ESCUELA POLITÉCNICA DEL EJÉRCITO

SEDE - LATACUNGA

CARRERA DE INGENIERÍA DE SISTEMAS E INFORMATICA

"PARADIGMA DE LA PROGRAMACIÓN ORIENTADA A ASPECTOS Y DESARROLLO DE UN PROTOTIPO SOFTWARE"

PROYECTO PREVIO A LA OBTENCIÓN DEL TITULO DE INGENIERA DE SISTEMAS E INFORMATICA

AUTOR:

MAYRA RAQUEL ROBALINO JÁCOME

LATACUNGA - 2008

ÍNDICE

ÍNDICE DE TABLAS

ÍNDICE DE CÓDIGO

ÍNDICE DE FIGURAS

ÍNDICE DE BNF

CAPITULO I

PROGRAMACIÓN ORIENTADA A ASPECTOS

1.1. 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 línea que los dividiera claramente. A esta etapa se la conoce como código spaghetti, debido a que se tenía una maraña entre datos y funcionalidad similar a la que se forma al comer un plato de esta comida italiana.

A medida que la ingeniería de software fue creciendo, se fueron introduciendo conceptos que llevaron a una programación de más alto nivel: la noción de tipos, bloques estructurados, agrupamientos de instrucciones a través de procedimientos y funciones como una forma primitiva de abstracción, unidades, módulos, tipos de datos abstractos, herencia. Como vemos los progresos más importantes se han obtenido aplicando tres principios, los cuales están estrechamente relacionados entre sí: abstracción, encapsulamiento, y modularidad. Esto último consiste en la noción de descomponer un sistema complejo en subsistemas más fáciles de manejar, siguiendo la antigua técnica de "dividir y conquistar".

Un avance importante lo introdujo la Programación Orientada a Objetos (POO), 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. La clase hace énfasis tanto en los algoritmos como en los datos. La POO está basada en cuatro conceptos [8]:

- Definición de tipos de datos abstractos.
- Herencia.
- Encapsulamiento.
- Polimorfismo por inclusión.

La herencia es un mecanismo lingüístico que permite definir un tipo de dato abstracto derivándolo de un tipo de dato abstracto existente. El nuevo tipo definido "hereda" las propiedades del tipo padre [8].

Ya sea a través de la POO o con otras técnicas de abstracción de alto nivel, se logra un diseño y una implementación que satisface la funcionalidad básica, y con una calidad aceptable. 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, se le denominan intereses transversales *{crosscutting concern).* Algunos de estos conceptos son: sincronización, manejo de memoria, distribución, chequeo de errores, seguridad o redes, entre otros. Así lo muestran los siguientes ejemplos:

1. Consideremos una aplicación que incluya conceptos de seguridad y sincronización, como por ejemplo, asegurarnos que dos usuarios no intenten acceder al mismo dato al mismo tiempo. Ambos conceptos requieren que los programadores escriban la misma funcionalidad en varias partes de la aplicación. Los programadores se verán forzados a recordar todas estas partes, para que a la hora de efectuar un cambio y / o una actualización puedan hacerlo de manera uniforme a través de todo el sistema. Tan solo olvidarse de actualizar algunas de estas repeticiones lleva al código a acumular errores [2]-
2. Manejo de errores y de fallas: agregar a un sistema simple un buen manejo de errores y de fallas requiere muchos y pequeños cambios y adiciones por todo el sistema debido a los diferentes contextos dinámicos que pueden llevar a una falla, y las diferentes políticas relacionadas con el manejo de una falla [3].
3. En general, los aspectos en un sistema que tengan que ver con el atributo performance, resultan diseminados por todo el sistema [3].

Algunos síntomas de un problema que presenta la POO pueden ser categorizados de la siguiente manera [9]:

1. *Código Mezclado* (Code Tangling): En un mismo módulo de un sistema de software pueden simultáneamente convivir más de un requerimiento. Esta múltiple existencia de requerimientos lleva a la presencia conjunta de elementos de implementación de más de un requerimiento, resultando en un *Código Mezclado.*

2. Código Diseminado (Code Scattering): Como los requerimientos están esparcidos sobre varios módulos, la implementación resultante también queda diseminada sobre esos módulos.

Estos síntomas combinados afectan tanto el diseño como el desarrollo de software, de diversas maneras [9]:

- Baja correspondencia: La implementación simultánea de varios conceptos oscurece la correspondencia entre un concepto y su implementación, resultando en un pobre mapeo.
- Menor productividad: La implementación simultánea de múltiples conceptos distrae al desarrollador del concepto principal, por concentrarse también en los conceptos periféricos, disminuyendo la productividad.
- Menor rehuso: Al tener en un mismo módulo implementado varios conceptos, resulta en un código poco reusable.
- Baja calidad de código: El *Código Mezclado* produce un código propenso a errores. Además, al tener como objetivo demasiados conceptos al mismo tiempo se corre el riesgo de que algunos de ellos sean subestimados.
- Evolución más dificultosa: Como la implementación no está completamente modula rizada los futuros cambios en un requerimiento implican revisar y modificar cada uno de los módulos donde esté presente ese requerimiento. Esta tarea se vuelve compleja debido a la insuficiente modularización.

Tenemos entonces que las descomposiciones actuales no soportan una completa separación de conceptos, la cual es clave para manejar un software entendible y evolucionable. Podemos 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.

Como respuesta a este problema nace la Programación Orientada a Aspectos (POA). La 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.

La POA es una nueva metodología de programación que aspira a soportar la separación de las propiedades para los aspectos antes mencionados. Esto implica separar la funcionalidad básica y los aspectos, y los aspectos entre sí, a través de mecanismos que permitan abstraerlos y componerlos para formar todo el sistema.

La POA es un desarrollo que sigue a la POO, y como tal, soporta la descomposición orientada a objetos, además de la procedimental y la funcional. Sin embargo, la programación orientada a aspectos no es una extensión de la POO, ya que puede utilizarse con los diferentes estilos de programación.

1.1.1. RESEÑA HISTÓRICA

Aún antes de que surgiera el concepto de programación orientada a aspectos, el grupo Demeter [5] había considerado y aplicado ideas similares.

El concepto principal detrás del trabajo del grupo Demeter es la programación adaptativa (PA), la cual puede verse como una instancia temprana de la POA. El término programación adaptativa se introduj o recién en 1991. La programación adaptativa constituye un gran avance dentro de la tecnología de software, basada en el uso de autómatas finitos y una teoría formal de lenguaje para expresar concisamente y procesar eficientemente conjuntos de caminos en un grafo arquitectónico, como por ejemplo los diagramas de clase en UML.

La relación entre la POA y la PA surge de la *Ley de Demeter: "Solo conversa con tus amigos inmediatos"*. Esta ley inventada en 1987 en la Northeastern University y popularizada en libros de Booch, Budd, Coleman, Larman, Page-Jones, Rumbaugh, entre otros, es una simple regla de estilo en el diseño de sistemas orientados a objetos.

Para poder escribir código respetando la Ley de Demeter observaron que los conceptos que se entrecruzan entre varias clases deberían y tendrían que ser claramente encapsulados. Esto resultaría en una clara separación de los conceptos de comportamiento y de aquellos conceptos de la funcionalidad básica.

Por lo tanto la separación completa de conceptos fue área de interés de este grupo aún antes de que la POA existiera como tal. En 1995 dos miembros de este grupo, Cristina Lopes, actualmente integrante del grupo Xerox PARC, y Walter Huersch, presentaron un reporte técnico sobre separación de conceptos incluyendo varias técnicas como filtros composicionales y PA para tratar con los conceptos que se entrecruzan. Este reporte

identificó el tema general de separación de conceptos y su implementación, y lo propuso como uno de los problemas a resolver más importante en el diseño y desarrollo de software.

La definición formal de PA puede considerarse como una definición inicial y general de la POA: en PA los programas se descomponen en varios bloques constructores de corte (crosscutting building blocks). Inicialmente separaron la representación de los objetos como un bloque constructor por separado. Luego agregaron comportamiento de estructura (structure-shy behavior) y estructuras de clase como bloques constructores de corte.

La primera definición del concepto de aspecto fue publicada en 1995, también por el grupo Demeter, y se describía de la siguiente forma: *Un aspecto es una unidad que se define en términos de información parcial de otras unidades.*

Cristina Lopes y Karl Lieberherr empezaron a trabajar con Gregor Kickzales y su grupo. A Lopes y Kickzales no les gustó el nombre "Programación Adaptativa" e introdujeron un mejor término "Programación Orientada a Aspectos" con su propia definición y terminología. En la literatura se lo considera a Gregor Kickzales como el creador de este nuevo paradigma.

Al crecer la POA como paradigma fue posible definir la PA como un caso de especial de la POA, esto es, la PA es igual a la POA con grafos y estrategias transversales. Las estrategias transversales podrían considerarse como expresiones regulares que especifican un recorrido en un grafo.

La POA es un paradigma que recién está naciendo. Se están realizando las primeras experiencias prácticas para mostrar su aplicabilidad, y obtener datos empíricos que estimulen la investigación en el tema. La POA está en su primera etapa, donde constantemente surgen nuevos problemas, nuevas herramientas, nuevos contextos en los cuales es posible aplicar aspectos. Este panorama hace pensar que la programación orientada a aspectos se encuentra en mismo lugar que se encontraba la POO hace veinte años.

1.2. POA: CONSIDERACIONES GENERALES

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 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?

Gregor Kickzales y su grupo [3], brinda un marco adecuado que facilita y clarifica la definición de un aspecto. Lo que propone es agrupar los lenguajes orientados a objetos, los procedurales y funcionales como Lenguajes de Procedimiento Generalizado (LPG), ya que sus mecanismos claves de abstracción y composición pueden verse como agrupados bajo una misma raíz. Esa raíz tendría la forma de un procedimiento generalizado. Los métodos de diseño que han surgido para los LPG tienden a dividir el sistema en unidades de comportamiento o funciones.

En general, decimos que dos propiedades se entrecruzan si al implementarse deben componerse de manera diferente, y aún deban ser coordinadas. Esto se debe a que tienen un comportamiento en común. Al proveer los LPG solamente un único medio de composición, es el programador quien debe realizar la tarea extra de co-componer manualmente las propiedades que se entrecruzan, lo cual lleva a un oscurecimiento y a un aumento de la complejidad del código.

Gracias a todo este marco [3], podemos diferenciar los aspectos de los demás integrantes del sistema: al momento de implementar una propiedad, tenemos que la misma tomará una de las dos siguientes formas:

1. Como un componente: si puede encapsularse claramente 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 encapsularse claramente en un procedimiento generalizado. Los aspectos tienden a ser propiedades que afectan la performance o la semántica de los componentes en forma sistemática (Ejemplo: sincronización, manejo de memoria, distribución, etc.)

A la luz de estos términos, podemos enunciar la meta principal de la POA: "brindar un contexto al programador que permita separar claramente componentes y aspectos, separando componentes entre sí, aspectos entre sí, y aspectos de componentes, a través de mecanismos que hagan posible abstraerlos y componerlos para producir el sistema completo". Tenemos entonces una importante y clara diferencia respecto de los LPG, donde todos los esfuerzos se concentran únicamente en los componentes, dejando de lado los aspectos.

Una vez diferenciados los aspectos de los componentes, estamos en condiciones de definir a un aspecto como un concepto que no es posible encapsularlo claramente, y que resulta diseminado por todo el código. Los aspectos son la unidad básica de la programación orientada a aspectos. Una definición más formal, y con la que se trabaja actualmente es: "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".

1.2.2. COMPARACIÓN GRÁFICA

Después de haber introducido los principales rasgos de la POA, podemos comparar la forma de una implementación basada en los LPG versus implementación basada en POA.

Figura 1.1 Implementación POA versus LPG

En la implementación sobre LPG, los aspectos se mezclan con la funcionalidad básica, y entre ellos. En cambio, en la implementación basada en el nuevo paradigma de la POA, la separación es completa, con una notable mejoría en lo que respecta a la modularización. También podemos comparar las diferentes tecnologías que han sido creadas hasta la actualidad, con el paradigma de aspectos, observando la evolución que han tenido las tecnologías.

Tecnología	Conceptos Claves	Constructores
Programación estructurada	Constructores de control Explícitos	Do, while, y otros Integradores
Programación modular	Ocultamiento de Información	Módulos con interfaces bien definidas
Abstracción de datos	Ocultar la representación del dato	Tipo de dato abstracto
Programación orientada a Objetos	Objetos con clasificación Especialización	Clases, objetos, Polimorfismo
Programación orientada a aspectos.	"6 C" y "4 S"	Aspectos

Tabla 1.1 Comparación entre los paradigmas

Las "6 C", de Mehmet Aksit, hacen referencia a los seis aspectos cuyos nombres en inglés comienzan con la letra 'c' y que son los siguientes:

- Entrecruzado (Crosscutting en inglés): el concepto ya ha sido introducido previamente en este trabajo.
- Canónico: brindar una implementación estable para los conceptos.

- Composición: proveer factores de calidad, como adaptabilidad, reusabilidad y extensibilidad.
- Clausura: mantener los factores de calidad del diseño en la etapa de implementación.
- Computabilidad: crear software ejecutable.
- Certificabilidad: evaluar y controlar la calidad de los modelos de diseño e implementación.

Las "4 S" hacen referencia a conceptos sobre Separación Exitosa, de Harold Ossher, y corresponden a:

- Simultáneo: la coexistencia de diferentes composiciones son importantes.
- Auto-Contenido (Self-Contained): Cada módulo debe declarar sus dependencias, para poderlo entenderlo individualmente.
- Simétrico: no debe haber distinción en la forma en que los diferentes módulos encapsulan los conceptos, para obtener una mayor confiabilidad en la composición.
- Espontaneidad (Spontaneous): debe ser posible identificar y encapsular nuevos conceptos, y aún nuevas formas de conceptos que se presenten durante el ciclo de vida del software.

1.2.3. FUNDAMENTOS DE LA **POA**

Los tres principales fundamentos de la POA son:

1. Un lenguaje para definir la funcionalidad básica, conocido como lenguaje base o componente. Podría ser un lenguaje imperativo, o un lenguaje no imperativo (C++, Java, Lisp, ML).
2. Uno o varios lenguajes de aspectos, para especificar el comportamiento de los aspectos. (COOL, para sincronización, RIDL, para distribución, AspectJ, de propósito general.)
3. 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.

Lenguaje base	Lenguaje de a aspectos
Java	-Aspectj -AspectWerkz
C/C++	-AspectC -AapectC++
Small Talk	-AspectS -Apostle
Pitón	-Pythius

Tabla 1.2 Correspondencia Lenguajes base/Lenguajes orientados a aspectos

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, provista a través de los puntos de enlace. 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.

1.2.3.1. ESTRUCTURA GENERAL

La estructura de una implementación basada en aspectos es análoga a la estructura de una implementación basada en los LPG.

Una implementación basada en LPG consiste en:

- Un lenguaje.
- Un compilador o intérprete para ese lenguaje.
- Un programa escrito en ese lenguaje que implemente la aplicación.

La figura 1.2 nos permite visualizar gráficamente la estructura anterior.

Figura 1.2 Estructura LGP

Una implementación basada en POA consiste en:

- El lenguaje base o componente para programar la funcionalidad básica.
- Uno o más lenguajes de aspectos para especificar los aspectos.
- Un tejedor de aspectos para la combinación de los lenguajes.
- El programa escrito en el lenguaje componente que implementa los componentes.
- Uno o más programas de aspectos que implementan los aspectos.

Gráficamente, se tiene una estructura como la siguiente:

sifiif

Figura 1.3 Estructura POA

Cabe notar que la función que realizaba el compilador se encuentra ahora incluida en las funciones del tejedor.

1.2.3.2. DESARROLLO ORIENTADO A ASPECTOS

Diseñar un sistema basado en aspectos requiere entender qué se debe incluir en el lenguaje base, qué se debe incluir dentro de los lenguajes de aspectos y qué debe compartirse entre ambos lenguajes. El lenguaje componente debe proveer la forma de implementar la funcionalidad básica y asegurar que los programas escritos en ese lenguaje componente 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. En general el desarrollo de una aplicación basada en aspectos consiste de tres pasos:

1. Descomposición de aspectos: es descomponer los requerimientos para distinguir aquellos que son componentes de los que son aspectos.
2. Implementadón de requerimientos: implementar cada requerimiento por separado.
3. Recomposición: dar las reglas de recomposición que permitan combinar el sistema completo.

John Lamping [6] propone una diferente visión del diseño. Asegura que la decisión sobre qué conceptos son base y cuáles deben ser manejados por aspectos es irrelevante, ya que no afectará demasiado a la estructura del programa, sino que la clave está en definir cuáles serán los ítems de la funcionalidad básica y cómo obtener una clara separación de responsabilidades entre los conceptos. La primera parte se hereda de la programación no orientada a aspectos y tiene la misma importancia dentro de la POA ya que la misma mantiene la funcionalidad básica. La segunda parte es inherente a la programación orientada a aspectos. Esta parte es fundamental para una descomposición de aspectos exitosa.

Diferentes aspectos pueden contribuir a la implementación de un mismo ítem de la funcionalidad básica y un sólo aspecto puede contribuir a la implementación de varios ítems. Una buena separación de responsabilidades entre los conceptos es lo que hace esto posible, porque el comportamiento introducido por los diferentes aspectos se enfoca en diferentes temas en cada caso, evitando gran parte de los conflictos. Lamping concluye que el trabajo del programador que utiliza POA es definir precisamente los ítems de la

funcionalidad básica y obtener una buena separación de responsabilidades entre los conceptos. Luego los aspectos le permitirán al programador separar los conceptos en el código.

Creemos que la propuesta de Lamping es solo una manera más compleja de llegar a los tres pasos del desarrollo orientado a aspectos, enumerados al comienzo de esta sección. Estos pasos nos proveen un camino más natural y directo. Tenemos entonces dos formas de lograr el mismo resultado pero al seguir los tres pasos logramos el resultado de una manera más apegada y coherente a la filosofía orientada a aspectos.

En un reporte técnico de la Universidad de Virginia [7], se establece que muchos de los principios centrales a la POO son ignorados dentro del diseño orientado a aspectos, como por ejemplo el ocultamiento de información, debido a que los aspectos tienen la habilidad de violar estos principios. Para esto se propone una filosofía de diseño orientada a aspectos que consiste de cuatro pasos:

1. Un objeto es algo: un objeto existe por sí mismo, es una entidad.
2. Un aspecto no es algo. Es algo sobre algo: un aspecto se escribe para encapsular un concepto entrecruzado. Por definición un aspecto entrecruza diferentes componentes, los cuales en la POO son llamados objetos. Si un aspecto no está asociado con ninguna clase, entonces entrecruza cero clases, y por lo tanto no tiene sentido en este contexto. Luego, para que un aspecto tenga sentido debe estar asociado a una o más clases; no es una unidad funcional por sí mismo.
3. Los objetos no dependen de los aspectos: un aspecto no debe cambiar las interfaces de las clases asociadas a él. Solo debe aumentar la implementación de dichas interfaces. Al afectar solamente la implementación de las clases y no sus interfaces, la encapsulación no se rompe. Las clases mantienen su condición original de cajas negras, aún cuando puede modificarse el interior de las cajas.
4. Los aspectos representan alguna característica o propiedad de los objetos, pero no tienen control sobre los mismos: esto significa que el ocultamiento de información puede ser violado en cierta forma por los aspectos porque éstos conocen detalles de un objeto que están ocultos al resto de los objetos. Sin embargo, no deben manipular la representación interna de los objetos más allá de lo que sean capaces de manipular el

resto de los objetos. A los aspectos se les permite tener esta visión especial, pero debería limitarse a manipular objetos de la misma forma que los demás objetos lo hacen, a través de la interface.

Esta filosofía de diseño permite a los aspectos hacer su trabajo de automatización y abstracción, respetando los principios de ocultamiento de información e interface.

Creemos que ésta filosofía es una política de diseño totalmente aceptable, y que se debe conocer y aplicar durante el desarrollo orientado a aspectos. Si bien es una excelente política de diseño, no deberíamos quedarnos solamente en esto, sino que deberíamos llevarla más allá, haciéndola formar parte de las reglas del lenguaje orientado a aspectos. Esto es, que no sea una opción, sino que sea forzada por el lenguaje. El mismo problema que antes existía entre las distintas componentes de un sistema, está presente ahora entre objetos y aspectos. Es decir, los aspectos no deben inmiscuirse en la representación interna del objeto, por lo tanto se necesita una solución similar: el lenguaje debería proveer una interface para que aspectos y objetos se comuniquen respetando esa restricción. Esta es una de las desventajas de los lenguajes orientados a aspectos actuales. El desafío está en construir lenguajes orientados a aspectos capaces de brindar mecanismos lingüísticos suficientemente poderosos para respetar por completo todos los principios de diseño.

1.2.4. 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.

1.2.4.1. GUIAS DE DISEÑO

Tomando como base la guía de diseño para la implementación de sistemas abiertos, descripta en [12], 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. En SMove **[11],** 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 minimal. 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.

1.2.5. LENGUAJES DE ASPECTOS ESPECÍFICOS VERSUS DE PROPÓSITO GENERAL

Existen dos enfoques principales en el diseño de los lenguajes de aspectos: los *lenguajes de aspectos de dominio específico* y los *lenguajes de aspectos de propósito general.*

Los lenguajes de aspectos de dominio específico son capaces de manejar uno o más aspectos, pero no pueden manejar otros aspectos más allá de los aspectos para los cuales fueron diseñados. Tienen un mayor nivel de abstracción que el lenguaje base, y expresan los conceptos de dominio específico en un nivel de representación más alto. Para garantizar que los aspectos del dominio sean escritos en el lenguaje de aspectos, y evitar conflictos de control y dependencia entre el lenguaje base y el lenguaje de aspectos, los lenguajes de dominio específicos imponen restricciones en la utilización del lenguaje base. Como

ejemplo de este tipo de lenguaje de aspectos se puede nombrar a COOL (Coordination Language), de Xerox, un lenguaje para sincronización de hilos concurrentes. En COOL, el lenguaje base es una restricción de Java, ya que se han eliminado los métodos wait, notify, y notifyAll, y la palabra reservada synchronized para evitar que el lenguaje base sea capaz de expresar sincronización. Se obtiene un mayor nivel de abstracción que en Java al especificar la sincronización de hilos de manera declarativa. Un segundo ejemplo es RIDL (Remote Interaction and Data transfers Language), para el aspecto de distribución: invocación remota y transferencia de datos.

Los lenguajes de aspectos de propósito general son diseñados para describir cualquier clase de aspecto, no solo específicos, por lo que no pueden imponer restricciones al lenguaje base. El nivel de abstracción del lenguaje base y del lenguaje de aspectos de propósito general es el mismo. Además, tienen el mismo conjunto de instrucciones, ya que debe ser posible expresar cualquier código al programar un aspecto.

La principal desventaja de los lenguajes de aspectos de propósito general es que no garantizan la separación de conceptos, esto es, que la unidad de aspecto se utilice únicamente para programar el aspecto. Esto último es uno de los objetivos principales de la POA. En comparación, los lenguajes de aspectos de dominio específico fuerzan la separación de conceptos.

Sin embargo, los de propósito general, proveen un ambiente más adecuado para el desarrollo de aspectos. Es una única herramienta capaz de describir todos los aspectos que se necesiten. Al pasar de una aplicación a otra, por ejemplo al pasar de trabajar en un contexto de sincronización a uno de manejo de excepciones, se mantiene la misma herramienta para programar los aspectos. No se debe aprender para cada aplicación todo de nuevo, que es costoso y hasta frustrante, sino que cada vez se obtiene una mayor experiencia, con lo cual programar nuevos aspectos es más sencillo, y permite evolucionar los conceptos sobre el paradigma. Con los específicos, el pasar de una aplicación a otra, significa comenzar de cero nuevamente, otros lenguajes, otras restricciones, que quizás sean similares, pero quizás sean completamente distintas. Si es el caso que se desee utilizar aspectos para una única aplicación, por una única vez, entonces un lenguaje de aspectos de dominio específico hará un mejor papel, con una separación de conceptos más clara.

Pero dada la gran variedad de aplicaciones en los cuales los aspectos pueden ser aplicados y observando las ventajas que trae un lenguaje de propósito general, es preferible esta última aproximación siendo capaz de manejar varios ambientes de trabajo con el mismo lenguaje, reduciendo costos de aprendizaje y desarrollo.

1.2.6. EL ROL DEL LENGUAJE BASE

Una de las decisiones relevantes a la hora de diseñar un lenguaje orientado a aspectos tiene que ver con el lenguaje base. Las distintas alternativas son: diseñar un *nuevo* lenguaje base o *adoptar* un lenguaje base existente.

En el caso de adoptar un lenguaje base existente, es preferible que el mismo sea un lenguaje de propósito general, de manera de no restringir la aplicabilidad del lenguaje orientado a aspectos. Una ventaja obvia de esta opción es el menor esfuerzo para diseñar e implementar lenguajes para ambientes orientados a aspectos, dado que nos ahorramos el trabajo que implica el diseño de un nuevo lenguaje base. También, al trabajar con un lenguaje existente nos evitamos los problemas de trabajar con algo que todavía no ha sido probado exhaustivamente. Una última ventaja radica en el hecho que el programador solo debe aprender el lenguaje de aspectos, y no también un nuevo lenguaje para la funcionalidad básica, la cual sigue siendo, aún en la POA, la parte más grande del sistema.

Si en cambio se elige diseñar un nuevo lenguaje base entonces se obtiene una mayor flexibilidad, una mayor libertad, y una manera más natural de expresar aquellas abstracciones que forman parte de la funcionalidad básica. Si bien estas ventajas son importantes, el costo de diseñar un nuevo lenguaje resulta demasiado alto. Sin embargo, existe otra, más oculta, pero de mucha mayor trascendencia, dado que afecta directamente al corazón mismo de la POA, al surgir del enunciado de la meta principal que persigue este nuevo paradigma, la separación completa de conceptos. Una gran parte de los investigadores que han trabajado con aspectos, establecen que no está claro si es posible lograr la separación completa de conceptos utilizando un lenguaje base existente. Si esta posición logra finalmente imponerse, entonces el futuro de la POA indica que el diseño de nuevos lenguajes base será altamente necesario. En [1], K. Mehner y A. Wagner, de la Universidad de Paderborn, plantean una futura investigación con respecto a este último

concepto. Estudiarán si en un domino de aplicación, como por ejemplo el dominio de sincronización de hilos concurrentes, la separación de conceptos puede lograrse utilizando un lenguaje base existente. De no ser así investigarán si el problema ocurrirá con cualquier lenguaje base de propósito general o si se debe construir un nuevo lenguaje base.

Hasta ahora los lenguajes orientados a aspectos específicos y de propósito general han elegido utilizar un lenguaje base existente. Aparte de las ventajas de esta elección esto trae consigo una serie de problemas. En el caso de los lenguajes de dominio específico uno no es completamente libre para diseñar los puntos de enlace sino que se restringe a aquellos que pueden ser identificados en el lenguaje base.
Otro problema surge de la restricción que impone sobre el lenguaje base. Dicha restricción es bastante difícil de lograr ya que se deben eliminar elementos de un sistema complejo, donde cada uno tiene su lugar y sus responsabilidades, y están coordinados entre sí. Tener en cuenta que ya es una tarea compleja el diseño de un lenguaje de programación, aún es más difícil realizarle cambios a un lenguaje que no fue diseñado para esta evolución.

Los lenguajes de propósito general son más fáciles de implementar sobre la base de un lenguaje de programación existente al no imponer restricciones sobre la base. También tienen el mismo problema de limitar los puntos de enlace a aquellos que pueden ser identificados en el lenguaje base.

1.2.7. ASPECTOS EN LENGUAJES PROCEDURALES

Las razones para utilizar aspectos dentro de los lenguajes procedurales son similares a aquellas para los lenguajes orientados a objetos. Los distintos procedimientos que forman parte del sistema quizás necesiten compartir datos e incluso otros procedimientos, y también quizás sea deseable agregarles comportamiento. Se podría pensar que la programación orientada a objetos solucionaría estos problemas, pero como hemos discutido antes, existen conceptos que no son capturados correctamente.

Sin utilizar aspectos, los lenguajes procedurales pueden compartir datos y procedimientos a través del pasaje de parámetros o a través de un espacio de nombre común. En los lenguajes estructurados por bloques, el espacio de nombre se particiona en una estructura

jerárquica, donde cada bloque tiene visibilidad sobre el(los) bloque(s) que lo contiene(n). Luego, si dos procedimientos *Proel* y *Proel* necesitan acceder a una variable *V,* entonces *V* deberá declararse en el mismo bloque que *Proel* y *Proel,* o en un bloque que contenga a los procedimientos, así como lo muestra el siguiente código:

```
Procedure ProcGlobal;
Var V: boolean;
Procedure Proel; {

};
Procedure Proc2;{

}; {

};
```

Código 1. Aspectos en Lenguajes Procedurales

Sin embargo, el mecanismo de alcance de los lenguajes estructurados por bloques sufre de importantes limitaciones. En el análisis de Wulf y Shaw sobre los lenguajes estructurados [19] se describen los cuatro problemas más importantes:

- Efectos colaterales: pueden definirse como la modificación del ambiente no local. Están presentes en la mayoría de los lenguajes, ya que deshabilitar todos los posibles efectos colaterales resulta en un lenguaje muy restrictivo, poco práctico, y de escasa utilidad.
- Acceso indiscriminado: toda entidad declarada en un bloque, será siempre visible a los bloques internos, y no hay manera de evitarlo.
- Vulnerabilidad: puede ser imposible mantener el acceso a una variable, cuando se lo desea. Puede ocurrir cuando un procedimiento *P* declara una variable *V,* y tiene un procedimiento interno *H* que accede a *V.* Ahora si se inserta entre ellos un procedimiento *J,* que declara una variable local *V* (oculta a la variable *V* declarada en *P)* entonces *H* accederá a la variable Kdeclarada en *J* y no a la declarada en *P,* que era lo deseado.
- No se permiten definiciones solapadas: Dados tres procedimientos, *Pl, P2,* y *P3,* que desean compartir dos variables, *VI* y *V2* de la siguiente forma: *Pl* y *P2* comparten *VI, P2* y *P3* comparten *V2, P3* no debe acceder a *VI,* y *Pl* no debe acceder a *V2.* Este tipo

de relación es imposible de implementar en un lenguaje estructurado por bloques. Gráficamente, se desea modelar el siguiente comportamiento:

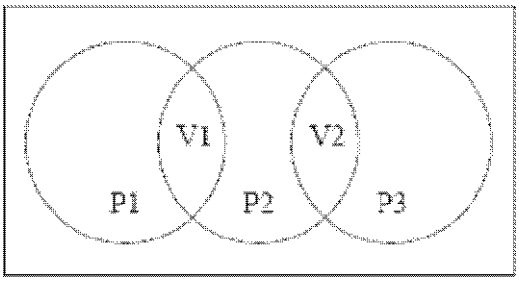


Figura 1.4 Definición solapada

Por lo tanto, es claro que la estructura de bloques no es un mecanismo apropiado para capturar los conceptos entrecruzados debido a los problemas de programación potenciales que puede ocasionar (acceso indiscriminado y vulnerabilidad), y la imposibilidad de capturar ciertos tipos de conceptos entrecruzados (definiciones solapadas). Luego, es la intención en [7] que los aspectos se empleen como un mecanismo que extienda las reglas de alcance de los lenguajes procedurales para capturar los conceptos entrecruzados sin la presencia de los problemas anteriormente mencionados. Entonces un aspecto se utiliza como un contenedor para almacenar procedimientos y variables específicas. El acceso a los aspectos es especificado por una palabra clave *(accessed by* <Lista_Procedimientos>) que determina los únicos procedimientos que pueden acceder a él, sin tener en cuenta el nivel de anidamiento de los mismos. De esta forma, los aspectos solucionan el acceso indiscriminado, la vulnerabilidad y permiten definiciones solapadas, al introducir un nuevo espacio de nombre, que solo es accesible por algunos procedimientos. Por ejemplo, el siguiente código resuelve el problema de definición solapada:

```
Aspee t A
 accessed byPl, P2; (var
 VI: boolean;
}
AspeeíB
 accessed byP2, P3; { var
 V2: boolean;
}
ProcedurePl;{
//puede acceder a VI y no a V2
}
ProcedureP2;{
//puede acceder a VI y a V2
```

```
}
ProcedureP3;{
//puede acceder a V2 y no a VI
}
```

Código 2. Aspectos en Lenguajes Procedurales sin Problemas

Finalmente, los aspectos descriptos también satisfacen las propiedades requeridas en el análisis de Wulf y Shaw [19] para una solución alternativa apropiada a las limitaciones de los lenguajes estructurados por bloques:

- No debe extender el alcance de un nombre a los bloques internos: Se especifica explícitamente los procedimientos que pueden acceder a un aspecto.
- El derecho de acceso a un nombre tiene que ser de mutuo acuerdo entre el creador y el que accede: se cumple por la misma razón de la propiedad anterior.
- Los accesos a una estructura y a sus sub-estructuras deben ser independientes: el acceso a una estructura no implica acceder a sus sub-estructuras.

Como conclusión, se observa que los aspectos juegan un rol diferente en los lenguajes procedurales que en los lenguajes orientados a objetos. Esto resulta de la diferencias entre los paradigmas, donde los componentes se describen como objetos en uno y como procedimientos en el otro. Al aplicar aspectos a los lenguajes procedurales, los procedimientos tienen conocimiento y dependen de los aspectos, a diferencia de los objetos, que no tienen conocimiento ni dependen de los aspectos.

1.2.8. APROXIMACIONES ALTERNATIVAS DE ASPECTOS

La meta-programación es la capacidad que tienen los programas de razonar sobre sí mismos. Un lenguaje permite la meta-programación cuando tiene la capacidad de razonar sobre programas escritos en ese lenguaje. Dentro del paradigma funcional se denominan lenguajes meta funcionales, y dentro del paradigma lógico, lenguajes meta lógicos.

El paradigma declarativo, que incluye el paradigma funcional y el lógico, permite naturalmente la meta-programación. Se puede pensar que los aspectos son a la

funcionalidad básica lo que los meta-programas son a los programas:

Meta - programación _ programas *Aspectos funcionalidad _ básica*

Siguiendo este razonamiento surgen dos aproximaciones alternativas para describir aspectos que serán abarcadas a continuación.

1.2.8.1. META-PROGRAMACIÓN LÓGICA DE ASPECTOS

Se basa en la utilización de lenguajes meta lógicos para razonar sobre aspectos, unificando el lenguaje para declarar aspectos con el lenguaje para implementar el tejedor.

Los aspectos no son declarados dentro de un lenguaje de aspectos diseñado especialmente para tal efecto, sino que son simplemente aserciones lógicas expresadas en un lenguaje lógico general. Estos hechos lógicos funcionan como "libros" de una librería de reglas lógicas que implementan el tejedor.

La principal ventaja de utilizar hechos lógicos para declarar aspectos en vez de un lenguaje de aspectos diseñados especialmente es que las declaraciones de aspectos se pueden acceder y declarar por reglas lógicas. Esta técnica permite al usuario adaptar o extender el lenguaje de aspectos.

El programa básico es representado indirectamente por un conjunto de proposiciones lógicas. Bajo esta aproximación un tejedor entendería un conjunto de declaraciones para describir aspectos como las que propone Brichau **[17]:**

- *introduceMethod(<nombre_de_clase>,{<código_del_método>})*\ introduce el método <*código_del_métodó*> en la clase <*nombre_de_clasé*>.
- *introduceVariable(<nombre_de_clasé> ,{<variablé>}):* introduce la variable <*variablé*> en la clase <*nombre_de_clasé*>.
- *adviceBeforeMethod(<nombre_de_clase>,<nombre_del_método>,{<código_ agregadó>})*\ agrega <*código_agregadó*> antes del método <*nombre_del_métodó*> en laclase <*nombre_de_clasé*>.
- *adviceAfterMethod(<nombre_de_clasé>,<nombre_del_métodó>,{<código_agregado >}):* agrega <*código_agregadó*> después del método <*nombre_del_métodó*> en la clase <*nombre_de_clasé*>.

Código 3. Proposiciones Lógicas

Dado un módulo de declaraciones el tejedor genera código para todas las declaraciones presentes en dicho módulo. Esto lo logra generando consultas lógicas para recoger todas las declaraciones mencionadas. Por ejemplo: el siguiente módulo implementa un aspecto simple de traza:

```
adviceBeforeMethod(claseA,unmétododeA,
{system.out.printlnfEntrando a claseA.unmétodoA');}).
adviceAfterMethod(claseA,unmétododeA,
{system.out.println('Saliendo de claseA.unmétodoA');}).
adviceBeforeMethod(claseB,unmétododeB,
{system.out.printlnfEntrando a claseB.unmétodoB');}).
adviceAfterMethod(claseB,unmétododeB,
{system.out.println('Saliendo de claseB.unmétodoB);}).
adviceBeforeMethod(claseC,unmétododeC,
{system.out.printlnCEntrando a claseC.unmétodoC');}).
adviceAfterMethod(claseC,unmétododeC,
{system.out.printlnfSaliendo de claseC.unmétodoC);}).
```

Código 4. Implementación de un Aspecto Simple Traza

Tomando ventaja del paradigma lógico podemos evitar la duplicación de código del módulo anterior reformulándolo a través de reglas lógicas:

```
adviceBeforeMethod(? clase,? unmétodo,
{system.out.printlnfEntrando a ?clase.?unmétodo);}):-traza(7clase,?unmétodo).
adviceAfterMethod(? clase,? unmétodo,
{system.out.printlnCSaliendo de ?clase.?unmétodo);}):-traza(?clase,?unmétodo).
traza(claseA,unmétododeA).
traza(claseB,unmétododeB).
traza(claseC,unmétododeC).
```

Código 5. Implementación de Aspectos atreves de Reglas Lógicas

Nota: los identificadores de variables comienzan con un signo de interrogación. Por ejemplo: *?clase.*

Las dos primeras declaraciones son reglas lógicas que declaran avisos para cada declaración de la forma *traza(?clase,?unmétodo).* Significa que las declaraciones de avisos son válidas solo si hay declaraciones de trazas. En este proceso, el motor de inferencia lógico liga las variables *?clase* y *?unmétodo* con las constantes especificadas en las declaraciones de trazas. A través de este proceso de evaluación obtenemos el mismo

aspecto que antes. También se observa la separación entre la implementación del aspecto y el uso del aspecto: las dos primeras reglas son implementadas por el programador del aspecto y las últimas tres son introducidas por el usuario del aspecto cuando el aspecto se combina con un programa básico particular. El verdadero código del aspecto se concentra en las dos primeras reglas mientras que las tres restantes completan los parámetros donde el código de aspecto debe insertarse.

También se pueden componer aspectos de una manera directa, aprovechando una vez más el poder de la lógica. Supongamos que deseamos implementar un nuevo módulo de aspectos a partir de dos aspectos ya definidos: *aspectoA* y *aspectoB.* El nuevo aspecto tendrá el comportamiento de *A* y *B* en aquellos métodos que ambos afecten; el comportamiento de *A* si *B* no afecta ese método y el comportamiento de *B ÚA* no afecta ese método. Contiene las variables y métodos de ambos aspectos.

introduceMethodf? clase,? códigométodo):-aspectA.introduceMethod(? clase,? códigométodo). introduceMethodf? clase,? códigométodo):-aspectB.introduceMethod(? clase,? códigométodo). introduceVariablef? clase,?variable):-aspectA.introduceVariable(? clase,? variable). introduceVariablef? clase,?variable):-aspectB.introduceVariable(?' clase,?variable). adviceBeforeMethod(?clase,?método,{?codigoA ?codigoB}):-aspectA. adviceBeforeMethodf? clase, ?método, (?codigoAJ) and aspectB.adviceBeforeMethod(?) clase,? método,{?)codigoB}). adviceBeforeMethod(? clase,? método,{? códigoA }):-aspectA. adviceBeforeMethodf? clase, ?método, {? códigoA}) and not(aspectB.adviceBeforeMethod(?) clase,? método,{? códigoB})). adviceBeforeMethod(? clase,? método,{? códigoB }):-notfaspectA. adviceBeforeMethodf? clase, ?método, {?códigoA})) and aspectB.adviceBeforeMethod(? clase,? método,{?'códigoB}). adviceAfterMethod(?clase,?método,{?códigoA ?códigoB}):-aspectA. adviceAfterMethodf?clase, ?método, {?códigoA}) and aspectB.adviceAfterMethod(? clase,? método,{? códigoB}). adviceAfterMethod(? clase,? método,{? códigoA }):-aspectA. adviceAfterMethodf?clase, ?método, {?códigoA}) and not(aspectB.adviceAfterMethod(? clase,? método,{? códigoB})). adviceAfterMethod(? clase,? método,{? códigoB }):-notfaspectA. adviceAfterMethodf?clase, ?método, {?códigoA})) and aspectB.adviceAfterMethod(? clase,? método,{? códigoB}).*

Código 6. Implementación de Aspectos atreves de Clases y Métodos

Nota: se introduce el operador de alcance ".", para referir a las declaraciones definidas en otros módulos. Por ejemplo: *aspectB.declaraciónS* especifica que *declaración^* se encuentra definida en el módulo *B.* Además se introduce el orden de la composición. Esto se refleja en la quinta y sexta regla donde el código de *A* precede al código de *B.* Una de las herramientas dentro de esta aproximación hacia los aspectos es TyRuBa, un lenguaje de meta-programación lógico para Java. Principalmente fue diseñado para generar código Java a partir de descripciones lógicas de la estructura del programa. Fue implementado como parte de la tesis de postgrado de Kris de Volder sobre meta-programación lógica orientada a tipos. El acrónimo TyRuBa deriva de TypeRuleBase, que significa basado en reglas con tipos. En pocas palabras, es un lenguaje de programación lógico con facilidades para la manipulación de código Java con el propósito de generar código. Para mayor referencia dirigirse a la página de TyRuBa [21].

Las ventajas de esta aproximación provienen del paradigma lógico, porque el programador se concentra en los aspectos y no en cómo estos serán combinados por el tejedor. La combinación es tarea del proceso de evaluación. Los aspectos se pueden componer fácilmente, como se vio anteriormente, reduciendo los conflictos entre ellos.

Cabe destacar que la utilización de un lenguaje lógico maduro, existente y probado con los años, resulta en una importante ventaja sobre un lenguaje de aspectos especialmente diseñado: el lenguaje lógico puede servir de manera uniforme como formalismo para declarar aspectos como hechos lógicos y como meta-lenguaje para la combinación como reglas lógicas.

1.2.8.2. META-PROGRAMACIÓN FUNCIONAL DE ASPECTOS

Se basa en la utilización de lenguajes funcionales de alto orden para describir aspectos. La meta-programación funcional es una propuesta general y efectiva para especificar el código de aspectos y el tejedor. Utilizando la terminología de aspectos se tiene que los componentes son programas funcionales mientras que los aspectos se implementan como transformación de programas. El proceso de tejer se considera como una composición de

programas que combina los componentes y los aspectos aplicando las transformaciones a los componentes modelados por los aspectos [18].

La razón de utilizar meta-programas funcionales radica en la facilidad de verificar propiedades y la naturalidad de los lenguajes funcionales de alto orden para definir manipulaciones abstractas de programas.
Ambas aproximaciones, lógica y funcional, proveen un ambiente adecuado para estudiar e investigar los cimientos teóricos del paradigma de la programación orientada a aspectos.

CAPITULO II

ANÁLISIS DE LENGUAJES ORIENTADOS A ASPECTOS

2.6. LENGUAJES ORIENTADOS A ASPECTOS

Como para el resto de los paradigmas de programación, un lenguaje que implemente el paradigma orientado a aspectos consta de dos partes:

Una especificación del lenguaje que describe las construcciones y sintaxis del lenguaje. Una implementación del lenguaje que verifica que el código se ajusta a la especificación del lenguaje y convierte el código a un formato que la máquina pueda ejecutar.

2.2. ESPECIFICACIÓN DEL LENGUAJE

Un lenguaje orientado a aspectos debe proporcionar construcciones para especificar las reglas de entretejido que guiarán al tejedor de aspectos en su tarea de integrar los aspectos con el código principal de la aplicación. Los lenguajes orientados a aspectos suelen basar sus **reglas de entretejido** en el concepto de **punto de enlace** *(j'oin point).* Un punto de enlace es un lugar bien definido en la ejecución de un programa.

Mediante las reglas de entretejido indicaremos aquellos puntos de enlace que se ven afectados por cierto aspecto.

Las construcciones del lenguaje para definir las reglas de entretejido deben ser potentes y expresivas, permitiendo expresar desde reglas muy específicas que involucren a uno o pocos puntos de enlace, hasta reglas muy genéricas que engloben a un gran número de puntos. Estos mecanismos y construcciones varían de unos lenguajes a otros. En el capítulo siguiente profundizaremos en el caso particular de AspectJ,

Figura 2.1 Generar ejecutable: enfoque tradicional & POA

A continuación describiremos informalmente algunos lenguajes orientados a aspectos disponibles actualmente. En el subcapítulo 2.3 se estudiará con mayor profundidad el lenguaje orientado a aspectos AspectJ, ya que se ha seleccionado este lenguaje por ser la herramienta con mayor usabilidad, futuro, popularidad y desarrollo.

2.2.1. 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). Por lo tanto la herramienta se llama JPAL que significa Junction Point Aspect Language [10], 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. De hecho, 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.

El código que agrega el Esquema del Tejedor invoca, cuando es alcanzado en ejecución, las acciones correspondientes para permitir la ejecución de los programas de aspectos. Esto permite una vinculación dinámica con los programas de aspectos, lo cual hace posible

modificar en tiempos de ejecución los programas de aspectos. Sin embargo, esta solución no es lo suficientemente poderosa como para agregar o reemplazar programas de aspectos en ejecución. Para tal efecto se agrega al Esquema del Tejedor una entidad llamada Administrador de Programas de Aspectos (APA), el cual puede registrar un nuevo aspecto de una aplicación y puede llamar a métodos de aspectos registrados. Es implementado como una librería dinámica que almacena los aspectos y permite dinámicamente agregar, quitar o modificar aspectos, y mandar mensajes a dichos aspectos. La comunicación entre el Administrador y el Esquema del Tejedor se logra a través de un Protocolo de Comunicación entre Procesos que permite registrar aspectos dinámicamente.

La siguiente figura resume la arquitectura JPAL:

Figura 2.2 Arquitectura JPAL

El costo de ejecución de una arquitectura basada en JPAL depende en la implementación del Administrador de Programas de Aspectos que tiene mayor eficiencia en ambientes interpretados [10].

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.
Tomando ventaja de esta redirección se obtiene un modelo de arquitectura para el manejo dinámico de aspectos. Su principal aplicación es para la implementación de sistemas distribuidos.

2.2.2. ASPECTS

AspectS [22] 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 [21].** Principalmente, AspectS está basado en dos proyectos anteriores: AspectJ de Xerox Pare 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 un metanivel, 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. En este lenguaje los aspectos se implementan a través de clases y sus instancias actúan como un objeto, respetando el principio de uniformidad. Un aspecto puede contener un conjunto de receptores, enviadores o clases enviadoras.

Estos objetos se agregan o se remueven por el cliente y serán usados por el proceso de tejer en ejecución para determinar si el comportamiento debe activarse o no. En Squeak, la interacción de los objetos está basada en el paradigma de pasaje de mensajes. Luego, en AspectS los puntos de enlace se implementan a través de envíos de mensajes. Los avisos asocian fragmentos de código de un aspecto con cortes y sus respectivos puntos de enlace, como los objetivos del tejedor para ubicar estos fragmentos en el sistema. Para representar esos fragmentos de código se utilizan bloques (instancias de la clase BlockContext). Los tipos de avisos definibles en AspectS son:

- Antes y después de la invocación a un método *(AsBeforeAfterAdvice).*
- Para manejo de excepciones *{AsHandlerAdvice).*
- Durante la invocación de un método *(AsAroundAdvice).*

Un calificador de avisos *(AsAdviceQualifier)* es usado para controlar la selección del aviso apropiado. Es similar al concepto de designadores de cortes de AspectJ. Utiliza un tejedor dinámico que transforma el sistema base de acuerdo a lo especificado en los aspectos. El código tejido se basa en el MethodWrapper y la meta-programación. MethodWrapper es un mecanismo que permite introducir código que es ejecutado antes, después o durante la ejecución de un método.

El proceso de tejer sucede cada vez que una instancia de aspectos es instalada. Para revertir los efectos de un aspecto al sistema, el aspecto debe ser desinstalado. A este proceso se lo conoce como "destejer", del inglés un weaving. El tejido de AspectS es completamente dinámico ya que ocurre en ejecución.

2.2.3. ASPECTC++

AspectC++ [16] es un lenguaje de aspectos de propósito general que extiende el lenguaje C++ para soportar el manejo de aspectos. En este lenguaje los puntos de enlace son puntos en el código componente donde los aspectos pueden interferir. Los puntos de enlaces son capaces de referir a código, tipos, objetos, y flujos de control.

Las expresiones de corte son utilizadas para identificar un conjunto de puntos de enlaces. Se componen a partir de los designadores 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.

La información del contexto del punto de enlace puede exponerse mediante cortes con argumentos y expresiones que contienen identificadores en vez de nombres de tipos, todas las veces que se necesite. Diferentes tipos de aviso pueden ser declarados, permitiendo que el aspecto introduzca comportamiento en diferentes momentos: el aviso después (after advice), el aviso antes (before advice) y el aviso durante (around advice).

Los aspectos en AspectC++ implementan en forma modular los conceptos entrecruzados y son extensiones del concepto de clase en C++. Además de atributos y métodos, los aspectos pueden contener declaraciones de avisos. Los aspectos pueden derivarse de clases y aspectos, pero no es posible derivar una clase de un aspecto.

La arquitectura del compilador de AspectC++ se muestra en la figura 2.2.

Primero el código fuente de AspectC++ es analizado léxica, sintáctica y semánticamente. Luego se ingresa a la etapa de planificación, donde las expresiones de corte son evaluadas y se calculan los conjuntos de puntos de enlace. Un plan para el tejedor es creado conteniendo los puntos de enlace y las operaciones a realizar en estos puntos. El tejedor es ahora responsable de transformar el plan en comandos de manipulación concretos basados en el árbol sintáctico de C++ generado por el analizador PUMA. A continuación el manipulador realiza los cambios necesarios en el código. La salida del manipulador es código fuente en C++, con el código de aspectos "tejido" dentro de él. Este código no

contiene constructores del lenguaje AspectC++ y por lo tanto puede ser convertido en código ejecutable usando un compilador tradicional de **C++.**

Si bien el manejo de aspectos en AspectC++ es similar al manejo que proporciona AspectJ, introduce modificaciones importantes dentro del modelo de puntos de enlace, permitiendo puntos de enlace sobre clases, objetos, y sobre el flujo de control. Esto resulta en un diseño del lenguaje más coherente con el paradigma de aspectos.

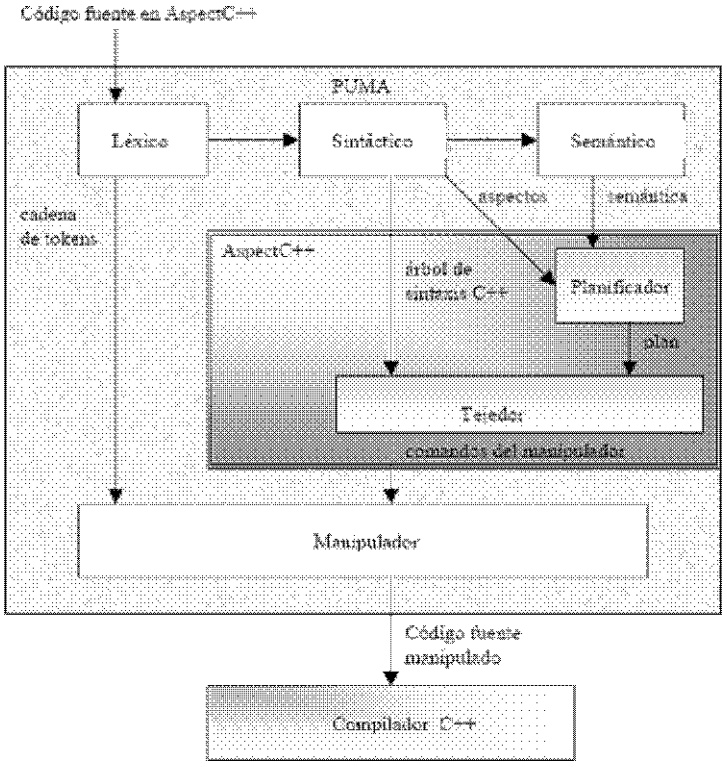


Figura 2.3 Arquitectura del compilador de AspectC++

2.2.4. MALAJ

Malaj [13,14], es un sistema que soporta la programación orientada a aspectos. Define constructores lingüísticos separados para cada aspecto de dominio específico, donde el código de los aspectos tiene una visibilidad limitada del código funcional, reduciendo los posibles conflictos con las características lingüísticas tradicionales y también, con el principio de encapsulación. Malaj es un lenguaje orientado a aspectos de dominio específico, concentrándose en dos aspectos: sincronización y relocación. Puede verse como un sucesor de los lenguajes COOL y RIDL por su filosofía, enfatizando la necesidad de restringir la visibilidad de los aspectos, y reglas claras de composición con

los constructores tradicionales. Para cada aspecto, provee un constructor lingüístico distinto, limitando así la visibilidad del aspecto sobre el módulo funcional asociado a él. Esto último se logra al estudiar cuidadosamente la relación entre el código funcional y un aspecto dado.

El lenguaje base de Malaj es una versión restringida de Java, donde se han removido los servicios que proveen los aspectos mencionados. Los servicios removidos son: la palabra clave synchronized, y los métodos wait, notify, and notifyAll. Para el aspecto de sincronización Malaj provee el constructor *guardián.* Cada guardián es una unidad distinta con su propio nombre, y se asocia con una clase en particular (esto es, "vigila" esa clase) y expresa la sincronización de un conjunto relacionado de métodos de esa clase, es decir, que el guardián de una clase K representa básicamente el conjunto de métodos sincronizados de *V.* Los guardianes no pueden acceder a los elementos privados de la clase que vigilan, y el acceso a los atributos públicos y protegidos se limita a un acceso de sólo lectura. El comportamiento adicional en un guardián para un método *m* de la clase asociada se especifica introduciendo código que se ejecutará antes o después de *m,* a través de las cláusulas before y añer. Una última característica de los guardianes es que pueden heredarse. Para reducir el problema de anomalía de herencia [4] las cláusulas befóte y after en una clase pueden referirse a las cláusulas before y after de su clase padre a través de la sentencia **super.**

El aspecto de relocación involucra el movimiento de objetos entre sitios en un ambiente de redes. Este tipo de relación es claramente dinámico. Para este aspecto Malaj provee el constructor *relocator,* que llamaremos retocador. Un retocador será una unidad diferente con su propio nombre, y se asocia con una clase en particular. Las acciones de relocación pueden ejecutarse antes o después de la ejecución de un método. Para modelar este comportamiento, el retocador brinda cláusulas before y after, que permiten la especificación deseada. También la visibilidad del retocador es limitada. Como el constructor guardián, un retocador puede heredarse, y las cláusulas before y after en una clase pueden referirse a las cláusulas before y after de su clase padre a través de la sentencia **super,** y así reducir el impacto de la anomalía de herencia [4]. Como conclusión Malaj provee una solución intermedia entre flexibilidad y poder por un lado, y entendimiento y facilidad de cambio por el otro. No permite describir cualquier

aspecto, pero sí captura el comportamiento de dos conceptos relacionados con el código funcional. El próximo paso en Malaj es extenderlo para abarcar otros aspectos.

2.2.5. HYPERJ

La aproximación por Ossher y Tarr sobre la separación multidimensional de conceptos (MDSOC en inglés) es llamada hyperspaces, y como soporte se construyó la herramienta HyperJ [15] 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. Cualquier modelo puede aumentar su comportamiento componiéndose con otro: HyperJ no exige una jerarquía base distinguida y no diferencia entre clases y aspectos, permitiendo así que los *hiperslices* puedan

extenderse, adaptarse o integrarse mutuamente cuando se lo necesite. Esto demuestra un mayor nivel de expresividad para la descripción de aspectos en comparación con las herramientas descriptas anteriormente.

Existe una versión prototipo de HyperJ que brinda un marco visual para la creación y modificación de las relaciones de composición, permitiendo un proceso simple de prueba y error para la integración de conceptos en el sistema. El usuario comienza especificando un *hipermódulo* eligiendo un conjunto de conceptos y un conjunto tentativo de reglas de integración. HyperJ crea *hiperslices* válidos para esos conceptos y los compone basándose en las reglas que recibe. Luego el *hiperslice* resultante se muestra en pantalla, si el usuario no está conforme puede en ese momento introducir nuevas reglas o modificar las reglas existentes y así continuar este proceso de refinación hasta obtener el modelo deseado.

2.3. TABLA DE RESUMEN CON LAS PRINCIPALES CARACTERÍSTICAS DE LOS LENGUAJES POA DE 2.2.

En la siguiente tabla se encuentran resumidas las principales características de las herramientas orientadas a aspectos descriptas en las secciones anteriores:

Leng. OA	Leng. **Base**	**Tejido**	**Propósito**	**Características salientes**
JPAL	Independiente	Dinámico	General	Meta-Tejedor. Independiente del leng. base. Totalmente dinámico.
AspectS	Squeak	Dinámico	General	Basado en el paradigma de pasaje de mensajes. Todo aspecto es un objeto.
Aspecto++	**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.

Tabla 2.1 Resumen con las principales características de los lenguajes POA

Como lo refleja la tabla anterior las tres decisiones de diseño: lenguaje base, tejido y propósito, son totalmente independientes entre sí para los Lenguajes Orientados a Aspectos (LOA). Esto es, que todas las combinaciones entre las decisiones son teóricamente posibles. Aunque existe la tendencia a implementar el comportamiento del tejedor según la naturaleza del lenguaje base: si el lenguaje base es estático entonces el tejedor es estático y si el lenguaje base es dinámico entonces el tejedor es dinámico.

Hasta donde conocemos nadie ha afrontado la tarea de desarrollar un nuevo lenguaje base. Se han tomado lenguajes bases existentes a los cuales se los restringe en algunos casos, como en los lenguajes de dominio específico, y en otros se los utiliza sin modificaciones.

2.4. ESTUDIO DEL LENGUAJE SELECCIONADO

Se ha seleccionado el lenguaje Aspecto por ser la herramienta con mayor usabilidad, futuro, popularidad y desarrollo.

2.4.1. INTRODUCCÍON A ASPECTOS

AspectJ [20] 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. La versión que analizamos en este trabajo es la versión actual a la fecha AspectJ 1.5.3; es importante hacer esta aclaración porque continuamente surgen nuevas versiones que mejoran la calidad final. 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 figura 2.3.

- 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.

... t

Figura 2.4 AspectJ es una extensión de Java para el manejo de aspectos.

Existen otras metas en la investigación del paradigma POA que AspectJ no tiene intención de abarcar. No es una traducción purista de los conceptos del paradigma, tampoco es un cálculo formal de aspectos ni tampoco representa un esfuerzo agresivo para explorar las posibilidades de un lenguaje orientado a aspectos.

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.

2.4.2. 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 NumComplejof
prívate real parte_imaginaria, parte_real;
NumComplejofreal x, realy){
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 partejmaginaria;}
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 7. Clase número complejo

Entonces lo que establece el código void ingresar_parte_ imaginaria(real x){

this.parte_imaginaria=x; } es que cuando el método *ingres*

ar_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.aumentarparterealprimera: Cuando el método *aumentar_parte_real_primera* es invocado sobre el objeto *ce.*

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 ncl = 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 jmaginaria* 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, y la figura 2.4:

```
Class CoordenadaCompleja {
NumComplejo NumComplejol ,NumComplejo2;

real distancia;

void aumentar_parte_real_primera(real x){

real s = this.distancia;

NumComplejol.aumentar_parte_real(x);

}

}
```

Código 8. Clase coordenada compleja

Figura 2.5 Modelo de puntos de enlace

La ejecución de las tres primeras líneas de la figura 2.4 provoca la creación de los objetos *ce*, *ncl* y *ncl*. 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 *ce*.
2. Un punto de enlace de recepción de llamada a un método, en el cual *ce* recibe la llamada *aumentar_parte_real_primera.*
3. Un punto de enlace de ej ecución 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 *ce* 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 *ncl.*
6. Un punto de enlace de recepción de llamada a un método, en el cual *ncl* 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 *ncl.*
9. Un punto de enlace de recepción de llamada a un método, en el cual *ncl* 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 método *ingresar_parte_real* es invocado sobre el objeto *ncl. ...* y así siguiendo, hasta que finalmente el control retorna por los puntos de enlace 3, 2 y 1.

2.4.3 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 accesoQ:*
get(NumComplejo.parte_real) \\ getfCoordenadaCompleja.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: *Cali :* 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(Patr onde Atributo)

Captura todos los puntos de enlace de referencia a los atributos que coinciden con el patrón correspondiente.

Ejemplo: get(NumComplejo.parteimaginaria) captura las referencias al atributo

parte_imaginaria de la clase *NumComplejo.*

Set: set(Patr onde Atributo)

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_forinales> expone el contexto del corte, y
< Corte> puede ser un corte primitivo o definido por el programador.
Ejemplo:

```
/* Corte 1 */
pointcut aumentar(): call(void aumentar*(...));
/* Corte2 */
pointcut aumentarsolocomplejoQ: aumentarQ && target(NumComplejo);
```

Cortel captura todas las llamadas a un método que comience con la palabra aumentar. *Corte!* utiliza *Cortel* 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 **(prívate** 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 algebraicos para crear nuevos cortes. Los operadores algebraicos soportados por AspectJ son: && ("y"), ||
("o"), !("no") 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>l && <Corte>2 : captura todos los puntos de enlace de Cortel y todos los puntos de enlace de Corte2.

<Corte>l|| <Corte>2 : captura todos los puntos de enlace de Cortel 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 composición es 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 *ApilarJsÁultiple,* 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 PilaQ {
arreglo = new Object[capacidadMaxima];
ultimo=0;
}
public void Apilar (Object elem) {
arreglo [ultimo] = elem; ultimo++;
}
public ObjectDesapilar•(){ Object elem
= arreglofultimo-1]; ultimo—; return
elem;
}
public void Apilar'_Multiple(Object[J arregloElem){ for (int
i=0,i< =arregloElem.length,i++){ this. apilar (arregloElem
[i] );
}
}
public intDar_Capacidad_Maxima(){ return capacidadMaxima;} public
int Cantidad_Elementos() {return ultimo;}
```

puhlic hoolean Pila_Vacia(){return (ultimo==0);}
public booleanPila_Llena(){return (ultimo==capacidadMaxima);}
}

Código 9 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 alapilardeauno captura todas las invocaciones al método *Apilar* **que no están dentro del código de** *Apilar JsÁultiple.* **El corte** *al_apilar**jnultiple*** **captura todas las invocaciones al método** *Apilar JsÁultiple.* 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[23]: en nuestro ejemplo, dentro de todas las invocaciones al método *Apilar* **necesitamos descartar las que se realicen desde el código de** *Apilar JsÁultiple.*

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

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(): (cali (int Pila.Dar_Capacidad_Maxima(..)) ||

cali (int Pila. Cantidad_Elementos(..))) &&

¡this(Pila);

El corte *atributos Jeidos* **captura todas las llamadas a los métodos** *DarJZapacidadJÁaxima* 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 atributosleidos(Pilap): (cali (int Pila.Dar_Capacidad_Maxima(..)) ||

cali (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 como sigue:

pointcut puntos igual es(Punto pl,Punto p2): target(pl) && args(p2)

&& call(boolean igual(Object)); donde *p 1* 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 pl, Pila p2): (target(pl) && call(void Pila.Apilar(..))) || (target(p2) && call(Object Pila.Desapilar(..)));

El corte anterior captura las llamadas al *método Apilar* en un objeto *pl* 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 *pl* 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 Patrón 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, prívate** 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. Cali (java.awt.*) captura todas las clases que comiencen con "java.awt." y no tengan más puntos. Esto es 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 cali (Pila.new() captura todas las llamadas a constructores donde una instancia exclusivamente de la clase *Pila* se crea, call(Pila+.newQ) 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"), || ("o"), !("no") 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

2.4.4. 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 determina cómo éste 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" (Afier): 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 10. Aspectos atreves 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 11. Aspectos atreves de Aviso Before operación Desapilar

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): alapilar(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):
alapilardeauno(p) && alapilarmultiple(p);
```

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

```
afier (Pila p):al_apilar_eficiente(p){
dibujaEstructura dibujante =new dibujaEstructura();
dibuj ante, dibuj arPila(p);
}
```

Código 12. Aspectos atreves 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_Mu Itiple.*

```
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+l);
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+l); 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** (< Parámetro s_formales>) <Forma_terminacion>
<Forma_terminacion>::= **returning** [(<Parametro_formal>) **]** |
throwing [(<Parametro_formal>)] |
L
<Aviso_durante>::= <Nombre_Clase> **around** (<Parámetros_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 como sigue:

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()

2.4.5. 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 el siguiente ejemplo se introduce el atributo booleano *atención* en la clase *Biblioteca* y lo inicializa con el valor true. Como está declarado **prívate** 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.

```
prívate 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 Pila Abstracta;
/* 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 fecha Vacuna;
}
```

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 *deformas de introducción* se define con la siguiente BNF:

```
<Introducciones>::=<IntroduccionesdeAtributos> |
<IntroduccionesdeMetodos> |
<IntroduccionesdeMetodosAbstractos>|
<IntroduccionesdeConstructores>
<IntroduccionesdeAtributos>: :=
[<Modificadores>] <Nombre_Clase> <PatrondeClase>.<Identificador>
[= <expresión>] ;
<IntroduccionesdeMetodos>::=
[<Modificadores>] <Nombre_Clase>
<PatrondeClase>.<Identificador>(< Parámetro s_formales>)
[ throws <Lista_Clases>] {<Cuerpo>}
<IntroduccionesdeMetodosAbstractos>::=
abstract [<Modificadores>] <Nombre_Clase>
<PatrondeClase>.<Identificador>(< Parámetro s_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.

2.4.6. 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 señalización. 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, 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 TrazadoPilaf
pointcut LlamadasaPilaQ :call(* Pila. *(..));
before():LlamadasaPila(){
System.out.println("Entrando al método: "+ thisJointPoint);
}
afterQ :LlamadasaPila () {
System.out.printlnf"Retornando del método: "+ thisJointPoint);
}
}
```

Código 13. 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(Pilap) : callfvoidPila.Apilar(..)) && target (p);
pointcutal_desapilar(Pilap) : callfObject Pila.Desapilarf..))
&& target(p);
beforefPila p) :al_apilar(p) {
if(p.Pila_Llena()){ throw newExcepcionPilaLlenaQ; }
}
beforefPila p) :al_desapilar(p) {
if(p.Pila_Vacia()){ throw newExcepcionPilaVaciaQ; }
}
}
```

Código 14. 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 TrazaSimplef
abstract pointcut PuntosTrazaQ;
before():PuntosTraza(){
System.out.println("Entrando "+ thisJoinPoint);
}
afterQ: Punto sTraza(){
System.out.printlnfRetornando "+ thisJoinPoint);

aspect TrazaPila extends TrazaSimplef
pointcut PuntosTraza():caU(* Pila. *(..));
}
```

Código 15. 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 {

 beforefPila p) :Cortel (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 Atienen 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> <PatrondeClasé>.<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>] <PatrondeClasé>.new(<Parametros_formales>)

[throws <Lista_Clases>] {<Cuerpó>}

<Avisó>::=<Tipo_avisó> : <Corté> {<Cuerpó>}

<Tipo_aviso>::= <Aviso_antes> \ <Aviso_despues> \ <Aviso_duranté>

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

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

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

***throwing** [(<Parametro_formal>)] \ L*

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

[throws <Lista_Clases>]

BNF 6 Definición de aspectos

2.4.7. EVALUACIÓN

En esta descripción de AspectJ no han sido tratados detalles de implementación para permitir una mejor comprensión y aprendizaje del lenguaje.

Se han priorizado la simpleza y forma general del lenguaje por sobre detalles de implementación que oscurecen el entendimiento. Además, no es necesario conocer a fondo estos detalles para iniciarse en la programación de AspectJ. En sí, se describe de AspectJ aquellos conceptos necesarios para poder programar con aspectos por primera vez y/o 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.

2.5. HERRAMIENTA MYECLIPSE 6.0.

2.5.1 INTRODUCCIÓN

MyEclipse Enterprise Workbench es una extensión que añadimos a Eclipse 3.3 para proporcionarle la posibilidad de realizar desarrollos, distribuciones e integraciones de aplicaciones J2EE.

Eclipse es una plataforma de software 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 integrados de desarrollo (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 Visual Age. 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.

2.5.2. 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 un 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 basada en SWT.

El entorno integrado de desarrollo (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 seria Gestión de la configuración. Se provee soporte para Java y CVS en el SDK de Eclipse. Y no tiene porque 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.

2.5.3. CARACTERÍSTICAS

La versión actual de **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 Subversión, vía Subclipse.
- Integración con Hibernate, vía Hibernate Tools.

2.5.4. PROYECTOS ECLIPSE

Eclipse está compuesto de muchos proyectos diferentes. Algunos proyectos se mencionan a continuación.

- **El proyecto Eclipse** per se que incluye la Plataforma Eclipse, Plataforma Eclipse de Cliente Enriquecido (RCP) y las herramientas de desarrollo de Java (JDT).
- **Plataforma de herramientas para pruebas y desempeño** (de sus siglas en Inglés TPTP) que provee una plataforma que permite a desarrolladores de software construir herramientas de pruebas y desempeño, como son Depuradores, profilers y aplicaciones Benchmark.
- **Proyecto Plataforma de Herramientas Web** (WTP) extiende la plataforma Eclipse con herramientas para desarrollar aplicaciones Web en Ja va EE. Está compuesta de: Editores de fuentes para HTML. JavaScript CSS. JSP. SQL. XML. DTD. XSD y

WSDL; Editores gráficos para XSD y WSDL; proyectos de naturaleza Java EE, constructores y modelos y un navegador de Java EE; un explorador y asistente para servicios Web y una herramienta de pruebas WS-I; herramientas para acceso a base de datos, filtrado y modelos; y herramientas para manejo de servidores de pruebas unitarias.

- Proyecto de herramientas para **inteligencia empresarial** y generación de reportes (BIRT), un sistema de reporteo Código abierto basado en Eclipse para aplicaciones Web, especialmente aquellas basadas en Java EE.
- Proyecto de Edición Visual (VE) una plataforma para crear constructores GUI para Eclipse
- **Plataforma de Modelado Eclipse** (EMF) una plataforma de modelado y generación de código para construir herramientas y otras aplicaciones basadas en un modelo de datos estructurado, desde una especificación de modelo descrita en XMI.
- **Herramientas de Modelado Generativo** (GMT) un grupo de herramientas para modelado por ejemplo para ejecutar transformaciones de modelo QVT.
- **Plataforma de Editor Gráfico** (GEF) permite a los desarrolladores tomar el modelo de una aplicación existente y fácilmente crear un editor de gráficos ricos.
- **UML2** una implementación de UML 2.0 metamodel para la plataforma Eclipse diseñada para soportar el desarrollo de herramientas de modelado.
- **Plataforma de comunicaciones de Eclipse Communication Framework** (ECF) habilita la creación de aplicaciones de comunicaciones en la plataforma de Eclipse.
- **Proyecto Plataforma de herramientas de Datos** (DTP)
- **Plataforma de Herramientas Paralelas** (PTP) entrega una plataforma de herramientas paralelas portables, escalables, basadas en estándares que habilita la integración de herramientas específicamente desarrolladas para computadoras con arquitectura paralela.
- **Plataforma de Cliente Rico incluido** (eRCP) la intención es extender la plataforma de Cliente Rico (de las siglas en Inglés RCP) para dispositivos incluidos. eRCP es en general un grupo de componentes que son subgrupos de los componentes RCP. Básicamente habilita el mismo modelo de aplicaciones usado en maquinas de escritorio para ser usados en dispositivos.

- **Plataforma de Desarrollo de Software para Dispositivos** (DSDP) es un proyecto de desarrollo de software colaborativo de código abierto dedicado a proveer una plataforma extendible basada en enstándares para cubrir un amplio rango de necesidades en el área del desarrollo de software para dispositivos usando la plataforma de Eclipse.

2.5.5. PROYECTOS IDE EN LENGUAJES

- **AspectJ** es una extensión del lenguaje Java orientado 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.
- **Subproyecto 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 implementan un IDE de Java, soportando el desarrollo de cualquier aplicación Java, incluyendo los plug-ins de Eclipse.
- **Photran** (photran) es un IDE completamente funcional para Fortran con soporte para Refactorización.
- **PHP Development Tools** trabaja para proveer un IDE completamente funcional para PHP para la plataforma Eclipse.
- **Wolfram *Workbench*** es un IDE basado en Eclipse (también disponible como plugin para Eclipse) para el lenguaje *Mathematica.*
- **PyDev** un IDE completamente funcional para pvthon con soporte para Refactorización, y depurador gráfico.

CAPITULO III

DESARROLLO DEL PROTOTIPO SOFTWARE APLICANDO ASPECTOS

3.5. GENERALIDADES

Para mostrar la utilización de Aspectos en un sistema real, se desarrollará un prototipo software, el cual consta de un Sistema de Facturación denominado "Sistema de Facturación CRC". Para el desarrollo inicial se utiliza UML *{UnifiedModeling Languagé)* hasta obtener el Diagrama de Clases, posteriormente para establecer el aporte de la Programación Orientada a Aspectos (POA), se toma como base el Diagrama de Clases desarrollado con UML.

3.6. DESARROLLO DEL PROTOTIPO

3.2.5. ANÁLISIS DEL SISTEMA

En este punto se analiza los requerimientos y necesidades que el usuario requiere para obtener un producto de calidad, para la obtención del sistema se va a utilizar el documento ERS **(Especificación de Requisitos del Software)** basados en el estándar **IEEE 830.** Este proyecto permite definir las especificaciones de requisitos de software (ERS) para el prototipo de un sistema de facturación, que estará enfocado a la empresa Bordados CRC en la ciudad de Latacunga con la aplicación de la programación orientada a aspectos.

Propósito

El documento trata de especificar de manera clara todas las funcionalidades del sistema, el cual deberá satisfacer las necesidades del usuario llegando así a una automatización de la realización de facturas.

Ámbito del Sistema

El motivo principal para la creación del Sistema de Facturación CRC, se basa que en la actualidad la empresa de bordados lleva un control manual de sus facturas sin tener un resultado de cuentas eficientes y con índices de pérdidas de dinero y tiempo considerables. Esto ayudará ha tener una información rápida, segura y que ayude a la gerencia en la toma de decisiones para lograr ser más productivos.

Definiciones Acrónimos Abreviaturas

Definiciones:

Administrador. Persona encargada en gestionar las funciones del sistema

Usuario (s). Persona encargada de realizar las transacciones presentadas por el sistema, con su respectiva seguridad.

Acrónimos:

ERS. Especificatión Requirements Software

Referencias:

"IEEE Recommended Practice for Software Requirements Specifícatíon ANSI/IEEE std 830 1998".

3.2.6. FUNCIONES **DEL** SISTEMA

En términos generales el sistema deberá proporcionar soporte a las siguientes tareas de gestión:

- Gestión de acceso al sistema.

La gestión de acceso al sistema permitirá el ingreso al manejo de Sistema de Facturación CRC, solicitará una clave de seguridad, la misma que será verificada y validada para el acceso al sistema y poder realizar o trabajar en cualquiera de las otras gestiones para la facturación.

- Gestión de Pedidos.

La gestión de pedidos permitirá realizar altas y consultas de pedidos para mantener actualizada la información del negocio.

- Gestión de Ventas.

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

Al realizar la venta automáticamente afecta al inventario para lo cual se disminuyen los productos del stock.

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

- Gestión de Productos.

El inventario se debe gestionar en forma conjunta y automática según los pedidos que nosotros realizamos y las ventas que realizamos.

Lo primero es ingresar todos los datos de los productos terminados y materia prima, al Sistema de Facturación CRC.

El Sistema de Facturación CRC nos permitirá ingresar a observar e imprimir un listado general de los productos existentes.

- Gestión de Clientes.

Los datos del cliente se podrán ingresar en el servidor manualmente. Se ingresará todos los datos personales del cliente, también se podrá actualizar y consultar los datos del cliente en forma individual.

Características de los usuarios

El Sistema de Facturación CRC 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) Realizar consulta individual o general del cliente.

Requisito (05) Actualizar datos de clientes

Gestión de Productos

Requisito (06) Ingresar los datos de los productos.

Requisito (07) Presentar un listado de los productos generales.

Requisito (08) Modificar un producto.

Requisito (09) Eliminar un producto.

Gestión de pedidos

Requisito (10) Ingresar pedido.

Requisito (11) Consultar información de pedidos

Gestión de Ventas

Requisito (12) Realizar ventas

Requisito (13) Generar facturas

Requisito (14) Imprimir factura

Requisitos de desarrollo

El ciclo de vida escogido es el interactivo incremental, con la ayuda del UML (lenguaje de modelaje unificado).

Requisitos tecnológicos

- Lenguaje de programación AspectJ.
- Integrador de lenguajes de programación MyEclipse 6.0.
- Base de datos MySQL WAMSERVER 2.0.
- Sistema Operativo Windows XP.
- JAVA Machine Virtual.

Requisitos de Hardware

La aplicación se ejecutara sobre un PC con las siguientes características como mínimo.

- Procesador Pentium III
- Memoria 512MB de RAM
- Disco duro 5 GB espacio libre.
- Unidades CD-ROM O DVD-ROM

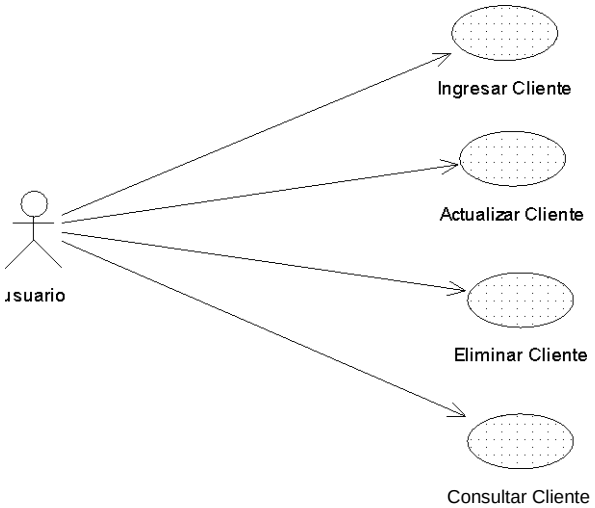
3.2.7. DIAGRAMA DE CASOS DE USO

GESTIÓN DE ACCESO AL SISTEMA

DEFINICIÓN DE CASOS DE USO ALTO NIVEL

NOMBRE DEL CASO DE USO	Ingresar clave de acceso
ACTOR	Usuario
TIPO	Primario
DESCRIPCIÓN	Este caso de uso inicia cuando el usuario selecciona la operación Ingresar clave de acceso, el sistema permite el ingreso, el usuario ingresa al sistema

GESTIÓN CLIENTES



DEFINICIÓN DE CASOS DE USO ALTO NIVEL

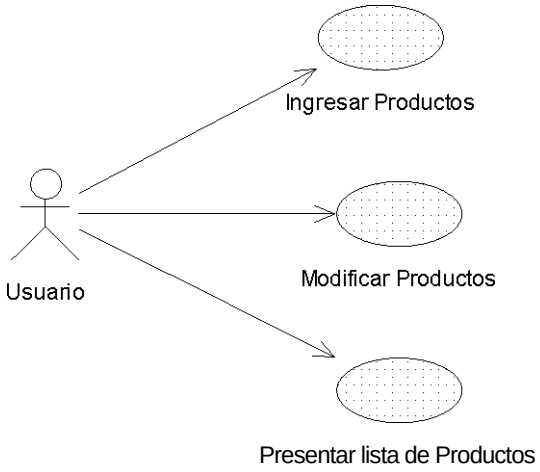
NOMBRE DEL CASO DE USO	Ingresar Cliente
ACTOR	Usuario
TIPO	Primario
DESCRIPCIÓN	Este caso de uso inicia cuando el usuario selecciona la operación Ingresar cliente, el sistema presenta el formulario de ingreso, el usuario ingresa datos

NOMBRE DEL CASO DE USO	Eliminar cliente
ACTOR	Usuario
TIPO	Primario
DESCRIPCIÓN	Este caso de uso inicia cuando el usuario selecciona la operación eliminar cliente, el sistema pide la cédula o Ruc del cliente y emite un mensaje de verificación, el usuario confirma.

NOMBRE DEL CASO DE USO	Consultar cliente
ACTOR	Usuario
TIPO	Primario
DESCRIPCIÓN	Este caso de uso inicia cuando el usuario selecciona la operación consultar cliente, el sistema pide la cédula o Ruc del cliente y presenta el formulario con los datos.

NOMBRE DEL CASO DE USO	Actualizar cliente
ACTOR	Usuario
TIPO	Primario
DESCRIPCIÓN	Este caso de uso inicia cuando el usuario selecciona la operación actualizar cliente, el sistema pide la cédula o Ruc del cliente y presenta el formulario con los datos, el usuario actualiza los datos.

GESTIÓN DE PRODUCTOS



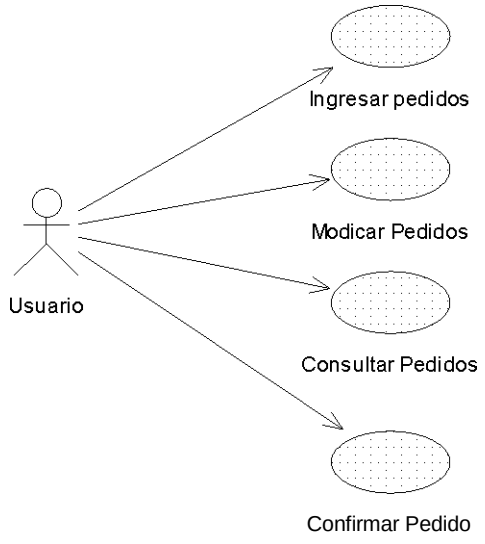
DEFINICIÓN DE CASOS DE USO ALTO NIVEL

NOMBRE DEL CASO DE USO	Ingresar productos
ACTOR	Usuario
TIPO	Primario
DESCRIPCIÓN	Este caso de uso inicia cuando el usuario selecciona la operación ingresar productos, el sistema presenta el formulario, el usuario ingresa los datos.

NOMBRE DEL CASO DE USO	Modificar producto
ACTOR	Usuario
TIPO	Primario
DESCRIPCIÓN	Este caso de uso inicia cuando el usuario selecciona la operación modificar productos, el sistema pide el código del producto y presenta el formulario con los datos.

NOMBRE DEL CASO DE USO	Presentar listado de productos
ACTOR	Usuario
TIPO	Primario
DESCRIPCIÓN	Este caso de uso inicia cuando el usuario selecciona la operación presentar listado de productos, el sistema pide el código del producto y presenta el formulario con los datos.

GESTIÓN DE PEDIDOS



DEFINICIÓN DE CASOS DE USO ALTO NIVEL

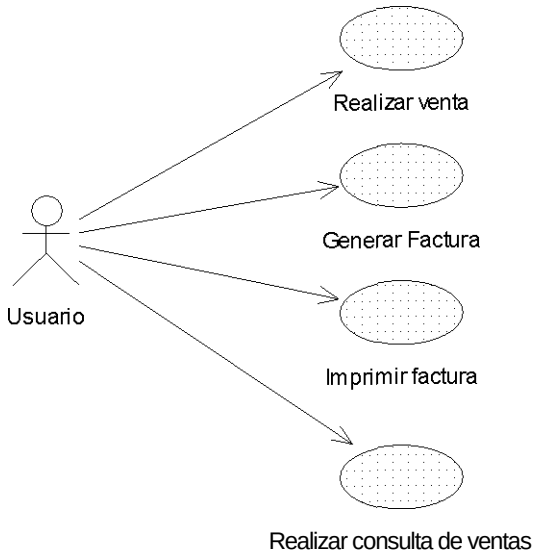
NOMBRE DEL CASO DE USO	Ingresar pedidos
ACTOR	Usuario
TIPO	Primario
DESCRIPCIÓN	Este caso de uso inicia cuando el usuario selecciona la operación ingresar pedidos, el usuario ingresa cédula o Ruc, el sistema muestra la información del cliente, el usuario selecciona productos, el sistema verifica stock.

NOMBRE DEL CASO DE USO	Modificar pedidos
ACTOR	Usuario
TIPO	Primario
DESCRIPCIÓN	Este caso de uso inicia cuando el usuario selecciona la operación modificar pedido, el sistema pide el número de pedido y presenta el formulario con los datos, el usuario procede a verificar los datos.

NOMBRE DEL CASO DE USO	Realizar consultas
ACTOR	Usuario
TIPO	Primario
DESCRIPCIÓN	Este caso de uso inicia cuando el usuario selecciona la operación presentar el listado de productos, el sistema pide el código del producto y presenta el formulario con los

NOMBRE DEL CASO DE USO	Confirmar pedido
ACTOR	Usuario
TIPO	Primario
DESCRIPCIÓN	Este caso de uso inicia cuando el usuario selecciona la operación confirmar pedido, el sistema pide el número de pedido y presenta el formulario con los datos, el usuario confirma el pedido.

GESTIÓN DE VENTAS



DEFINICIÓN DE CASOS DE USO ALTO NIVEL

NOMBRE DEL CASO DE USO	Realizar venta
ACTOR	Usuario
TIPO	Primario
DESCRIPCIÓN	Este caso de uso inicia cuando el usuario selecciona la operación realizar pedido, el sistema muestra el formulario de ventas el mismo que está relacionado con un pedido, el usuario realiza la venta.

NOMBRE DEL CASO DE USO	Generar factura
ACTOR	Usuario
TIPO	Primario
DESCRIPCIÓN	Este caso de uso inicia cuando el usuario selecciona la operación generar nota de venta, el sistema guarda los datos de la venta, genera una factura.

NOMBRE DEL CASO DE USO	Imprimir factura
ACTOR	Usuario
TIPO	Primario
DESCRIPCIÓN	Este caso de uso inicia cuando el usuario selecciona la operación imprimir nota de venta, el sistema guarda los datos de la venta, genera una factura e imprime la misma.

NOMBRE DEL CASO DE USO	Realizar consulta de venta
ACTOR	Usuario
TIPO	Primario
DESCRIPCIÓN	Este caso de uso inicia cuando el usuario selecciona la operación realizar consulta de ventas, el sistema pide el número de venta y muestra los datos en pantalla.

GESTIÓN DE ACCESO AL SISTEMA

PARA EL USUARIO

DEFINICIÓN DE CASOS DE USO EXPANDIDOS

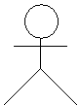
NOMBRE DEL CASO DE USO	Ingresar Clave de acceso
PROPÓSITO	Permitir a un usuario ingresar al sistema.
REFERENCIA	Requisito 01
ACTOR	Usuario
TIPO	Primario Real
VISIÓN GENERAL	Este caso de uso inicia cuando el usuario selecciona la operación Ingresar clave de acceso, el sistema verifica la clave y permite el acceso al usuario al sistema

CURSO TÍPICO DEL EVENTO

ACTOR	SISTEMA
1. El usuario selecciona la operación ingresar clave de acceso	2. El sistema verifica y permite el acceso

CASOS ALTERNATIVOS

2* Si no es válida la clave termina el caso de uso.



Sistema

Usuario Seleccionar la operación ingresar clave de acceso

CONTRATO DE OPERACIONES

NOMBRE DEL CONTRATO DE OPERACIÓN	Seleccionar operación ingresar clave de acceso
PROPÓSITO	Permitir al usuario ingresar al sistema.
TIPO	Sistema
SALIDA	Acceso al sistema
PRECONDICION	
POST CONDICIÓN	
EXCEPCIÓN	Que no sea valida la clave

DIAGRAMAS INTERACTIVOS

DIAGRAMA DE COLABORACIÓN

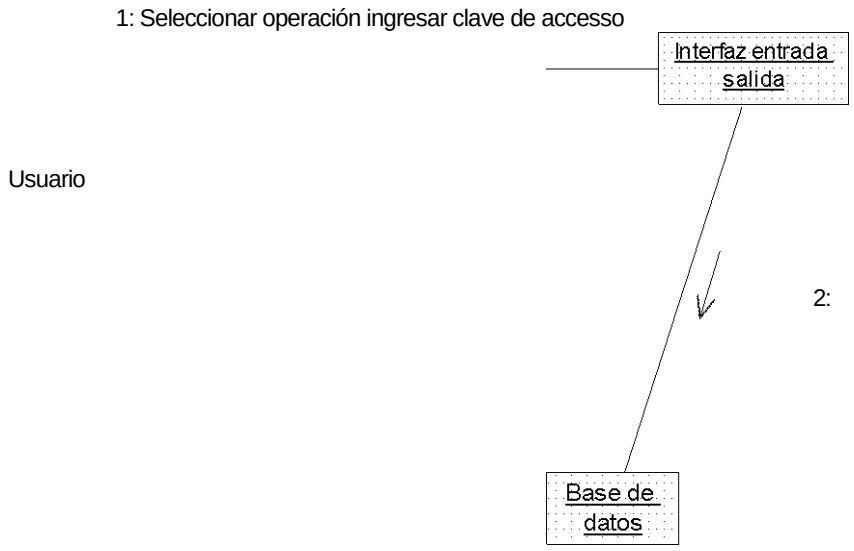
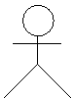


DIAGRAMA DE SECUENCIA



Interfaz

Base de datos

Usuario

Seleccionar opción ingresar clave

->

Verificar clave

GESTIÓN DE CLIENTES

PARA EL USUARIO

DEFINICIÓN DE CASOS DE USO EXPANDIDOS

NOMBRE DEL CASO DE USO	Ingresar cliente
PROPÓSITO	Permitir a un usuario ingresar datos de un nuevo cliente.
REFERENCIA	Requisito 02
ACTOR	Usuario
TIPO	Primario Real
VISIÓN GENERAL	Este caso de uso inicia cuando el usuario selecciona la operación Ingresar cliente, el sistema presenta el formulario de ingreso, el usuario ingresa datos

CURSO TÍPICO DEL EVENTO

ACTOR	SISTEMA
1. El usuario selecciona la operación ingresar cliente.	2. El sistema presenta el formulario del cliente
3. El usuario ingresa los datos.	4. El sistema guarda información del cliente.

CASOS ALTERNATIVOS

4* Si el cliente ya existe termina el caso de uso.

Sistema

: Usuario

I Seleccionar la operación ingresar cliente

T

| Ingresar datos

CONTRATO DE OPERACIONES

NOMBRE DEL CONTRATO DE OPERACIÓN	Seleccionar operación Ingresar cliente
PROPOSITO	Presentar formulario de clientes
TIPO	Sistema
SALIDA	Formulario de clientes
PRECONDICION	Exista formulario de clientes
POST CONDICIÓN	
EXCEPCIÓN	No existe formulario de clientes

DIAGRAMAS INTERACTIVOS

DIAGRAMA DE COLABORACIÓN

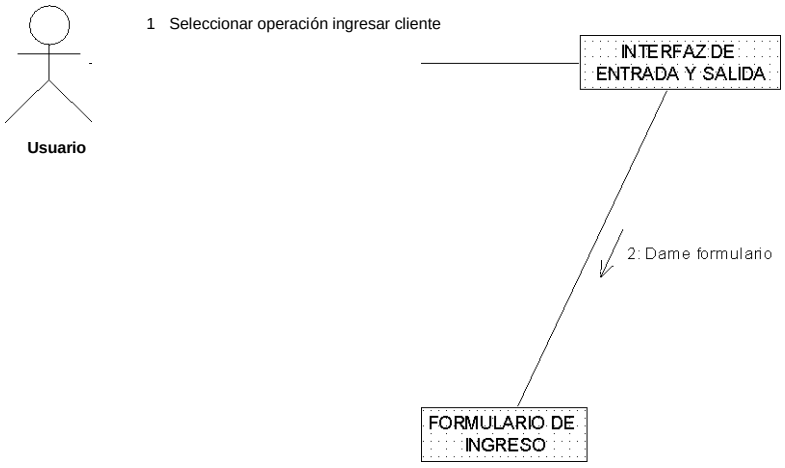
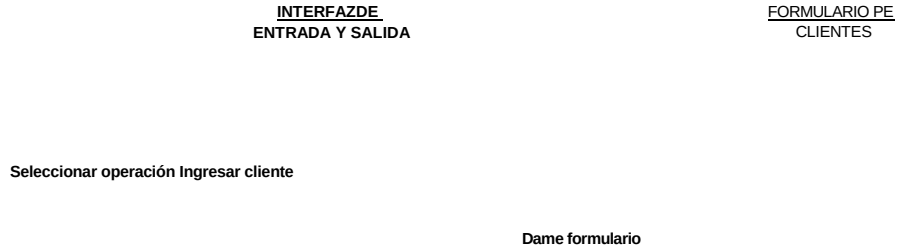


DIAGRAMA DE SECUENCIA



NOMBRE DEL CONTRATO DE OPERACIÓN	Almacenar datos del cliente
PROPOSITO	Ingresar nuevo cliente
TIPO	Sistema
SALIDA	Datos del cliente almacenados
PRECONDICION	Cliente no existe
POST CONDICIÓN	
EXCEPCIÓN	Los datos del cliente no fueron almacenados

DIAGRAMA DE COLABORACIÓN

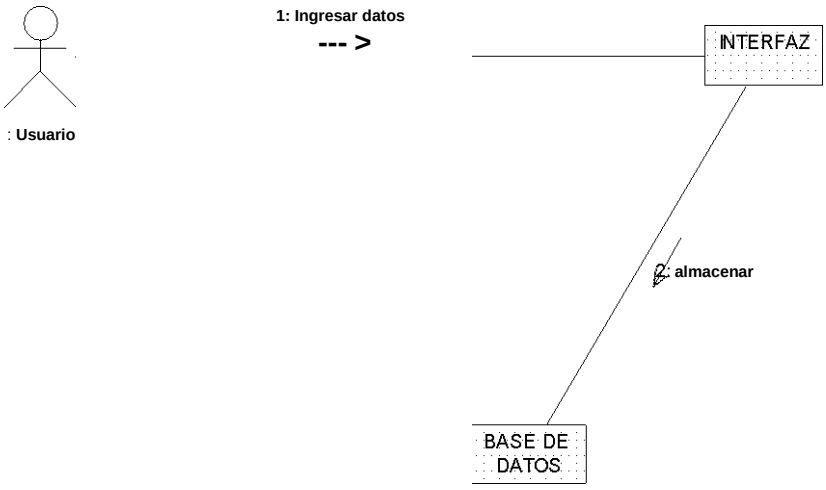


DIAGRAMA DE SECUENCIA

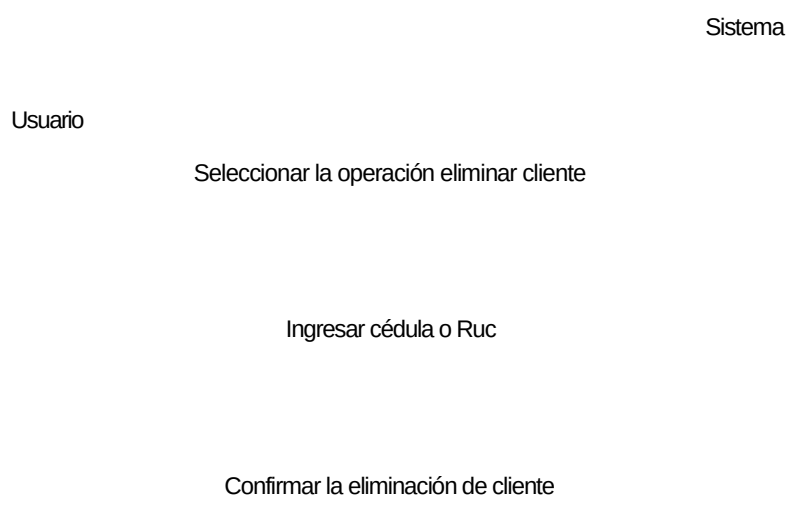
NOMBRE DEL CASO DE USO	Eliminar cliente
PROPÓSITO	Permitir a un usuario eliminar un cliente.
REFERENCIA	Requisito 03
ACTOR	Usuario
TIPO	Primario Real
VISIÓN GENERAL	Este caso de uso inicia cuando el usuario selecciona la operación eliminar cliente, el sistema pide la cédula o Ruc del cliente y emite un mensaje de verificación, el usuario confirma.

CURSO TÍPICO DEL EVENTO

ACTOR	SISTEMA
1. El usuario selecciona la operación eliminar cliente.	2. El sistema pide cédula o Ruc del cliente.
3. El usuario ingresa la cédula o Ruc.	4. El sistema verifica si existe el cliente y no tiene pedidos o compras y devuelve un mensaje.
5. El usuario confirma	6. El sistema elimina al cliente

CASOS ALTERNATIVOS

4* Si el cliente ya existe termina el caso de uso.

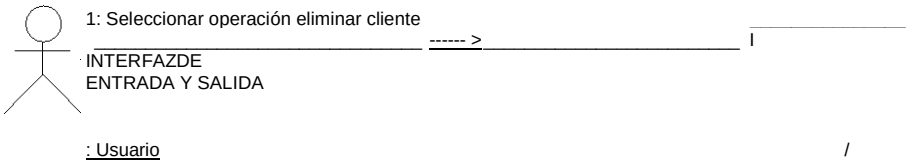


CONTRATO DE OPERACIONES

NOMBRE DEL CONTRATO DE OPERACIÓN	Seleccionar operación eliminar cliente
PROPÓSITO	Permitir a un usuario eliminar un cliente.
TIPO	Sistema
SALIDA	Datos del cliente
PRECONDICION	Exista cliente
POST CONDICIÓN	Cliente no tenga pedidos o compras
EXCEPCIÓN	No existe el cliente

DIAGRAMAS INTERACTIVOS

DIAGRAMA DE COLABORACIÓN



FORMULARIO
ELIMINAR

DIAGRAMA DE SECUENCIA

INTERFAZ DE ENTRADA Y SALIDA

FORMULARIO DE

Seleccionar operación eliminar cliente

Dame caja de texto

NOMBRE DEL CONTRATO DE OPERACIÓN	Confirmar eliminación del cliente
PROPOSITO	Eliminar cliente
TIPO	Sistema
SALIDA	Cliente eliminado
PRECONDICION	Cliente no tenga pedidos ni compras
POST CONDICIÓN	
EXCEPCIÓN	Cliente tenga pedido o compra

DIAGRAMA DE COLABORACIÓN

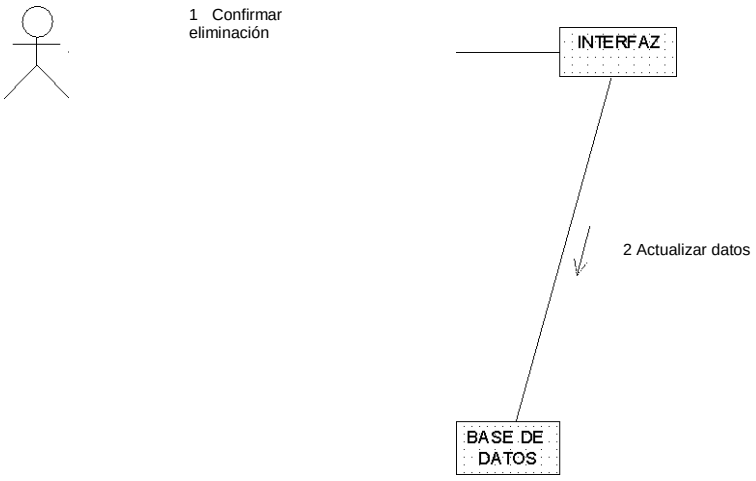
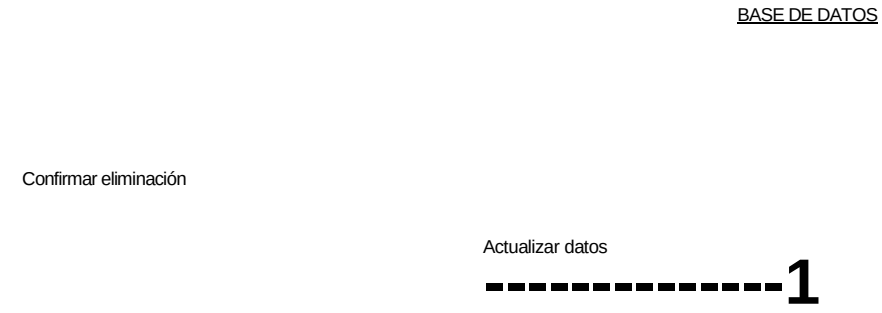


DIAGRAMA DE SECUENCIA



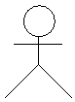
NOMBRE DEL CASO DE USO	Consultar clientes
PROPÓSITO	Permitir a un usuario obtener información de clientes
REFERENCIA	Requisito 04
ACTOR	Usuario
TIPO	Primario Real
VISIÓN GENERAL	Este caso de uso inicia cuando el usuario selecciona la operación consultar cliente, el sistema pide la cédula o Ruc del cliente y presenta el formulario con los datos.

CURSO TÍPICO DEL EVENTO

ACTOR	SISTEMA
1. El usuario selecciona la operación consultar cliente.	2. El sistema devuelve datos del cliente

CASOS ALTERNATIVOS

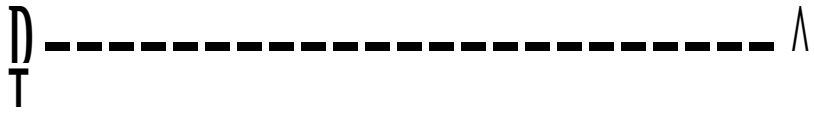
2* Si no existe clientes termina el caso de uso.



Sistema

: Usuario

Seleccionar operación consultar clientes



CONTRATO DE OPERACIONES

NOMBRE DEL CONTRATO DE OPERACIÓN	Seleccionar operación consultar cliente
PROPÓSITO	Permitir a un usuario obtener información de los clientes.
TIPO	Sistema
SALIDA	Datos del cliente
PRECONDICION	Exista cliente
POST CONDICIÓN	
EXCEPCIÓN	No existe el cliente

DIAGRAMAS INTERACTIVOS

DIAGRAMA DE COLABORACIÓN

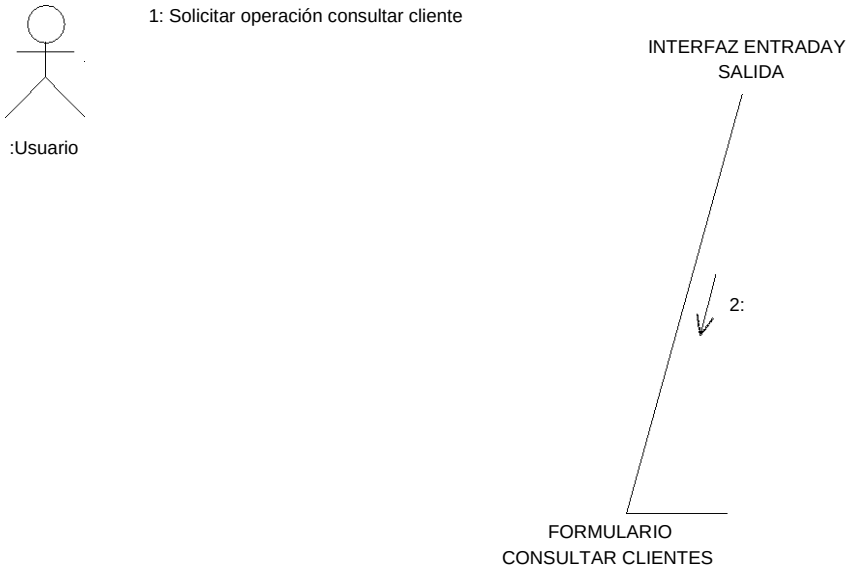
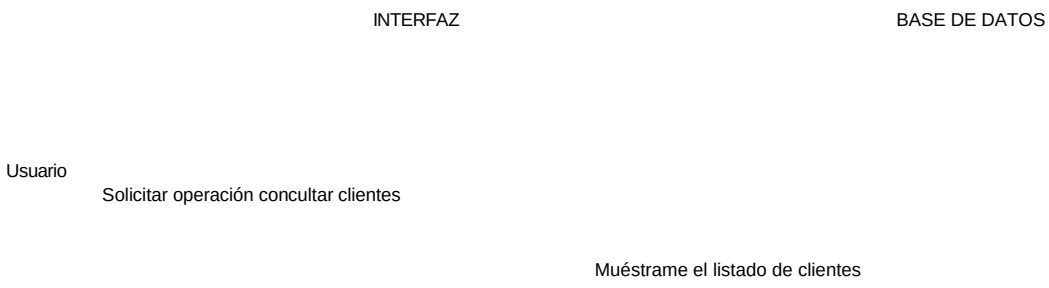


DIAGRAMA DE SECUENCIA



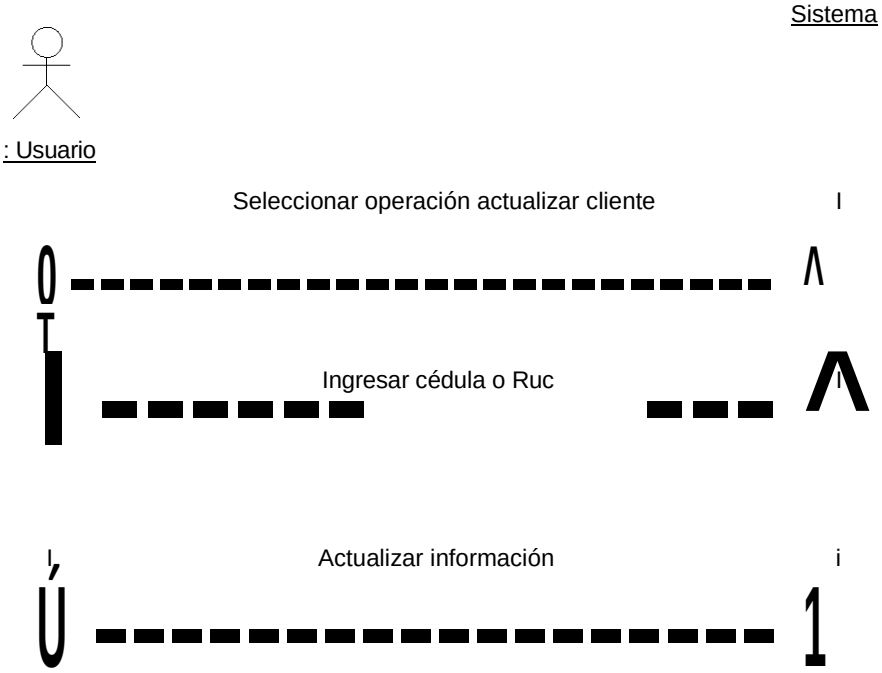
NOMBRE DEL CASO DE USO	Actualizar clientes
PROPÓSITO	Permitir a un usuario actualizar datos de un cliente
REFERENCIA	Requisito 05
ACTOR	Usuario
TIPO	Primario Real
VISIÓN GENERAL	Este caso de uso inicia cuando el usuario selecciona la operación actualizar cliente, el sistema pide la cédula o Ruc del cliente y presenta el formulario con los datos, el usuario actualiza los datos.

CURSO TÍPICO DEL EVENTO

ACTOR	SISTEMA
1. El usuario selecciona la operación eactualizar cliente.	2. El sistema pide cédula o Ruc del cliente.
3. El usuario ingresa la cédula o Ruc.	4. El sistema verifica si existe el cliente y muestra el formulario de clientes.
5. El usuario actualiza la información del cliente	6. El sistema graba la nueva información del cliente

CASOS ALTERNATIVOS

4* Si no existe el cliente termina el caso de uso.



CONTRATO DE OPERACIONES

NOMBRE DEL CONTRATO DE OPERACIÓN	Seleccionar operación actualizar cliente
PROPÓSITO	Permitir a un usuario actualizar datos de un cliente
TIPO	Sistema
SALIDA	Formulario clientes
PRECONDICION	Exista el cliente
POST CONDICIÓN	
EXCEPCIÓN	No existe el cliente

DIAGRAMAS INTERACTIVOS

DIAGRAMA DE COLABORACIÓN

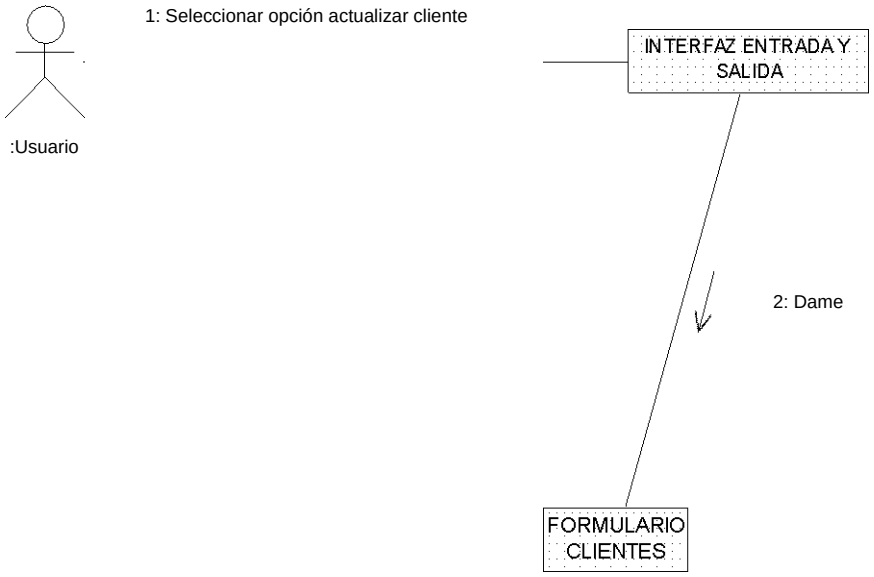
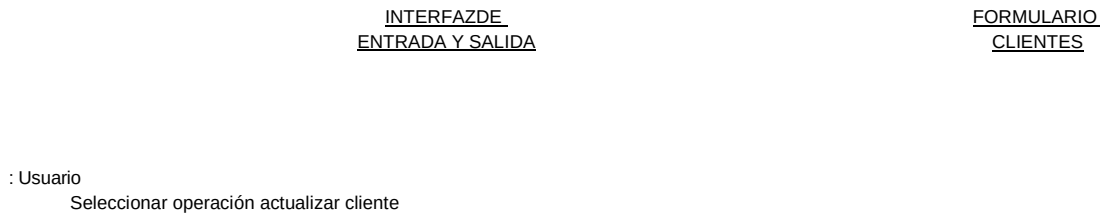


DIAGRAMA DE SECUENCIA



NOMBRE DEL CONTRATO DE OPERACIÓN	Ingresar cédula o Ruc
PROPOSITO	Actualizar datos del cliente
TIPO	Sistema
SALIDA	Datos del cliente actualizado
PRECONDICION	Exista el cliente
POST CONDICIÓN	
EXCEPCIÓN	No existe el cliente

DIAGRAMA DE COLABORACIÓN

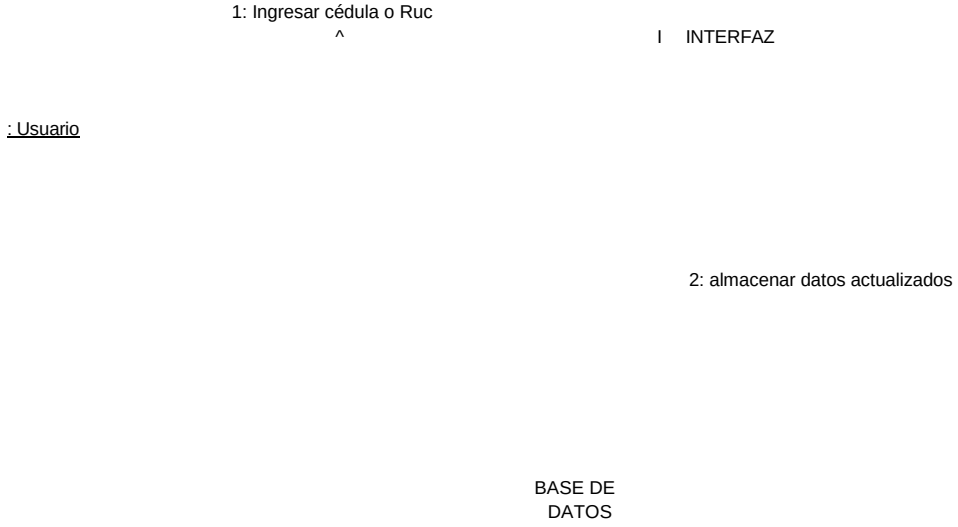
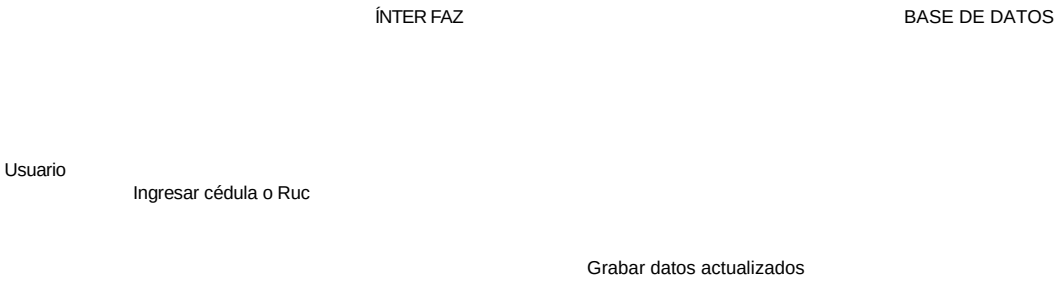


DIAGRAMA DE SECUENCIA



GESTIÓN DE PRODUCTOS

PARA EL USUARIO

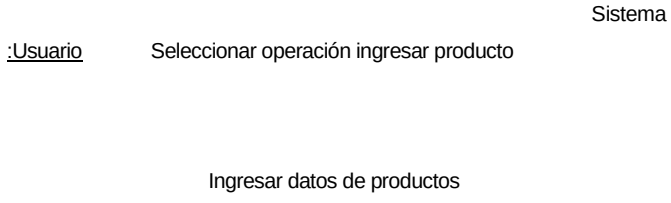
NOMBRE DEL CASO DE USO	Ingresar producto
PROPÓSITO	Permitir a un usuario ingresar un nuevo producto.
REFERENCIA	Requisito 06
ACTOR	Usuario
TIPO	Primario Real
VISIÓN GENERAL	Este caso de uso inicia cuando el usuario selecciona la operación Ingresar cliente, el sistema presenta el formulario de ingreso, el usuario ingresa los datos

CURSO TÍPICO DEL EVENTO

ACTOR	SISTEMA
1. El usuario selecciona la operación ingresar cliente.	2. El sistema presenta el formulario de productos
3. El usuario ingresa los datos.	4. El sistema guarda la nueva información.

CASOS ALTERNATIVOS

4* Si el cliente ya existe termina el caso de uso.



CONTRATO DE OPERACIONES

NOMBRE DEL CONTRATO DE OPERACIÓN	Seleccionar operación Ingresar producto
PROPÓSITO	Permitir al usuario ingresar un nuevo producto.
TIPO	Sistema
SALIDA	Formulario productos
PRECONDICION	Exista formulario de productos
POST CONDICIÓN	
EXCEPCIÓN	No existe formulario de clientes

DIAGRAMAS INTERACTIVOS

DIAGRAMA DE COLABORACIÓN

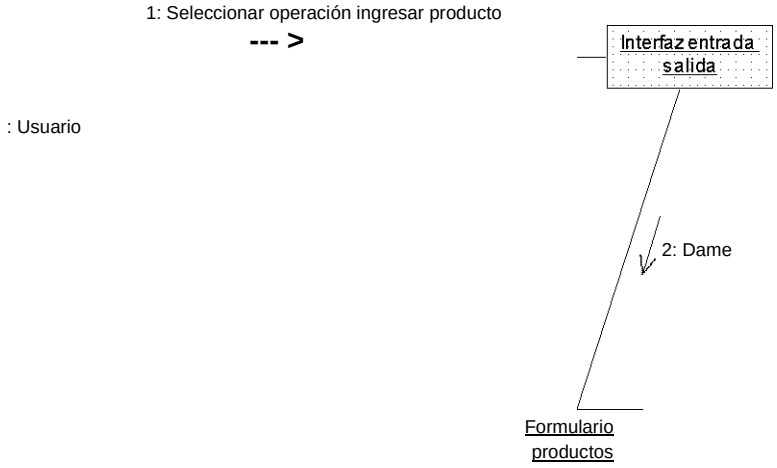
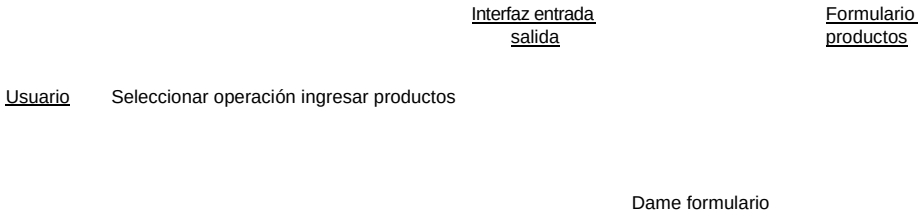


DIAGRAMA DE SECUENCIA



NOMBRE DEL CONTRATO DE OPERACIÓN	Almacenar datos del producto
PROPOSITO	Ingresar nuevo producto
TIPO	Sistema
SALIDA	Datos del producto almacenados
PRECONDICION	Producto no existe
POST CONDICIÓN	
EXCEPCIÓN	Los datos del producto no fueron almacenados

DIAGRAMA DE COLABORACIÓN

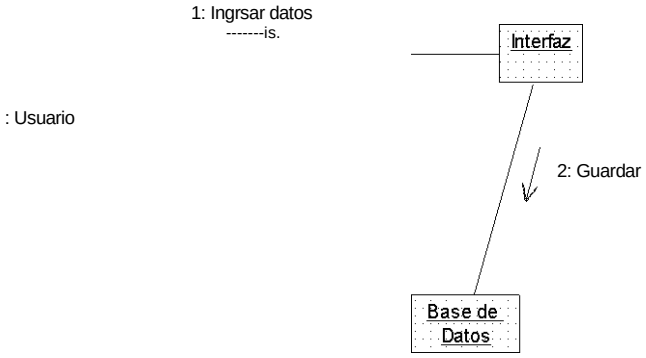


DIAGRAMA DE SECUENCIA

NOMBRE DEL CASO DE USO	Presentar listado de productos
PROPÓSITO	Permitir a un usuario obtener un listado de productos
REFERENCIA	Requisito 07
ACTOR	Usuario
TIPO	Primario Real
VISIÓN GENERAL	Este caso de uso inicia cuando el usuario selecciona la operación presentar listado de productos, el sistema pide el código del producto y presenta el formulario con los datos el usuario confirma.

CURSO TÍPICO DEL EVENTO

ACTOR	SISTEMA
1. El usuario selecciona la operación presentar listado de productos	2. El sistema presenta datos de productos.

CASOS ALTERNATIVOS

Sistema

Usuario Seleccionar operación presentar listado de productos

CONTRATO DE OPERACIONES

NOMBRE DEL CONTRATO DE OPERACIÓN	Seleccionar operación presentar listado productos
PROPÓSITO	Permitir a un usuario obtener un listado de productos.
TIPO	Sistema
SALIDA	Datos de productos
PRECONDICION	Exista productos
POST CONDICIÓN	
EXCEPCIÓN	

DIAGRAMAS INTERACTIVOS

DIAGRAMA DE COLABORACIÓN

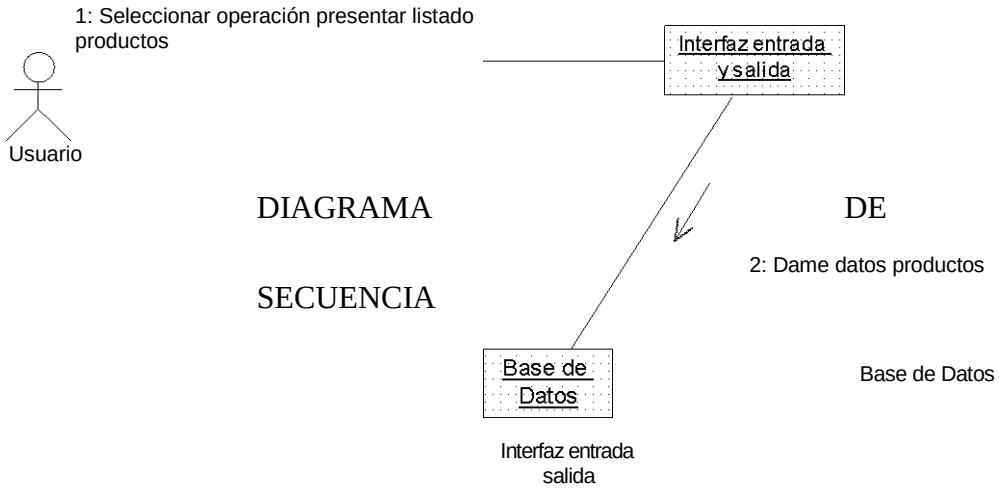


DIAGRAMA DE SECUENCIA

UsiiariSeleccionar operación presentar listado de productos

Dame datos de productos

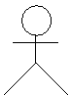
NOMBRE DEL CASO DE USO	Modificar producto
PROPÓSITO	Permitir a un usuario modificar los datos de un producto
REFERENCIA	Requisito 08
ACTOR	Usuario
TIPO	Primario Real
VISIÓN GENERAL	Este caso de uso inicia cuando el usuario selecciona la operación modificar producto y presenta el formulario con lo datos.

CURSO TÍPICO DEL EVENTO

ACTOR	SISTEMA
1. El usuario selecciona la operación modificar producto	2. El sistema pide código del producto.
3. El usuario ingresa el código del producto.	4. El sistema presenta el formulario de productos con los datos.
5. El usuario modifica los datos	6. El sistema graba los datos actuales del producto

CASOS ALTERNATIVOS

4* Si el producto no existe termina el caso de uso.



Usuario

Sistema

Seleccionar la operación modificar producto

Ingresar el código el producto

Modificar datos

CONTRATO DE OPERACIONES

NOMBRE DEL CONTRATO DE OPERACIÓN	Seleccionar operación modificar producto
PROPÓSITO	Permitir a un usuario modificar datos.
TIPO	Sistema
SALIDA	Caja de texto
PRECONDICION	
POST CONDICIÓN	
EXCEPCIÓN	

DIAGRAMAS INTERACTIVOS

DIAGRAMA DE COLABORACIÓN

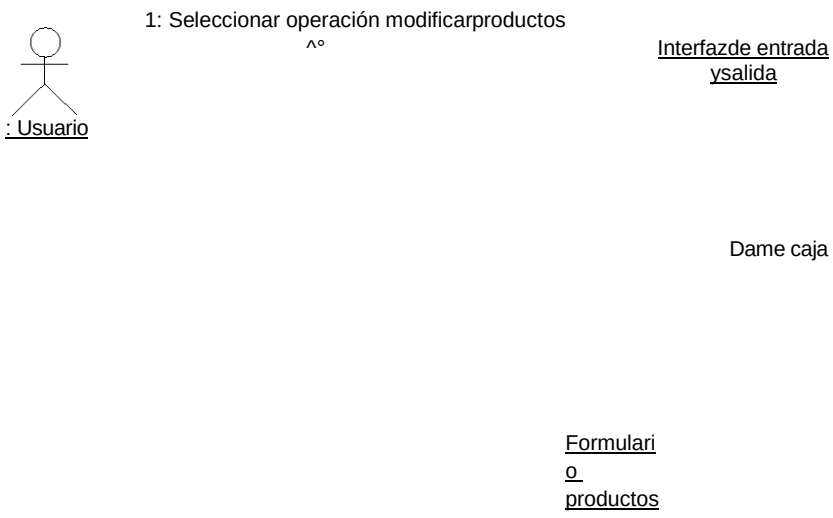
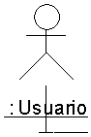


DIAGRAMA DE SECUENCIA



Seleccionar operacón modificar producto

Interfaz entrada salida

Formulario productos

Dame caja de texto

T

NOMBRE DEL CONTRATO DE OPERACIÓN	Ingresar código de producto
PROPOSITO	Ingresar el código de un producto
TIPO	Sistema
SALIDA	Formulario de productos con datos
PRECONDICION	Producto existe
POST CONDICIÓN	
EXCEPCIÓN	Producto no existe

DIAGRAMA DE COLABORACIÓN

1: Ingresar código del producto

^

Interfaz

:Usuario

\L 2: Dame formulario con datos

Formulario productos

DIAGRAMA DE SECUENCIA

Interfaz

Formulario productos

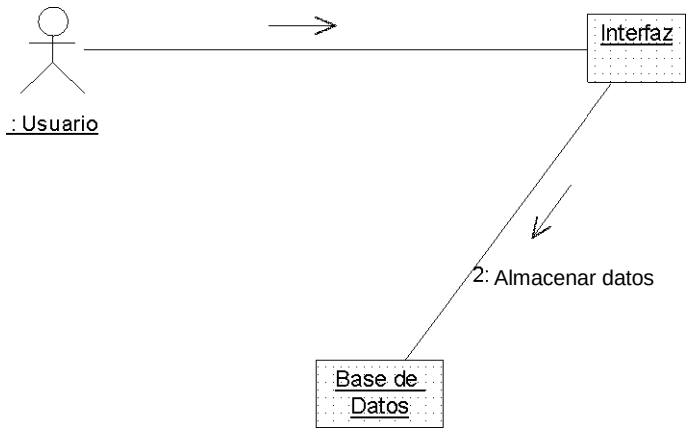
Usuario

Ingresar código del producto

Dame formulario con datos

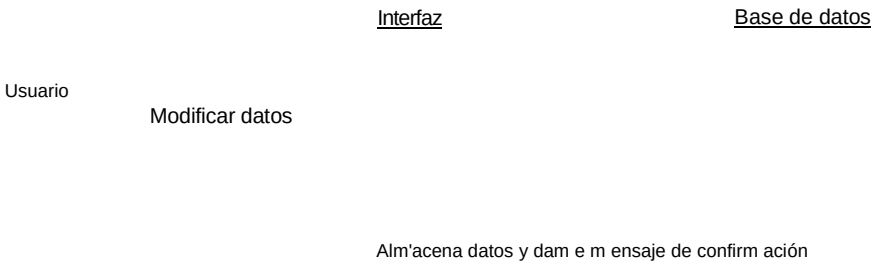
NOMBRE DEL CONTRATO DE OPERACIÓN	Modificar datos
PROPOSITO	Modificar datos del producto
TIPO	Sistema
SALIDA	Mensaje de confirmación
PRECONDICION	
POST CONDICIÓN	
EXCEPCIÓN	

DIAGRAMA DE COLABORACIÓN



1: Modificar datos

DIAGRAMA DE SECUENCIA



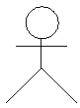
NOMBRE DEL CASO DE USO	Eliminar Producto
PROPÓSITO	Permitir a un usuario eliminar un producto.
REFERENCIA	Requisito 09
ACTOR	Usuario
TIPO	Primario Real
VISIÓN GENERAL	Este caso de uso inicia cuando el usuario selecciona la operación eliminar producto, el sistema pide el código de producto y emite un mensaje de verificación, el usuario confirma.

CURSO TÍPICO DEL EVENTO

ACTOR	SISTEMA
1. El usuario selecciona la operación eliminar producto.	2. El sistema pide código del producto.
3. El usuario ingresa el código de producto.	4. El sistema verifica si existe el producto y devuelve un mensaje.
5. El usuario confirma	6. El sistema elimina al producto

CASOS ALTERNATIVOS

4* Si el producto no existe termina el caso de uso.



Sistema

: Usuario

Seleccionar la operación eliminar producto

Ingresar código

confirmar eliminación del producto

CONTRATO DE OPERACIONES

NOMBRE DEL CONTRATO DE OPERACIÓN	Seleccionar operación eliminar producto
PROPÓSITO	Permitir a un usuario eliminar un producto.
TIPO	Sistema
SALIDA	Datos del cliente
PRECONDICION	Exista producto
POST CONDICIÓN	
EXCEPCIÓN	No existe el producto

DIAGRAMAS INTERACTIVOS

DIAGRAMA DE COLABORACIÓN

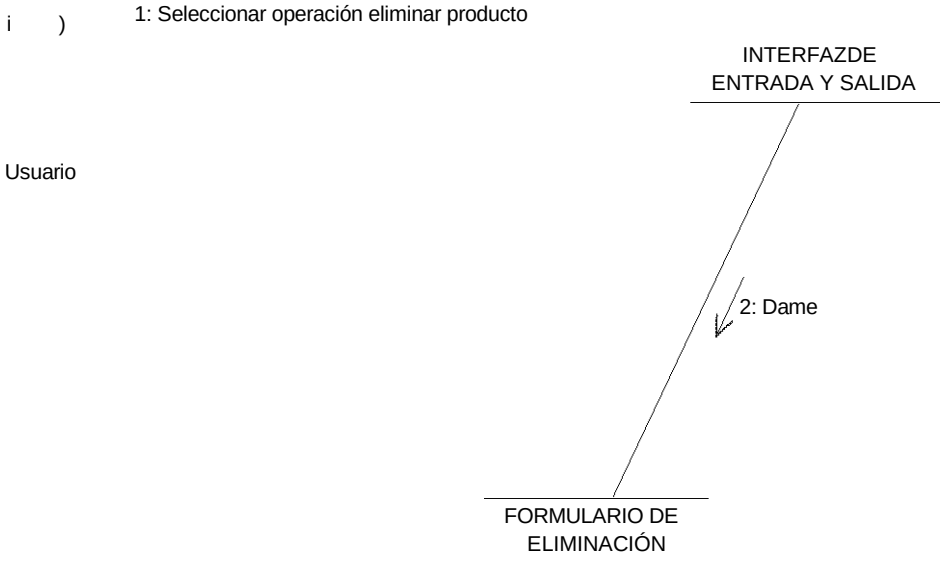
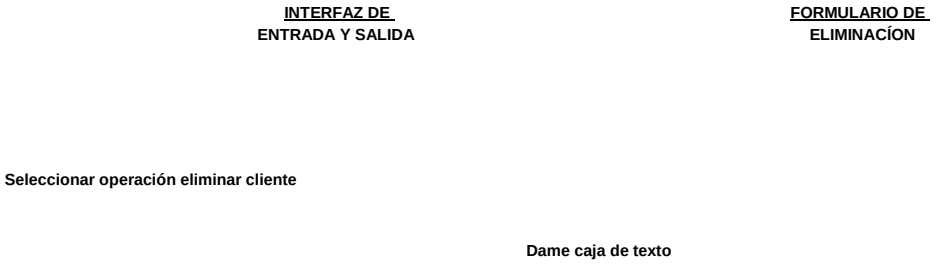


DIAGRAMA DE SECUENCIA



NOMBRE DEL CONTRATO DE OPERACIÓN	Confirmar eliminación del producto
PROPOSITO	Eliminar producto
TIPO	Sistema
SALIDA	Producto eliminado
PRECONDICION	
POST CONDICIÓN	
EXCEPCIÓN	No exista producto

DIAGRAMA DE COLABORACIÓN

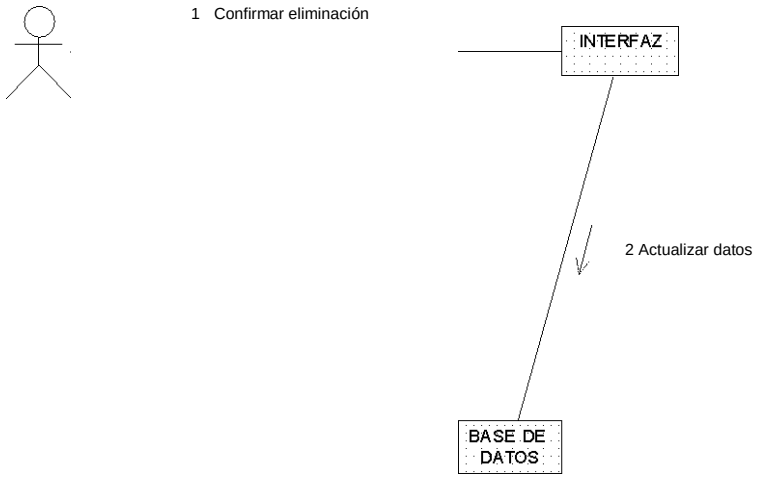


DIAGRAMA DE SECUENCIA

INTERFAZ BASE DE DATOS

Confirmar eliminación

Actualizar datos

GESTIÓN DE PEDIDOS

PARA EL USUARIO

DEFINICIÓN DE CASOS DE USO EXPANDIDOS

NOMBRE DEL CASO DE USO	Ingresar pedidos
PROPÓSITO	Permitir a un usuario ingresar un nuevo pedido.
REFERENCIA	Requisito 10
ACTOR	Usuario
TIPO	Primario Real
VISIÓN GENERAL	Este caso de uso inicia cuando el usuario selecciona la operación Ingresar pedido, el usuario ingresa la cédula o Ruc, el sistema muestra la información del cliente, el usuario selecciona productos, el sistema verifica stock.

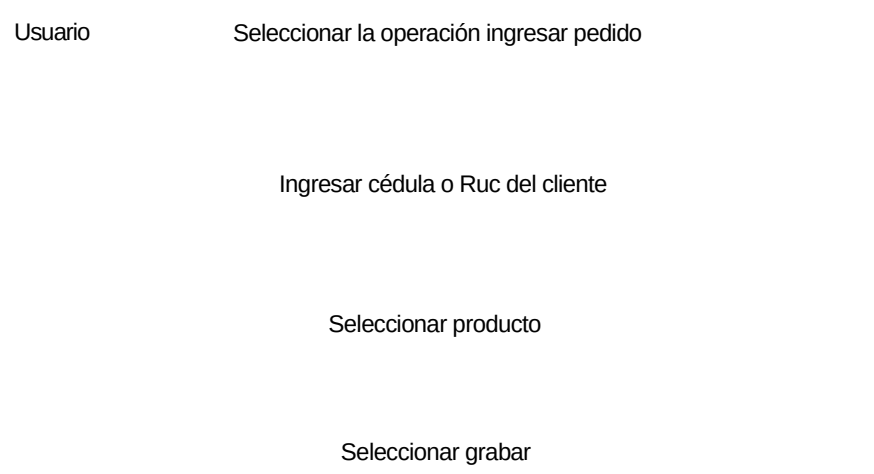
CURSO TÍPICO DEL EVENTO

ACTOR	SISTEMA
1. El usuario selecciona la operación ingresar pedido	2. El sistema pide el formulario de pedidos.
3. El usuario ingresa cédula o Ruc del cliente	4. El sistema comprueba si existe el cliente.
5. El usuario selecciona el producto	6. El sistema comprueba que exista en stock
7. el usuario selecciona la opción grabar	8. el sistema almacena el nuevo pedido y actualiza stock.

CASOS ALTERNATIVOS

4* Si el cliente ya existe termina el caso de uso.

6* Si no existe stock termina el caso de uso



CONTRATO DE OPERACIONES

NOMBRE DEL CONTRATO DE OPERACIÓN	Seleccionar operación Ingresar pedido
PROPOSITO	Presentar formulario de pedidos
TIPO	Sistema
SALIDA	Formulario de pedidos
PRECONDICION	Exista formulario de pedidos
POST CONDICIÓN	
EXCEPCIÓN	No existe formulario de pedidos

DIAGRAMAS INTERACTIVOS

DIAGRAMA DE COLABORACIÓN

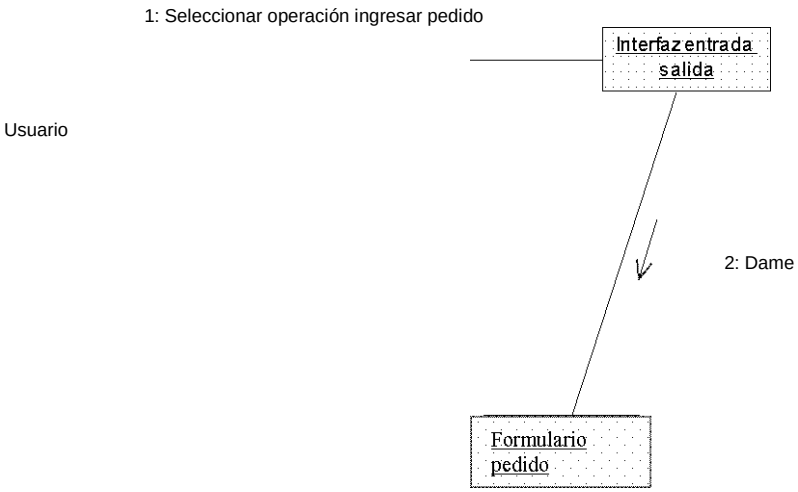
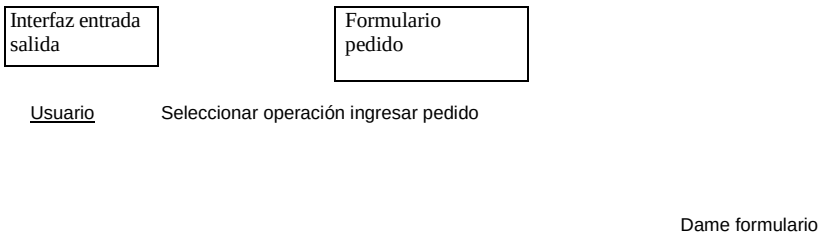
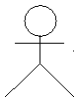


DIAGRAMA DE SECUENCIA



NOMBRE DEL CONTRATO DE OPERACIÓN	Comprobar stock de producto
PROPÓSITO	Comprobar si hay existencia de productos
TIPO	Sistema
SALIDA	Datos de productos
PRECONDICION	Exista stock de productos
POST CONDICIÓN	
EXCEPCIÓN	No exista stock de producto

DIAGRAMA DE COLABORACIÓN



1: Selecionar producto
^ Interfaz

: Usuario /

/ 2: Verifica existencia

Base de
Datos

DIAGRAMA DE SECUENCIA

Interfaz Base de datos

Usuario
Seleccionar producto

Verifica existencia
----------^

NOMBRE DEL CONTRATO DE OPERACIÓN	Almacenar pedido
PROPOSITO	Grabar datos del nuevo pedido.
TIPO	Sistema
SALIDA	Mensaje de verificación
PRECONDICION	
POST CONDICIÓN	
EXCEPCIÓN	No se puede grabar pedido

DIAGRAMA DE COLABORACIÓN

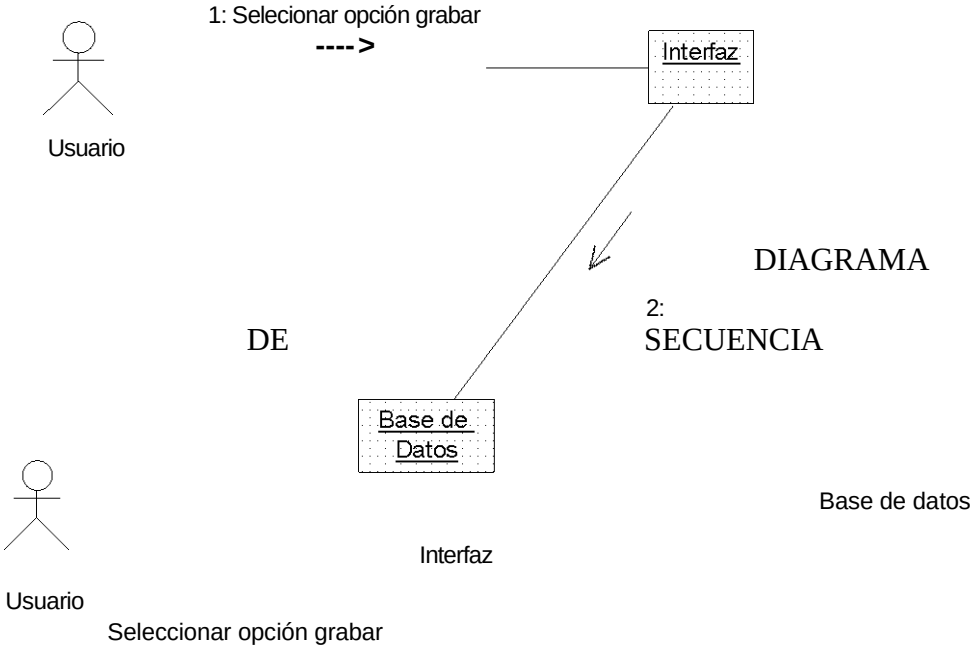


DIAGRAMA DE SECUENCIA

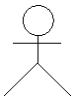
NOMBRE DEL CASO DE USO	Consultar pedidos
PROPÓSITO	Permitir a un usuario obtener información de pedidos
REFERENCIA	Requisito 11
ACTOR	Usuario
TIPO	Primario Real
VISIÓN GENERAL	Este caso de uso inicia cuando el usuario selecciona la operación información de pedidos, el sistema pide el número de pedido y presenta el formulario con lo datos.

CURSO TÍPICO DEL EVENTO

ACTOR	SISTEMA
1. El usuario selecciona la operación generar factura	2. El sistema devuelve factura en pantalla

CASOS ALTERNATIVOS

2* Si no existe la venta termina el caso de uso.



Usuario

Sistema

Seleccionarla operación consultar pedidos

CONTRATO DE OPERACIONES

NOMBRE DEL CONTRATO DE OPERACIÓN	Seleccionar operación Consultar pedidos
PROPÓSITO	Permitir al obtener usuario información de
TIPO	Sistema
SALIDA	Datos de pedidos
PRECONDICION	Exista pedidos
POST CONDICIÓN	
EXCEPCIÓN	

DIAGRAMAS INTERACTIVOS

DIAGRAMA DE COLABORACIÓN

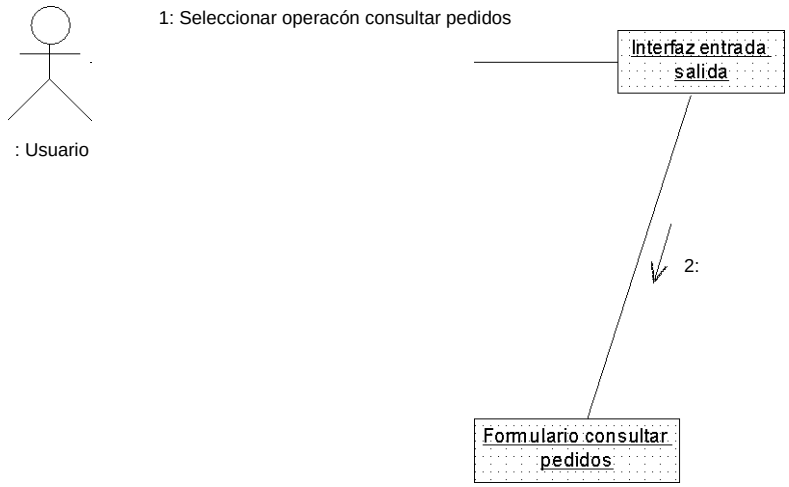
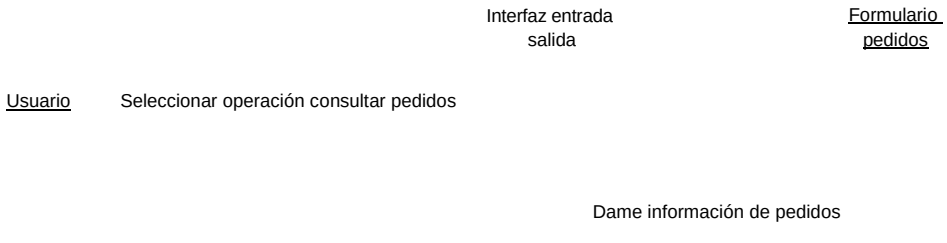


DIAGRAMA DE SECUENCIA



GESTIÓN DE VENTAS

PARA EL USUARIO

DEFINICIÓN DE CASOS DE USO EXPANDIDOS

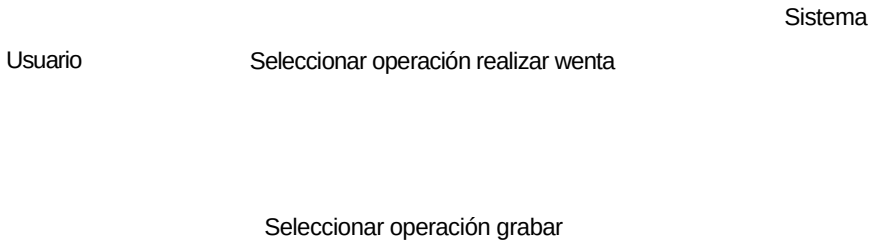
NOMBRE DEL CASO DE USO	Realizar venta
PROPÓSITO	Permitir a un usuario realizar una venta
REFERENCIA	Requisito 12
ACTOR	Usuario
TIPO	Primario Real
VISIÓN GENERAL	Este caso de uso inicia cuando el usuario selecciona la operación realizar venta, el sistema muestra el formulario ventas el mismo que esta relacionado con un pedido el usuario realiza la venta.

CURSO TÍPICO DEL EVENTO

ACTOR	SISTEMA
1. El usuario selecciona la operación realizar venta 3. El usuario selecciona la operación grabar	2. El sistema devuelve formulario de ventas 4. El sistema almacena los datos de la venta

CASOS ALTERNATIVOS

2* Si no existe un pedido termina el caso de uso.



CONTRATO DE OPERACIONES

NOMBRE DEL CONTRATO DE OPERACIÓN	Seleccionar operación realizar venta
PROPÓSITO	Permitir al usuario realizar una nueva venta.
TIPO	Sistema
SALIDA	Formulario de venta
PRECONDICIÓN	Exista un pedido y este confirmado el pedido
POST CONDICIÓN	
EXCEPCIÓN	Que no exista el pedido o no este confirmado

DIAGRAMAS INTERACTIVOS

DIAGRAMA DE COLABORACIÓN

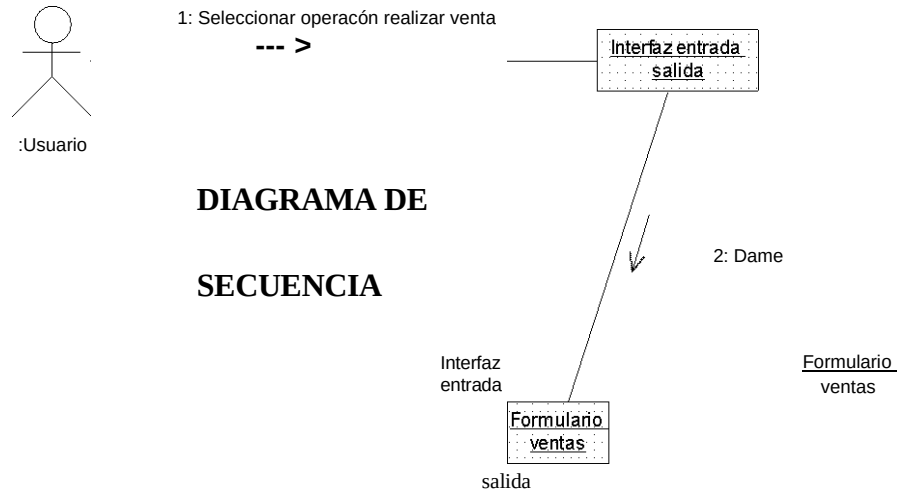


DIAGRAMA DE SECUENCIA

Usuario Seleccionar operación realizar \«ntas

Dame formulario

NOMBRE DEL CONTRATO DE OPERACIÓN	Almacenar datos de ventas
PROPÓSITO	Permitir que un usuario almacene una nueva venta
TIPO	Sistema
SALIDA	Mensaje de verificación
PRECONDICION	Exista pedido y este confirmado
POST CONDICIÓN	
EXCEPCIÓN	No exista pedido o no este confirmado

DIAGRAMAS INTERACTIVOS

DIAGRAMA DE COLABORACIÓN

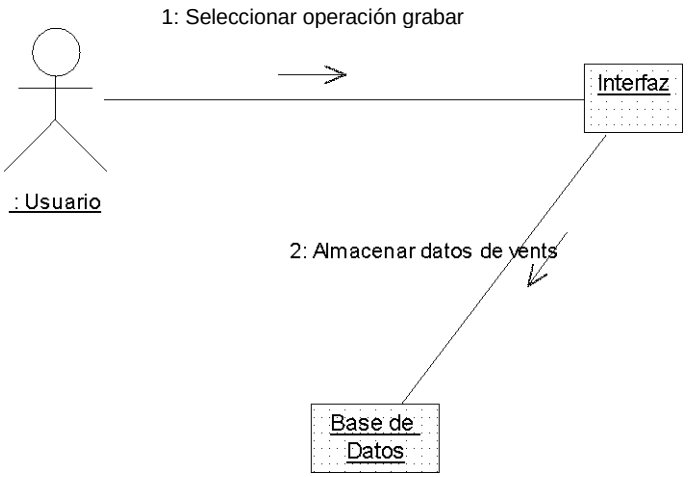


DIAGRAMA DE SECUENCIA

Interfaz Base de datos

Usuario

Seleccionar operación grabar

Almacenar datos de wentas

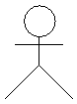
NOMBRE DEL CASO DE USO	Generar factura
PROPÓSITO	Permitir a un usuario generar factura
REFERENCIA	Requisito 13
ACTOR	Usuario
TIPO	Primario Real
VISIÓN GENERAL	Este caso de uso inicia cuando el usuario selecciona la operación grabar venta el sistema guarda los datos de la venta y genera una factura.

CURSO TÍPICO DEL EVENTO

ACTOR	SISTEMA
1. El usuario selecciona la operación generar factura	2. El sistema devuelve la factura en pantalla

CASOS ALTERNATIVOS

2* Si no existe la venta termina el caso de uso.



Usuario

Seleccionar operación generar factura

Sistema

CONTRATO DE OPERACIONES

NOMBRE DEL CONTRATO DE OPERACIÓN	Seleccionar operación generar factura
PROPÓSITO	Permitir al usuario generara una factura.
TIPO	Sistema
SALIDA	Factura
PRECONDICION	Exista una factura
POST CONDICIÓN	
EXCEPCIÓN	Que no exista una factura

DIAGRAMAS INTERACTIVOS

DIAGRAMA DE COLABORACIÓN

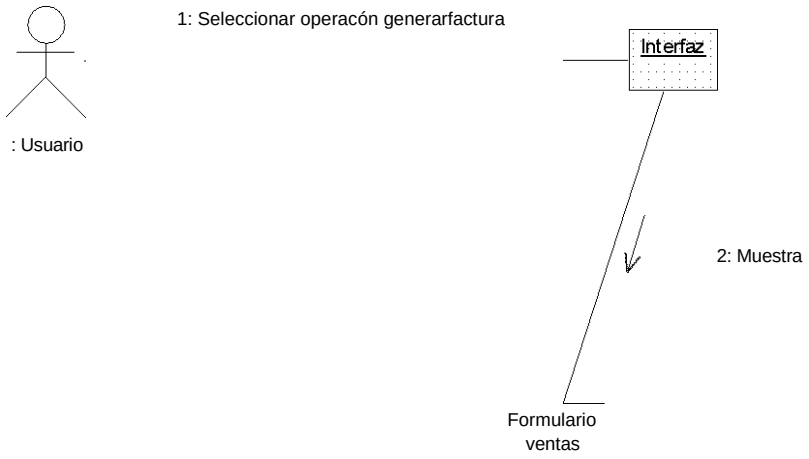
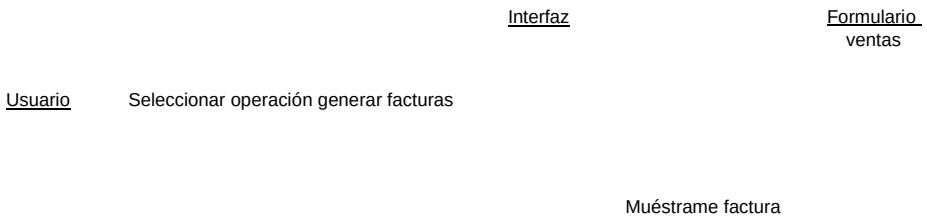


DIAGRAMA DE SECUENCIA



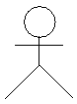
NOMBRE DEL CASO DE USO	Imprimir factura
PROPÓSITO	Permitir a un usuario imprimir una factura
REFERENCIA	Requisito 14
ACTOR	Usuario
TIPO	Primario Real
VISIÓN GENERAL	Este caso de uso inicia cuando el usuario selecciona la operación grabar venta, el sistema guarda los datos de la venta, genera una factura e imprime una factura.

CURSO TÍPICO DEL EVENTO

ACTOR	SISTEMA
1. El usuario selecciona la operación imprimir factura	2. El sistema devuelve factura impresa.

CASOS ALTERNATIVOS

2* si no existe el factura termina el caso de uso



Usuario

Seleccionar operación imprimir factura

Sistema

->

CONTRATO DE OPERACIONES

NOMBRE DEL CONTRATO DE OPERACIÓN	Seleccionar operación imprimir nota de venta
PROPÓSITO	Permitir a un usuario que imprima una factura.
TIPO	Sistema
SALIDA	Factura impresa
PRECONDICION	Exista una venta
POST CONDICIÓN	
EXCEPCIÓN	Que no exista una venta

DIAGRAMAS INTERACTIVOS

DIAGRAMA DE COLABORACIÓN

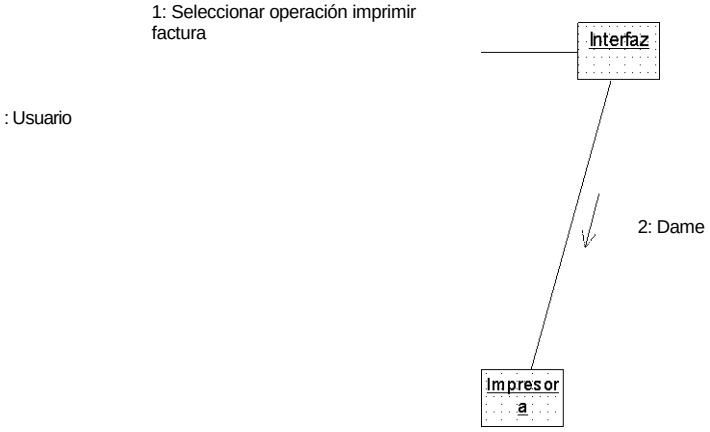
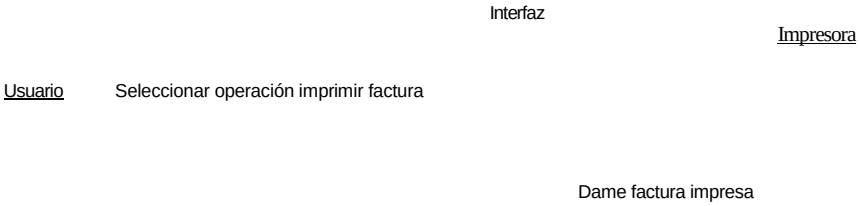


DIAGRAMA DE SECUENCIA



3.2.8. DIAGRAMA DE CLASES

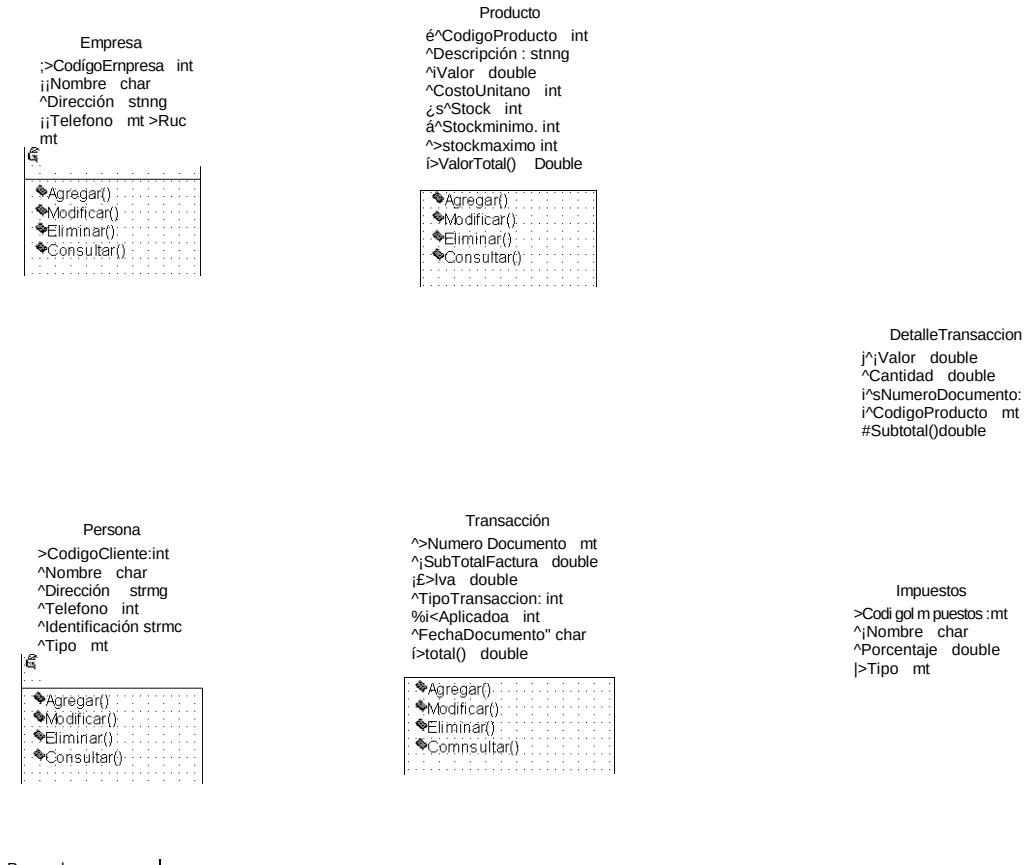


Figura 3.1 Diagrama de Clases producto de UML

3.7. IMPLEMENTACION DEL PROTOTIPO UTILIZANDO PROGRAMACIÓN ORIENTADA A ASPECTOS - POA

Al haber utilizado como herramienta de desarrollo UML el cual se fundamenta en la programación orientada a objetos la implementación sugiere utilizar uno de los lenguajes de programación orientado a objetos, sin embargo este trabajo utiliza programación orientada a objetos y la programación orientada a aspectos para el cual se tomará como base el diagrama de clases como producto generado al aplicar UML, el mismo que se le utiliza como elemento para aplicar el nuevo paradigma de programación.

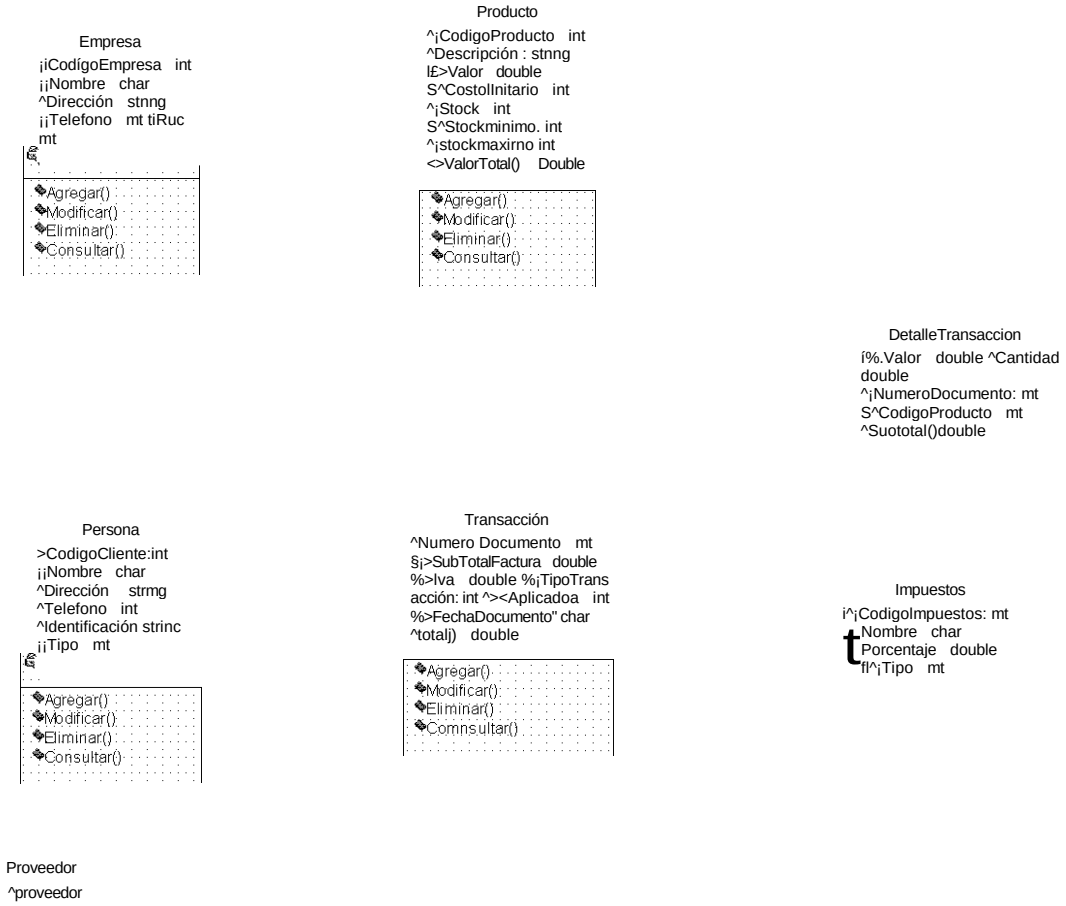


Figura 3.2 Diagrama de Clases generado en Rational Rose utilizando UML

Partiendo del diagrama de clases para tener una idea clara se señala paso a paso como se realiza el análisis para poder trabajar con aspectos, teniendo claro en el estudio realizado

del paradigma decimos que aún no existe una herramienta específica o estandarizada para poder diagramar un aspecto ya que es un paradigma que aún está siendo estudiado.

Para poder representar de una manera gráfica los aspectos se ha elegido la herramienta MyEclipse versión 6.0.1 GA teniendo en cuenta las limitaciones que presenta el plug-in de aspectj versión 1.5.0.200.706070619 para dicha versión.

A continuación se observa en la figura 3.3 el resultado de pasar el diagrama de clases realizado en UML a la herramienta MyEclipse.

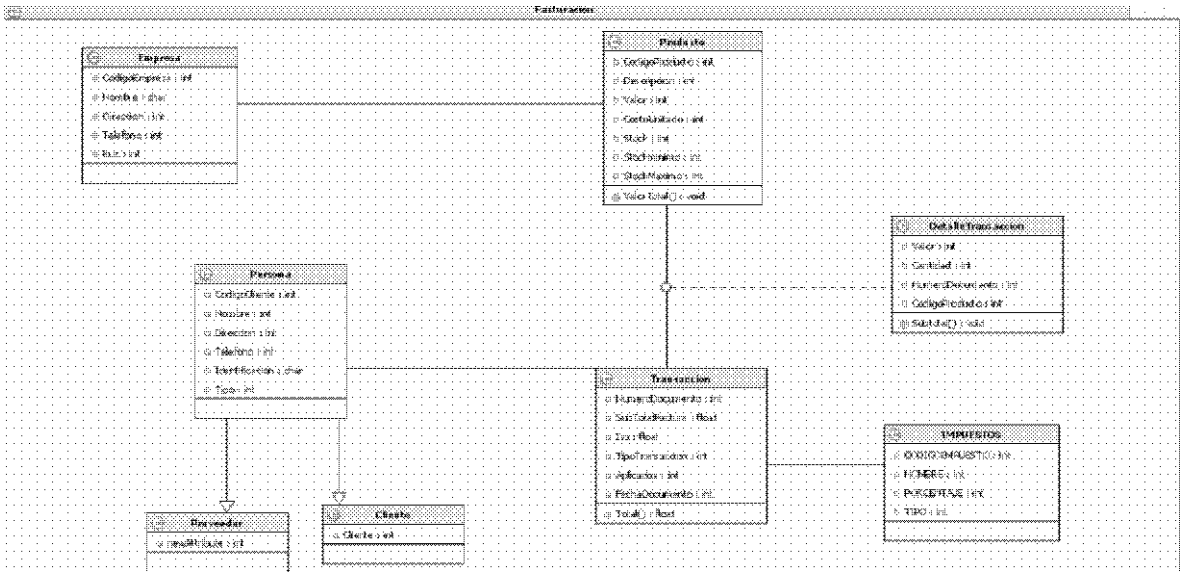


Figura 3.3 Diagrama de Clases generado en MyEclipse

Hasta aquí se ha representado el diagrama de clases generado en UML en la herramienta MyEclipse es decir no hay nada de novedoso es más se puede decir que MyEclipse también es una herramienta que nos permite realizar diagramas de clases para lo que es la programación orientada a objetos. A continuación se muestra de una manera gráfica donde se aplicará POA

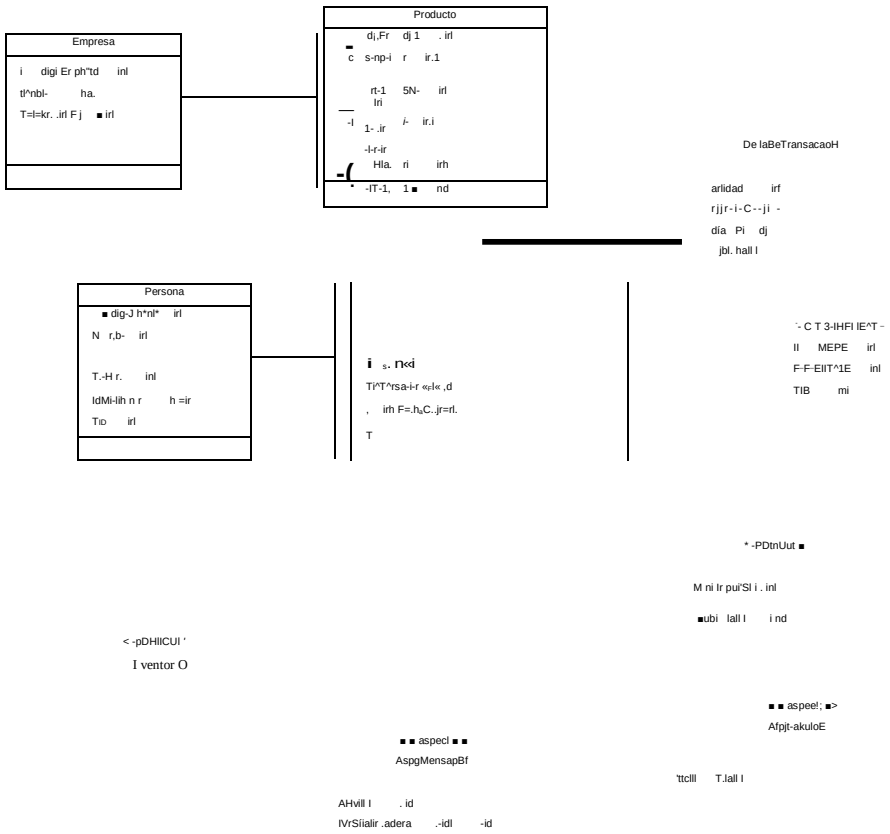


Figura 3.4 Diagrama de Clases con Aspectos generado en MyEclipse

Como se puede observar en la figura 3.4 el diagrama del prototipo Sistema de Facturación con aspectos, se especifica que se trabajará con POA para llevar el control de todos los cálculos matemáticos que requiera la aplicación y la manipulación de mensajes.

Teniendo claramente diagramado como se va aplicar y en que instancia actuará la programación orientada a aspectos se procede a programar.

Para un mejor entendimiento se ha separado en dos funciones la programación con aspectos, una de las funciones va ha controlar todo lo correspondiente a mensajería esto quiere decir, realizar un Punto de Corte *{desvió de la ejecución normal del programa hacia el aspecto para realizar ciertas tareas y devolver el control al programa para continuar con su normal ejecución)* en cada acción que realice la base de datos como resultado se obtendrá un cuadro de texto dando a conocer cuál fue la acción que se realizó, por ejemplo

si realiza ya sea una alta, baja, modificación o consulta en la aplicación, se tendrá un cuadro de texto notificando lo que acaba de realizar.

Con las siguientes líneas de código se muestra la programación con Aspectos de la función mensajes.

Nota:
PUNTO DE CORTE PARA LA FUNCIÓN PROPIA DE JPA QUE MANEJA LOS LOG DE LA BD.

```
pointcut Evento sEmpresaQ : call(* EntityManagerHelper.log(..));
after () returningQ: Evento sEmpresaQ {
Object[] args =thisJoinPoint.getArgsQ;
String MensajeEvento = (String) argsfOJ;
mensajes(MensajeEvento, "El Proces de
:"+thisJoinPointStaticPart.getSignature().getName());
}
```

Código 16. Aspectos atreves de Puntos de Corte

Con estas líneas de código se consigue realizar 90 puntos de corte o pinchazos a los procesos que interactúan con la base de datos, la opción after especifica que se notifique con un mensaje la acción que ejecutó después de realizada la acción como podemos ver en la siguiente figura generada por MyEclipse:

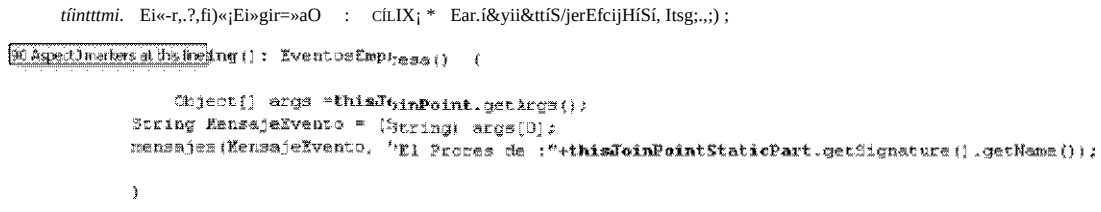


Figura 3.5 Etiqueta del Número de Aspectos

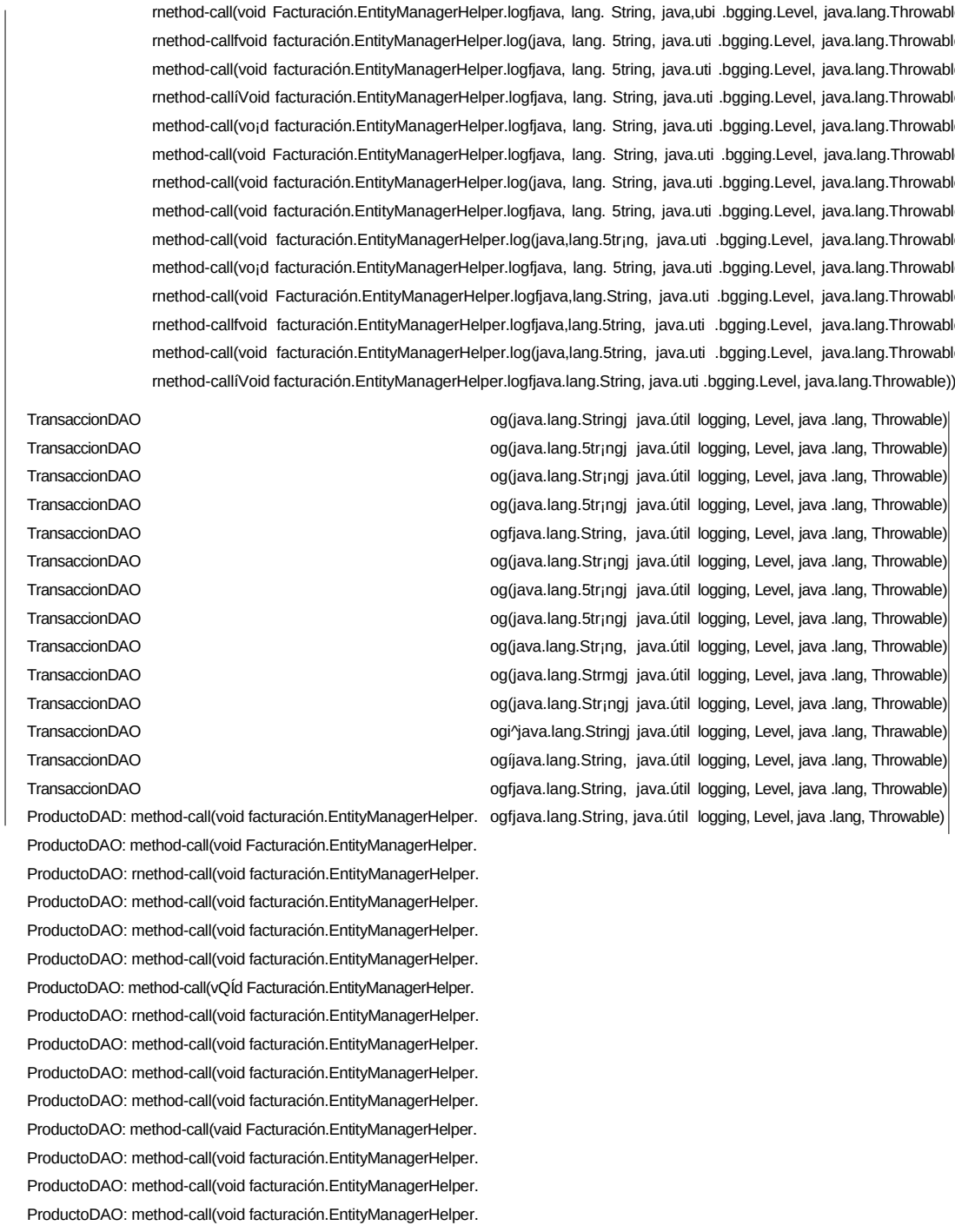


Figura 3.6 Pantalla del Listado de Aspectos

Se puede presenciar un listado de las 90 acciones que realiza la función de mensajería con aspectos, con esto se diría la ventaja que tiene el utilizar POA ya que para poder realizar esta acción con POO se debería programar en cada uno de las clases, eso implica que se tiene 7 líneas de código esto multiplicado por los 6 clases que contiene la aplicación, se estaría hablando de 42 líneas de código tomando en cuenta que la aplicación es un prototipo corto.

Se ha trabajado también con aspectos en el manejo de un mensaje de aviso de cierre de ventana en el botón salir, se aplica para todas las ventanas que cuentan con dicha opción esto con la finalidad de poner en funcionamiento varias alternativas de usar aspectos en una aplicación, a continuación se verá el código que se utiliza en la función

AspjMensajes.aj que interactúa con la acción del botón salir al contrario del código anterior lo hacía con la base de datos.

Nota:
*PUNTO DE CORTE PARA EL OBJETO SALIR A NIVEL DE **TODO** EL SISTEMA. AL UTILIZAR PROGRAMACIÓN ORIENTACIÓN A OBJETOS SE TENDRÍA QUE PROGRAMAR POR CADA UNO DE LOS BOTONES SALIR DE LAS PANTALLAS DEL SISTEMA.*

```
pointcut Evento sSalidaQ : call(* *cmdSalirActionPerformed*(..)) &&
!within(Mmensajes);
before () : Evento sSalidaQ {mensajesf'Se cerrara la Ventana", " Cierre de Pantalla");}
```

Código 17. Aspectos atreves de Puntos de Corte

En este caso son 3 líneas de código si se quisiera hacer sin utilizar aspectos se debería programar en cada uno de los botones de la opción salir, entonces se puede comprender claramente cómo actúan los aspectos con el ahorro de líneas de código en esta pequeña aplicación.

Hasta aquí se ha visto claramente cómo funciona la programación con aspectos en la función AspjMensajes.aj.

A continuación se detalla la función AspjCalculos.aj, la cual controla todos los cálculos que requiere el Sistema de Facturación CRC, al mismo tiempo llevará un control del stock de los productos ya sea mercadería de salida como de ingreso, tomando en cuenta que si la aplicación requiriera algún cambio en lo concerniente a cálculos o impuestos es de fácil rectificación o agregación sin tener que alterar cada una de las clases involucradas si no solo la función AspjCalculos.aj.

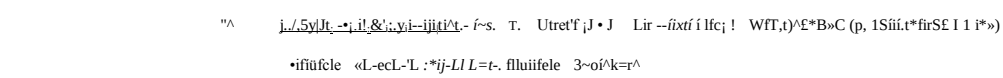


Figura 3.7 Etiqueta del Número de Aspectos

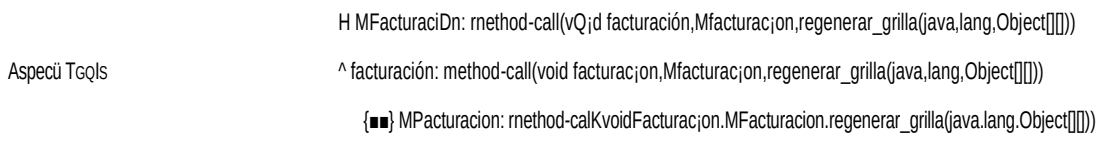


Figura 3.8 Etiqueta del Listado de Aspectos

Como se visualiza en la figura 3.7 se ha notificado los 3 aspectos que realiza en ese punto de corte, y de haber alguna alteración en la aplicación en esta consola de trabajo seria donde deberíamos modificar sin que altere las clases programadas en java con las que consta nuestro sistema.

MyEclipse es una herramienta que permite mostrar gráficamente el porcentaje de trabajo con aspectos además de indicar claramente como interactúa con las clases y en que instancia lo hace.

Figura 3.9 Pantalla del Porcentaje de Trabajo con Aspectos

Finalmente concluimos con la Figura 3.9 donde se especifica de una manera gráfica el trabajo que se realizó con POA en el Sistema de Facturación CRC, quedando claramente justificado el objetivo del proyecto de tesis.

3.8. CÓDIGO FUENTE DEL PROTOTIPO

Debido a la amplitud de código se va a señalar los párrafos específicos donde está la programación con aspectos. Todo el código fuente estará en el anexo **1.**

3.4.3. CÓDIGO DE LA FUNCIÓN ASPJCALCULOS

```
package facturación; public
aspect AspjCalculos {

//Punto de corte a mfacturacion para realizar validación y cálculos
pointcut métodosSet(Mfacturación p, ObjectfJfJ Datosgrilla): call( *
*regenerar_grilla*(*)) && target(p) && args'(Datosgrilla);

//valido el stock y que no se repita
 before(Mfacturacion p, ObjectfJfJ Datosgrilla): metodosSet(p,Datosgrilla) {
 double cantidad=0;
 double codProducto=0;
 double subtotal=0;
 double stock=0;
 int VarInicio=0;
 try{
cantidad=Double.parseDouble(Datosgrillafp.getRow()-1J f3J .toStringQ);
codProducto=Double.parseDouble(Datosgrillafp.getRow()-1J fOJ .toStringQ);
} catch (Exception e) { e.printStackTraceQ;
 }
 stock=p.STOK;
 if (cantidad > stock) {
p.TotRow—;
 mensajes("Solo tiene en stock : " + stock, "ERROR EN LA CANTIDAD");
 throw new IllegalArgumentException("Solo tiene en stock : " + stock);
}

for (Varlnicio = 0; Varlnicio < p.getRow()-l; VarInicio++) {
 if (codProducto==Double.parseDouble(DatosgrillafVarInicioJ fOJ .toString())){
p.TotRow—;
 mensajes("El Producto ya esta Ingresado : " + Datosgrilla fVarlnicioJflJ. toStringQ ,
  "ERROR PRODUCTO REPETIDO ");
 throw new IllegalArgumentException("El Producto ya esta Ingresado : " +
Datosgrilla fVarlnicioJflj. toStringQ );
}
}
}

 after(Mfacturacion p, ObjectfJfJ Datosgrilla): metodosSet(p,Datosgrilla) {
```

```
double subtotal=0;
double total=0; double
impuesto=0; int
VarInicio=0;
try{
for (Varlnicio = 0; Varlnicio < p.getRowQ; VarInicio++) { subtotal=subtotal+
DoubleparseDouble(DatosgrillafVarInicioJf4J.toStringQ);
}
p.SubTotal=subtotal;
impuesto=impuestos(subtotal, "102 ");
total=subtotal+impuesto;
p.Impuesto=impuesto;
}
catch (Exception e) {
eprintStackTraceQ;
}
}

pointcutSubtotalFacturaaddQ : call(* *setValoresTotales(..));
beforeQ : SubtotalFacturaaddQ {
System.outprintln( "Recalculamos el valor de la Factura cuando adiciona");
}

//Punto de corte a mrecepcion productos para realizar validación y cálculos pointcut
metodosSetrecepcion(MrecepcionProductos p, ObjectfJfJ Datosgrilla): call(
*.regenerar_grilla*(*)) && target(p) && args'(Datosgrilla);
before(MrecepcionProductos p, ObjectfJfJ Datosgrilla) :
metodosSetrecepcion(p,Datosgrilla){ double cantidad=0;
double codProducto=0; double stock=0; int VarInicio=0;
try{
cantidad=DoubleparseDouble(Datosgrillafp.getRow()-1J f3J .toStringQ);
codProducto=DoubleparseDouble(Datosgrillafp.getRow()-1J fOJ .toStringQ); } catch
(Exception e) { eprintStackTraceQ;
} stock=p.STOKMAXIMO;

if (cantidad > stock) {
p.TotRow—;
mensajes("Se sobrepaso el Stock Máximo : " + stock, "ERRORENLA CANTIDAD");
throw new IllegalArgumentException("El Stock Máximo es : " + stock);
}

for (Varlnicio = 0; Varlnicio < p.getRow()-l; Varlnicio++) {
```

```
if (codProducto==Double.parseDouble(Datosgrilla[VarInicio] [0] .toString())){
p.TotRow—;

mensajes("El Producto ya esta Ingresado : " + Datosgrilla[VarInicioJ[lJ.toString() ,
 "ERROR PRODUCTO REPETIDO ");
throw new IllegalArgumentException("El Producto ya esta Ingresado : " + codProducto
y,
}
}

}

after(MrecepcionProductos p, ObjectfJfJ Datosgrilla):
metodosSetrecepcion(p,Datosgrilla){
double subtotal=0;
double total=0;
double impuesto=0;
int VarInicio=0;
System.out.println("desde el aspecto :" + p.getRowQ);
for (Varlnicio = 0; Varlnicio < p.getRowQ; VarInicio++) {
subtotal=subtotal+ Double.parseDouble(Datosgrilla[VarInicio] [4] .toStringQ);
}
p.SubTotal=subtotal;
impuesto=impuestos(subtotal, "102 ");
total=subtotal+impuesto;
p.Impuesto=impuesto;
}

prívate double impuestos (double valorBase, String codigoImpuesto){
double porcentaje=0;

ImpuestosDAO dao = new ImpuestosDAOQ;
Impuestos newlmpuesto = dao.findByld(codigoImpuesto);
porcentaje=newImpuesto.getPorcentaje();
valorBase=valorBase*porcentaje;
return valorBase;

}

//DISPARA LA PANTALLA DE ALERTA
prívate void mensajesfString TexMensaje, String Titulo){
System.out.printlnfTexMensaje);
Mempresa Mempresa = new MempresaQ;
Mmensajes Mmensajes = new Mmensajes (Mempresa,true,TexMensaje,Titulo);
Mmensajes.show ();
} }
```

3.4.4. CÓDIGO DE LA FUNCIÓN ASPJMENSAJES

```
package facturación;
import java. útil. logging.Level; import
javax.swing.JOptionPane; public
aspect AspjMensajes {

//PUNTO DE CORTE OPCIÓN 2 solo como ejemplo en la clase empresa
pointcutEventosEmpresal() : executionf* EmpresaDAO.savef..)) || executionf*
EmpresaDAO.update(..)) \\ execution(*EmpresaDAO.delete(..));
 after 0 returning: Evento sEmpre sal () {
 mensajesf'elemplo de aspecto opción 2 ", " ejemplo 2 Aspectos El Proces de :");
}

// PUNTO DE CORTE PARA LA FUNCIÓN PROPIA DE JPA QUE MANEJA LOS LOG
DELABD.
pointcut Evento sEmpre sa() : call(* EntityManagerHelper.log(..));
 after () returningQ: Evento sEmpre sa() {
 Object[] args =thisJoinPoint.getArgsQ;
 String MensajeEvento = (String) argsfOJ;
 mensajes(MensajeEvento, "El Proces de
 :"JrthisJoinPointStaticPart.getSignature().getName());
}

// PUNTO DE CORTE PARA EL OBJETO SALIR A NIVEL DE TODO EL SISTEMA, SI
UTILIZÁRAMOS ORIENTACIÓN A OBJETOS ESTO TENDRÍAMOS QUE PROGRAMAS
POR CADA UNO DE LOS BOTONES SALIR DE LAS PANTALLAS DEL SISTEMA

 pointcut Evento sSalidaQ : call(* *cmdSalirActionPerformed *(..)) &&
 iwithin (Mmensajes) ;

 before () : Evento sSalidaQ {
 mensajesf'Se cerrara la Ventana", " Cierre de Pantalla");
}

//FUNCIO PARA PRESENTAR MENSAJES A PANTALLA
private void mensajesfString TexMensaje, String Titulo){
 System.out.printlnfTexMensaje);
Mempresa Mempresa = new MempresaQ;
Mmensajes Mmensajes = newMmensajes(Mempresa,true,TexMensaje,Titulo);
Mmensajes.show ();

}

}
```

CAPITULO IV

CONCLUSIONES Y RECOMENDACIONES

4.1. CONCLUSIONES

- Estamos ante la presencia de un paradigma que si bien todavía sufre de cierto grado de inmadurez, muestra bases lo suficientemente sólidas como para pensar que se convertirá en la solución definitiva al problema de complejidad en la construcción del software.

- La POA es un paradigma muy reciente y por lo tanto poco consolidado, pero las indiscutibles posibilidades que ofrece a los desarrolladores de software están haciendo que su popularidad crezca a gran velocidad.

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

- La POA encapsula estos elementos del sistema 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 sustituto de la POO, sino como una extensión de la misma, dando lugar a un paradigma de programación en el que coexisten clases y aspectos. Las clases modelan 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 una separación de intereses completa.

- Las ventajas de aplicar POA son numerosas: obtenemos diseños más 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 más fácil razonar sobre los intereses principales de la aplicación. También existen inconvenientes, la mayoría relacionados con el carácter novedoso del paradigma, que hace que no existen estándares que faciliten su consolidación.

- Se ha utilizado MyEclipse versión 6.0.1 GA 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.

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

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

4.2. 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 llegado a desarrollarse en su plenitud.

- Una vez superada todos los inconvenientes se ha demostrado en este trabajo las ventajas que provee el POA recomendando su utilización de sistemas de mayor envergadura.

- Se recomienda realizar trabajos de investigación en el área de desarrollo de software con la finalidad de facilitar el desarrollo y la programación, como se pudo demostrar en el trabajo la POA facilita la programación dejando en claro que los lenguajes de programación y metodologías de desarrollo actuales pueden evolucionar y no son la última palabra.

4.3. REFERENCIAS

[1] K. Mehner, A. Wagner, *"An Assessment OfAspect Language Design"*, Position Paper in Young Researchers Workshop, GCSE '99, 1999.

[2] Claire Tristram, *"Untangling Code"*, in the January/February 2001 issue of the Technology Review.

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

[4] S. Matsuoka, A. Yonezawa, *'Analysis of inheritance anomaly in objectoriented concurrent programming languajes"*, in Research Directions in Concurrent Object Oriented Programming (G. Agha, P.Wegner, and A. Yonezawa, eds.),pp. 107-150, Cambridge, MA: MIT Press, 1993.

[5] Página del grupo Demeter: http://www.ccs.neu.edu/research/demeter/.

[6] John Lamping, *"The role ofthe base in aspect oriented programming"*, in Proceedings of the European Conference on Obj ect-Oriented Programming (ECOOP) Workshops 1999.

[7] Timothy Highley, Michael Lack, Perry Myers, *"Aspect-Oriented Programming: A Critical Analysis of a new programming paradigm"* , University of Virginia, Department of Computer Science, Technical Report CS-99-29, Mayo 1999.

[8] Cario Ghezzi, Mehdi Jazayeri, *"Programming language concepts"*, 3º Edición, John Wiley&Sons, 1998.

[9] Ramnivas Laddad, *"I want myAOP", Part 1,2 and 3*, from JavaWorld, Enero-Marzo-Abril 2002.

[10] L.Berger, *"Junction Point Aspect: A Solution to Simplify Implementation ofAspect Languages and Dynamic Management ofAspect Programs* ", in Proceedings of ECOOP 2000, Junio de 2000, Francia.

[11] Peter Kenens, Sam Michiels, Frank Matthijs, Bert Robben, Eddy Truyen, Bart Vanhaute, Wouter Joosen, Pierre Verbaeten, *"An AOP Case with Static and Dynamic Aspects"*, Departament of Computer Science, K.U. Leuven, Bélgica.

[12] Gregor Kickzales, John Lamping, Cristina Lopes, Chris Maeda, Anurag Mendhekar, Gail Murphy, *"Open Implementation Design guidelines"*, in Proceedings of the 19th International Conference on Sofware Engineering, (Bostón, MA), mayo de 1997.

[13] Mattia Monga, *"Concern Specific Aspect-Oriented Programming with Malaj"*, Politécnico de Milano, Dip. Di Elettronica e Informazione, Milán, Italia.

[14] Gianpaolo Cugola, Cario Ghezzi, Mattia Monga, Gian Pietro Picco, *"Malaj: A Proposal to Eliminate Clashes Between Aspect-Oriented and Object-Oriented Programming"*, Politécnico de Milano, Dip. Di Elettronica e Informazione, Milán, Italia.

[15] Harold Ossher, Peri Tarr, *"Multi-Dimensional Separation of Concerns and the Hyperspace Approach"*, in Proceedings of the Symposium on Software Arquitectures and Component Technology: The state of the Art in Sofware Development. Kluwer, 2001.

[16] Andreas Gal, Wolfgan Schroder, Olaf Spinczyk, *"AspectC++: Language Proposal andPrototypeImplementation"*, University of Magdeburg, Alemania, 2001.

[17] Johan Brichau, *"Declarative Composable Aspects"* , in Proceedings of OOPSLA 2000 Workshop on Advanced Separation of Concerns, Septiembre de 2000.

[18] Ralf Lammel, *"Declarative aspect-oriented programming*", in Proceedings PEPM'99, 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM'99, San Antonio (Texas), BRICS Notes Series NS-99-1, páginas 131-146, Enero del999.

[19] B. Wulf, M. Shaw, *"Global Variables Considered Harmful"*, SIGPLAN Notices, vol.8, No. 2, February, 1972.

[20] El sitio de AspectJ: www.aspectj.org . Palo Alto Research Center.

[21] Página de Squeak: http://squeak.org .

[22] Robert Hirschfeld, *"AspectS-AOP with Squeak"*, in Proceedings of OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented System, Agosto de 2001.

[23] Johan Brichau, Wolfgang De Meuter, Kris de Volder, *"Jumping Aspect"*, Programming Technology Lab, Vrije Universiteit Brussel, Bélgica, Abril de 2000.

4.4. REFERENCIAS BIBLIOGRÁFICAS

- http://www.ccs.neu.edu/research/demeter/
- http://www2.parc.com/csl/groups/sda/
- http://eclipse.org/aspectj/
- http://www.germany.net/teilnehmer/101,199268/
- www.cs.ub.ca/labs/spl/projects/aspectc.html
- www.aspectc.org
- www.prakinf.tuilmenau.de/~hirsch/Projects/Squeak/AspectS
- www.cs.ub.ca/labs/spl/projects/apostle/
- http://sourceforge.net/proiects/pvthius/
- http://aspectr.sourceforge.net
- http://jac.aopsys.com
- http://aspectwerkz.codehaus.org
- http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectihome/ai5announce.htm 1
- http://patterntesting.sourceforge.net
- http://www.eclipse.org/aspecti/doc/devguide/progguide/index.html
- http://www.cs.ubc.ca/~ian/AODPs/
- www.eclipse.org/ajdt
- http ://www. springframework. org/
- http://www.eclipse.org/aspectj/doc/released/progguide/index.html

4.5. ANEXOS

Anexo 1. En el siguiente CD contiene el código fuente de todo lo concerniente al prototipo software "Sistema de Facturación CRC".