientados a aspectos**, de propósito específico o general, encapsulando cada competencia en unidades llamadas aspectos. Estos lenguajes orientados a aspectos deben ser

compatibles con el lenguaje base para que los aspectos puedan ser combinados con el código que implementa la funcionalidad básica y así obtener el sistema final. Normalmente los lenguajes orientados a aspectos suelen ser extensiones del lenguaje base, como es el caso de AspectJ y AspectC, pero también puede ser lenguajes totalmente independientes.

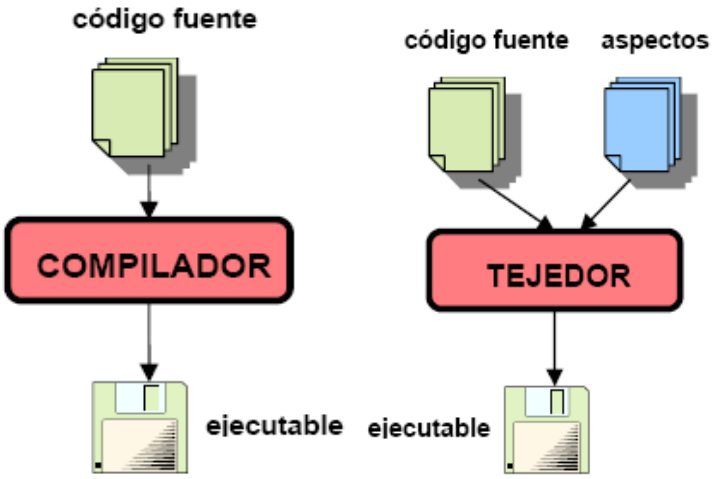
TABLA Nº 1. 2 CORRESPONDENCIA LENGUAJES BASE/LENGUAJES ORIENTADOS A ASPECTOS

Lenguaje base	Lenguaje de aspectos
Java	-AspectJ -AspectWerkz
C/C++	-AspectC -AspectC++
SmallTalk	-AspectS -Apostle
Python	-Pythius

Etapa 3: Componer el sistema final

Este proceso se conoce como **entretejido** (*weaving*) o **integración.** Consiste en combinar los aspectos con los módulos que implementan la funcionalidad básica del sistema dando lugar al sistema final. El modulo encargado de realizar este proceso recibe el nombre de **tejedor de aspectos** (*weaver*).

FIGURA Nº 1. 2 EJECUTABLE ENFOQUE TRADICIONAL & POA



Tejido estático versus dinámico

La primera decisión que hay que tomar al implementar un sistema Orientado a Aspectos es relativa a las dos formas de entrelazar el lenguaje componente con el lenguaje de aspectos. Dichas formas son el entrelazado o tejido estático y el entrelazado o tejido dinámico.

El entrelazado estático implica modificar el código fuente escrito en el lenguaje base, insertando sentencias en los puntos de enlace. Es decir, que el código de aspectos se introduce en el código fuente.

El entrelazado dinámico requiere que los aspectos existan y estén presentes de forma explícita tanto en tiempo de compilación como en tiempo de ejecución. Para conseguir esto, tanto los aspectos como las estructuras entrelazadas se deben modelar como objetos y deben mantenerse en el ejecutable. Un tejedor dinámico será capaz añadir, adaptar y remover aspectos de forma dinámica durante la ejecución.

El tejido estático evita que el nivel de abstracción introducido por la POA derive en un impacto negativo en la eficiencia de la aplicación, ya que todo el trabajo se realiza en tiempo de compilación, y no existe sobrecarga en ejecución. Si bien esto es deseable el costo es una menor flexibilidad: los aspectos quedan fijos, no pueden ser modificados en tiempo de ejecución, ni existe la posibilidad de agregar o remover nuevos aspectos.

Otra ventaja que surge es la mayor seguridad que se obtiene efectuando controles en compilación, evitando que surjan errores catastróficos o fatales en ejecución. Podemos agregar también que los tejedores estáticos resultan más fáciles de implementar y consumen menor cantidad de recursos.

El tejido dinámico implica que el proceso de composición se realiza en tiempo de ejecución, decrementando la eficiencia de la aplicación. El postergar la

composición brinda mayor flexibilidad y libertad al programador, ya que cuenta con la posibilidad de modificar un aspecto según información generada en ejecución, como también introducir o remover dinámicamente aspectos.

La característica dinámica de los aspectos pone en riesgo la seguridad de la aplicación, ya que se puede dinámicamente remover comportamiento de un aspecto que quizás luego se requiera, o más grave aún, qué sucede si un aspecto en su totalidad es removido, y luego se hace mención al comportamiento de ese aspecto de otra manera. El tener que llevar mayor información en ejecución, y tener que considerar más detalles, hace que la implementación de los tejedores dinámicos sea más compleja.

Es importante notar que la naturaleza dinámica hace referencia a características o propiedades de un aspecto, y no al aspecto en sí mismo. Es decir, una vez que se identificó un aspecto, el mismo se mantendrá como concepto a lo largo de todo el ciclo de vida de la aplicación. Lo que puede variar son las características o niveles de aplicación de ese concepto.

Por ejemplo, la comunicación es estática, en el sentido que el concepto de comunicación está siempre presente, pero algunas características de la misma, como la velocidad o el costo, pueden cambiar por distintas razones en ejecución. Es más, quizás algunas propiedades de la comunicación no se conozcan hasta ejecución. Por lo tanto es necesario que el aspecto de comunicación sea dinámico para adaptarse correctamente a los distintos casos de uso.

Guías de Diseño

Tomando como base la guía de diseño para la implementación de sistemas abiertos, podemos enunciar las siguientes pautas para aquellos tejedores que soporten tanto aspectos dinámicos como estáticos:

- Los aspectos dinámicos deben separarse claramente de los aspectos estáticos, como por ejemplo, a través de un constructor o palabra reservada que indique la naturaleza del aspecto. Los aspectos estáticos están precedidos por la palabra reservada “static”, y los dinámicos por la palabra reservada “dynamic”.

- La especificación de la naturaleza del aspecto debe ser opcional.

- El alcance de la influencia de dicha especificación tiene que ser controlado de una forma natural y con suficiente granularidad. Esta pauta ayuda al programador a entender y razonar sobre su programa.

- El conocimiento que tiene el cliente sobre los detalles internos de implementación debe ser mínimas. Por ejemplo, el cliente podría elegir diferentes niveles de seguridad para el aspecto Seguridad-Edificio (alto, mediano o bajo) en ejecución, sin conocer cómo está implementada cada alternativa.

1.3.3 Indicios del lenguaje de programación Java

Java surgió en 1991 cuando un grupo de ingenieros de *Sun Microsystems* trataron de diseñar un nuevo lenguaje de programación destinado a electrodomésticos. La reducida potencia de cálculo y memoria de los electrodomésticos llevó a desarrollar un lenguaje sencillo capaz de generar código de tamaño muy reducido.

FIGURA Nº 1.3 LOGOTIPO DE JAVA

Debido a la existencia de distintos tipos de CPUs y a los continuos cambios, era importante conseguir una herramienta independiente del tipo de CPU utilizada.

Desarrollaron un código "neutro" que no dependía del tipo de electrodoméstico, el cual se ejecutaba sobre una *"máquina hipotética o virtual"* denominada ***Java Virtual Machine*** (**JVM**). Era la JVM quien interpretaba el código neutro convirtiéndolo a código particular de la CPU utilizada. Esto permitía lo que luego se ha convertido en el principal lema del lenguaje: *"Write Once, Run Everywhere".* A pesar de los esfuerzos realizados por sus creadores, ninguna empresa de electrodomésticos se interesó por el nuevo lenguaje.

Como lenguaje de programación para computadores, Java se introdujo a finales de 1995. La clave fue la incorporación de un intérprete Java en la versión 2.0 del programa *Netscape Navigator*, produciendo una verdadera revolución en Internet. ***Java 1.1*** apareció a principios de 1997, mejorando sustancialmente la primera versión del lenguaje. ***Java 1.2***, más tarde rebautizado como ***Java 2***, nació a finales de 1998.

Al programar en Java no se parte de cero. Cualquier aplicación que se desarrolle "cuelga" (o se apoya, según como se quiera ver) en un gran número de clases preexistentes. Algunas de ellas las ha podido hacer el propio usuario, otras pueden ser comerciales, pero siempre hay un número muy importante de clases que forman parte del propio lenguaje (el **API** o ***Application Programming Interface*** de ***Java***).

Java incorpora en el propio lenguaje muchos aspectos que en cualquier otro lenguaje son extensiones propiedad de empresas de software o fabricantes de ordenadores (threads, ejecución remota, componentes, seguridad, acceso a base de datos, etc.).

Por eso muchos expertos opinan que Java es el lenguaje ideal para aprender la informática moderna, porque incorpora todos estos conceptos de un modo

estándar, mucho más sencillo y claro que con las citadas extensiones de otros lenguajes. Esto es consecuencia de haber sido diseñado más recientemente y por un único equipo.

El principal objetivo de Java es llegar a ser el *"nexo universal"* que conecte a los usuarios con la información, esté ésta situada en el ordenador local, en un servidor de **Web,** en una base de datos o en cualquier otro lugar.

Existen distintos programas comerciales que permiten desarrollar código ***Java***. La compañía Sun, creadora de Java, distribuye gratuitamente el ***Java™ Development Kit*** (***JDK***). Se trata de un conjunto de programas y librerías que permiten desarrollar, compilar y ejecutar programas en Java. Existe también una versión reducida del JDK, denominada ***JRE*** (*Java Runtime Environment*) destinada únicamente a ejecutar código Java (no permite compilar).

El compilador de Java: Se trata de las herramientas incluidas en el ***JDK***. Realiza un análisis de sintaxis de código escrito en los ficheros fuente de Java (con extensión ****.java***). Si no encuentra errores en el código genera los ficheros compilados (con extensión ****.class***). En otro caso muestra la línea o líneas erróneas. En el ***JDK*** de ***Sun*** dicho compilador se llama ***javac.exe***. Tiene numerosas opciones, algunas de las cuales varían de una versión a otra. Se aconseja consultar la documentación de la versión del JDK utilizada para obtener una información detallada de las distintas posibilidades.

El Java Virtual Machine: La existencia de distintos tipos de procesadores y ordenadores llevó a los ingenieros de ***Sun*** a la conclusión de que era muy importante conseguir un software que no dependiera del tipo de procesador utilizado. Se planteo la necesidad de conseguir un código capaz de ejecutarse en cualquier tipo de máquina. Una vez compilado no debería ser necesaria ninguna modificación por el hecho de cambiar de procesador o de ejecutarlo en otra máquina.

La clave consistió en desarrollar un código "neutro" el cual estuviera preparado para ser ejecutado sobre una *"máquina virtual",* denominada ***Java Virtual Machine*** (***JVM***). Es esta ***JVM*** quien interpreta este código neutro convirtiéndolo a código particular de la CPU utilizada. Se evita tener que realizar un programa diferente para cada CPU o plataforma.

La ***JVM*** es el intérprete de Java. Ejecuta los *"bytecodes"* (ficheros compilados con extensión ****.class***) creados por el compilador de Java (***javac.exe***). Tiene numerosas opciones entre las que destaca la posibilidad de utilizar el denominado ***JIT*** (*Just-In-Time Compile*r), que puede mejorar entre 10 y 20 veces la velocidad de ejecución de un programa.

Java es uno de los lenguajes de programación y de desarrollo de aplicaciones muy estable, eficiente y desde luego que aporta con grandes eficiencias y ventajas a la hora de construir un software por propia cuenta. Es notoria la diferencia que ofrece el lenguaje de programación Java con respecto a la interfaz que se puede diseñar, a otras herramientas como Visual Basic, .NET, Fox, Objetive-C, Python, Ruby, Eiffel, Ocaml, Clarion, Ecplice, etc.

En conjunto Java con el programa AspectJ para desarrollo de software con Programación Orientada a Aspectos son una de las mayores herramientas utilizadas para este tipo de aplicaciones a nivel mundial.

1.3.4 AspectJ

AspectoJ es un lenguaje orientado a aspectos de propósito general, cuya primera versión fue lanzada en 1998 por el equipo conformado por Gregor Kickzales (líder del proyecto), Ron Bodkin, Bill Griswold, Erik Hilsdale, Jim Hugunin, Wes Isberg y Mik Kersten. Es una herramienta que está en desarrollo y las nuevas versiones pueden tanto corregir errores de su versión predecesora como modificar el lenguaje.

AspectJ es una extensión compatible de Java para facilitar el uso de aspectos por parte de los programadores de Java. Por compatible se entiende:

- Compatibilidad base: todos los programas válidos de Java deben ser programas válidos de AspectJ.

- Compatibilidad de plataforma: todos los programas válidos de AspectJ deben correr sobre la máquina virtual estándar de Java.

- Compatibilidad de programación: la programación en AspectJ debe ser una extensión natural de la programación en Java.

Esta última meta fue una de las guías más importante a la hora de tomar decisiones sobre el lenguaje.

Extiende Java para soportar el manejo de aspectos agregando a la semántica de Java cuatro entidades principales. Esto se ve reflejado en la Figura1.4.

- Los *puntos de enlace* son puntos bien definidos en la ejecución de un programa, entre ellos podemos citar llamadas a métodos y accesos a atributos.

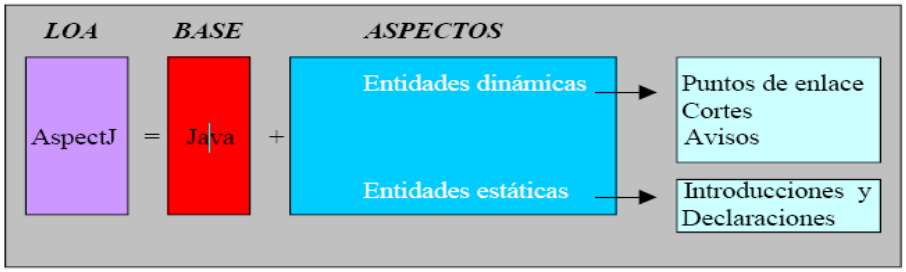
- Los *cortes* agrupan puntos de enlace y permiten exponer el contexto en ejecución de dichos puntos. Existen cortes primitivos y también definidos por el usuario.

- Los *avisos* son acciones que se ejecutan en cada punto de enlace incluido en un corte. Los avisos tienen acceso a los valores expuestos por el corte. Las tres entidades anteriores son dinámicas porque permiten definir comportamiento adicional que actuará en tiempo de ejecución.

- Las *introducciones y declaraciones* permiten cambiar la estructura de clases de un programa agregando o extendiendo interfaces y clases con nuevos atributos, constructores o métodos. Esta última entidad es estática porque afecta la signatura estática del programa. Obtenemos entonces que AspectJ soporta tanto implementación estática como dinámica de conceptos entrecruzados.

Un aspecto es un tipo entrecruzado que encapsula cortes, avisos y las propiedades estáticas. Por tipo se entiende una unidad modular de código con una interface bien definida sobre la cual es posible razonar en tiempo de compilación.

FIGURA Nº 1. 4 ESPECIFICACIÓN DE ASPECTJ



La intención de AspectJ es ser un LOA práctico, que provea un conjunto sólido y maduro de características orientadas a aspectos, compatible con Java para aprovechar su popularidad.

1.3.4.1 Puntos de enlace

Para entender el concepto de punto de enlace consideremos el siguiente ejemplo, en el cual se define una clase para manejar números complejos:

```
Class NumComplejo{
private real parte_imaginaria, parte_real;
NumComplejo(real x, real y){
this.parte_imaginaria=x;
this.parte_real=y;
```

```
}
void ingresar_parte_imaginaria(real x){ this.parte_imaginaria=x; }
void ingresar_parte_real(real y){ this.parte_real=y; }
real devolver_parte_ imaginaria(){ return parte_imaginaria; }
real devolver_parte_real(){ return parte_real; }
void aumentar_parte_real(real x) {
real a = devolver_parte_real();
a= a+x;
ingresar_parte_real(a);
}
}
```

Código 1 Clase número complejo

Entonces lo que establece el código void ingresar_parte_ imaginaria (real x){ this.parte_imaginaria=x; } es que cuando el método ingresar_parte_imaginaria es invocado con un real como argumento sobre un objeto de tipo NumComplejo entonces se ejecuta el cuerpo del método.

De igual forma cuando un objeto de tipo NumComplejo es instanciado a través de un constructor con dos argumentos de clase real, entonces se ejecuta el cuerpo del constructor.

El patrón que surge de esta descripción es que cuando “algo” pasa entonces “algo” se ejecuta. El conjunto de las “cosas que pasan” representan los puntos de enlace.

Los puntos de enlace son entonces puntos bien definidos en la ejecución de un programa y AspectJ define los siguientes conceptos:

*Llamadas a métodos***:** Cuando un método es invocado, no incluye llamadas a **super**.

Ejemplo: cc.aumentar_parte_real_primera(5.5); Cuando el método *aumentar_parte_real_primera* es invocado sobre el objeto cc.

Ejecución de un método: Cuando el cuerpo de un método se ejecuta.
Ejemplo: Cuando el cuerpo del método *aumentar_parte_real_primera* es ejecutado.

Llamada a un constructor: Cuando un objeto es creado y un constructor es invocado, sin incluir la llamada al constructor **this** o **super**.
Ejemplo: NumComplejo nc1 = new NumComplejo(3.0,5.3); El objeto es creado por **new** y luego el constructor *NumComplejo* es invocado.

Ejecución de un inicializador: Cuando los inicializadores no estáticos de una clase se ejecutan.
Ejecución de un constructor: Cuando se ejecuta el código de un constructor, luego de su llamada al constructor **this** o **super**.
Ejemplo: Cuando el código del constructor *NumComplejo* se ejecuta.

Ejecución de un inicializador estático: Cuando se ejecuta el inicializador estático para una clase.

Pre-inicialización de un objeto: Antes que el código de inicialización para una clase particular se ejecute. Comprende el tiempo entre el comienzo de la llamada al primer constructor y el comienzo del constructor de su clase padre. Luego, la ejecución de estos puntos de enlace comprenden los puntos de enlace del código encontrado en las llamadas a constructores **this** y **super**

Inicialización de un objeto: Cuando el código de inicialización para una clase particular se ejecuta. Comprende el tiempo entre el retorno del constructor de su clase padre y el retorno de la llamada a su primer constructor. Incluye todos los inicializadores dinámicos y constructores usados para crear el objeto.

Referencia a un atributo: Cuando se referencia a un atributo no final. Un atributo final es un atributo que no cambia su valor una vez inicializado.
Asignación a un atributo: Cuando se realiza la asignación a un atributo.
Ejemplo: numerocomplejo.parte_imaginaria=5.5;

Cuando al atributo *parte_imaginaria* del objeto *numerocomplejo* se le asigna el valor real 5.5.

Ejecución de un manejador: Cuando el manejador de una excepción se ejecuta.
El modelo de punto de enlaces es un elemento crítico en el diseño de cualquier LOA ya que provee la interface entre el código de aspectos y el código de la funcionalidad básica.

En AspectJ este modelo puede considerarse como nodos en un grafo de llamadas a objetos en ejecución. Estos nodos incluyen puntos en los cuales un objeto recibe una llamada a un método o puntos en los cuales un atributo de un objeto es referenciado.

Los arcos representan el flujo de control entre los nodos. En este modelo el control pasa por cada punto de enlace dos veces, una cuando realiza el cálculo y otra al regresar del mismo.
Como ejemplo del modelo, consideremos la siguiente clase.

```
Class CoordenadaCompleja {
NumComplejo NumComplejo1,NumComplejo2;
real distancia;
...void aumentar_parte_real_primera(real x){
...real s = this.distancia;
…NumComplejo1.aumentar_parte_real(x);
}
...
}
```

Código 2. Clase coordenada compleja

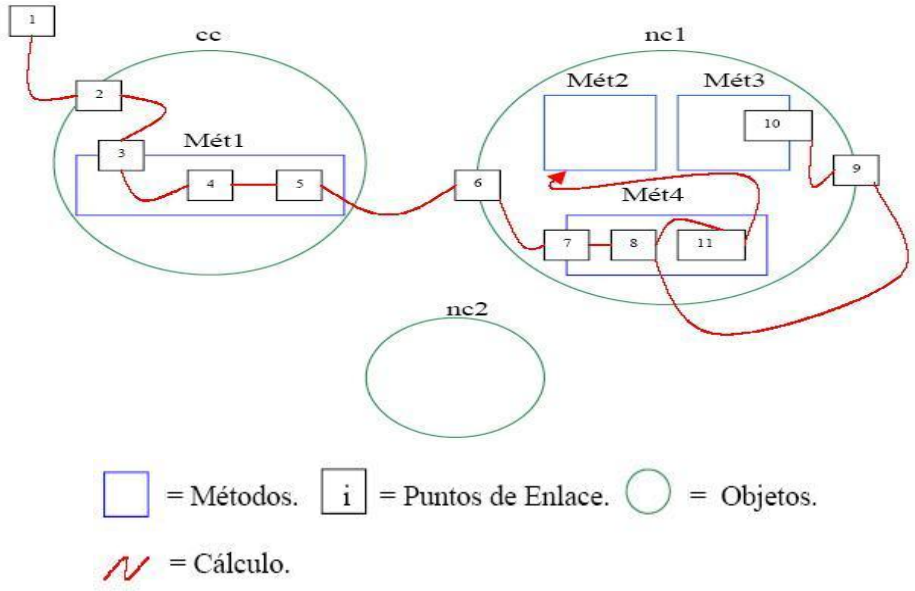
FIGURA Nº 1.5 MODELO DE PUNTOS DE ENLACE

NumComplejo nc1 = new NumComplejo(3.0,5.3);

NumComplejo nc2 = new NumComplejo(6.2,1.3);

CoordenadaCompleja cc = new CoordenadaCompleja(nc1,nc2);

cc.aumentar_parte_real_primera(5.5);



Mét1 = aumentar_parte_real_primera, de la clase CoordenadaCompleja.

Mét2 = ingresar_parte_real, de la clase NumComplejo.

Mét3 = devolver_parte_real, de la clase NumComplejo.

Mét4 = aumentar_parte_real, de la clase NumComplejo.

La ejecución de las tres primeras líneas de la figura 1.5 provoca la creación de los objetos cc, nc1 y nc2. La ejecución de la última línea inicia una computación que irá siguiendo los puntos de enlace:

1. Un punto de enlace de llamada a un método, correspondiente al método *aumentar_parte_real_primera* invocado sobre el objeto cc.

2. Un punto de enlace de recepción de llamada a un método, en el cual cc recibe la llamada aumentar_parte_real_primera.

3. Un punto de enlace de ejecución de un método, en el cual el método aumentar_parte_real_primera definido en la clase *CoordenadaCompleja* empieza su ejecución.
4. Un punto de enlace de acceso a un atributo, en el cual el atributo *distancia* de cc es referenciado.
5. Un punto de enlace de llamada a un método, en el cual el método *aumentar_parte_real* es invocado sobre el objeto nc1.
6. Un punto de enlace de recepción de llamada a un método, en el cual nc1 recibe la llamada *aumentar_parte_real*.
7. Un punto de enlace de ejecución de un método, en el cual el método *aumentar_parte_real* definido en la clase *NumComplejo* empieza su ejecución.
8. Un punto de enlace de llamada a un método, en el cual el método *devolver_parte_real* es invocado sobre el objeto nc1.
9. Un punto de enlace de recepción de llamada a un método, en el cual nc1 recibe la llamada devolver_parte_real.
10. Un punto de enlace de ejecución de un método, en el cual el método *devolver_parte_real* definido en la clase *NumComplejo* empieza su ejecución. El *control retorna por los puntos de enlace* 10 y 9.
11. Un punto de enlace de llamada a un método, en el cual el métodoingresar_parte_real es invocado sobre el objeto nc1, y así siguiendo, hasta que finalmente el control retorna por los puntos de enlace 3, 2 y 1.

1.3.4.2 Cortes

Un corte es un conjunto de puntos de enlace más datos del contexto de ejecución de dichos puntos. Los cortes principalmente son usados por los avisos y pueden ser compuestos con operadores booleanos para crear otros cortes. Aspecto provee varios designadores de cortes primitivos. El programador luego puede componerlos para definir designadores de cortes anónimos o con nombres.

Los cortes no son de alto orden, porque sus argumentos y resultados no pueden ser cortes ni existen designadores de cortes paramétricos.

Un designador de corte captura en ejecución varios puntos de enlace, por ejemplo el designador: *call(void NumComplejo.ingresar_parte_real(real))* captura todas las llamadas al método *ingresar_parte_real* de la clase *NumComplejo* con un argumento de clase real.

Otro ejemplo sería:
pointcut acceso():
get(NumComplejo.parte_real) // get(CoordenadaCompleja.distancia); este designador de corte captura todas las referencias al atributo parte_real de la clase *NumComplejo* o todas las referencias al atributo distancia de la clase *CoordenadaCompleja*. Como vemos los designadores de cortes pueden capturar puntos de enlaces de diferentes clases, esto es, entrecruzan las clases.

Cortes primitivos

AspectJ incluye una variedad de designadores de cortes primitivos que identifican puntos de enlace de diferentes formas.

Los designadores de cortes primitivos son:

Call: call(PatróndeMétodo), call(PatróndeConstructor).
Captura todos los puntos de enlace de llamadas a métodos o
constructores cuyo encabezamiento coincide con el respectivo patrón.
Ejemplo: call(* NumComplejo.* (..)) captura todos los puntos de enlace de llamadas a cualquier método, que devuelve cualquier tipo y con cualquier número y tipo de argumentos, de la clase *NumComplejo*. Los patrones se refieren a expresiones como NumComplejo.* que permite identificar todos los métodos de la clase *NumComplejo*.

Execution: execution (PatróndeMétodo), execution(PatróndeConstructor).
Captura todos los puntos de enlace de ejecución de métodos o
Constructores cuyo encabezamiento coincide con el respectivo patrón.
Ejemplo: execution(NumComplejo.new(..)) captura la ejecución de los constructores de la clase *NumComplejo* con cualquier número de argumentos.

Get: get(PatróndeAtributo)
Captura todos los puntos de enlace de referencia a los atributos que coinciden con el patrón correspondiente.
Ejemplo: get(NumComplejo.parte_imaginaria) captura las referencias al atributo *parte_imaginaria* de la clase *NumComplejo.*

Set: set(PatróndeAtributo)
Captura todos los puntos de enlace de asignación a los atributos que coinciden con el patrón correspondiente.
Ejemplo: set(NumComplejo.parte_*)captura las asignaciones a los atributos de la clase *NumComplejo* que coinciden con el patrón "parte_*", en este caso coinciden *parte_imaginaria y parte_real.*

Initialization: initialization(PatróndeConstructor)
Captura los puntos de enlace de las inicializaciones de los objetos cuyos constructores coinciden con el patrón.

Staticinitialization:staticinitialization(PatróndeClase)
Captura los puntos de enlace de ejecución de un inicializador estático de las clases que coinciden con el patrón.

Handler: handler (PatróndeClase)
Captura los manejadores de excepción de las clases de excepciones que coinciden con el patrón.

This: this(PatróndeClase), this(identificador).
Captura todos los puntos de enlace donde el objeto actual (el objeto ligado a **this**) es una instancia de una clase que coincide con el PatróndeClase, o es instancia de una clase que coincide con la clase asociada al identificador.

Target: target(PatróndeClase), target(identificador) Captura todos los puntos de enlace donde el objeto objetivo (el objeto sobre el cual se invoca un método o una operación de atributo) es una instancia de una clase que coincide con el PatróndeClase, o es instancia de una clase que coincide con la clase asociada al identificador.

Args: args(PatróndeClase, . . .),args(identificador, . . .)
Captura todos los puntos de enlace donde los argumentos son instancias de una clase que coincide con el PatróndeClase o con la clase del identificador.

Si es un PatróndeClase entonces el argumento en esa posición debe ser instancia de una clase que coincida con el PatróndeClase. Si es un identificador entonces el argumento en esa posición debe ser instancia de una clase que coincida con la clase asociada al identificador (o cualquier clase si el identificador está asociado a Object.

Este caso especial se verá nuevamente en la sección Exposición de contexto).
Ejemplo: args(*,int) captura todos los puntos de enlace con dos argumentos, el primero puede ser cualquiera y el segundo debe ser un entero.

Within: within(PatróndeClase) Captura todos los puntos de enlace donde el código que se está ejecutando está definido en una clase que coincide con el patrón. Incluye la inicialización de la clase, del objeto, puntos de enlaces de ejecución de métodos y constructores, y puntos de enlaces asociados con las sentencias y expresiones de la clase.

Ejemplo: within(NumComplejo) captura todos los puntos de enlace donde el código que se está ejecutando está definido dentro de la clase *NumComplejo*.

Withincode: withincode(PatróndeMétodo),withincode(PatróndeConstructor)
Captura todos los puntos de enlace donde el código que se está ejecutando está definido en un método o constructor que coincide con el patrón correspondiente.

Incluye todos los puntos de enlace de ejecución de métodos y constructores, y puntos de enlaces asociados con las sentencias y expresiones del método o constructor.
Ejemplo: withincode(real devolver_parte_real()) captura todos los puntos de enlace donde el código que se está ejecutando está definido en el método devolver_parte_real.

Estos dos últimos cortes primitivos, within y withincode, tratan con la estructura léxica del programa. Cflow: cflow(Corte)

Captura todos los puntos de enlace en el flujo de control de los puntos de enlace capturados por el corte pasado como parámetro.
Ejemplo: cflow(withincode(real devolver_parte_real())) captura todos los puntos de enlace que ocurren entre el comienzo y la finalización de los puntos de enlace especificados por withincode(real devolver_parte_real()).

Cflowbelow: cflowbelow(Corte) captura todos los puntos de enlace en el flujo de control debajo de los puntos de enlace capturados por el corte pasado como parámetro, sin incluir el punto de enlace inicial del flujo de control.

Ejemplo: cflowbelow(withincode(real devolver_parte_real())) captura todos los puntos de enlace que ocurren entre el comienzo y la finalización de los puntos de enlace especificados por withincode(real devolver_parte_real()), pero no lo incluye a éste.
If: if(Expresiónbooleana)

Captura todos los puntos de enlace cuando la expresión booleana se satisface.

Cortes definidos por el programador. El programador puede definir nuevos cortes asociándoles un nombre con la declaración **pointcut**.

pointcut nombre(<Parametros_formales>)**:** <Corte>**;** donde <Parametros_formales> expone el contexto del corte, y
< Corte> puede ser un corte primitivo o definido por el programador.
Ejemplo:

```
/* Corte1 */
pointcut aumentar(): call(void aumentar*(…)) ;
/* Corte2 */
pointcut aumentarsolocomplejo(): aumentar() && target(NumComplejo);
```

Corte1 captura todas las llamadas a un método que comience con la palabra aumentar. Corte2 utiliza Corte1 para capturar aquellas llamadas a métodos que empiecen con aumentar pero que sean invocados sólo sobre un objeto de clase.

NumComplejo.
Un corte nombrado puede ser definido tanto en una clase como en un aspecto y es tratado como un miembro de esa clase o aspecto. Como miembro puede tener modificadores de acceso privado o público (**private** o **public**). No se permite la sobrecarga de cortes definidos por el programador. Por lo tanto será un error de compilación cuando dos cortes en una misma clase o aspecto tengan asociado el mismo nombre. Esta es una de las diferencias con la declaración de métodos.

El alcance de un corte definido por el programador es la declaración de la clase o aspecto que lo contiene. Es diferente al alcance de otros miembros que sólo se limitan al cuerpo de la clase o aspecto.
Los cortes pueden declararse como abstractos, y son definidos sin especificar su cuerpo. Estos cortes sólo pueden declararse dentro de un aspecto declarado abstracto.

Composición de cortes

Tanto los cortes definidos por el programador como los primitivos pueden componerse entre sí a través de la utilización de operadores lógicos para crear nuevos cortes. Los operadores algebraicos soportados por AspectJ son: && ("y"), || ("or"), !("not") y los paréntesis.

Lo anterior puede modelarse a través de una BNF:
<Corte>::= <Corte_Primitivo>| <Corte_Definido_por_el_programador> |
<Corte> **&&** <Corte> |
<Corte> || <Corte> |
!<Corte> |
(<Corte>**)**

BNF 1 Definición de cortes

Donde la semántica asociada a los operadores es la siguiente:
<Corte>1 **&&** <Corte>2: captura todos los puntos de enlace de Corte1 y todos los puntos de enlace de Corte2.
<Corte>1|| <Corte>2: captura todos los puntos de enlace de Corte1 o todos los puntos de enlace de Corte2.
!<Corte>: captura todos los puntos de enlace que no captura Corte.
(<Corte>**)**: captura todos los puntos de enlace que captura Corte.

Esto es, el resultado de componer uno o más cortes también es un corte.

La composición de los cortes resulta uno de los mecanismos más importantes para permitir la expresividad en el manejo de aspectos de AspectJ. La composiciónes sencilla y simple, a través de operadores bien conocidos. Daría la impresión que esta simplicidad tendría como efecto secundario una pérdida importante en el poder de expresividad; pero esto no es así ya que se pueden obtener expresiones tan complejas como se requiera. La composición es entonces un mecanismo muy poderoso, y aún así mantiene su simplicidad.

Como ejemplo, consideremos la siguiente implementación de una pila para la cual cada vez que un elemento es agregado, se desea mostrar la pila resultante. Para esto, se define un corte que captura todas las llamadas al método apilar:
pointcut al_apilar() : call(void Pila.Apilar(Object));

Si bien con este corte se capturan los puntos de enlace deseados, caemos en un problema de eficiencia porque en el método Apilar_Multiple, por cada elemento se invoca al método Apilar. Idealmente, en el caso de este método, la pila se debería mostrar una vez que se apilaron todos los elementos, y no cada vez que se apila un elemento.

```
Class Pila {
private Object [] arreglo ;
private int ultimo;
private int capacidadMaxima=100;
public Pila() {
arreglo = new Object[capacidadMaxima];
ultimo=0;
}
public void Apilar(Object elem) {
arreglo[ultimo] = elem;
ultimo++;
}
public Object Desapilar(){
Object elem = arreglo[ultimo-1];
ultimo--; return elem;
}
public void Apilar_Multiple(Object[] arregloElem){
for (int i=0,i<=arregloElem.length,i++){
this.apilar(arregloElem[i] );
}
}
```

```
public int Dar_Capacidad_Maxima(){ return capacidadMaxima;}
public int Cantidad_Elementos() {return ultimo;}
public boolean Pila_Vacia(){return (ultimo==0);}
public boolean Pila_ Llena(){return (ultimo==capacidadMaxima);}
}
```

Código 3. Definición clase pila

Para discriminar estos dos casos y solucionar el problema, utilizamos la composición de cortes:

```
pointcut al_apilar_deauno(): al_apilar() &&
!withincode(void Pila.Apilar_Multiple(..));
pointcut al_apilar_multiple(): call(void Pila.Apilar_Multiple(..));
pointcut al_apilar_eficiente(): al_apilar_deauno() && al_apilar_multiple();
```

El corte al_apilar_deauno captura todas las invocaciones al método Apilar que no están dentro del código de Apilar_Multiple. El corte al_apilar_multiple captura todas las invocaciones al método Apilar_Multiple. Y el último corte, al_apilar_eficiente, captura los puntos de enlace del primer y segundo corte, solucionado así el problema de eficiencia.

Como vemos la composición de cortes de AspectJ nos permite modelar una situación dinámica, en el cual el punto de enlace depende del contexto dinámico de ejecución:

Ejemplo, dentro de todas las invocaciones al método Apilar se necesita descartar las que se realicen desde el código de Apilar_Multiple.

Para ello debemos analizar dinámicamente cada invocación al método Apilar y descartar aquellas que estén en el contexto de Apilar_Multiple.

Otro ejemplo más sencillo de composición sería obtener los momentos donde otros objetos requieran el valor de los atributos de la pila, en particular capacidadMaxima y ultimo. Esto se logra con el siguiente corte:

pointcut atributos_leidos(): (call (int Pila.Dar_Capacidad_Maxima(..)) ||
call (int Pila.Cantidad_Elementos(..))) && !this(Pila) ;
El corte atributos_leidos captura todas las llamadas a los métodos Dar_Capacidad_Maxima y Cantidad_Elementos tal que el llamador no es un objeto de la clase Pila.

Exposición de contexto

Los cortes pueden exponer el contexto de ejecución en sus puntos de enlace a través de una interface. Dicha interface consiste de parámetros formales en la declaración de los cortes definidos por el programador, análogo a los parámetros formales en un método.

Una lista de parámetros vacía, como en el ejemplo anterior, significa que cuando los eventos ocurren ningún contexto está disponible, en cambio pointcut atributos_leidos(Pila p): (call (int Pila.Dar_Capacidad_Maxima(..)) || call (int Pila. Cantidad_Elementos(..))) && !this(Pila) && target(p); sería obtener los momentos donde otros objetos requieran el valor de los atributos de la pila y disponer del objeto de clase Pila receptor de los mensajes a través del parámetro p, que es instanciado por el corte primitivo target.

El contexto disponible en este caso es p.
En la parte derecha de la declaración de un corte o de un aviso se permite un identificador en lugar de especificar una clase o PatróndeClase. Existen tres cortes primitivos donde esto se permite: this, target y args. Usar un identificador en vez de un PatróndeClase es como si la clase seleccionada fuera de la clase del parámetro formal, por lo tanto en el ejemplo anterior target(p) captura todos los objetos receptores de clase Pila.

Otro ejemplo de exposición de contexto sería:
pointcut puntosiguales(Punto p): target(Punto) && args(p)&& call(boolean igual(Object)); donde suponemos definida la clase Punto con un método público igual que recibe como parámetro otro objeto de clase Punto y retorna un valor booleano estableciendo si las coordenadas de ambos objetos son iguales o no.

El corte tiene un parámetro de clase Punto, entonces cuando los eventos descriptos en la parte derecha ocurren un objeto p de clase Punto está disponible.
Pero en este caso si miramos con atención la parte derecha el objeto disponible no es el objeto receptor del método sino el objeto pasado como parámetro en dicho método.

Si queremos acceder a ambos objetos el corte debe ser definido:
pointcut puntosiguales(Punto p1,Punto p2): target(p1) && args(p2) && call(boolean igual(Object)); donde p1 expone el objeto de clase Punto receptor del método igual y p2 expone el objeto de clase Punto que es pasado como parámetro al método igual .

La definición de los parámetros en un corte es flexible. Sin embargo la regla más importante a tener en cuenta es que cuando cada uno de los eventos definidos ocurre todos los parámetros del corte deben ligarse a algún valor. Por consiguiente este corte daría un error de compilación:

```
pointcut enDosPilas(Pila p1, Pila p2) :
(target(p1) && call(void Pila.Apilar(..))) ||
(target(p2) && call(Object Pila.Desapilar(..)));
```

El corte anterior captura las llamadas al método Apilar en un objeto p1 de clase Pila o las llamadas al método Desapilar en un objeto p2 de clase Pila. El problema es que la definición de los parámetros intenta acceder a dos objetos de clase Pila. Pero cuando el método Apilar es llamado sobre un objeto no existe otro objeto de clase de Pila accesible, por lo que el parámetro p2 quedaría sin un valor ligado y de ahí el error de compilación. La otra situación sería cuando el método Desapilar

es llamado sobre un objeto y el parámetro p1 quedaría sin un valor asociado resultando en el mismo error.

Un caso especial de exposición de contexto se da al utilizar la clase Object:
pointcut llamadaspublicas( ): call(public *.*(..)) && args(Object); este corte captura todos los métodos públicos unarios que toman como único argumento subclases de Object pero no los tipos primitivos como **int**. Pero:
pointcut llamadaspublicas(Object o): call(public *.*(..)) && args(o); captura todos los métodos unarios con cualquier clase de argumento, si el argumento fuera de tipo **int** el valor ligado a o sería de la clase java.lang.Integer.

Patrones

En esta sección se describen los patrones que permite especificar AspectJ.
Un Patrónde de Método es una descripción abstracta que permite agrupar uno o más métodos según su encabezamiento.
Un patrón para los métodos sigue el siguiente esquema:
<PatróndeMétodo>::=
[<ModificadoresdeMetodos>]
<PatróndeClase>[<PatróndeClase>**.**]
<PatróndeIdentificador>**(**<PatrondeClase>**,...)**[**throws**<PatróndeExcepción>]

BNF 2 Patrones de métodos

El esquema incluye los modificadores de métodos como **static**, **private** o **public**, luego las clases de los objetos retornados, las clases a las cuales pertenece el método, el nombre del método, que puede ser un patrón, los argumentos, que también pueden ser patrones, y la cláusula throws que corresponda.
La lista de parámetros formales puede utilizar el wildcard ".." para indicar cero o más argumentos. Luego execution(void m(..,int)) captura los métodos m cuyo último parámetro es de tipo entero, incluyendo el caso de que tenga un sólo parámetro entero. Mientras que execution(void m(..)) captura todos los métodos m con cualquier cantidad de argumentos.

Los nombres de métodos pueden contener el wildcard "*" que indica cualquier número de caracteres en el nombre del método. Luego call(boolean *()) captura todos los métodos que retornan un valor booleano sin tener en cuenta el nombre de los métodos. En cambio call(boolean dar*()) captura todos los métodos que retornan un valor booleano cuyo nombre empiece con el prefijo "dar". También se puede utilizar este wildcard en el nombre de la clase que contiene al método.

call(*.Apilar(..)) captura todas las llamadas al método Apilar definido en cualquier clase. Mientras que call(Pila.Apilar(..)) captura todas las llamadas al método Apilar definido en únicamente en la clase Pila. De no estar presente un nombre de clase se asume el wildcard "*" indicando cualquiera de las clases definidas.

Un PatróndeClase es un mecanismo para agrupar un conjunto de clases y usarlo en lugares donde de otra forma se usaría una sola clase. Las reglas para usar los patrones de clase son simples.

Todos los nombres de clases son patrones de clase. Existe un nombre especial "*" que también es un patrón. El "*" captura todas las clases y también los tipos primitivos. Luego call(void Apilar(*)) captura todas las llamadas a los métodos Apilar que tienen un único argumento de cualquier clase.

Los patrones de clase pueden también utilizar los wildcard "*" y "..". El wildcard "*" indica cero o más caracteres a excepción del punto("."). Luego puede ser usado cuando las clases respetan una convención para formar sus nombres. Call(java.awt.*) captura todas las clases que comiencen con "java.awt." y no tengan más puntos. Esto captura las clases en el paquete java.awt pero no clases interiores como java.awt.datatransfer o java.awt.peer.

El wildcard "..." indica cualquier secuencia de caracteres que comienza y termina con punto (".") , por lo tanto puede ser utilizado para agrupar todas las clases en cualquier subpaquete o todas las clases internas. Luego target (java.awt...*)

captura todos los puntos de enlaces donde el objetivo es de una clase que comienza con “java.awt.”.

Incluye clases definidas en java.awt tanto como clases definidas en java.awt.datatransfer o java.awt.peer, entre otras.

A través del wildcard “+” es posible agrupar todas las subclases de una clase o conjunto de clases. Este wildcard sigue inmediatamente al nombre de una clase.

Entonces mientras que call (Pila.new ()) captura todas las llamadas a constructores donde una instancia exclusivamente de la clase Pila se crea, call(Pila+.new()) captura todas las llamadas a constructores donde una instancia de cualquier subclase de la clase Pila, incluyendo a ella misma, es creada.

Los patrones de clase pueden componerse utilizando los operadores lógicos **&&** (“y”), || (“or”)! (“not”) y los paréntesis.

Formalmente, la sintaxis de un PatróndeClase es especificada por la siguiente BNF.

<PatróndeClase>::= <PatróndeIdentificador> [+] [**[] .. ***] |
!<PatróndeClase> |
<PatróndeClase> **&&** <PatróndeClase> |
<PatróndeClase> || <PatróndeClase>
(<PatróndeClase>**)**

BNF 3 Patrones de clase

1.3.4.3 Avisos

Los avisos definen el código adicional que se ejecuta en los puntos de enlace capturados por los cortes. Los avisos definen el comportamiento que entrecruza toda la funcionalidad, están definidos en función de los cortes. La forma en que el código del aviso es ejecutado depende del tipo del aviso. AspectJ soporta tres tipos de avisos.

El tipo de un aviso determinado interactúa con el punto de enlace sobre el cual está definido. AspectJ divide a los avisos en:

- Aviso "Antes" (Before): aquellos que se ejecutan antes del punto de enlace.
- Aviso "Después" (After): aquellos que se ejecutan después del punto de enlace.
- Aviso "Durante" (Around): aquellos que se ejecutan en lugar del punto de enlace.

El aviso "antes" no agrega mayores dificultades. Por ejemplo, consideremos ahora agregar los controles al método Apilar de la clase Pila definida anteriormente para que no efectúe la operación Apilar si es el caso de qué la pila esté llena.

```
pointcut al_apilar(Pila p) : call(void Pila.Apilar(..)) && target (p);
before(Pila p):al_apilar(p){
if(p.Pila_Llena()){
throw new ExcepcionPilaLlena();
}
}
```

Código 5 Aspectos a través de Aviso Before

De igual forma podemos controlar también que la operación Desapilar no se realice cuando la pila está vacía.

```
pointcut al_desapilar(Pila p) : call(Object Pila.Desapilar(..))
&& target(p);
```

```
before(Pila p):al_desapilar(p){
if(p.Pila_Vacia()){
throw new ExcepcionPilaVacia();
}
}
```

Código 6 Aspectos atreves de Aviso Befote operación Desafilar

El aviso “después” se puede subdividir según como sea el retorno del método: normal, señalando una excepción o sin importar el retorno. Como ejemplo consideremos nuevamente el problema de mostrar la pila cada vez que se agrega un elemento de manera eficiente.

```
pointcut al_apilar_deauno(Pila p): al_apilar(p)
&& !withincode(void Pila.Apilar_Multiple(..));
pointcut al_apilar_multiple(Pila p): call(void Pila.Apilar_Multiple(..))
&& target(p);
pointcut al_apilar_eficiente(Pila p):
al_apilar_deauno(p) && al_apilar_multiple(p);
```

Completemos el ejemplo agregando el comportamiento adicional para mostrar la pila.

```
after (Pila p):al_apilar_eficiente(p){
dibujaEstructura dibujante =new dibujaEstructura();
dibujante.dibujarPila(p);
}
```

Código 7 Aspectos a través de Avisos After

Una vez que se efectúe una modificación en la pila, en este caso agregar un elemento, se insertará el código necesario para dibujar la pila. Suponemos definida una clase dibujaEstructura la cual muestra en pantalla diversas estructuras como pilas, listas, tablas hash, etc. En este aviso no importa la forma en que retornan los métodos Apilar y Apilar_Multiple.

```
After () throwing (Exception e):call( public *.*(..)) {
```

```
contadorDeExcepciones++;
System.out.println(e); }
```

Este aviso se ejecuta cada vez que se genera una excepción en un método público definido en cualquier clase, aumenta el contador de excepciones y muestra en pantalla la excepción. Este aviso tiene en cuenta la forma de retorno de los métodos, cuando retornan señalando una excepción, y señala nuevamente la excepción una vez que se ejecuta su código.

```
After () returning (int cantidaddeelementos):
call(int Pila.Cantidad_Elementos(..)){
System.out.println("Retornando un valor entero"+cantidaddeelementos);
}
```

Este aviso se ejecuta luego de cada punto de enlace capturado por el corte pero sólo en el caso de que su retorno sea normal. El valor retornado puede accederse a través del identificador cantidaddeelementos. Una vez que se ejecutó el aviso se retorna el valor.

El aviso "durante" se ejecuta en lugar del punto de enlace asociado al mismo y no antes o después de éste. El aviso puede retornar un valor, como si fuera un método y por lo tanto debe declarar un tipo de retorno. El tipo de retorno puede ser **void**, indicando el caso en que en el aviso no se desee devolver valor alguno; en esta situación el aviso retorna el valor que devuelve el punto de enlace.

En el cuerpo del aviso se puede ejecutar la computación original del punto de enlace utilizando la palabra reserva **proceed(**<Lista_Argumentos>**)**, que toma como argumentos el contexto expuesto por el aviso y retorna el valor declarado en el aviso.

```
int around(int i) : call(int Numero.suma(int)) && args(i) {
int nuevoRetorno = proceed(i+1);
return nuevoRetorno*3;
}
```

Este aviso captura todas las llamadas al método suma, y en vez de ejecutar directamente el cuerpo del método, empieza a computar el código del aviso.

Este código reformula la llamada, cambiando el argumento de entrada i por i+1; invoca al método suma a través de **proceed** y finalmente el aviso retorna la respuesta multiplicada por tres. Si el valor de retorno del aviso es de clase Object, entonces el resultado del **proceed** es convertido a una representación Object, aún cuando ese valor sea un tipo primitivo. Sin embargo, cuando un aviso retorna un valor de clase Object, ese valor es convertido nuevamente a su representación original. Luego, el siguiente aviso es equivalente al anterior:

```
Object around(int i) : call(int Numero.suma(int)) && args(i) {
Integer nuevoRetorno = proceed(i+1);
return new Integer(nuevoRetorno*3);
}
```

En todos los tipos de avisos todos los parámetros se comportan exactamente como los parámetros de los métodos. La semántica del pasaje de parámetros es pasaje de parámetros por valor. La única manera de cambiar los valores de los parámetros o los valores de retorno de los puntos de enlace es utilizando el aviso "durante".

La declaración de avisos puede describirse formalmente a partir de la siguiente BNF.

<Aviso>::=<Tipo_aviso> **:** <Corte> **{**<Cuerpo>**}**
<Tipo_aviso>::= <Aviso_antes> | <Aviso_despues> | <Aviso_durante>
<Aviso_antes>::= **before (**<Parametros_formales>**)**
<Aviso_despues> ::= **after (**<Parametros_formales>**)** <Forma_terminacion>
<Forma_terminacion>::= **returning** [**(**<Parametro_formal>**)**] |
throwing [**(**<Parametro_formal>**)**] |
[
<Aviso_durante>::= <Nombre_Clase> **around (**<Parametros_formales>**)**
[**throws** <Lista_Clases>]

BNF 4 Definición de avisos

Modelo de comportamiento

Múltiples avisos se pueden aplicar a un mismo punto de enlace. Para determinar el orden de aplicación de los avisos se define una relación de precedencia entre ellos. Las reglas de precedencia se describirán en la sección 3.9.5 Precedencia de aspectos. Una vez que se arriba a un punto de enlace, todos los avisos del sistema son examinados para ver si alguno se aplica al punto de enlace. Aquellos que sí, son agrupados, ordenados según las reglas de precedencia, y ejecutados:

1. Primero, cualquier aviso "durante" es ejecutado, los de mayor precedencia primero. Dentro del código del aviso "durante", la llamada a **proceed()** invoca al próximo aviso "durante" que le sigue en precedencia. Cuando ya no quedan más avisos "durante" se pasa al punto 2.
2. Se ejecutan todos los avisos "antes" según su orden de precedencia (de mayor a menor)
3. Se ejecuta la computación del punto de enlace.
4. La ejecución de los avisos "después normal" y "después con excepción" depende de cómo resultó la ejecución en el punto 3 y de la terminación de avisos "después normal" y "después con excepción" ejecutados anteriormente.
5. Si la terminación es normal todos los avisos "después normal" se ejecutan, los de menor precedencia primero.
6. Si ocurre una excepción todos los avisos "después con excepción" que coinciden con la excepción se ejecutan, los de menor precedencia primero. (Esto significa que los avisos "después con excepción" pueden manejar excepciones señaladas por avisos "después normal" y "después con excepción" de menor precedencia.
7. Se ejecutan todos los "avisos después", los de menor precedencia primero.
8. Una vez que se ejecutan todos los avisos "después" el valor de retorno del punto 3 (si es que hay alguno) es devuelto a la llamada **proceed** correspondiente del punto 1 y ese aviso "durante" continúa su ejecución.

9. Cuando el aviso “durante” finaliza, el control pasa al próximo aviso “durante” con mayor precedencia.
10. Cuando termina el último aviso “durante”, el control retorna al punto de enlace.

Acceso reflexivo

AspectJ provee un variable especial, thisJointPoint, que contiene información reflexiva sobre el punto de enlace actual para ser utilizada por un aviso. Esta variable sólo puede ser usada en el contexto de un aviso y está ligada a un objeto de clase JoinPoint que encapsula la información necesaria.

La razón de su existencia es que algunos cortes pueden agrupar un gran número de puntos de enlaces y es deseable acceder a la información de cada uno.

```
before(Punto p): target(p) && call(*.*(..)){
System.out.println(“accediendo”+ thisJointPoint +“en”+p);
}
```

El ejemplo anterior muestra una manera “bruta” de efectuar trazas sobre todas las llamadas a métodos de la clase Punto.

La variable thisJointPoint puede usarse para acceder tanto a información estática como dinámica sobre los puntos de enlace. Para aquellos casos en los que sólo se desea la información estática, AspectJ provee otra variable especial thisJoinPointStaticPart, de la clase JoinPoint. StaticPart, la cual está ligada a un objeto que contiene únicamente la información estática. Dicho objeto es el mismo objeto que se obtendría con la siguiente expresión:

```
thisJointPoint.getStaticPart()
```

1.3.4.4 Introducciones y declaraciones

Las declaraciones de avisos cambian el comportamiento de sus clases asociadas pero no modifican su estructura estática. Para aquellos conceptos que operan sobre la estructura estática de la jerarquía de clases, AspectJ provee formas de introducción. Cada forma de introducción será miembro del aspecto que lo define, pero introduce un nuevo miembro en otra clase.

En este ejemplo se introduce el atributo booleano atencion en la clase Biblioteca y lo inicializa con el valor true. Como está declarado **private** solamente puede ser accedido dentro del aspecto que incluye la declaración. Si fuera declarado como **public** podría ser accedido por cualquier código. Por omisión se asume como declarado **package-protected** y en este caso puede ser accedido por cualquier código dentro del paquete.

```
private boolean Biblioteca.atencion=true;
```

También se pueden introducir métodos, incluyendo constructores, en una o más clases utilizando un patrón de clase.

```
public Pila.new(Object o){
arreglo = new Object[capacidadMaxima];
ultimo=0;
this.Apilar (o) ;
}
```

El ejemplo anterior introduce un nuevo constructor en la clase Pila. No se puede introducir un constructor en una interface, y si el patrón de clase incluye una interface se producirá un error.

Las **formas de introducción** permiten también declarar que una o más clases objetivo implementarán una interface o heredarán de otra clase desde ese momento en adelante.

```
/* implementa una interface */
```

declare parents: Pila implements PilaAbstracta;
/* hereda de un clase */
declare parents: PilaOrdenada extends Pila;
Como efecto de estas declaraciones PilaOrdenada hereda de la clase Pila y Pila implementa la interface PilaAbstracta.

Una misma declaración puede introducir varios elementos en más de una clase.
public unaFecha (Perro||Gato).fechaUltimaVacuna(){
return fechaVacuna;
}

Introduce dos métodos, uno en la clase Perro y otro en clase Gato; ambos métodos tienen el mismo nombre, no tienen parámetros, retornan un objeto de clase unaFecha y tienen el mismo cuerpo. En el caso que las clases Perro o Gato no tengan definido un atributo fechaVacuna resultará en un error.

Un aspecto puede introducir atributos y métodos tanto en interfaces como en clases.

Formalmente la declaración de formas de introducción se define con BNF:
<Introducciones>::=<IntroduccionesdeAtributos> |
<IntroduccionesdeMetodos> |
<IntroduccionesdeMetodosAbstractos>|
<IntroduccionesdeConstructores>
<IntroduccionesdeAtributos>::=
[<Modificadores>] <Nombre_Clase> <PatrondeClase>**.**<Identificador>
[= <expresión>] **;**
<IntroduccionesdeMetodos>::=
[<Modificadores>] <Nombre_Clase>
<PatrondeClase>**.**<Identificador>**(**<Parametros_formales**>)**
[**throws** <Lista_Clases**>**] **{**<Cuerpo**>}**
<IntroduccionesdeMetodosAbstractos>::=

abstract [<Modificadores>] <Nombre_Clase>
<PatrondeClase>**.**<Identificador>**(**<Parametros_formales>**)**
[**throws** <Lista_Clases>] **;**
<IntroduccionesdeConstructores>::=
[<Modificadores>]
<PatrondeClase>**.new(**<Parametros_formales>**)**
[**throws** <Lista_Clases>] **{**<Cuerpo>**}**

BNF 5 Definición de formas de introducción

La semántica de las formas de introducción es la siguiente:

- Introducción de atributos: el efecto de este tipo de introducción es que todas las clases que coincidan con el patrón de clase soportarán el nuevo atributo especificado en la introducción. Las interfaces que coincidan con el patrón de clase también soportarán el nuevo atributo.

- Introducción de métodos: el efecto de este tipo de introducción es que todas las clases que coincidan con el patrón de clase soportarán el nuevo método especificado en la introducción. Las interfaces también soportarán el nuevo método, aún si el método no es público o abstracto.

- Introducción de métodos abstractos: tiene el mismo efecto que la introducción de un método.

- Introducción de constructores: el efecto de este tipo de introducción es que todas las clases que coincidan con el patrón de clase soportarán el nuevo constructor especificado en la introducción.

Como hemos mencionado anteriormente no se pueden introducir constructores en una interface, luego es un error si el patrón de clase incluye una interface.

La ocurrencia del identificador **this** en el cuerpo de la introducción de un constructor o método, o en el inicializador de una introducción de atributo, hace referencia a la clase objetivo y no al aspecto.

1.3.4.5 Aspectos

Los aspectos son declarados en forma similar a la declaración de clases. El aspecto es la unidad que encapsula puntos de enlace, cortes, avisos, introducciones y declaraciones, además de poder definir sus propios métodos y atributos.

La diferencia con una clase es que un aspecto puede entrecruzar otras clases o aspectos, y que no son directamente instanciados con una expresión **new**, o procesos de clonado o serialización. Los aspectos pueden incluir la definición de un constructor pero dicha definición no debe contener argumentos y no debe señalar excepciones chequeadas.

Un aspecto puede ser definido en un paquete, o como un aspecto interno, esto es o, puede ser miembro de una clase, una interface o un aspecto.

Como primer ejemplo de la definición de aspecto podemos implementar una traza sobre la clase Pila a través de un aspecto TrazadoPila. Por cada llamada a los métodos de la clase Pila se imprime el nombre del método.

```
aspect TrazadoPila{
pointcut LlamadasaPila():call(* Pila.*(..));
before():LlamadasaPila(){
System.out.println("Entrando al método: "+ thisJointPoint);
}after():LlamadasaPila(){
System.out.println("Retornando del método: "+ thisJointPoint);
}
}
```

Código 8. Aspecto de traza para la clase pila

De no utilizar AspectJ, el código de TrazadoPila estaría diseminado por toda la clase Pila al comienzo y al final de cada uno de sus métodos. Si se quiere modificar el mensaje que se muestra o los métodos involucrados en la traza, implicaría revisar todo el código de la clase y realizar los cambios necesarios. Al tenerlo encapsulado como un aspecto dichas modificaciones son triviales y no corremos el riesgo de afectar la funcionalidad básica.

Reconsideremos el problema de agregar controles en los métodos Apilar y Desapilar de la clase Pila. Sin AspectJ dicho código queda diseminado en ambos métodos con los problemas ya mencionados. En cambio si tomamos al control como un aspecto ControlPila quedaría encapsulado en una única unidad.

```
aspect Control Pila{
pointcut al_apilar(Pila p) : call(void Pila.Apilar(..)) && target (p);
pointcut al_desapilar(Pila p) : call(Object Pila.Desapilar(..))
&& target(p);
before(Pila p):al_apilar(p){
if(p.Pila_Llena()){ throw new ExcepcionPilaLlena(); }
}
before(Pila p):al_desapilar(p){
if(p.Pila_Vacia()){ throw new ExcepcionPilaVacia(); }
}
}
```

Código 9. Aspecto de control para la clase pila

Extensión de aspectos

Para manejar la abstracción y composición de los conceptos entrecruzados, los aspectos pueden extenderse en una forma similar a las clases. Sin embargo, la extensión de aspectos agrega nuevas reglas.

Un aspecto, abstracto o concreto, puede heredar de una clase e implementar un conjunto de interfaces. Heredar de una clase no le da al aspecto la habilidad de

instanciarse con una expresión **new**. Una clase no puede heredar o implementar un aspecto. Tal situación es considerada un error.

Los aspectos pueden heredar de otros aspectos, heredando atributos, métodos y cortes. Una restricción importante es que los aspectos solo pueden heredar aspectos abstractos. Se considera un error que un aspecto concreto herede de otro aspecto concreto.

Como ejemplo podemos definir al aspecto TrazaPila en función de un aspecto abstracto TrazaSimple.

```
abstract aspect TrazaSimple{
abstract pointcut PuntosTraza();
before():PuntosTraza(){
System.out.println("Entrando "+ thisJoinPoint);
}
after():PuntosTraza(){
System.out.println("Retornando "+ thisJoinPoint);
}
}
aspect TrazaPila extends TrazaSimple{
pointcut PuntosTraza():call(* Pila.*(..));
}
```

Código 10. Ejemplo de herencia en aspectos

Privilegio de aspectos

El código escrito en un aspecto está sujeto a las mismas reglas de acceso del código Java para referenciar a miembros de clases o aspectos. Por ejemplo, un aspecto no puede referir a miembros con visibilidad package-protected sino está definido en el mismo paquete.

Existen algunas situaciones donde es necesario para los aspectos acceder a recursos privados o protegidos de otras clases. Para permitir esto se puede declarar al aspecto como privilegiado (**privileged**). Un aspecto privilegiado puede acceder a todos los atributos y métodos, incluso a los privados. Por ejemplo:

```
privileged aspect A {
. . .
before(Pila p):Corte1(p){
. . .
p.capacidadMaxima=25;
. . .
}
. . .
}
```

El aspecto A al ser declarado como privilegiado puede acceder al atributo privado capacidadMaxima de la clase Pila.

Precedencia de aspectos

Existen situaciones en donde los avisos de un aspecto deben preceder a los avisos de otro aspecto. Por tal motivo, un aspecto puede declarar que "domina" a otro aspecto estableciendo que los avisos definidos en él tienen prioridad sobre los avisos en el aspecto que domina.

aspect <Identificador> **dominates** < PatróndeClase> {. . .}

Como ejemplo, supongamos que tenemos definidos dos aspectos Precondicion y Trazado, y queremos que el aspecto Precondicion tenga mayor precedencia que el aspecto Trazado para que la traza se efectúe sólo si las precondiciones son satisfechas. Luego aspect Precondicion dominates Trazado {. . .} logra que el aspecto Precondicion, incluyendo sus avisos, tenga mayor precedencia que el aspecto Trazado y sus avisos serán ejecutados primero.

La declaración “dominates” forma parte de las reglas de precedencia de avisos, además de otros elementos como la herencia de aspectos.Las reglas que determinan la precedencia de avisos son las siguientes:

Si dos avisos están definidos en diferentes aspectos entonces:

- Si el aspecto A domina al aspecto B entonces los avisos en A tienen precedencia sobre los avisos en B.
- En otro caso, si el aspecto A hereda del aspecto B entonces todos los avisos en A tienen precedencia sobre los avisos en B.
- En otro caso, si no hay relación entre los aspectos entonces es indefinido quien tiene mayor precedencia.

Si dos avisos están definidos en el mismo aspecto:

- Si ambos son avisos “después” entonces el que esté definido último (según la estructura léxica del programa) tiene mayor precedencia.
- En otro caso, aquel que aparezca primero tiene mayor precedencia.

Cuando múltiples avisos se aplican a un mismo punto de enlace, el orden de resolución se basa en la precedencia de los avisos.

BNF completa

La siguiente BNF describe la declaración de aspectos en AspectJ agrupando todos los conceptos que vimos hasta ahora.

<Aspecto>::=[**privileged**] **aspect** <Identificador>
[**extends** <Nombre_Clase>]
[**implements** <Lista_Clases>]
[**dominates** <Lista_Clases>]
{
{<Atributos>}
{<Métodos>}
{<Def_Cortes>}
{<Introducciones>}

{<Aviso>}**}**

<Def_Cortes>::= **abstract** [<Modificadores>]

pointcut <Identificador>**(**<Parametros_formales>**);** |

[<Modificadores>]

pointcut <Identificador>**(**<Parametros_formales>**):**<Corte>**;**

<Introducciones>::=<IntroduccionesdeAtributos> |

<IntroduccionesdeMetodos> |

<IntroduccionesdeMetodosAbstractos>|

<IntroduccionesdeConstructores>

<IntroduccionesdeAtributos>::=

[<Modificadores>] <Nombre_Clase> <PatrondeClase>**.**<Identificador>

[= <expresión>] **;**

<IntroduccionesdeMetodos>::=

[<Modificadores>] <Nombre_Clase>

<PatrondeClase>**.**<Identificador>**(**<Parametros_formales>**)**

[**throws** <Lista_Clases>] {<Cuerpo>}

<IntroduccionesdeMetodosAbstractos>::=

abstract [<Modificadores>] <Nombre_Clase>

<PatrondeClase>**.**<Identificador>**(**<Parametros_formales>**)**

[**throws** <Lista_Clases>] **;**

<IntroduccionesdeConstructores>::=

[<Modificadores>] <PatrondeClase>**.new(**<Parametros_formales>**)**

[**throws** <Lista_Clases>] {<Cuerpo>}

<Aviso>::=<Tipo_aviso> : <Corte> {<Cuerpo>}

<Tipo_aviso>::= <Aviso_antes> | <Aviso_despues> | <Aviso_durante>

<Aviso_antes>::= **before (**<Parametros_formales>**)**

<Aviso_despues> ::= **after (**<Parametros_formales>**)** <Forma_terminacion>

<Forma_terminacion>::= **returning** [**(**<Parametro_formal>**)**] |

throwing [**(**<Parametro_formal>**)**] | λ

<Aviso_durante>::= <Nombre_Clase> **around (**<Parametros_formales>**)**

[**throws** <Lista_Clases>]

BNF 6 Definición de aspecto

1.3.4.6 Evaluación

En sí, describimos de AspectJ aquellos conceptos necesarios para poder programar con aspectos por primera vez y entender la semántica de un programa escrito en AspectJ.

La programación en AspectJ es directa si el programador conoce el lenguaje Java. Los aspectos son muy similares a las clases, los avisos a los métodos. Luego la declaración de aspectos es sencilla, el programador con poco esfuerzo y sin mayores dificultades agrega aspectos a su implementación con las ventajas que esto trae. Al ser una extensión compatible de Java los costos de aprendizaje son mínimos.

AspectJ soporta la facilidad "plug-in plug-out" de aspectos. Los aspectos pueden agregarse o removerse de la funcionalidad básica con sólo incluirlos o no en la compilación. Así el programador puede intentar introducir aspectos en su aplicación y de no satisfacerle los puede remover, manteniendo la aplicación en su estado original.

Uno de los puntos fuertes de AspectJ es la composición de cortes a través de operadores booleanos, la cual es a la vez simple y poderosa. Por lo tanto la expresividad del lenguaje es destacable.

Recomendamos a los estudiantes o particulares que quieren construir un software destinado a la aplicabilidad del paradigma de Programación Orientada a Aspectos a enfocarse en el lenguaje *AspectJ* para el desarrollo de sus aplicaciones, porque es una herramienta que presenta grandes condiciones y facilidades para la separación de "incumbencias" y el trabajo con programación de aspectos.

1.3.5 Eclipse

Eclipse es un entorno de desarrollo integrado de código abierto independiente de una plataforma para desarrollar lo que el proyecto llama "Aplicaciones de Cliente Enriquecido", opuesto a las aplicaciones "Cliente-liviano" basadas en navegadores. Esta plataforma, típicamente ha sido usada para desarrollar entornos de desarrollo integrados (IDE), como el IDE de Java llamado Java Development Toolkit (JDT) y el compilador (ECJ) que se entrega como parte de Eclipse (y que son usados también para desarrollar el mismo Eclipse).

Eclipse fue desarrollado originalmente por IBM como el sucesor de su familia de herramientas para VisualAge. Eclipse es ahora desarrollado por la Fundación Eclipse, una organización independiente sin ánimo de lucro que fomenta una comunidad de código abierto y un conjunto de productos complementarios, capacidades y servicios.

Arquitectura

La base para Eclipse es la Plataforma de cliente enriquecido (del Inglés Rich Client Platform RCP). Los siguientes componentes constituyen la plataforma de cliente enriquecido:

- Plataforma principal - inicio de Eclipse, ejecución de plugins
- OSGi - una plataforma para bundling estándar.
- El Standard Widget Toolkit (SWT) - Un widget toolkit portable.
- JFace - manejo de archivos, manejo de texto, editores de texto
- El Workbench de Eclipse - vistas, editores, perspectivas, asistentes

Los widgets de Eclipse están implementados por una herramienta de widget para Java llamada SWT, a diferencia de la mayoría de las aplicaciones Java, que usan las opciones estándar Abstract Window Toolkit (AWT) o Swing. La interfaz de

usuario de Eclipse también tiene una capa GUI intermedia llamada JFace, la cual simplifica la construcción de aplicaciones basadas en SWT.

El entorno de desarrollo integrado (IDE) de Eclipse emplea módulos (en inglés *plug-in*) para proporcionar toda su funcionalidad al frente de la plataforma de cliente rico, a diferencia de otros entornos monolíticos donde las funcionalidades están todas incluidas, las necesite el usuario o no.

Este mecanismo de módulos es una plataforma ligera para componentes de software. Adicionalmente a permitirle a Eclipse extenderse usando otros lenguajes de programación como son C/C++ y Python, permite a Eclipse trabajar con lenguajes para procesado de texto como LaTeX, aplicaciones en red como Telnet y Sistema de gestión de base de datos. La arquitectura plugin permite escribir cualquier extensión deseada en el ambiente, como sería Gestión de la configuración. Se provee soporte para Java y CVS en el SDK de Eclipse. Y no tiene por qué ser usado únicamente para soportar otros lenguajes de programación.

La definición que da el proyecto Eclipse acerca de su software es: "*una especie de herramienta universal - un IDE abierto y extensible para todo y nada en particular*".

En cuanto a las aplicaciones clientes, eclipse provee al programador con frameworks muy ricos para el desarrollo de aplicaciones gráficas, definición y manipulación de modelos de software, aplicaciones web, etc. Por ejemplo, GEF (Graphic Editing Framework - Framework para la edición gráfica) es un plugin de Eclipse para el desarrollo de editores visuales que pueden ir desde procesadores de texto wysiwyg hasta editores de diagramas UML, interfaces gráficas para el usuario (GUI), etc. Dado que los editores realizados con GEF "viven" dentro de Eclipse, además de poder ser usados conjuntamente con otros plugins, hacen uso de su interfaz gráfica personalizable y profesional.

El SDK de Eclipse incluye las herramientas de desarrollo de Java, ofreciendo un IDE con un compilador de Java interno y un modelo completo de los archivos fuente de Java. Esto permite técnicas avanzadas de refactorización y análisis de código. El IDE también hace uso de un espacio de trabajo, en este caso un grupo de metadata en un espacio para archivos plano, permitiendo modificaciones externas a los archivos en tanto se refresque el espacio de trabajo correspondiente.

Características

Eclipse dispone de las siguientes características.

- Editor de texto
- Resaltado de sintaxis.
- Compilación en tiempo real.
- Pruebas unitarias con JUnit.
- Control de versiones con CVS.
- Integración con Ant,
- Asistentes (*wizards*) para creación de proyectos, clases, tests, etc.
- Refactorización.

Asimismo, a través de "plugins" libremente disponibles es posible añadir.

- Control de versiones con Subversion, via Subclipse.
- Integración con Hibernate, via Hibernate Tools.

Proyectos IDE en lenguajes sobre eclipse

- **AspectJ** es una extensión del lenguaje Java orientada a aspectos.
- **Proyecto de herramientas de desarrollo en C/C++** (CDT) trabaja para proveer un ambiente integrado de desarrollo completamente funcional para C y C++ para la Plataforma Eclipse.

- **Proyecto IDE de COBOL para Eclipse** (COBOL) construye un Ambiente integrado de desarrollo (IDE) completamente funcional para COBOL en la plataforma Eclipse.
- **Herramientas de Desarrollo de Java** (JDT) provee las herramientas que implementa un IDE de Java, soportando el desarrollo de cualquier aplicación Java, incluyendo los plug-ins de Eclipse.
- **Photran** es un IDE completamente funcional para Fortran con soporte para Refactorizacion.
- **PHP Development Tools** trabaja para proveer un IDE completamente funcional para PHP para la plataforma Eclipse.
- **Wolfram Wowkbench** es un IDE basado en Eclipse también disponible como plugin para Eclipse para el lenguaje Matemática.
- **PyDev** un IDE completamente funcional para Python con soporte para Refactorizacion, y depurador graficó.

1.3.6 Base de Datos Mysql

MySQL es un sistema de administración de bases de datos (*Database Management System, DBMS*) para bases de datos relacionales. Así, MySQL no es más que una aplicación que permite gestionar archivos llamados de bases de datos .Existen muchos tipos de bases de datos, desde un simple archivo hasta sistemas relacionales orientados a objetos. MySQL, como base de datos relacional, utiliza multiples tablas para almacenar y organizar la información.

MySQL fue escrito en C y C++ y destaca por su gran adaptación a diferentes entornos de desarrollo, permitiendo su interactuación con los lenguajes de programación más utilizados como PHP, Perl y Java y su integración en distintos sistemas operativos.

1.3.7 Servidor Web Tomcat

Tomcat es un servidor web con soporte de servlets y JSPs. Tomcat no es un servidor de aplicaciones, como JBoss o JOnAS. Incluye el compilador Jasper, que compila JSPs convirtiéndolas en servlets. El motor de servlets de Tomcat a menudo se presenta en combinación con el servidor web Apache.

Tomcat puede funcionar como servidor web por sí mismo. En sus inicios existió la percepción de que el uso de Tomcat de forma autónoma era sólo recomendable para entornos de desarrollo y entornos con requisitos mínimos de velocidad y gestión de transacciones.

Hoy en día ya no existe esa percepción y Tomcat es usado como servidor web autónomo en entornos con alto nivel de tráfico y alta disponibilidad. Dado que Tomcat fue escrito en Java, funciona en cualquier sistema operativo que disponga de la máquina virtual Java.

Estado de su desarrollo

Tomcat es mantenido y desarrollado por miembros de la Apache Software Foundation y voluntarios independientes. Los usuarios disponen de libre acceso a su código fuente y a su forma binaria en los términos establecidos en la Apache Software Licence. Las primeras distribuciones de Tomcat fueron las versiones 3.0.x. Las versiones más recientes son las 6.x, que implementan las especificaciones de Servlet 2.5 y de JSP 2.1. A partir de la versión 4.0, Jakarta Tomcat utiliza el contenedor de servlets.

Estructura de directorios

La jerarquía de directorios de instalación de Tomcat incluye:

- bin - arranque, cierre, y otros scripts y ejecutables

- common - clases comunes que pueden utilizar y las aplicaciones web
- conf - ficheros XML y los correspondientes DTD para la configuración de Tomcat
- logs - logs de Catalina y de las aplicaciones
- server - clases utilizadas solamente por Catalina
- shared - clases compartidas por todas las aplicaciones web
- webapps - directorio que contiene las aplicaciones web
- work - almacenamiento temporal de ficheros y directorios

1.3.8 Microsoft Virtual PC

Windows Virtual PC llamado Microsoft Virtual PC es un software gestor de virtualización desarrollado por Connectix y comprado por Microsoft para crear equipos virtuales. Es decir, su función es emular mediante virtualización, un hardware sobre el que funcione un determinado sistema operativo. Con esto se puede conseguir ejecutar varios sistemas operativos en la misma máquina a la vez y hacen que se comuniquen entre ellos.

En informática una máquina virtual es un software que emula a una computadora y puede ejecutar programas como si fuese una computadora real. Este software en un principio fue definido como "un duplicado eficiente y aislado de una máquina física". La acepción del término actualmente incluye a máquinas virtuales que no tienen ninguna equivalencia directa con ningún hardware real.

Una característica esencial de las máquinas virtuales es que los procesos que ejecutan están limitados por los recursos y abstracciones proporcionados por ellas. Estos procesos no pueden escaparse de esta "computadora virtual".

Microsoft Virtual PC es una herramienta que permite utilizar diferentes Sistemas Operativos en el ordenador sin la necesidad de reiniciarlo o instalar cada uno en una partición distinta.

Si bien manejar dos plataformas diferentes hace algunos años atrás parecía imposible, con el paso del tiempo y el avance de los programas ahora se puede llevar a cabo de manera muy sencilla.

Cambiar de sistema será una tarea de niños: sólo hay que ejecutar Microsoft Virtual PC y seleccionar cual es la que se desea utilizar.

Para llevar adelante su función crea ordenadores virtuales y cada uno de ellos se encarga de emular una plataforma diferente. Vale la pena mencionar que únicamente sólo puede instalarse en las versiones de Windows, logrando emular eficientemente: 98, NT, 2000, 2003, XP, Vista y OS/2.

1.3.9 Herramientas CASE

Power Designer

Es una herramienta para el análisis, diseño inteligente y construcción sólida de una base de datos y un desarrollo orientado a modelos de datos a nivel físico y conceptual, que da a los desarrolladores Cliente/Servidor la más firme base para aplicaciones de alto rendimiento.

Ofrece un acercamiento de diseño para optimizar las estructuras de las bases de datos. Capturando el flujo de datos de su organización, puede crear un modelo conceptual y físico de la base de datos.

PowerDesigner para Arquitectura Empresarial también brinda un enfoque basado en modelos, el cual permite alinear al negocio con la tecnología de información, facilitando la implementación de arquitecturas efectivas de información empresarial. Brinda potentes técnicas de análisis, diseño y gestión de metadatos a la empresa.

PowerDesigner combina varias técnicas estándar de modelamiento con herramientas líder de desarrollo, como .NET, Sybase WorkSpace, Sybase Powerbuilder, Java y Eclipse, para darle a las empresas soluciones de análisis de negocio y de diseño formal de base de datos. Además trabaja con más de 60 bases de datos relacionales.

Rational Rose

Rational Rose es la herramienta case que comercializan los desarrolladores de UML. Esta herramienta propone la utilización de cuatro tipos de modelo para realizar un diseño del sistema, utilizando una vista estática y otra dinámica de los modelos del sistema, uno lógico y otro físico. Permite crear y refinar estas vistas creando de esta forma un modelo completo que representa el dominio del problema y el sistema de software.

- Diagramas de Implementación. Los diagramas de implementación muestran los aspectos físicos del sistema. Incluyen la estructura del código fuente y la implementación.
- Diagramas de Comportamiento o Interacción. Muestran las interacciones entre un conjunto de objetos, ordenadas según el tiempo en que tienen lugar. Un diagrama de secuencia representa una forma de indicar el período durante el que un objeto está desarrollando una acción directamente o a través de un procedimiento.
- Diagramas de Casos de uso. Los diagramas de casos de uso se utilizan para ilustrar los requerimientos del sistema al mostrar cómo reacciona una respuesta a eventos que se producen en el mismo.
- Diagramas de Clases. Los diagramas de clases representan un conjunto de elementos del modelo que son estáticos, como las clases y los tipos, sus contenidos y las relaciones que se establecen entre ellos, tenemos los elementos que se pueden clasificar como estáticos son los siguientes:

  Paquete: Es el mecanismo de que dispone UML para organizar sus elementos en grupos, se representa un grupo de elementos del modelo. Un

sistema es un único paquete que contiene el resto del sistema, por lo tanto, un paquete debe poder anidarse, permitiéndose que un paquete contenga otro paquete.

Clases: Una clase representa un conjunto de objetos que tienen una estructura, un comportamiento y unas relaciones con propiedades parecidas. Describe un conjunto de objetos que comparte los mismos atributos, operaciones, métodos, relaciones y significado.

En UML una clase es una implementación de un tipo. Los componentes de una clase son:

Atributo. Se corresponde con las propiedades de una clase o un tipo. Se identifica mediante un nombre. Existen atributos simples y complejos. **Operación.** Conocido como método, es un servicio proporcionado por la clase que puede ser solicitado por otras clases y que produce un comportamiento en ellas cuando se realiza.

UML (Unified Modeling Language) es un lenguaje para especificar, construir, visualizar y documentar los artefactos de un sistema de software orientado a objetos (OO). Un artefacto es una información que es utilizada o producida mediante un proceso de desarrollo de software.

El UML es una técnica de modelado de objetos y como tal supone una abstracción de un sistema para llegar a construirlo en términos concretos. El modelado no es más que la construcción de un modelo a partir de una especificación.

Un modelo es una abstracción de algo, que se elabora para comprender ese algo antes de construirlo. El modelo omite detalles que no resultan esenciales para la comprensión del original y por lo tanto facilita dicha comprensión.

CAPÍTULO II

"DESCRIPCIÓN, ANÁLISIS E INTERPRETACIÓN DE RESULTADOS"

2.1 Breve descripción de la Universidad Técnica de Cotopaxi

La Universidad Técnica de Cotopaxi, es una Institución de Educación Superior Pública, Laica y Gratuita, creada mediante Ley promulgada en el Registro Oficial N.- 618 del 24 de enero del 1995, y que forma parte del Sistema Nacional de Educación Superior Ecuatoriano. Se rige por la Constitución Política del Estado, la de Educación Superior y otras leyes conexas. Es una institución universitaria sin fines de lucro que orienta su trabajo hacia los sectores urbanos, marginales y campesinos; que busca la verdad y la afirmación de la identidad nacional, y que asume con responsabilidad el aseguramiento de la libertad en la producción y difusión de los conocimientos y del pensamiento democrático y progresista para el desarrollo de la conciencia antiimperialista del pueblo.

En la institución actualmente se forman profesionales en las siguientes áreas de especialidades: Ciencias Aplicadas, Ciencias Agropecuarias, Ambientales y Veterinarias, Ciencias Humanísticas y del Hombre.

La Universidad Técnica de Cotopaxi; es una institución educativa con adecuados niveles de pertinencia y calidad, logrados a través de la concientización y difusión de la ciencia, cultura, arte y los conocimientos ancestrales. Contribuye con una acción transformadora en la lucha por alcanzar una sociedad más justa equitativa y solidaria. Por ello, la Universidad Técnica de Cotopaxi asume su identidad con gran responsabilidad: "Por la vinculación de la Universidad con el Pueblo".

Gracias a la información obtenida de la Guía Académica de la Universidad Técnica de Cotopaxi desarrollada por el Ing. M.SC. Hernán Yánez.
La Universidad Técnica de Cotopaxi, es una Institución Educativa en evolución, que forma profesionales útiles para la sociedad, con conciencia social, identidad propia, amor a la provincia y al país; pero sobre todo con valores de colaboración a la gente pobre y olvidada por el estado. Forma profesionales inquietantes a la investigación, al trabajo; buscadores de un mundo más humano, más justo y progresista.

2.1.1 Descripción de la especialidad de Informática y Sistemas Computacionales

La Informática y los sistemas de Computadora es una carrera de tecnología y futuro, además; el Ingeniero en Informática y sistemas Computacionales responde a la realidad científica y tecnológica de nuestro país; diestro en la utilización de herramientas informáticas; diseña, opera, evalúa proyectos y procesos de desarrollo informático, redes de computadoras; es un eficiente administrador informático, capacitado para resolver grandes cambios tecnológicos y ponerlos a disposición de la colectividad.

Un profesional en la especialidad de Informática y Sistemas Computacionales está inmerso en el diseño de sistemas informáticos, base de datos, sistemas de computación, redes de computadoras, arquitecturas de equipos de computo, mantenimiento de computadoras; también es considerado como analista, programador de sistemas, diseñador de aplicaciones; considerado como asesor de procesos de automatización, un excelente administrador de sistemas basados en Internet, hábil como auditor informático, Gerente o Jefe de departamentos de sistemas.

La Especialidad de Informática y Sistemas Computacionales es una ciencia que forma profesionales capaces de abordar conocimientos científicos y prácticos en

todo lo referente al uso-utilidad de hardware y software, que a su vez forman un equipo de cómputo.

En la actualidad una computadora se ha convertido en la herramienta básica para todo profesional y para toda aquella persona que enmarca su éxito en conjunto con la tecnología. Materias como base de datos, programación, ingeniería de software, mantenimiento y arquitectura de computadoras, programación visual, redes, electricidad, electrónica, sistemas lineales, sistemas distribuidos, inteligencia artificial, entre muchas otras materias; hacen de la Informática y Sistemas Computacionales una profesión de mucha actitud científica e investigativa.

2.2 Análisis e interpretación de resultados de las entrevistas realizadas a los Docentes de Informática y Sistema Computacionales, encargados de la materia respectiva

La investigación del proyecto: "**DESARROLLO DE UN SOFTWARE PROTOTIPO DE VENTAS PARA EL APRENDIZAJE DEL PARADIGMA DE PROGRAMACIÓN ORIENTADA A ASPECTOS EN LA ESPECIALIDAD DE INGENIERÍA EN INFORMÁTICA Y SISTEMAS COMPUTACIONALES DE CIYA - UTC"**, llevó a la necesidad de aplicar los instrumentos de investigación como son la entrevista y la encuesta (ver Anexo I y Anexo II Entrevista y encuesta) realizadas con el fin de recolectar la información necesaria para el desarrollo del software prototipo.

2.2.1 Entrevista dirigida a los docentes

La entrevista se realizó con los Docentes encargados de la materia de Programación Orientada a Objetos, que es la que abarca el paradigma de Programación Orientado a Aspectos. En este caso el número de docentes muestra la siguiente tabla.

TABLA Nº 2.1 DOCENTES POO

DOCENTES	Nº	%
Docentes de Programación Orientada a Objetos	2	100
TOTAL	2	100

FUENTE: Docentes de Ingeniería en Informática y Sistemas Computacionales UTC
ELABORADO POR: Autora del proyecto

De la entrevista realizada a los *Docentes encargados de la materia de POO,* en la especialidad de Informática y Sistemas Computacionales, sobre el conocimiento del paradigma de POA, se obtuvieron los siguientes resultados:

Pregunta Nº 1: ¿Conoce usted sobre el paradigma de Programación Orientada a Aspectos, le parece interesante este tema? ¿Por qué?

Docente 1: Si, porque la Programación Orientada a Aspectos, se basa en heredar propiedades, características y aspectos de otros objetos, también se puede decir que es un nuevo paradigma que se encuentra en investigación.

Docente 2: Si, porque se aplican temas muy interesantes sobre encapsulamiento y manejo de eventos; Ya que este paradigma de programación POA consiste en descomponer un sistema en módulos independientes.

Análisis e interpretación*:* Satisfactoriamente para el desarrollo de esta tesis, los docentes entrevistados, y que a su vez son encargados de la cátedra de Programación Orientada a Objetos, materia que influye o prepara para el entendimiento de la Programación Orientada a Aspectos; están capacitados y tienen conocimiento del tema, lo que ayudará a un buen desarrollo sistemático del tema de tesis y el desarrollo del software fruto de las concepciones científicas de la POA y su manejo.

Pregunta Nº 2: ¿Piensa usted que la Programación Orientada a Aspectos es una evolución de la Programación Orientada a Objetos?

Docente 1: Si, porque aparte de utilizar características y propiedades se tiene aspectos, es decir es una adaptación del paradigma de Programación Orientada a Aspectos con Programación Orientada a Objetos.

Docente 2: Si, se asume que este tipo de programación optimiza la tradicional orientada a objetos. La Programación Orientada a Aspectos intenta resolver interés para facilitar código repetitivo.

Análisis e interpretación*:* La concordancia para la pregunta dos es muy clara al tener como resultado que los dos docentes entrevistados saben de la importancia de la Programación Orientada a Aspectos y sobre todo que presenta mayores ventajas en el desarrollo de software, y que por lo tanto es una evolución de la tradicional programación, es decir, la Programación Orientada a Objetos.

Pregunta Nº 3: ¿Cree usted que un software desarrollado con Programación Orientada a Objetos será mejor que uno con Programación Orientada a Aspectos?

Docente 1: Igual capacidad, porque esto depende de la calidad del software no incluye en mucho el paradigma que se escoja.

Docente 2: No, porque si se habla de mejora, lógicamente la orientada a aspectos es la mejor. Ya que con la POO el desarrollo de un software se complica demasiado.

Análisis e interpretación*:* Los docentes entrevistados no coinciden en sus respuestas en esta pregunta, al decir el un docente que la POO es de igual capacidad que la Programación Orientada a Aspectos; el otro docente dice que la Programación Orientada a Objetos no tiene la misma capacidad que la POA. Ante estos dos criterios, la respuesta correcta es que las condiciones y características hacen de la POA una evolución de la POO, por lo tanto esta será mejor que su antecesora.

Pregunta Nº 4: ¿Cree usted que la Programación Orientada a Aspectos facilita el desarrollo de un software?

Docente 1: Si, porque esta metodología o paradigma maneja aspectos más productibles que la orientada a objetos. Ayudando en el desarrollo de software.

Docente 2: Si, porque aplica nuevas metodologías como por ejemplo la métrica. También una interfaz fácil de manejar.

Análisis e interpretación*:* Está pregunta acredita el desarrollo de la tesis al saber que los dos catedráticos encuestados están de acuerdo que la POA facilita y mejora el desarrollo de software y aplicaciones. Al saber que al utilizar el paradigma POA optimiza la escritura de líneas de código y por ende la corrección de errores.

Pregunta Nº 5: ¿Piensa usted que teoría y práctica son necesarias para el aprendizaje de nuevos conocimientos?

Docente 1: Si, porque al realizar la práctica se la combina con la teoría para reforzar los conocimientos impartidos.

Docente 2: Si, porque aplicando la práctica mediante ejemplos se divisa el conocimiento de la teoría aprendida.

Síntesis: Los docentes entrevistados coinciden en la respuesta al tener muy en cuenta ellos; por su propia experiencia, que la práctica siempre será el complemento ideal para toda teoría estudiada, al igual, que la teoría debe ser reforzado con la práctica para tener una excelente enseñanza pedagógica.

Pregunta Nº 6: ¿Si usted tuviera la oportunidad de utilizar un software, aplicando Programación Orientada a Aspectos debería ser?

Docente 1: Entendible y eficiente, porque se basa en aspectos más específicos y entendibles. Para conocer como se aplica la Programación Orientada a Aspectos.

Docente 2: Entendible y eficiente, porque si son lenguajes de quinta generación se asume que son de fácil entendimiento. Para el conocimiento de este paradigma.

Análisis e interpretación*:* En la pregunta seis los docentes entrevistados confirman en los resultados que cuando ellos tengan la oportunidad de manipular un software aplicando Programación Orientada a Aspectos, debe ser entendible y eficiente; al igual que el software contenga la eficacia exacta para lo que fue creado, las seguridades posibles, y que sea de un interfaz agradable al usuario.

Pregunta Nº 7: ¿Cree usted que los estudiantes deberían conocer sobre la Programación Orientada a Aspectos?

Docente 1: Si, porque en la actualidad es necesario conocer y manejar ahora objetos puros y eso es lo que los estudiantes deben llegan a aprender y tener interés por la investigación ya que día a día se aprende nuevas cosas.

Docente 2: Si, porque se aplica en el campo laboral. Esto les permite conocer y actualizarse en la programación, ya que algunos estudiantes no conocen la existencia de este nuevo paradigma de programación.

Análisis e interpretación*:* Los datos obtenidos para la pregunta siete, como resultado de la entrevista realizada con los dos docentes de Informática y Sistemas Computacionales; aportan satisfactoriamente al desarrollo de la tesis, al encontrar en sus respuestas que, en verdad es indispensable que los estudiantes conozcan sobre la Programación Orientada a Aspectos; ya que esto deben saber manejar, y que les será útil en su vida profesional y en el campo laboral.

2.2.2 Análisis e interpretación de resultados de las encuestas realizadas a los estudiantes de Informática y Sistemas Computacionales

A continuación se muestra los resultados obtenidos luego de la aplicación del instrumento de investigación, como es la encuesta (ver Anexo Nº 2.2 Encuesta.) a los estudiantes de Informática y Sistemas Computacionales, a partir de sexto hasta octavo ciclo de dicha carrera sobre el conocimiento del paradigma de Programación Orientada a Aspectos, los mismos que son representados en tablas, para luego realizar la interpretación a través de gráficos estadísticos en pastel y finalmente concluir con la realización de un análisis e interpretación de resultados obtenidos de cada pregunta aplicada.

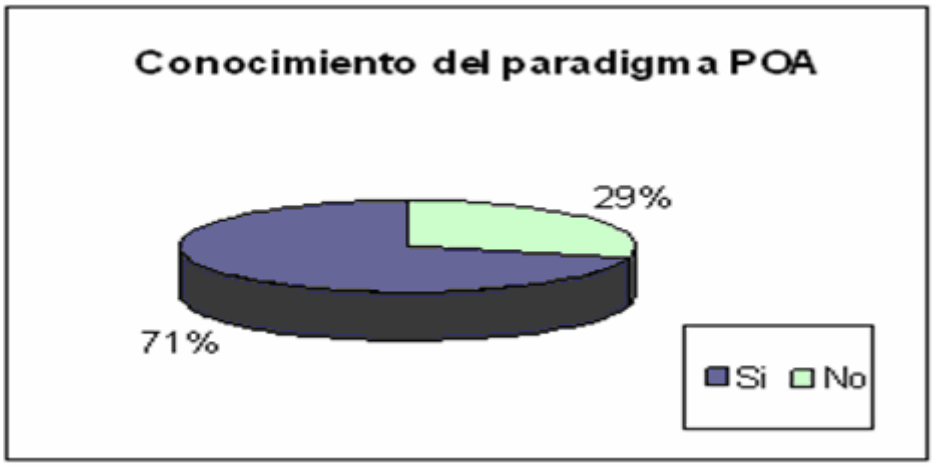
Pregunta Nº 1: ¿Conoce usted sobre el paradigma de Programación Orientada a Aspectos?

TABLA Nº 2.2 CONOCIMIENTO DEL PARADIGMA POA

Alternativas	Frecuencias	Porcentaje
Si	71	71
No	29	29
Total	100	100%

FUENTE: Estudiantes de Ingeniería en Informática y Sistemas Computacionales UTC
ELABORADO POR: Autora del proyecto

GRAFICÓ Nº 2. 1 CONOCIMIENTO DEL PARADIGMA POA



FUENTE: Estudiantes de Ingeniería en Informática y Sistemas Computacionales UTC
ELABORADO POR: Autora del proyecto

Análisis e interpretación de resultados.

Las encuestas realizadas a los estudiantes de la especialidad de informática y sistemas computacionales muestran que los estudiantes tienen pocos conocimientos y otros desconocen la existencia sobre el paradigma de Programación Orientada a Aspectos, dejando una emoción a nivel de investigadora, ya que los resultados da una excelente apertura al tema de tesis y a la ejecución del mismo.

Pregunta Nº 2: ¿El aprendizaje a través de la Programación Orientada a Aspectos con una computadora; para usted es?

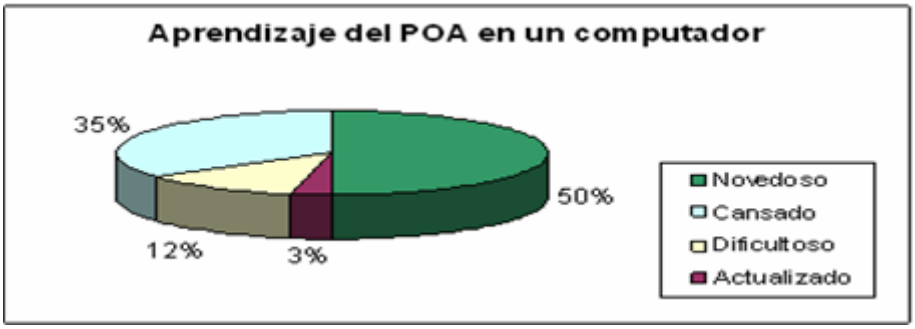
TABLA Nº 2. 3 APRENDIZAJE DEL POA EN UN COMPUTADOR

Alternativas	Frecuencias	Porcentaje
Novedoso	50	50
Cansado	35	35
Dificultoso	12	12
Actualizado	3	3
Total	100	100%

FUENTE: Estudiantes de Ingeniería en Informática y Sistemas Computacionales UTC

ELABORADO POR: Autora del proyecto

GRAFICÓ Nº 2.2 APRENDIZAJE DEL POA EN UN COMPUTADOR



FUENTE: Estudiantes de Ingeniería en Informática y Sistemas Computacionales UTC
ELABORADO POR: Autora del proyecto

Análisis e interpretación de resultados.

El mayor porcentaje de estudiantes encuestados, consideran de lo novedoso; que es aprender el paradigma POA en una computadora, los grandes beneficios que les haría el recibir una práctica del paradigma POA en sus laboratorios de informática. El menor porcentaje de encuestados dicen que el aprender el POA en una computadora será muy cansado y dificultoso. No es dable saber esto, ya que un estudiante y futuro profesional de la carrera en informática y sistemas computacionales, toda su vida tendrá que utilizar una computadora y equipos de cómputo.

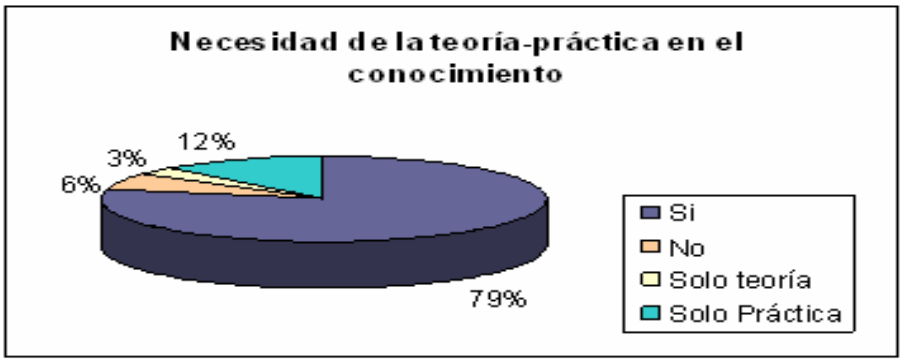
Pregunta Nº 3: ¿Piensa que teoría-práctica son necesarias para la asimilación de conocimientos de la Programación Orientada a Aspectos?

TABLA Nº 2. 4 NECESIDAD DE LA TEORÍA-PRÁCTICA EN EL CONOCIMIENTO

Alternativas	Frecuencias	Porcentaje
Si	79	79
Solo práctica	12	12
No	6	6
Solo teoría	3	3
Total	100	100%

FUENTE: Estudiantes de Ingeniería en informática y sistemas computacionales UTC
ELABORADO POR: Autora del proyecto

GRAFICÓ Nº 2. 3 TEORÍA-PRÁCTICA EN EL CONOCIMIENTO



FUENTE: Estudiantes de Ingeniería en informática y sistemas computacionales UTC
ELABORADO POR: Autora del proyecto

Análisis e interpretación de resultados.

Es factible aplicar y desarrollar el software aplicando la POA, ya que un buen porcentaje de los estudiantes de la especialidad de Informática y Sistemas Computacionales, están de acuerdo en la necesidad de tener la teoría y la práctica como la mejor estrategia para el aprendizaje eficiente y eficaz de conocimientos. Un pequeño porcentaje piensan que para el aprendizaje de conocimientos solo se requiere la práctica; el otro porcentaje que no es necesario teoría y práctica en la asimilación de conocimientos; y el mínimo porcentaje se inclinan a tan solo la teoría en el aprendizaje de nuevas concepciones y definiciones.

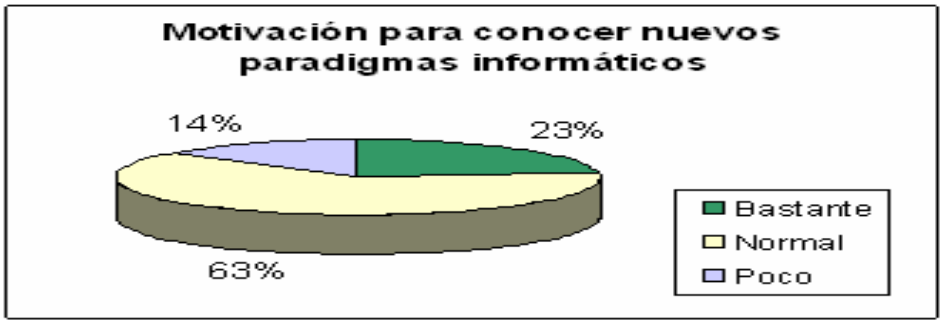
Pregunta Nº 4: ¿Su motivación para conocer nuevos programas informáticos y sobre todo del paradigma de Programación Orientada a Aspectos, es?

TABLA Nº 2. 5 MOTIVACIÓN PARA CONOCER NUEVOS PARADIGMAS INFORMÁTICOS

Alternativas	Frecuencias	Porcentaje
Normal	63	63
Bastante	23	23
Poco	14	14
Total	100	100%

FUENTE: Docentes de Ingeniería en informática y sistemas computacionales UTC
ELABORADO POR: Autora del proyecto

GRAFICÓ Nº 2. 4 MOTIVACIÓN PARA CONOCER NUEVOS PARADIGMAS INFORMÁTICOS



FUENTE: Docentes de Ingeniería en informática y sistemas computacionales UTC
ELABORADO POR: Autora del proyecto

Análisis e interpretación de resultados.

Los datos mencionan que los estudiantes encuestados sienten gran motivación para aprender sobre el paradigma POA; un mínimo porcentaje les despreocupa y no se sienten motivados por cultivar conceptos y definiciones que agrupan el paradigma POA. Al observar estos resultados hay mucha satisfacción, ya que si sumamos las cantidades iníciales en las que hay motivación por conocer el paradigma POA, obtenemos un porcentaje mayor de los estudiantes encuestados que busca conquistar el conocimiento del paradigma POA y tiene gusto por nuevas teorías y conocimientos en su carrera.

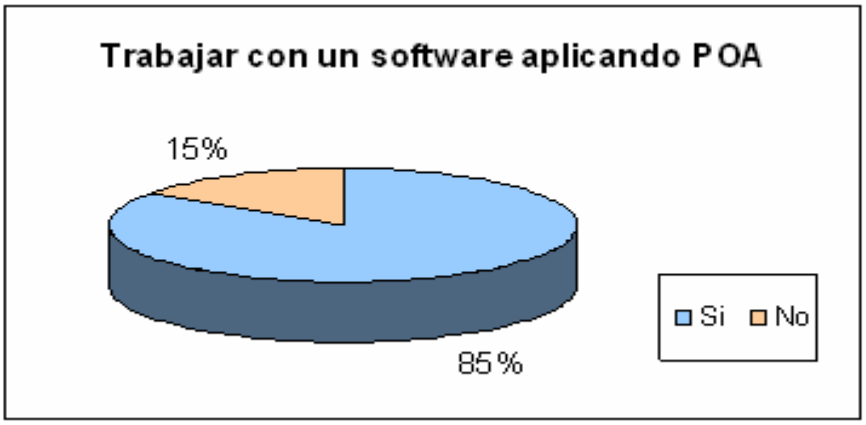
Pregunta Nº 5: ¿Le gustaría trabajar con un software aplicando la Programación Orientada a Aspectos?

TABLA Nº 2.6 TRABAJAR CON UN SOFTWARE APLICANDO POA

Alternativas	Frecuencias	Porcentaje
Si	85	85
No	15	15
Total	100	100%

FUENTE: Docentes de Ingeniería en informática y sistemas computacionales UTC
ELABORADO POR: Autora del proyecto

GRAFICÓ Nº 2. 5 TRABAJAR CON UN SOFTWARE APLICANDO POA



FUENTE: Docentes de Ingeniería en informática y sistemas computacionales UTC
ELABORADO POR: Autora del proyecto

Análisis e interpretación de resultados.

Un gran porcentaje de los estudiantes encuestados piensan que la POA será mejor asimilada si se trabaja con un software aplicando dicho paradigma. Un pequeño porcentaje de estudiantes no están de acuerdo a la utilidad de cualquier tipo de software que se haya aplicado el paradigma POA. Estas cifras estadísticas indican bastante información a nuestro interés investigativo ya que promulgan el requerimiento en los docentes universitarios, a una correcta enseñanza evolutiva, con temas de actualidad y las nuevas tendencias en el desarrollo de software.

2.3 Verificación de la hipótesis

2.3.1. Enunciado

"DESARROLLO DE UN SOFTWARE PROTOTIPO DE VENTAS PARA EL APRENDIZAJE DEL PARADIGMA DE PROGRAMACIÓN ORIENTADA A ASPECTOS EN LA ESPECIALIDAD DE INGENIERÍA EN INFORMÁTICA Y SISTEMAS COMPUTACIONALES DE CIYA - UTC".

2.3.2. Comprobación

Son claros los resultados obtenidos con las entrevistas y encuestas realizadas a los docentes y estudiantes de la especialidad de Ingeniería en Informática y Sistemas Computacionales respectivamente; ya que muestran que los estudiantes recibirán gran ayuda, a la par, serán aventajados al aprender y tratar un nuevo paradigma de programación; como lo viene siendo el paradigma de Programación Orientada a Aspectos.

Además de ser un tema actualizado y que está en evolución, los estudiantes informáticos podrán, en lo posterior, con la ayuda de sus docentes empaparse del mencionado paradigma de programación. Tiene un ambiente de motivación el apoyo de los docentes entrevistados, al igual que los conocimientos que ellos tienen sobre este tema que está en crecimiento.

2.3.3. Conclusión

EL término del desarrollo del software denominado ***VENTSOFT 1.0***; logrará realizar el manejo de facturación, productos, clientes y otras características que harán de este SOFTWARE PROTOTIPO DE VENTAS una aplicación eficiente del uso de la POA Cumpliendo con los objetivos de esta tesis, que exigen obtener un software fruto del paradigma de POA; una herramienta entendible, una interfaz agradable al usuario, eficiencia en la tarea que desempeña, innovación, y una

aplicación evolutiva tecnológicamente. Todo esto ha incidido en la hipótesis con un resultado favorable en toda su dimensión.

El VENTSOFT 1.0 es una aplicación consecuencia de la POA que servirá para los estudiantes de la especialidad de Informática y Sistemas Computacionales, participantes de las cátedras de Programación Orientada a Objetos y materias afines según la malla curricular establecida por la Universidad Técnica de Cotopaxi; mismos que podrán aprender y conocer argumentos, fundamentos, sintaxis y otras demostraciones al contar con el VENTSOFT 1.0 como laboratorio para el respectivo estudio del paradigma.

El desarrollo del VENTSOFT 1.0 es factible promoverle en la materia de Programación Orientada a Objetos, ya que al ser la Programación Orientada a Aspectos una extensión de dicha programación, y al tener herramientas eficientes para ese tipo de paradigma como es Java y AspecJ, servirá de gran ayuda para los estudiantes; futuros profesionales que ocuparan lugares de trabajo destinados a la creación de aplicaciones informáticas, concisamente actualizando y culturalizando sus conocimientos, sobre todo para salir muy bien preparados de la Universidad hacia el campo laboral.

Junto con el software prototipo se ha incluido una guía de usuario que ayudara en un 90% al manejo de esta herramienta básica para los estudiantes y docentes de Informática y Sistemas Computacionales.

CAPÍTULO III

“FACTIBILIDAD EN EL DESARROLLO DE UN SOFTWARE PROTOTIPO DE VENTAS PARA EL APRENDIZAJE DEL PARADIGMA DE PROGRAMACIÓN ORIENTADA A ASPECTOS EN LA ESPECIALIDAD DE INGENIERÍA EN INFORMÁTICA Y SISTEMAS COMPUTACIONALES”

3.1 Presentación

En esta era informática, el desarrollo de software para la automatización de procesos, sin competidor alguno, es la estrategia más importante para el crecimiento de instituciones, empresas, organizaciones y multinacionales; por lo que varias entidades de software buscan la manera de agilitar el proceso de creación de una aplicación, mejorar las condiciones y trayectoria que sigue el programador, haciendo a este trabajo menos agotador y rutinario.

La pericia de automatizar el mundo que nos rodea, se hace cada vez algo tan cercano y no imposible, y todas estas destrezas tiene que ser debidamente encausadas y aprovechadas para comodidad de los seres humanos y el planeta.
El conocimiento de esta realidad ha inducido proponer, el desarrollo e implementación del software VENTSOFT 1.0; utilizando técnicas de la Programación Orientada a Aspectos, el que se presenta a fin de incursionar en el estudio de mencionado paradigma, para que en complicidad con el uso de Java y AspectJ, se motiva la creación del proyecto informático, denominado

VENTSOFT 1.0, este software realiza el ingreso de un cliente con sus respectivos datos, el registro del producto, una factura con su respectivo detalle, realiza reportes de producto eliminado en caja, reportes de error al ingreso del software. Se tiene tres clases de usuario el Administrador, Vendedor, Bodeguero cada uno con su respectiva contraseña.

3.2 Justificación de la propuesta

La evolución tecnológica, la automatización de procesos computacionales, la ingeniería de software y el desarrollo de aplicaciones informáticas, son ítems que globalizan al planeta, acrecientan la competitividad de los estudiantes de la computación y por ende de las universidades que ofertan carreras tecnológicas, además de orientar su pensamiento a la dependencia de la investigación y la ciencia.

La arquitectura de este proyecto de tesis está respaldada con el patrocinio y colaboración de un director, el correspondiente material científico, la adecuada documentación virtual y una excelente información en la Internet; aspectos que permiten una contextura bien fundamentada en la esporádica finalización de el presente proyecto investigativo.

La Programación Orienta a Aspectos, es un paradigma que está en auge con el motivo principal de mejorar el desarrollo de herramientas informáticas bien constituidas, adicional a esto al ser una extensión de la Programación Orientada a Objetos, ha logrado obtener diseños más modulares, mejorar la trazabilidad, corregir la evolución del sistema, aumentar la reutilización, reducir el tiempo de desarrollo y el coste de futuras implementaciones.

El VENTSOFT 1.0 es una aplicación consecuencia de la Programación Orientada a Aspectos, que servirá para los estudiantes de la Carrera de Informática y Sistemas Computacionales, participantes de las cátedras de Programación Orientada a Objetos y materias afines según la malla curricular establecida por la

Universidad Técnica de Cotopaxi; mismos que podrán aprender y conocer argumentos, fundamentos, sintaxis y otras demostraciones al contar con el VENTSOFT 1.0 como laboratorio para el respectivo estudio del paradigma.

El desarrollo del VENTSOFT 1.0 es factible promoverle en la materia de Programación Orientada a Objetos, ya que al ser la Programación Orientada a Aspectos una extensión de dicha programación, y al tener herramientas eficientes para ese tipo de paradigma como es Java y AspecJ, servirá de gran ayuda para los estudiantes; futuros profesionales que ocuparan lugares de trabajo destinados a la creación de aplicaciones informáticas, concisamente actualizando y culturalizando sus conocimientos, sobre todo para salir muy bien preparados de la Universidad hacia el campo laboral.

Las características que justifican al VENTSOFT 1.0 son las siguientes:

- Mejorar el desarrollo del software en la etapa de construcción de algoritmos y programación.
- Inculcar cimientos conceptúales y prácticos de Programación Orientada a Aspectos en los estudiantes de informática.
- Orientar la creación de aplicaciones de computadora en otros tipos de paradigmas y lenguajes estudiados en horas clase.
- Inspirar la utilización de aspectos en la programación y en la ingeniería de software para la búsqueda de estándares que fomenten el crecimiento de este paradigma.
- Tener una herramienta práctica, fruto de la Programación Orientada a Aspectos para su tratado y respectivo discernimiento de teorías.

Las herramientas y lenguajes de programación utilizados son Java y AspectJ, permitiendo de alguna manera mejorar la utilidad que se le da en la especialidad a estos potentes y muy avanzados programas, que constan con librerías, algoritmos e interfaces estupendas para la elaboración de proyectos computacionales.

La finalidad de esta tesis es motivar a los estudiantes de Informática y Sistemas Computacionales a generar investigación del paradigma de Programación Orientada a Aspectos y de hecho a solucionar problemas a la hora de crear software, presentar diversas posibilidades para el manejo de aspectos y ser el origen de muchas más tesis que den referencia a este tema.

3.3 Objetivos de la propuesta

Existe diversidad de objetivos que puede desembocar de un paradigma que está en proceso evolutivo y en investigación constante, pero la delimitación de este tema a una mencionada tarea permite plantearnos objetivos tanto generales como específicos, mencionados en las siguientes líneas:

3.3.1 Objetivo del Software prototipo VENTSOFT 1.0

- Desarrollar un software prototipo de ventas para el aprendizaje del paradigma de Programación Orientada a Aspectos en la especialidad de Ingeniería en Informática y Sistemas Computacionales de C.I.Y.A.

3.3.2 Objetivos Específicos del Software prototipo VENTSOFT 1.0

- Hacer una revisión y ver el estado del arte de esta tecnología, comprobar cuáles son las líneas de los distintos trabajos de investigación.
- Aprender información relevante sobre el paradigma de la Programación Orientada a Aspectos.
- Vincular el uso de un software prototipo utilizando POA como ejemplo para los estudiantes de la Carrera de Ingeniería en Informática y Sistemas Computacionales
- Detallar información específica y comprensible para el aprendizaje del POA para los estudiantes de dicha especialidad.

3.3.3 Objetivos Conceptuales:

- Conocer las principales estrategias, principios que utiliza la Programación Orientada a Aspectos.
- Establecer diferencias coherentes, lógicas y fructíferas de la programación tradicional con el paradigma tratado en esta tesis.

3.3.4 Objetivos Procedimentales

- Analizar críticamente información actualizada y relevante.
- Optimizar la construcción de algoritmos al momento del diseño de la aplicación.
- Aprovechar las estrategias de la Programación Orientada a Aspectos, para ser capaces de aplicarlas a problemas reales que requieren de una automatización.

3.4 Factibilidad para aplicar el software prototipo, en la especialidad de Informática y Sistemas Computacionales.

Los continuos avances en la ingeniería del software han ido incrementando la capacidad de los desarrolladores de software para descomponer un sistema en módulos independientes cada uno con una función bien definida, esto es, facilitar la separación de intereses. Dentro de estos avances quizás el más importante en estas últimas décadas ha sido la aparición de la programación orientada a objetos.

El paradigma orientado a objetos proporciona un potente mecanismo para separar intereses, sobre todo aquellos relacionados con la lógica del negocio de la aplicación, pero presenta dificultades a la hora de modelar otros intereses que no pueden ser encapsulados en una única clase, ya que afectan a distintas partes del sistema. Es por esto que han ido surgiendo distintas técnicas o paradigmas que intentan solucionar los vacios que mantiene la POO; como por ejemplo la programación adaptativa, la programación subjetiva, la transformación de programas, los filtros composicionales, de las cuales la más popular y que con el tiempo a ganado muchos adeptos es la Programación Orientada a Aspectos.

La Ingeniería de Software tradicional carece actualmente de mecanismos adecuados para abstraer, y encapsular conceptos que no forman parte de la funcionalidad básica de los sistemas, tales como debugging, sincronización, distribución, seguridad, administración de memoria, y otros. El resultado de esta insuficiente abstracción es una notable disminución de la calidad del software final. Cabe recalcar que una de las alternativas más prometedoras para resolver este problema es la Programación Orientada a Aspectos. Contiguo a los nuevos descubrimientos; progresos más importantes se han obtenido aplicando la programación de alto nivel junto con tres principios estrechamente relacionados entre sí:

Abstracción, Encapsulamiento, Modularidad.
En una carrera en la que la programación es la parte medular de toda su estructura, es imperioso abordar el tema de la diversidad de estrategia de programación para el desarrollo de aplicaciones informáticas; y mucho mejor involucrarse con la Programación Orientada a Aspectos.

Los tres elementos constitutivos principales de la POA son:

- Un lenguaje para definir la funcionalidad básica, conocido como *lenguaje base* o *componente*. El mismo puede ser un lenguaje imperativo, o no, como por ejemplo C++, Java, Lisp.

- Uno o varios *lenguajes de aspectos*, para especificar el comportamiento de los distintos aspectos. Algunos ejemplos son AspectJ, *COOL*, para especificar sincronización, y *RIDL* para especificar distribución.

- Un *tejedor* de aspectos (*weaver)*, que se encarga de combinar los lenguajes en tiempo de ejecución o de compilación.

El campo laboral de la informática requiere en la actualidad profesionales de la rama que tengan conocimientos de varios lenguajes de programación, y sobre todo de aquellas técnicas, herramientas y lenguajes de programación actualizados. Un profesional exitoso en el área computacional será aquel que domine y se haga muy buen amigo de la tecnología cambiante.

La automatización de procesos es una tarea que le compete primordialmente al ordenador y su técnico programador; razón por lo que es lógico saber que una tesis destinada a la investigación de un proceso que mejora la metodología de creación de software, como es el caso de la Programación Orientada a Aspectos, no se debe pasar por alto la designación directa que tiene con la especialidad de Ingeniería en Informática y Sistemas Computacionales.

3.5 Impacto y Vida útil del Software Prototipo

El software VENTSOFT 1.0 es una herramienta producto de la utilización de los programas Java y AspectJ, sus funciones y librerías de estos, y especialmente la funcionalidad de estos lenguajes de programación con el cual los usuarios destinados, (estudiantes de la especialidad de Informática y Sistemas Computacionales) pueden interactuar con un programa sencillo y de una interfaz de usuario comprensible a la hora de hacer una transacción comercial de ventas, todo esto como "simulación", ya que el software que desarrolla esta tesis está destinado como objetivo principal la percepción visual y práctica de la utilización del Paradigma de Programación Orientada a Aspectos.

El VENTSOFT 1.0 posee una guía de usuario con el cual los estudiantes podrán tener la facilidad de manipular el programa y ver donde se encuentra empleado el paradigma de Programación Orientada a Aspectos.

Por ello VENTSOFT 1.0 tendrá un excelente impacto en aquellos estudiantes y usuarios que requieren de una herramienta rápida y eficiente en la venta de un producto, el VENTSOFT 1.0 servirá como ayuda para estudiantes de Ingeniería en Informática y Sistemas Computacionales que hagan de la POA un tema para concluir su carrera.

La vida útil de VENTSOFT 1.0 por lógicas razones está delimitada por el avance tecnológico; y en el mundo Java y AspectJ que evoluciona a cada instante tendrá una escaza durabilidad en la parte de interfaz gráfica de usuario, porque con respecto a la programación que entraña dicha aplicación unos seis años serán escasos para ver su declive.

Software y Hardware se despliegan en un mundo cambiante cada día, por tal motivo no será ajeno encontrar un programa o una página de Internet que cumpla con similares tareas que el VENTSOFT 1.0 y que sea víctima beneficiosa de la POA. Sin duda alguna las raíces que marcan el caminar para la complementación

de lo que hoy en día es tan solo un paradigma poco conocido, será de una gran ayuda.

3.6 Marco de trabajo de la POA

3.6.1 Generalidades

Para demostrar la utilización de Aspectos en un sistema real, se desarrollará un prototipo software de Ventas, el cual se denomina ***VENTSOFT 1.0***, ya que esta aplicación está destina a la automatización del proceso de ventas y compras en cualquier empresa comercial pequeña, pero inicialmente consignada para los estudiantes de la especialidad de informática y sistemas computacionales como aplicación del paradigma de programación orientada a aspectos.

Para el desarrollo inicial se utiliza UML(Unified Modelig lenguaje) hasta obtener el Diagrama de Clases, posteriormente para establecer el aporte de la Programación Orientada A aspectos, se toma como base el diagrama de clases desarrollado con UML.

Se analiza los requisitos y necesidades que el usuario requiere para obtener un producto, para la obtención del sistema se va a utilizar el documento ERS(Especificación de Requisitos del Software) basado en la estándar IEEE 830. Este proyecto permite definir las especificaciones de requisitos de Software para el prototipo de un sistema de ventas, como ejemplo de la utilización de la Programación Orientada a Aspectos.

Recolección de requisitos

Requerimientos y necesidades que el usuario requiere para la venta de un producto, el manejo de cliente, producto, facturación.

Funciones del sistema

En términos generales el sistema deberá proporcionar las siguientes tareas de gestión.

Gestión de acceso al sistema. Permitirá el ingreso al VENTSOFT 1.0 para el acceso al sistema y poder realizar o trabajar en cualquier de las otras gestiones para la venta.

Gestión de ventas. Nos permite seleccionar con los datos del cliente los productos que el cliente desea comprar.

Cuando exista un cambio de venta, se debe buscar el pedido, y automáticamente se podrá modificar los productos. Para realizar una consulta de un pedido se puede realizar por medio del nombre del cliente.

Gestión de Producto. Ingresar los datos de los productos al sistema VENTSOFT 1.0.

Gestión de Clientes. Los datos del cliente se podrán ingresar manualmente en el sistema, se ingresará todos los datos personales del cliente, también se podrá actualizar y eliminar.

El VENTSOFT 1.0. Deberá proporcionar una interfaz gráfica de usuario fácil de utilizar.

Requisitos Funcionales

Se debe especificar cada gestión del sistema de una manera clara y concisa.

Gestión de acceso al sistema: requisito (01) Ingresar clave de acceso.

Gestión de clientes: el sistema permitirá.
Requisito (02) Ingresar un nuevo cliente, Requisito (03) Eliminar un cliente, Requisito (04) Actualizar datos del cliente.

Gestión de Productos: Requisito (05) Ingresar los datos de los productos,
Requisito (06) modificar un producto, Requisito (07) Eliminar un producto.

Gestión de ventas: Requisito (08) realizar ventas, Requisito (09) Generar facturas.

El VENTSOFT 1.0 a la hora de desempeñarse automatizará el proceso de una venta y por ende de una compra; mejorando la eficiencia del vendedor, en el área de Caja.

Requisitos tecnológicos

- Lenguaje de programación AspectJ.
- Integrador de lenguaje Eclipse.
- Base de Datos Mysql.
- Java.
- Servidor Web Tomcat.

Instalación de Tomcat

1. Bajar la versión binaria de Tomcat en:
   http://jakarta.apache.org/tomcat . (La versión de Código Fuente(src) solo es necesaria si quiere experimentar y/o Instalar Apache con Tomcat).

2. Descomprimir el archivo Tar de Tomcat en `/usr/local/`, esto genera un directorio llamado `jakarta-tomcat-<numero_de_version>`, para dar mayor uniformidad se recomienda cambiar el nombre de este directorio a `tomcat`.

3. Posteriormente se debe definir una variable ambiental la cual le indicará al sistema la ubicación de Tomcat , esta variable se llama `CATALINA_HOME` la cual debe ser agregada a `/etc/bashrc` , si no esta familiarizado con ambientes *nix, esto significa agregar la línea: `export CATALINA_HOME=/usr/local/tomcat;;` para instalaciones Windows esta variable ambiental puede ser definida de la misma manera que CLASSPATH, descrita en las instrucciones del JDK.

Instalación de Mysql

Para descargar mysql mos vamos a la web oficial de MySQL y descargamos la última versión gratuita disponible llamada "MySQL Community Serve(dev.mysql.com/get/Downloads/mysql-5.1/mysql-essential-5.1.31-win32.msi/from/http://mysql.easynet.be/).

Pasos para la instalación.

- Descomprimir el archivo de MySQL a través de *WinZip* y colocarlo dentro de un directorio temporal.
- Dentro de este directorio, oprima el *icono* de PC llamado Setup para iniciar el proceso de instalación.
- Siga las indicaciones de instalación seleccionando el directorio donde desea instalar MySQL (Se recomienda el directorio "default" C:\mysql).
- Elija la instalación Typical.
- Solo si instaló MySQL en un directorio **distinto a C:\mysql** siga las siguientes instrucciones :

- Abra un editor de textos y genere el siguiente archivo llamado my.cnf :

```
[mysqld]
basedir = e:\mysql
datadir = e:\mysql\data
```

- El archivo anterior describe donde fue instalado MySQL, en este caso el directorio raíz de instalación corresponde a e:\mysql.
- Debe colocar este archivo en la ruta: c:\my.cnf para que MySQL sea ejecutado correctamente.

- Arranque la Base de Datos MySQL a través del Administrador *winmysqladmin*, esta herramienta reside en el directorio bin de la instalación MySQL.

- Al oprimir el *icono* de esta utilería aparecerá una consola indicando el estado de MySQL; en esta misma consola aparecerá un *semáforo* indicando el estado de MySQL: Una luz verde indica MySQL activo y una luz roja inactivo.

NOTA: Para cambiar el estado de operación de MySQL, cuando la consola administrativa (*winmysqladmin*) este abierta, oprima el botón derecho del "Mouse" para activar/desactivar el servicio.

Instalación de Driver J para mysql

Es necesario que nuestra aplicación desarrollada en Java acceda a, y manipule, datos que se encuentra en algún DBMS, como por ejemplo MySQL. Para esto debemos ingresar a API JDBC de Java, y además de esto, usar los Drivers "específicos" para manipular DBMS's "específicos" (Oracle, MySQL, etc.).

MySQL provee conectividad para aplicaciones cliente desarrolladas en Java mediante este driver JDBC, llamado *MySQL Connector/J*, que es el driver JDBC oficial para MySQL. La ultima versión de este conector, es el Connector/J 5.1, que incluye soporte para las funcionalidades del JDBC-4.0.

MySQL Connector/J es un driver JDBC tipo 4. Existen otras versiones que son compatibles con las especificaciones JDBC-3.0 y JDBC-4.0. Se dice que es tipo 4 ya que el driver es una implementación del protocolo de MySQL hecha puramente en Java, y no requiere de clientes binarios de MySQL. Por supuesto es necesario que comprenda de manera clara el funcionamiento de la API de JDBC, para poder usar cualquier conector.

Instalando el MySQL Connector

Lo primero que debemos hacer, es ir a la página de descargas del conector de MySQL y descargarnos la última versión (si es necesario). Escogemos el tipo de archivo que nos convenga (por ejemplo el tar.gz si usamos Gnu/Linux, o zip si usamos Windows). Descomprimimos el archivo y observamos que nos proporcionan:

- Los binarios ejecutables (en caso ***mysql-connector-java-5.1.5-bin.jar***)
- Los archivos fuente
- La documentación del conector

Debemos entonces copiar el archivo JAR ejecutable, en el siguiente path: \ruta_instalacion_java\jre\lib\ext. Por ejemplo en Windows podría ser en C:\Program Files\Java\jre1.6.0_01\lib\ext,

Luego de copiar dicho archivo, ya podemos usar el conector de MySQL desde nuestras aplicaciones en Java, JSP, servlets, etc.

Instalación de Eclipse

El proceso de instalación de Eclipse tiene cuatro pasos principales:

1. Instale el software de Java 5.0.
2. Instale el software de Eclipse.
3. Lanzamiento de Eclipse.
4. Instalar un conjunto de plugins.
5. Los archivos que se necesitan para descargar son bastante grandes por lo que es necesario tener una conexión rápida a Internet Proceso de instalación para Windows.

1. Instalar Java.
   - Descargue el archivo jdk-1_5_0_04-windows-i586-p.exe .
   - Haga doble clic en el archivo descargado previamente y siguiendo las instrucciones de instalación.

2. Instale el software Eclipse.
   - Descargue el archivo eclipse-SDK-3.1-win32.zip.
   - Una vez que haya descomprimido el archivo descargado anteriormente, podrás ver una carpeta llamada eclipse. En esa carpeta se encuentra el eclipse de la aplicación (un gran punto azul). Le recomendamos crear un acceso directo en el escritorio para simplificar la puesta en marcha de eclipse. A diferencia de Java, Eclipse no tiene un proceso de instalación. Una vez que haya descomprimido el archivo que está hecho.

3. Lanzamiento de Eclipse
    - Siga los pasos en el lanzamiento de Eclipse .
4. Instalación de los plugins
    - Lanzamiento eclipse.
    - Siga las instrucciones que aparecen en el sitio web http://www.cs.umd.edu/ pugh ~ / eclipse . Después de instalar los plugins nuevos, tendrá que reiniciar Eclipse. Tenga en cuenta que no será capaz de presentar proyectos si no se completa la instalación de estos complementos.

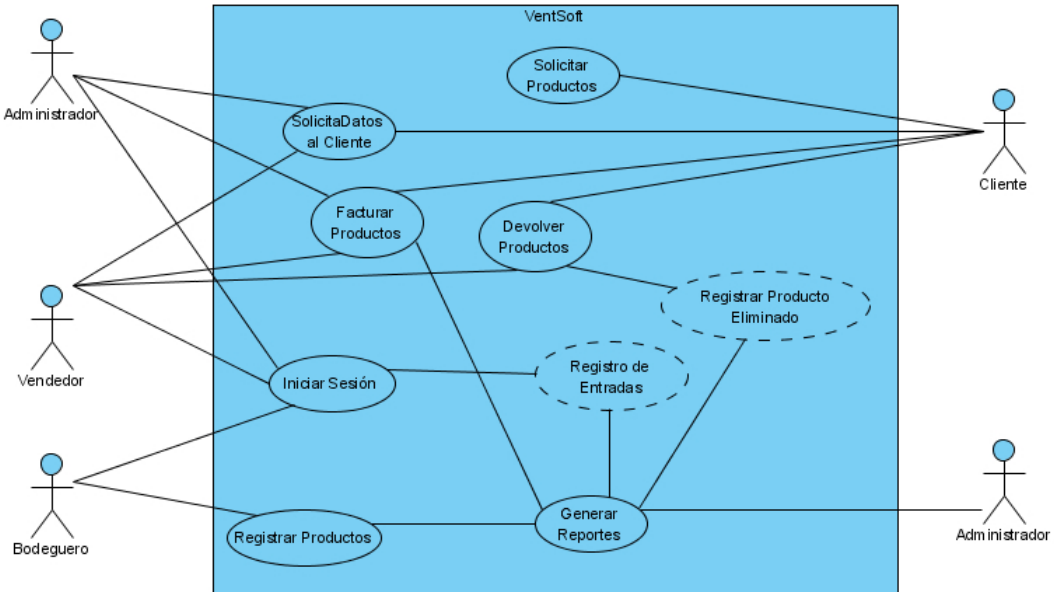
Requisitos de Hardware

La aplicación se ejecuta sobre un PC con las siguientes características como mínimo. Procesador Pentium III, memoria1Giga de RAM, Disco duro 5 GB espacio libre, Unidades CD-ROM O DVD-ROM.

3.6.2 Construcción del Modelo mediante marco de Trabajo

Diagramación de la Solución

FIGURA Nº 3. 1 DIAGRAMA CASOS DE USO



En la figura 3.1 tenemos tres tipos de usuarios que son Administrador, Vendedor y Bodeguero, este caso de uso inicia cuando uno de los usuarios selecciona la operación Iniciar para ello se ingresa el tipo de usuario y su clave de acceso, esto le permite ingresar al sistema Ventsoft 1.0.

El usuario Administrador: en el momento de ingresar al sistema podrá manejar las opciones de manejo de Solicitar Datos al cliente, Facturar Productos, reportes de producto eliminado, etc.

El usuario Vendedor: ingresa al sistema y podrá manejar lo que corresponde Solicitar Datos del cliente y Facturar.

El usuario Bodeguero: ingresará al sistema y podrá manejar lo que corresponde a registro de producto.

FIGURA Nº 3. 2 DIAGRAMA DE CASOS DE SECUENCIA

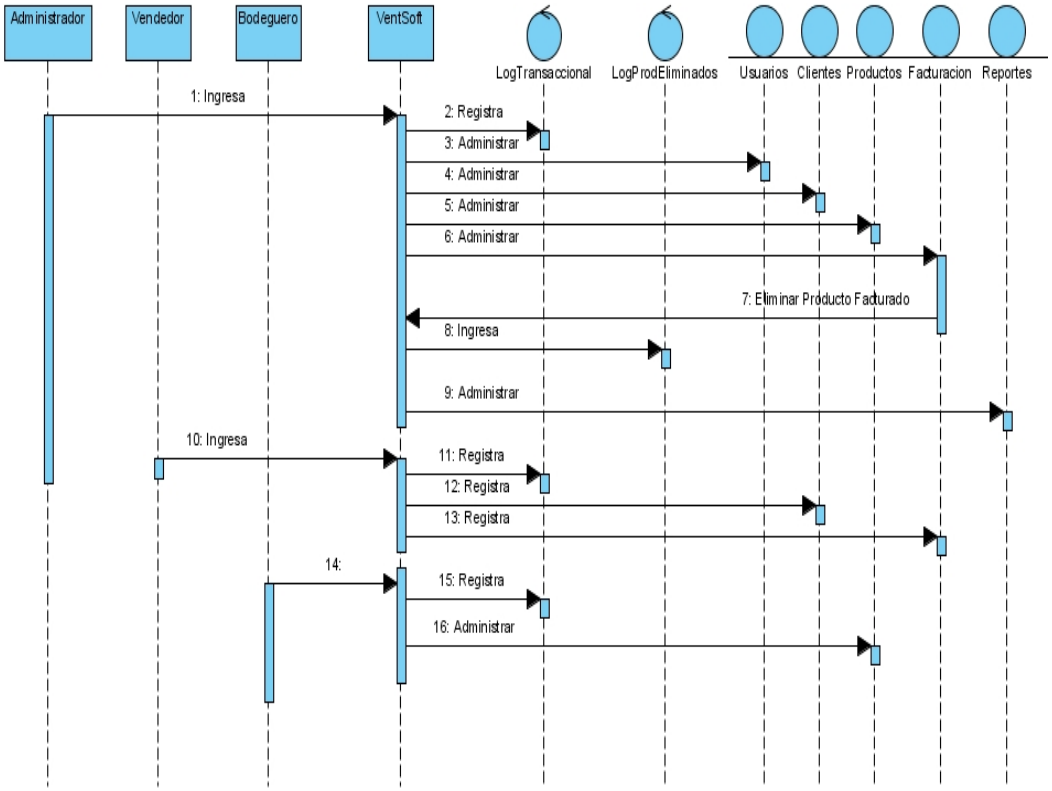


FIGURA Nº 3. 3 DIAGRAMA DE CLASES

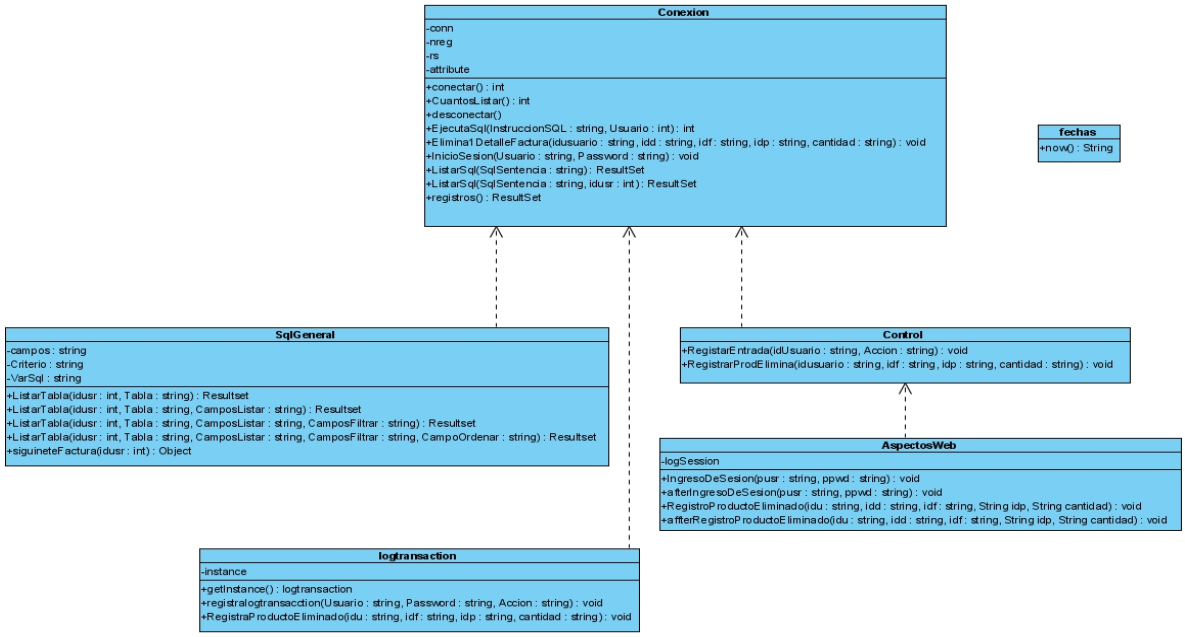
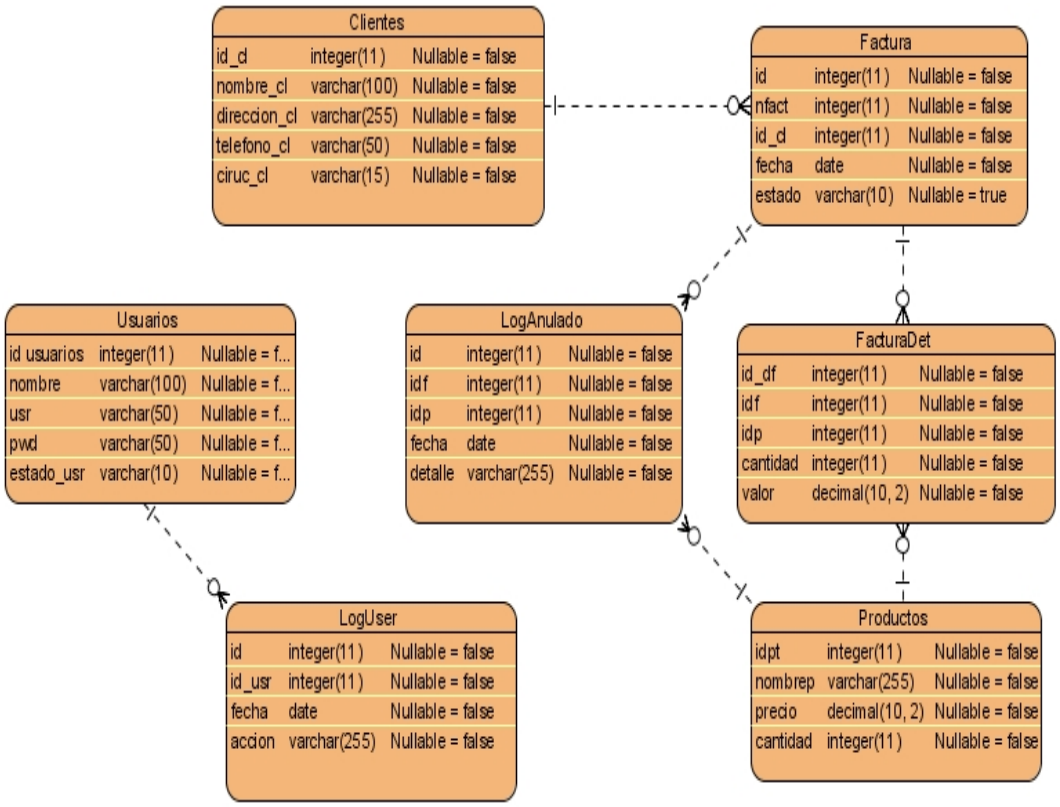


FIGURA Nº 3. 4 DIAGRAMA ENTIDAD RELACIÓN



DICCIONARIO DE DATOS

En este diccionario de datos describimos las diferentes tablas que se utilizó en el prototipo VentSoft 1.0. Base de Datos: ventas

Tabla: clientes

Índices:

Nombre	Tipo
Índice principal	id_cl

Campos:

Nombre	Tipo	NULL	Extras	Descripción
id_cl	int(11)		auto_increment	Identificación del Cliente
Nombre _cl	varchar(100)	NULL		Nombre del Cliente
Dirección _cl	varchar(255)	NULL		Dirección del Cliente
Teléfono_cl	varchar(50)	NULL		Teléfono del Cliente
Ceruc _cl	varchar(15)	NULL		Cédula o Ruc del Cliente

Tabla: factura

Índices:

Nombre	Tipo
Índice principal	id

Campos:

Nombre	Tipo	NULL	Extras	Descripción
id	int(11)		auto_increment	Identificación de transacción
nfact	int(11)	NULL		Número de Factura
id_cl	int(11)	NULL		Identificación del Cliente
fecha	Date			Fecha de Facturación
estado	varchar(10)			Estado de Factura

Tabla: facturadet

Índices:

Nombre	Tipo
Índice principal	id_df

Campos:

Nombre	Tipo	NULL	Extras	Descripción
id_df	int(11)		auto_increment	Identificación de Identificación de Transacción
idf	int(11)			Identificación de factura
idp	int(11)	NULL		Identificación de Producto
cantidad	int(11)	NULL		Cantidad de producto
valor	decimal(10,2)	NULL		Valor unitario del producto

Tabla: logpanulado

Índices:

Nombre	Tipo
Índice principal	id

Campos:

Nombre	Tipo	NULL	Extras	Descripción
id	int(11)		auto_increment	Identificación del Anulaciones
idf	int(11)	NULL		Identificaión de Factura
idp	int(11)	NULL		Identificación de Producto
fecha	datetime	NULL		Fecha de Anulación de Producto
detalle	varchar(255)	NULL		Detalle de eliminación de producto

Tabla: loguser

Índices:

Nombre	Tipo
Índice principal	id

Campos:

Nombre	Tipo	NULL	Extras	Descripción
id	int(11)		auto_increment	Identificación de registro de usuarios
id_usr	int(11)			Identificación del usuario
fecha	datetime			Fecha de registro de log
acción	varchar(255)			Actividad registrada

Tabla: productos

Índices:

Nombre	Tipo
Índice principal	idpt
Producto repetido	nombrep

Campos:

Nombre	Tipo	NULL	Extras	Descripción
idpt	int(11)		auto_increment	Identificación del Producto
nombrep	varchar(255)	NULL		Nombre del Producto
precio	double(10,2)	NULL		Precio Unitario del Producto
cantidad	int(11)	NULL		Cantidad De Producto Existente

Tabla: usuarios

Índices:

Nombre	Tipo
Índice principal	id_usuarios

Campos:

Nombre	Tipo	NULL	Extras	Descripción
id_usuarios	int(11)		auto_increment	Identificación de Usuarios
nombre	varchar(100)	NULL		Nombre de Usuario
usr	varchar(50)	NULL		Nickde acceso al sistema
pwd	varchar(50)	NULL		Contraseña de Acceso al Sistema
estado_usr	varchar(10)	NULL		Estado del Usuario
tipo_usr	varchar(50)			Tipo de Usuario

Aquí se encuentra el Script de la Base de Datos MySql.

```
DROP DATABASE IF EXISTS `ventas`;
CREATE DATABASE `ventas`
USE `ventas`;

// Table structure for table clients//
CREATE TABLE `clientes` (
`id_cl` int(11) NOT NULL auto_increment COMMENT' Identificación del Cliente',
`nombre_cl` varchar(100) default NULL COMMENT 'Nombre del Cliente',
`direccion_cl` varchar(255) default NULL COMMENT 'Dirección del Cliente',
`telefono_cl` varchar(50) default NULL COMMENT 'Teléfono del Cliente',
`ciruc_cl` varchar(15) default NULL COMMENT 'Cédula o Ruc del Cliente',
PRIMARY KEY  (`id_cl`)
)
ENGINE=MyISAM AUTO_INCREMENT=24 DEFAULT CHARSET=latin1;

# Table structure for table factura#

CREATE TABLE `factura` (
`id` int(11) NOT NULL auto_increment COMMENT 'Identificación de transacción',
`nfact` int(11) default NULL COMMENT 'Número de Factura',
`id_cl` int(11) default NULL COMMENT 'Identificación del Cliente',
`fecha` date NOT NULL COMMENT 'Fecha de Facturación',
`estado` varchar(10) NOT NULL COMMENT 'Estado de Factura',
PRIMARY KEY  (`id`)
)ENGINE=MyISAM AUTO_INCREMENT=35 DEFAULT CHARSET=latin1;
```

```
# Table structure for table facturadet#

CREATE TABLE `facturadet` (
`id_df` int(11) NOT NULL auto_increment COMMENT 'Identificación de Identificación de Transacción',
`idf` int(11) NOT NULL default '0' COMMENT 'Identificación de factura',
`idp` int(11) default NULL COMMENT 'Identificación de Producto',
`cantidad` int(11) default NULL COMMENT 'Cantidad de producto',
`valor` decimal(10,2) default NULL COMMENT 'Valor unitario del producto',
PRIMARY KEY  (`id_df`)
)
ENGINE=MyISAM AUTO_INCREMENT=33 DEFAULT CHARSET=latin1;

# Table structure for table logpanulado#

CREATE TABLE `logpanulado` (
`id` int(11) NOT NULL auto_increment COMMENT 'Identificación del Anulaciones',
`idf` int(11) default NULL COMMENT 'Identificación de Factura',
`idp` int(11) default NULL COMMENT 'Identificación de Producto',
`fecha` datetime default NULL COMMENT 'Fecha de Anulación de Producto',
`detalle` varchar (255) default NULL COMMENT 'Detalle de eliminación de producto',
PRIMARY KEY  (`id`)
)
ENGINE=MyISAM AUTO_INCREMENT=10 DEFAULT CHARSET=latin1;
```

Table structure for table loguser#

```
CREATE TABLE `loguser` (
`id` int(11) NOT NULL auto_increment COMMENT 'Identificación de registro de usuarios',
`id_usr` int(11) NOT NULL COMMENT 'Identificación del usuario',
`fecha` datetime NOT NULL COMMENT 'Fecha de registro de log',
`Acción` varchar(255) NOT NULL COMMENT 'Actividad registrada',
PRIMARY KEY  (`id`)
)
ENGINE=MyISAM AUTO_INCREMENT=16 DEFAULT CHARSET=latin1;
```

Table structure for table productos

```
CREATE TABLE `productos` (
`idpt` int(11) NOT NULL auto_increment COMMENT 'Identificación del Producto',
`nombrep` varchar(255) default NULL COMMENT 'Nombre del Producto',
`precio` double(10,2) default NULL COMMENT 'Precio Unitario del Producto',
`cantidad` int(11) default NULL COMMENT 'Cantidad De Producto Existente',
PRIMARY KEY  (`idpt`),
UNIQUE KEY `productorepetido` (`nombrep`)
)
ENGINE=MyISAM AUTO_INCREMENT=8 DEFAULT CHARSET=latin1;
```

3.6.3 Desarrollo del Software Prototipo

***Programación gráfica en* Java (JSP)**

La interfaz del programa es la que permite interactuar a éste con el usuario. Las interfaces de usuario pueden adoptar muchas formas, que van desde la simple línea de comandos hasta las interfaces gráficas (GUI) que proporcionan las aplicaciones más modernas.

La interfaz de usuario es uno de los aspectos más importante de cualquier aplicación. Una aplicación sin un interfaz fácil, impide que los usuarios saquen el máximo rendimiento del programa. **Java Server Pages** (**JSP**) es una tecnología Java que permite generar contenido dinámico para Web, en forma de documentos HTML, XML o de otro tipo.

Cada página es automáticamente compilada a servlet por el motor de JSP, en primer lugar es recogida y ejecutada. JSP tiene gran variedad de formas para comunicarse con las clases de Java, servlets, applets y el servidor web; por esto se puede aplicar una funcionalidad a nuestra web a base de componentes.

Una página JSP es un archivo de texto simple que consiste en contenido HTML o XML con elementos JSP. Cuando un cliente pide una página JSP del sitio web y no se ha ejecutado antes, la página es inicialmente pasada al motor de JSP, el cual compila la página convirtiéndola en Servlet, la ejecuta y devuelve el contenido de los resultados al cliente.

Es posible ver el código del servlet generado, este código debe estar en el directorio que se informa en la estructura de directorios del servidor.

Las clases son: JSP Page y Http JspPage
Ellas definen la interface para el compilador de páginas JSP.

También se tiene tres métodos:

- Jsp Init ()
- Jsp Destroy ()
- _jspService(HttpServletRequest request,HttpServletResponse response).

Los dos primeros métodos pueden ser definidos por el autor de la página JSP, pero el tercer método es una versión compilada de la página JSP, y su creación es responsabilidad del motor de JSP.
Una página JSP simple
Para el código fuente de una página JSP incluye las variables y etiquetas:

1. Directivas: Dan información global de la página, por ejemplo, importación de estamentos, página que maneja los errores o cuando la página forma parte de una sesión. Una directiva de JSP es un estamento que proporciona la información del motor de JSP para la página que la pide.

Su sintaxis general es <%@ directiva {atributo ="valor"} %> dónde la directiva debe tener un número de atributos.

Las directivas disponibles son:
Page: Información para la página.
Include: Incluye archivos completos palabra por palabra.

Atributo	Sintaxis	Utilización
Import	<%@ page import="class; class" %>	Importa clases y paquetes Java para ser utilizadas dentro del fichero JSP.
Sesión	<%@ page session="false" %>	Especifica si utiliza los datos contenidos en sesión; por defecto "true".
contentType	<%@ page contentType="class; class" %>	Especifica el tipo MIME del objeto "response"; por defecto "text/html; charset=ISO-8859-1".
Buffer	<%@ page buffer="12KB" %>	Buffer utilizado por el objeto writer "out"; puede tomar el valor de "none"; por defecto "8KB".
errorPage	<%@ page errorPage="/path_to_error_page" %>	Especifica la ruta de la página de error que será invocada en caso de producirse una excepción durante la ejecución de este fichero JSP.
isErrorPage	<%@ page isErrorPage="true" %>	Determina si este fichero JSP es una página que maneja excepciones. Únicamente a este tipo de páginas pueden acceder a la variable implícita "excepción", que contiene la excepción que provocó la llamada a la página de error.

Taglib: La dirección de la librería de tags que se usará en la página, page posee varios atributos.

2. Declaraciones: Sirven para declarar métodos y variables.

Una declaración de JSP, puede definirse como una definición de variables y métodos a nivel de clase que son usadas en la página.

Un bloque de declaraciones típico sería <%! declaración %>

Ejemplo de declaración de script sería el siguiente:

```
<HTML>
<HEAD>
<TITLE>Patina simple JSP</TITLE>
</HEAD>
<BODY>
<%! String strCadena = "x";
```

```
int intContador = 0;
%>
</BODY>
</HTML>
```

3. Scripts de JSP: Es el código Java embebido en la página.

Scripts son bloques de código Java residentes entre los tags <% y %>.
Este bloques de código estarán dentro del servlets generado incluidos en método _jspService().

Los Scripts pueden acceder a cualquier variable o Beans que haya sido declarado. También hay algunos objetos implícitos disponibles para los Scripts desde entorno del Servlet.

Objetos implícitos	**Descripción**
request	Es la petición del cliente. Es normalmente una Subclase de la clase HttpServletRequest.
response	Es la página JSP de respuesta y es una subclase de HttpServletResponse.
pageContext	Los atributos de la página y los objetos implícitos necesitan ser accesibles a través de API, para permitir al motor de JSP compilar la página. Pero cada servidor tiene implementaciones específicas de cada uno de esos atributos y objetos. Para solucionar este problema, el motor de JSP utilizar la clase Factory para devolver la implementación de clase PageContext del servidor. Esta clase PageContext es inicializada con los objetos response y request y algunos atributos de la directiva de la página (erropage, session, buffer and autoflush) y facilita los otros objetos implícitos para la pagina de petición.
session	El objeto de sesión HTTP asociado a la petición.
application	Lo que devuelve el servlet cuando se llama a getServletConfig().getContext()
out	El objeto que representa la salida de texto por pantalla.
config	El objeto ServletConfig de la página.
page	Es la forma que tiene la página para referirse a si misma. Se usa como alternativa al objeto this
exception	Es una subclase libre de Throwable que es pasada a la página que maneja los errores.

El siguiente fragmento de código muestra cómo obtener el valor de una parámetro mediante el objeto request, y como pasarlo a una cadena para mostrarlo en pantalla.

```
<%String strNombre = request.getParameter("nombre");
out.println(strNombre);
%>
```

4. Expresiones de JSP: Formatea las expresiones como cadenas para incluirlas en la página de salida.
Las expresiones son una magnifica herramienta para insertar código embebido dentro de la página HTML. Cualquier cosa que este entre los tags <%= y %> será evaluado, convertido a cadena y posteriormente mostrado en pantalla.

La conversión desde el tipo inicial a String es manejada automáticamente.
Es importante remarcar que la expresión no termina en punto y coma (;) . Esto es así porque el motor de JSP, pondrá la expresión automáticamente entre out.println().

Las expresiones JSP te permiten parametrizar las páginas HTML (es parecido a cuando parametrizas una consulta SQL pero difieren la forma de los valores). Una y otra vez, en el código de la página HTML, se verán bucles o condiciones usando código Java, simplemente empezando y acabando las condiciones o bucles entre los tags <% y %>.

Un ejemplo sería:

```
<% for (int i=0;i<5;i++) { %>
<BR>El valor del contador es <%=i%>
<% } %>
```

La herramienta más importante que se usa a la hora de desarrollar web con Jsp son los Servlets; los servlets son la primera línea de batalla del desarrollo de las aplicaciones web.
Estos aportan una manera fácil para que nuestro servidor se comunique con el lado cliente. Los servlets dan un modelo general de clases para ejecutar servicios.

La principal diferencia entre los servlets y los JSPs es el enfoque de la programación: un JSP es una página Web con etiquetas especiales y código Java incrustado, mientras que un servlet es un programa que recibe peticiones y genera a partir de ellas una página web.

La principal ventaja de JSP frente a otros lenguajes es que permite integrarse con clases Java (.class) lo que permite separar en niveles las aplicaciones web, almacenando en clases java las partes que consumen más recursos (así como las que requieren más seguridad) y dejando la parte encargada de formatear el documento HTML en el archivo JSP.

Desde el punto de vista de arquitectura, podríamos situar esta tecnología como una capa superior a las Servlets dentro de nuestra aplicación, ya que extiende la especificación Servlet .Ambas tecnologías fueron desarrolladas originalmente por Sun Microsystem.

3.7 Guía de Usuario

La guía de usuario del Software desarrollado VENTSOFT 1.0 y su código se muestra en el Anexo IV.

CONCLUSIONES Y RECOMENDACIONES

CONCLUSIONES

- La POA ofrece una solución elegante al problema de modelar los intereses que afectan a diferentes partes del sistema y que con la POO se encontraban dispersos y enredados en múltiples clases.

- La plataforma Java y en especial con el AspectJ son las herramientas más dotadas para la consecución de proyectos de ingeniería de software, en la utilización del paradigma de POA.

- La POA encapsula los intereses en entidades llamadas aspectos eliminando el código disperso y enredado y dando lugar a implementaciones más comprensibles, adaptables y reusables.

- La POA no se tiene que ver como un sustitutivo de la POO, si no como una extensión de la misma, dando lugar a un paradigma de programación en el que coexisten clases y aspectos.

- La ideología de POA es que sus clases modelen la funcionalidad básica del sistema, mientras que los aspectos se encargan de modelar comportamiento cruzado que no puede ser encapsulado en una única clase, dando lugar a la separación de intereses completa.

- Las ventajas de aplicar POA son numerosas: obtenemos diseños mas modulares, se mejora la trazabilidad, se consigue una mejor evolución del sistema, aumenta la reutilización, se reduce el tiempo de desarrollo, porque reduce las líneas de código y el coste de futuras implementaciones y se consigue retrasar decisiones de diseño haciendo mas fácil razonar sobre los intereses principales de la aplicación.

- Se ha utilizado Eclipse por ser un integrador de lenguajes de programación, el cual nos permite cargar los plug-in de aspectj de acuerdo a las necesidades de la programación.

- Eclipse facilita la programación con aspectos debido a que maneja un entorno gráfico, obteniendo una idea clara como actúan los aspectos en las aplicaciones.

- Eclipse es una herramienta más estable para el manejo de aspectos en comparación con herramientas existentes.

RECOMENDACIONES

- Por tratarse de una técnica de programación nueva y que aún no hay soporte ni las herramientas estandarizadas se recomienda por el momento utilizar POA en sistemas pequeños debido a que este nuevo paradigma no a llegado a desarrollarse en su plenitud.

- Se recomienda realizar trabajos de investigación en el área de desarrollo de software con la finalidad de facilitar el desarrollo y la programación, utilizando POA teniendo claro que los lenguajes de programación y metodologías de desarrollo actuales pueden evolucionar y no son la última palabra.

- Investigar la sintaxis y la forma de trabajar con el lenguaje de programación AspectJ será el camino más optimizado para el desarrollo de un software.

- Es indispensable comprender las definiciones, conceptos y teorías de los distintos lenguajes de programación que hay que emplear para el desarrollo total de una aplicación con ideología del paradigma de Programación Orientada a Aspectos.

- Las líneas de código en los algoritmos empleados para la implementación de una aplicación en cualquier lenguaje de programación deben conservar la estructura, saltos de línea, espacios y sintaxis correctas. Programar razonablemente las clases, eventos, aspectos, etc., evitaran el depurar gran cantidad de errores.

- Al instante de convertir la aplicación en un instalador debemos extraer las librerías necesarias para que funcione el software en cualquier ordenador sin dificultad alguna.

REFERENCIAS BIBLIOGRÁFICAS

Bibliografía Consultada

[1] **ASTEASUAIN**, Fernando y **CONTRERAS**, Bernardo. Programación Orientada a Aspectos, análisis del paradigma, Tesis de Licenciatura Departamento de Ciencias e Ingeniería de la Computación, UNIVERSIDAD NACIONAL DEL SUR, Noviembre del 2002.

[2] KICILLOF, Nicolás. Programación Orientada a Aspectos (AOP) MSDN – Microsoft, enero 2001.

[3] Gregor Kickzales, John Lamping, Anurag Mendhekar, Chris Maeda,Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin, "*Aspect-Oriented Programming",* in Proceedings of the European Conference onObject-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. Junio 1997.

[4] **KRZYSZTOF, Czarnecki** y **EISENECKER, Ulrich.** Generative Programming, Methods, Tools and Applications, Addison-Wesley, 2000.

[5] **REINA, Antonia.** *"Visión General de la Programación Orientada a Aspectos"*, Departamento de Lenguajes y Sistemas Informáticos, Universidad de Sevilla, diciembre de 2000.

[8] **Mendhekar A., Kiczales G. and Lamping J.** "*RG: A Case-Study for Aspect-Oriented Programming".* Xerox PARC, 1997.

[9] **Gamma E., Helm R., Johnson R., Vlissides J., *"Design Patterns**: Elements of Reusable Object-Oriented Software"*. Addison-Wesley, 1994.

[10] **Hannemann J., Kiczales G.,** *"Design Pattern Implementation in Java and AspectJ"*. OOPSLA'2002.

[11] **Ramnivas Laddad, "*Aspectj** *in action. Practical aspect-oriented programming"*. Ed. Manning, 2003.

[12] **Joseph D. Gradecki**, Nicholas Lesiecki, "*Mastering AspectJ. Aspect-Oriented Programming in Java"*. Ed. Wiley, 2003

[13] Russell Miles, "*AspectJ Cookbook"*. Ed. O'Reilly, 2004.

[14] AspectJ and AspectWerkz to Join Forces

Bibliografía Citada

- Wikipedia. Programación Orientada a Aspectos en línea. España: Enciclopedia de contenido libre, 29 Septiembre 2009. Disponible en http://es.wikipedia.org/wiki/Programacion_Orientada_a_Aspectos
- Página de AspectJ. Palo Alto Research Center, 30 Noviembre 2009: http://www.aspectcj.org
- Página de Karl Lieberherr, septiembre 2009: http://www.ccs.neu.edu/home/lieber/.
- Página Web del grupo Demeter: Septiembre 2009 http://www.ccs.neu.edu/research/demeter/.
- Página de Mira Mezini Noviembre 2009: http://www.informatik.uni-siegen.de/~mira/.
- Xerox Corporation, *"The AspectJtm Programming Guide"*, 2002. http://www.eclipse.org/aspectj/doc/released/progguide/index.html
- Página Web del plug-in de integración de AspectJ con Eclipse:2009 www.eclipse.org/ajdt
- AspectJ and AspectWerkz to Join Forces: 2009 http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectjhome/aj5announce.html
- Proyecto PatternTesting: 2009 http://patterntesting.sourceforge.net