34

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES

"ACATLAN"

PROGRAMACION ORIENTADA A ASPECTOS

T E S I N A

QUE PARA OBTENER EL TITULO DE:

LICENCIADO EN MATEMATICAS APLICADAS Y COMPUTACION

P R E S E N T A :

GABRIEL RAMIREZ CAZARES

ASESOR: ING. RUBEN ROMERO RUIZ

UNAM
CAMPUS ACATLÁN

ACATLAN, ESTADO DE MEXICO **OCTUBRE 2002**

A mis padres Leticia y Carlos por todo el apoyo que me brindaron no nada mas en mis estudios sino en mi vida en general, y que sin el cual no hubiera sido posible la terminación de este trabajo y además por ser determinantes en la formación de mi persona.

A todos los maestros que tuve a lo largo de mi formación escolar, que de alguna u otra forma aportaron a la misma.

"To my taste the main characteristic of intelligent thinking is that one is willing and able to study in depth an aspect of one's subject matter in isolation, for the sake of its own consistency, all the time knowing that one is occupying oneself with only one of the aspects. The other aspects have to wait their turn, because our heads are so small that we cannot deal with them simultaneously without getting confused."

Edsger Dijkstra *"A discipline of programming"*

A mi parecer la principal característica del pensamiento es que, se tiene la capacidad y la disposición de estudiar a profundidad un solo aspecto de algún cuestionamiento de manera aislada, y por el bien de su propia consistencia, estar siempre consciente que se esta tratando con tan solo uno de los aspectos. Los demás tendrán que esperar su turno, debido a que nuestras cabezas son tan pequeñas que no podemos tratar con ellos simultáneamente sin perdernos en el camino. *(traducción propia)*

Introducción

¿Hay necesidad de otra metodología de programación?, para responder a esta cuestión habría que responder las siguientes preguntas, ¿cuál es la meta clave en el diseño de software?, y ¿se ha alcanzado satisfactoriamente esa meta?, la meta clave en el diseño de software es descomponer el sistema a analizar en unidades significativas, cada una de las cuales con una funcionalidad perfectamente definida y sin que sus características se empalmen entre sí; obligando de alguna forma a que el diseño del sistema a implementar utilice unidades funcionales comprensibles, perfectamente definidas y claras, además estas unidades pueden ser reutilizadas si la interfaz esta bien definida.
Actualmente los objetos se han estado utilizando extensivamente para definir las unidades significativas de los sistemas de software. Los objetos permiten a los programadores encapsular las unidades funcionales dentro de clases perfectamente bien definidas; sin embargo, los objetos como otros paradigmas de programación, algunas veces fallan al tratar de modularizar cuestiones que no se localizan en los límites de un sólo módulo. Algunas cuestiones de diseño atraviesan varios módulos y pueden inclusive extenderse por todo el sistema. Un ejemplo de este tipo de cuestiones es el manejo de errores o excepciones, en la programación orientada a objetos, los programadores tienen que añadir fragmentos de código para la comprobación de errores en cada uno de los módulos del programa; resultando en un sistema de software con fragmentos de código de comprobación de errores por todos lados, enmarañando así todas las clases y por lo tanto el software resultante, no tan sólo es difícil de entender sino además difícil de darle mantenimiento, ya que las modificaciones hechas a un fragmento de código podrían causar problemas en otros fragmentos sin que el programador, siquiera, se de cuenta; y debido a que estos fragmentos de código están diseminados por todo el sistema, la raíz de determinado problema puede no ser aparente.
Por estas razones, y contestando al primer cuestionamiento, es necesario el surgimiento de alguna forma que ataque efectivamente estos problemas.

Una nueva metodología de programación ha emergido recientemente. Esta metodología permitirá a los diseñadores y programadores de sistemas de software modularizar las cuestiones que se diseminan, en el código por todos lados, en entidades perfectamente bien definidas. Los investigadores en esta materia la llaman *programación orientada a aspectos*.

Hasta ahora la idea primaria para organizar los sistemas de software ha sido descomponer el sistema en módulos funcionales como subrutinas, procedimientos, funciones, clases, etc. Pero muchas cuestiones que se necesitan programar no siguen limpiamente o no encajan de manera natural dentro de estas abstracciones disponibles. La programación orientada a aspectos permite a los programadores expresar cada una de estas cuestiones en una forma natural y mucho más apropiada, esto quiere decir que el sistema en el que se esta trabajando se descompone en piezas que tratan únicamente un solo aspecto, y después se integran para formar el sistema completo, permitiendo así programar usando aspectos naturales de cada uno de los ámbitos dentro de un sistema por muy complejo que este sea, inclusive si estos aspectos se revuelven entre sí. La programación orientada a aspectos hace posible programar sistemas extremadamente complejos así como mejora significativamente la reutilización del software.

El construir sistemas complejos de software es una tarea extremadamente difícil debido a que nuestra mente no puede manejar tal grado de complejidad a la vez, más aún, casi siempre surgen nuevos requerimientos incrementando todavía más la complejidad con la que inevitablemente se tiene que tratar. Es por eso que la meta en la programación orientada a aspectos es permitir a los programadores desarrollar expresiones claras para trabajar con un sistema de una manera más fácil, se pretende ser capaz de escribir programas que limpiamente capturen todas las decisiones de diseño que los programadores hagan y que estas decisiones sean capturadas en una forma que les permita pensar y tratar en cada aspecto ya sea de manera conjunta o en una manera relativamente aislada.

La motivación para este trabajo de investigación nació de la inquietud de conocer uno de los avances de una de las áreas de desarrollo de los egresados de la carrera de *MAC* (Matemáticas Aplicadas y Computación) y con ello mostrar y difundir el nacimiento de una tecnología que esta en pleno desarrollo: la programación orientada a aspectos. que es muy prometedora y que si bien apenas esta empezando existen los suficientes trabajos de investigación para tomarla en cuenta.

Esta tesina pretende dar una visión general de la programación orientada a aspectos, sentando las bases sobre las que se apoya esta tecnología y definiendo los conceptos que maneja; sirviendo de punto de partida a los interesados que quieran introducirse en este campo. Se tratan además algunas de las formas de aplicar los conceptos orientados a aspectos por ejemplo utilizando el lenguaje *AspectJ* y se analiza un ejemplo de una pila tanto en las formas convencionales como con las orientadas a aspectos.

CAPITULO I

Filosofía y Fundamentos

Este capítulo expone el porqué y qué es la orientación a aspectos haciendo un recorrido por la evolución de los lenguajes de programación y sus paradigmas; se ve además el principio fundamental de *Dijkstra* en el cual se basa la orientación a aspectos; se clasifica y explica como surge la idea de todo esto, dando una breve historia a sus orígenes y se definen conceptos básicos como el de aspecto; se muestra de donde nace el fenómeno del código enmarañado; los aspectos no se quedan sólo en la implementación, sino que se llevan también a la fase de diseño, utilizando una extensión del modelo *UML*; una parte muy importante de la programación orientada a aspectos es que las clases y los aspectos se integran, se presentan dos enfoques para realizarlo.

1.1 Evolución de los lenguajes de programación

Un lenguaje de programación es un lenguaje intermedio entre la computadora y el lenguaje natural humano. El diseño y evolución de los lenguajes de programación es una de las áreas más importantes dentro de las ciencias de la computación. Los lenguajes de programación definen la manera en la cual nos comunicamos con nuestras máquinas, al darnos capas de abstracción con las cuales podemos interactuar con dichas máquinas.

Los lenguajes de programación han evolucionado tremendamente desde inicios de los cincuentas y esta evolución ha resultado en cientos de diferentes lenguajes utilizados por todos lados. Esta revolución es propiciada en parte por el gran desarrollo del hardware, el cual hasta ahora va por delante del software, con procesadores cada vez mejores, más rápidos y con una gran capacidad de almacenamiento que día a día aumenta cada vez más. La historia de los lenguajes de programación realmente empieza con el trabajo de *Charles Babbage* en los inicios del siglo XIX, quien desarrolló cálculos automatizados para funciones matemáticas.

Cualquier computadora sólo puede entender directamente su propio lenguaje máquina. El lenguaje máquina es el lenguaje natural de una computadora particular, esta relacionado con el diseño del hardware de cada computadora, y consiste de cadenas de números (al final reducidos a unos y ceros) que indican a las computadoras ejecutar sus operaciones más elementales, una a la vez. Los lenguajes máquina son difíciles de manejar por los seres humanos; no existían interpretes o compiladores para facilitar el trabajo. *Micro-code* es un ejemplo de lenguaje de primera generación residente en el microprocesador escrito para realizar multiplicaciones y divisiones; luego las computadoras fueron programadas en notación binaria, lo cual era muy propenso a errores; un algoritmo sencillo resultaba en una gran cantidad de código. Lo que obligo a aligerar el trabajo usando mnemónicos[1] para representar las operaciones. Los mnemónicos formaron la base de los lenguajes

[1] los programadores empezaron a usar abreviaturas similares al idioma inglés para representar las operaciones elementales de la computadora.

ensambladores, los cuales son lenguajes de traducción de los programas en lenguaje ensamblador a lenguaje máquina.

El código ensamblador simbólico llegó a mediados de los cincuentas, ejemplos de esto son: *AUTOCODER, SAP y SPS*. Las direcciones simbólicas permitieron a los programadores representar localidades de memoria, variables e instrucciones con nombres. Los programadores ya tenían la flexibilidad de no cambiar las direcciones por nuevas localidades de variables cuando las modificarán. Este tipo de programación es aún considerada rápida, aunque para programar en lenguaje máquina se requiere un alto grado de conocimiento del microprocesador y del conjunto de instrucciones de la máquina que se esta utilizando; además propicia una gran dependencia con el hardware y muy poca portabilidad. El código máquina y el código ensamblador no puede correr en diferentes máquinas. Por ejemplo, código escrito para un procesador de la familia *Intel* se ve completamente diferente a un código escrito para uno de la familia *Motorola*. El traslado significaría cambiar prácticamente todo el código.

Aunque los lenguajes ensamblador fue un gran avance, todavía se necesita de muchas instrucciones para llevar a cabo inclusive las tareas más sencillas. Para acelerar el proceso de programación, se desarrollaron los lenguajes de alto nivel y con ellos los compiladores: que son programas de traducción que convierten los programas de lenguaje de alto nivel al lenguaje máquina.

En el período comprendido entre 1960 y 1980 aparecieron los lenguajes de alto nivel. En los lenguajes de alto nivel se utilizaba algo conocido como programación estructurada, significa que el lenguaje contiene funciones definidas en procedimientos específicos para establecer como se va a hacer determinada tarea. La ventaja es la rapidez de desarrollo comparada con los lenguajes anteriores. La independencia con la máquina permitía que el código pudiera correr en diferentes arquitecturas. Otras ventajas de los lenguajes de alto nivel es el soporte para la abstracción de ideas, con lo cual los programadores pueden concentrarse en encontrar la solución al problema en vez de preocuparse por detalles de bajo nivel como la representación de datos. La facilidad de uso y aprendizaje, mejoró la portabilidad y simplifico la depuración, las modificaciones y el mantenimiento eran confiables y se tenían costos de software más bajos.

Aunque la sintaxis era diferente entre todos estos lenguajes, compartían construcciones similares y se leían y entendían mucho mejor comparados con los lenguajes ensambladores. Algunos lenguajes fueron mejorados a lo largo del tiempo y algunos fueron influenciados por lenguajes previos, tomando las características deseadas para hacer el lenguaje más poderoso.

Después llegó la orientación a objetos (*OO*) y con esto los lenguajes visuales orientados a eventos: las características de estos lenguajes son: amigables con el usuario, utilizables por no programadores, tienen opciones inteligentes por defecto acerca de lo que el usuario quiere y permiten al mismo obtener resultados rápidos usando el mínimo de código requerido, lo cual no es posible hacer con *COBOL , C o PL/I*. Ejemplos de estos lenguajes son *Visual Basic, Delphi, ADRS2, APL, Power Builder y Access*.

1.1.1 Paradigmas de Programación

Si se consulta cualquier diccionario en busca de la palabra paradigma se encuentra que significa: "ejemplo, modelo", pero para el mundo científico tiene un significado más amplio, se refiere, según *Thomas S. Khun* [1] a: "un desplazamiento de la conceptualización de entidades y, en consecuencia, de las soluciones que aplica a sus problemas". Ejemplos de estos paradigmas son: la tierra no es el centro del universo, la generación espontánea de vida no existe, la tierra no es plana, y recientemente que la programación orientada a objetos (*POO*) no es lo suficientemente poderosa y versátil para atacar ciertos problemas informáticos.
Basado en lo anterior *Peter Wegner* [2] clasifica los paradigmas de programación en: paradigmas declarativos y paradigmas imperativos.

1.1.1.1 Paradigmas Declarativos

Los paradigmas declarativos son aquellos donde un programa especifica una relación o función. Cuando se programa al estilo declarativo, no se hacen asignaciones a variables en el programa. El intérprete o compilador del lenguaje administra la memoria por el programador. Estos lenguajes son de más alto nivel que los lenguajes imperativos.
Dentro de los paradigmas declarativos encontramos los: orientados a funciones, orientados a la lógica y orientados a bases de datos.

1.1.1.1.1 Lenguajes Orientados a Funciones

Los matemáticos llevan un tiempo considerable resolviendo problemas usando el concepto de función: una función convierte ciertos datos en resultados y al saber cómo evaluar una función, usando la computadora, se pueden resolver automáticamente muchos problemas, y de allí que se generaran los lenguajes de programación funcionales, además, se aprovecha la posibilidad que tienen las funciones para manipular datos simbólicos, y no solamente numéricos, y la propiedad de las funciones que les permite componer, creando de esta manera, la oportunidad para resolver problemas complejos a partir de las soluciones de otros más sencillos, también incluyen la posibilidad de definir funciones recursivamente.
Un lenguaje funcional ofrece conceptos que son muy entendibles y relativamente fáciles de manejar. El lenguaje funcional más antiguo, y seguramente el más popular hasta la fecha, es *LISP*, diseñado por *McCarthy* [3] en la segunda mitad de los años 50's, donde su área de aplicación es principalmente la Inteligencia Artificial.
Programar en un lenguaje funcional significa construir funciones a partir de las ya existentes. Por lo tanto es importante conocer y comprender bien las funciones que conforman la base del lenguaje, así como las que ya fueron definidas previamente. De esta manera se pueden ir construyendo aplicaciones cada vez más complejas. La desventaja de este modelo es que resulta bastante alejado del modelo de la máquina de *von Neumann*[2] y,

[2] el cual se describirá en el paradigma imperativo

por lo tanto, la eficiencia de ejecución de los intérpretes en los lenguajes funcionales no es comparable con la ejecución de los programas imperativos precompilados.
En estos lenguajes en vez de examinar la serie de estados a través de los cuales debe pasar la máquina para obtener una respuesta, la pregunta que se debe hacer es: ¿cuál es la combinación de funciones que se debe aplicar a los datos iniciales para obtener los resultados esperados?: en vez de examinar los estados sucesivos de cómputo se analizan las transformaciones funcionales sucesivas que se deben aplicar a los datos para llegar a la respuesta. en general, la sintaxis de esta clase de lenguajes es similar a:

función(...función2(función1(datos))...)

1.1.1.1.2 Lenguajes Orientados a la Lógica

Otra forma de razonar para resolver problemas en matemáticas se fundamenta en la lógica. El conocimiento básico de las matemáticas se puede representar en la lógica en forma de axiomas, a los cuales se añaden reglas formales para deducir cosas verdaderas (teoremas) a partir de los axiomas. Gracias al trabajo de algunos matemáticos se encontró la manera de automatizar computacionalmente el razonamiento lógico que permitió que la lógica matemática diera origen a otro tipo de lenguajes de programación, conocidos como lenguajes lógicos.

En los lenguajes lógicos se utiliza el formalismo de la lógica para representar el conocimiento sobre un problema y para hacer preguntas que, si se demuestra que se pueden deducir a partir del conocimiento dado en forma de axiomas y de las reglas de deducción estipuladas, se vuelven teoremas. Así se encuentran soluciones a problemas formulados como preguntas. Con base en la información expresada dentro de la lógica, se formulan las preguntas sobre el dominio del problema y el intérprete del lenguaje lógico trata de encontrar la respuesta automáticamente. El conocimiento sobre el problema se expresa en forma de predicados (axiomas) que establecen relaciones sobre los símbolos que representan los datos del dominio del problema.

PROLOG es el primer lenguaje lógico y el más conocido y utilizado. Sus orígenes se remontan a los inicios de la década de los 70's con los trabajos del grupo de *A. Colmerauer* [4] en Marsella, Francia. También en este caso, las aplicaciones a la Inteligencia Artificial mantienen el lenguaje vivo y útil.

En el caso de la programación lógica, el trabajo del programador se restringe a la buena descripción del problema en forma de hechos y reglas. A partir de ésta se pueden encontrar muchas soluciones dependiendo de como se formulen las preguntas (metas), que tienen sentido para el problema. Si el programa está bien definido, el sistema encuentra automáticamente las respuestas a las preguntas formuladas. En este caso ya no es necesario definir el algoritmo de solución, como en la programación imperativa, en cambio, lo fundamental aquí es expresar bien el conocimiento sobre el problema mismo. En la programación lógica, al igual que en programación funcional, el programa, en este caso los hechos y las reglas, al igual que los anteriores, están muy alejados del modelo de *von Neumann* que posee la máquina en la que tienen que ser interpretados: por lo tanto, la eficiencia de la ejecución no puede ser comparable con la de un programa equivalente escrito en un lenguaje imperativo. Sin embargo, para cierto tipo de problemas, la

formulación del programa mismo puede ser mucho más sencilla y natural (para un programador experimentado).

En estos lenguajes basados en reglas se ejecutan verificando la presencia de una cierta condición habilitadora. que. cuando se satisface. ejecuta una acción apropiada. Las condiciones habilitadoras son expresiones lógicas de predicados. La ejecución de este tipo de lenguajes es similar a la de un lenguaje imperativo. excepto que las instrucciones no son secuenciales. Las condiciones habilitadoras determinan el orden de ejecución. En general la sintaxis es similar a:

```
Condición habilitadora_1 -> acción_1
Condición habilitadora_2 -> acción_2
        :                    :          :
Condición habilitadora_n -> acción_n
```

Herramientas como *YACC* (Yet Another Compiler Compiler: todavía otro compilador de compiladores) para el análisis sintáctico de programas. usa reglas que usan la sintaxis formal como condición. Otra herramienta que usa reglas son los *Shells* de sistemas expertos basados en reglas como *NEXPERT. LEVEL-5*. que proporcionan algoritmos para realizar la máquina de inferencias de sistemas expertos.

1.1.1.1.3 Lenguajes Orientados a Bases de Datos.

Los lenguajes orientados a bases de datos tienen propiedades que los distinguen y son: la persistencia[1] y la administración de cambios. Las entidades de bases de datos no desaparecen después de que finaliza un programa. sino que permanecen activas por tiempo indefinido como fueron estructuradas originalmente.

Un sistema de administración de bases de datos incluyen un lenguaje de definición de datos (*DDL*) para describir una nueva colección de hechos o datos y un lenguaje de manipulación de datos(*DML*) para la interacción con las bases de datos existentes. El ejemplo más conocido es *SQL*.

1.1.1.2 Paradigmas Imperativos

Los paradigmas imperativos son lenguajes controlados por comandos u orientados a instrucciones. Un programa se compone de una secuencia de instrucciones. y la ejecución de cada una de éstas hace que el intérprete o compilador cambie el valor de una o más ubicaciones de memoria. es decir que cambie de estado. La sintaxis de esta clase de lenguajes tiene por lo general la forma:

```
instrucción 1;
instrucción 2;
      :
instrucción n;
```

[1] es persistente en el sentido que tanto sus entidades como las relaciones entre ellas son preservadas de un uso al siguiente

El desarrollo de programas consiste en construir los estados sucesivos que se necesitan para llegar a la solución. En este tipo de lenguajes, cuyo origen está ligado a la propia arquitectura de *von Neumann*, la arquitectura consta de una secuencia de celdas, llamadas memoria, en la cual se pueden guardar en forma codificada, lo mismo datos que instrucciones; y de un procesador, el cual es capaz de ejecutar de manera secuencial una serie de operaciones, principalmente aritméticas y booleanas, llamadas comandos. En general, un lenguaje imperativo ofrece al programador conceptos que se traducen de forma natural al modelo de la máquina.
El programador, al utilizar un lenguaje imperativo, por lo general tiene que traducir la solución abstracta del problema a términos muy primitivos, cercanos a la máquina. La distancia entre el nivel del razonamiento humano y lo expresable por los lenguajes imperativos causa que sus programas sean más "comprensibles" para la máquina que para el hombre. Esta desventaja para nosotros, reflejada en la dificultad que tenemos al construir programas en un lenguaje imperativo, se vuelve una ventaja en el momento de la generación del código. El programa está expresado en términos tan cercanos a la máquina, que el código generado es relativamente parecido al programa original, lo que permite cierta eficiencia en la ejecución.
El paradigma imperativo disfruta del hecho de que las máquinas de *Turing* son imperativas, y de que las implementaciones *hardware* de las computadoras modernas (que son equivalentes a *LBA*[4]) también son de naturaleza imperativa.
Dentro de los paradigmas imperativos encontramos: los orientados a procedimientos, los orientados a objetos, los orientados al procesamiento concurrente y recientemente los orientados a aspectos.

1.1.1.2.1 Lenguajes Orientados a Procedimientos

Los principios de diseño en programación empezaron a emerger en la década de los 70's como resultado de la crisis creada por el incremento de la complejidad en el código. Este incremento de la complejidad, combinada con la creciente necesidad por mantener y evolucionar los sistemas llevó al concepto de programación estructurada.
La programación estructurada se concentra en una mejor organización del proceso de desarrollo del sistema para alcanzar objetivos como: simplicidad, comprensión, verificación, mantenimiento y modificación; y se refiere a bloques anidados, los cuales pueden estar anidados dentro de otros bloques, y además tener sus propias variables. El estado representa una pila con una referencia al bloque actualmente activo en la parte superior. En los lenguajes estructurados, el procedimiento es el principal bloque de construcción de los programas. Ejemplos de estos lenguajes son: *Ada, ALGOL 60, Pascal, y C*.
La programación estructurada fue sólo el inicio de la revolución del software. Los requerimientos de programación continuaron siendo cada vez más complejos lo cual implicaba el tener más programadores sobre un mismo sistema. Temas como la reutilización del software vinieron a ser relevantes, ya que la gente se comenzó a dar cuenta que estaba reinventando la rueda cada vez que creaban un nuevo programa. La mayoría de los programadores ya habían reutilizado su propio software, pero esto no era suficiente ya

[4] *linear-bounded automaton* lo que significa autómatas linealmente acotados o limitados

que los programas sobrevivían a sus creadores. *Fred Brooks* [5] establece: "*The best way to attack the essence of building software is not to build it at all*", traducido significa, la mejor manera de crear software es no crearlo en sí.

La idea de modularidad pareció ser un buen punto de inicio para la reutilización de software. La modularidad, introducida junto con la programación estructurada, dio al programador interfaces claras para las unidades funcionales de un programa. Inclusive programadores que no estaban inmersos en el desarrollo de esas unidades podían reutilizarlas muchas veces sin modificación alguna, siempre y cuando la interfaz estuviera bien definida. La implementación de un módulo estaba abierta y por lo tanto sujeta a cambios a gusto y placer del programador, creando aún más problemas al mantenimiento del código, y propiciando que surgiera la idea de caja negra, donde si la implementación estaba encapsulada en el módulo y los programadores podían sólo ver la interfaz, la implementación podría entonces permanecer constante.

Pero, ¿qué pasaba cuando un programador no quisiera exactamente lo que le ofrece un módulo, sino alguna variante de este con una leve diferencia en funcionalidad? el programador tendría que modificar totalmente tal módulo. Esta necesidad dio lugar a la introducción de la herencia, la cual permitió a los programadores extender o limitar la funcionalidad de un módulo sin redefinir el módulo original. Estas ideas fueron combinadas y evolucionaron en la concepción moderna de la *OO*.

1.1.1.2.2 Lenguajes Orientados a Objetos

A mediados de los años 60's se empezó a vislumbrar el uso de las computadoras para la simulación de problemas del mundo real. Pero el mundo real está lleno de objetos, en la mayoría de los casos, complejos, los cuales difícilmente se traducen a los tipos de datos primitivos de los lenguajes anteriores a los *OO*. Así es que a dos noruegos, *Dahl* y *Nygaard* [6], se les ocurrió el concepto de objeto y sus colecciones, llamadas clases de objetos, que permitieron introducir abstracciones de datos a los lenguajes de programación. La posibilidad de reutilización del código y sus indispensables modificaciones, se reflejaron en la idea de las jerarquías de herencia de clases. A ellos también se les debe el concepto de polimorfismo introducido vía procedimientos virtuales.

La comunidad informática ha tardado demasiado en entender la utilidad de los conceptos básicos de lo que hoy identificamos como conceptos del modelo de objetos. Se tuvo que esperar hasta los años 80's para vivir una verdadera ola de propuestas de lenguajes de programación con conceptos de objetos encabezada por *Smalltalk*, *C++*, *Eiffel*, *Modula-3*, *Ada 95* y terminando con *Java*. La moda de objetos se ha extendido de los lenguajes de programación a la Ingeniería de Software.

El modelo de objetos, y los lenguajes que lo usan, parecen facilitar la construcción de sistemas o programas en forma modular. Los objetos ayudan a expresar programas en términos de abstracciones del mundo real, lo que aumenta su comprensión. La clase ofrece cierto tipo de modularización que facilita las modificaciones al sistema. La reutilización de clases previamente probadas en distintos sistemas también es otro punto a favor; sin embargo, el modelo de objetos, a la hora de ser interpretado en la arquitectura de *von Neumann* conlleva un excesivo manejo dinámico de memoria debido a la constante creación de objetos, así como a una carga de código fuente causada por la constante invocación de métodos, por lo tanto, los programas en lenguajes orientados a objetos

siempre pierden en eficiencia, en tiempo y memoria, contra los programas equivalentes en lenguajes anteriores a la *OO*. Para consolarnos, los expertos dicen que les ganan en la comprensión de código.

1.1.1.2.3 Lenguajes Orientados al Procesamiento Concurrente

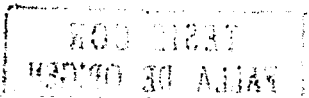
La necesidad de ofrecer concurrencia en el acceso a los recursos computacionales se remonta a los primeros sistemas operativos. Por ejemplo mientras que un programa realizaba una operación de entrada o salida otro podría gozar del tiempo del procesador para sumar dos números. Aprovechar al máximo los recursos computacionales fue una necesidad apremiante, sobre todo en la época en que las computadoras eran caras y escasas; el sistema operativo tenía que ofrecer la ejecución concurrente y segura de programas de varios usuarios, que desde distintas terminales utilizaban un sólo procesador, y así surgió la necesidad de introducir algunos conceptos de programación concurrente para programar los sistemas operativos.

Posteriormente, cuando los procesadores cambiaron de tamaño y de precio, se abrió la posibilidad de contar con varios procesadores en una máquina y ofrecer el procesamiento en paralelo, es decir, procesar varios programas al mismo tiempo. Esto le dio impulso a la creación de lenguajes que permitían expresar el paralelismo. Finalmente, llegaron las redes de computadoras, que también ofrecen la posibilidad de ejecución en paralelo, pero con procesadores distantes, lo cual conocemos como la programación distribuida.

En resumen, el origen de los conceptos para el manejo de concurrencia, paralelismo y distribución está en el deseo de aprovechar al máximo la arquitectura de *von Neumann* y sus modalidades reflejadas en conexiones paralelas y distribuidas.

Por ejemplo se pueden encontrar algunas soluciones conceptuales y mecanismos tales como: semáforos, regiones críticas, monitores, envío de mensajes (CSP), llamadas a procedimientos remotos (RPC), que posteriormente se incluyeron como partes de los lenguajes de programación en *Concurrent Pascal*, *Modula*, *Ada*, *OCCAM*, y últimamente en *Java*. Un ejemplo de un lenguaje de programación muy utilizado dentro del computo distribuido es PVM (Parallel Virtual Machine) y para computo paralelo se tiene el lenguaje MPI (Message Passing Interface).

Es difícil evaluar las propuestas existentes de lenguajes para la programación concurrente, paralela y distribuida. Primero, porque los programadores están acostumbrados a la programación secuencial y cualquier uso de estos mecanismos les dificulta la construcción y el análisis de programas. Por otro lado, este tipo de conceptos en el pasado se manejaba principalmente a nivel de sistemas operativos, protocolos de comunicación, etc. donde la eficiencia era crucial, y por lo tanto no se utilizaban lenguajes de alto nivel para la programación. Hoy en día, la programación de sistemas complejos tiene que incluir las partes de comunicaciones, programación distribuida y concurrencia. Esto lo saben los creadores de los lenguajes más recientes, que integran conceptos para manejar: los hilos de control, comunicación, sincronización y no determinismo: el hardware y las aplicaciones se los exigen.



1.1.1.2.4 Lenguajes Orientados a Aspectos

La orientación a aspectos (*OA*, tema de esta tesina) es un paso más en la ingeniería de software por crear cada vez más un software de mejor calidad, atacando los problemas de los ámbitos diseminados y el enmarañamiento del código: al encapsular los ámbitos diseminados se garantiza una mejor evolución, mantenimiento, y comprensión de los sistemas informáticos. Existen varias técnicas para lograr lo anterior como: el modelo de objetos de los filtros de integración, la programación subjetiva de IBM, los hiperespacios en la descomposición multidimensional, AOP (*Aspect-Oriented Programming*) de *Xerox PARC*, la programación adaptativa del grupo *DEMETER*, etc.
Ejemplos de los lenguajes hasta ahora utilizados y todavía en constante desarrollo están: AspectJ (Java), Sina (Smalltalk), HyperJ (Java), AspectR (Ruby), AspectS (Squeak/Smalltalk), JAC (Java), AspectC# (C#), Jade (Java), Phytius (Phyton), etc.

1.2 Descomposición en Ámbitos

¿Cuál es la parte central o más importante en el desarrollo de los lenguajes de programación?, es aquello que *Dijkstra* [7] estableció como el principio fundamental de descomposición en ámbitos o competencias. El cual establece la necesidad de tratar con tan sólo una parte importante a la vez, es decir, la descomposición de un sistema complejo en partes que sean fáciles de manejar. Desafortunadamente, mientras el principio expresa una importante cualidad del código y del proceso de desarrollo, no dice como lograrlo.
El principio puede ser aplicado en varias formas: por ejemplo la descomposición en fases en el proceso de ingeniería de software puede ser visto como la descomposición en actividades de ingeniería en tiempo y metas, también la definición de subsistemas, objetos y componentes puede ser visto como la aplicación de este mismo principio. No es exagerado establecer que este es un principio de ingeniería de software que está presente en todos lados.

1.2.1 Ámbitos

A pesar del acuerdo común de la necesidad de aplicar el principio de descomposición en ámbitos, no esta del todo establecido y comprendido la noción de ámbito. Por ejemplo en los métodos orientados a objetos los ámbitos son modelados como objetos o clases, los cuales son generalmente derivados de las entidades en la especificación de requerimientos. En métodos de programación estructurada, los ámbitos son representados como procedimientos. En programación orientada a aspectos, se extiende el término ámbito con lo que es llamado propiedades no funcionales como por ejemplo la sincronización de procesos. En un sentido uno puede pensar esto como una generalización de la noción de ámbito en el contexto de lenguajes de programación. Aunque se considere esto como un desarrollo natural, este incrementa la necesidad de renovar el concepto de lo que son los ámbitos porque estos ya no están restringidos a sólo objetos y funciones. Más aún la tarea de separar los ámbitos correctos o adecuados es complicada porque se tiene que tratar cada vez con un conjunto más grande y más variado de estos.

1.2.2 Los Ámbitos desde la Perspectiva Problema-Solución

Para discutir la noción de ámbito, se debe considerar al desarrollo de software como un proceso de problema-solución en el cual se alcanza una solución de software para ciertos requerimientos dados. Típicamente, esto puede ser expresado como:

$$R \rightarrow S$$

Donde la R representa los requerimientos, S la solución y la $\rightarrow$ el proceso de transformación. Por ejemplo para el diseño de un sistema distribuido los requerimientos R podrían ser la preservación de la consistencia bajo la presencia de fallas. La solución S podría ser representada por un protocolo de recuperación de fallas.
Se asume generalmente que los requerimientos R incluyen la especificación de los problemas relevantes. En la práctica encontrar los problemas exactos y expresarlos en la forma adecuada no es una tarea fácil. Para simplificar esto, valdría la pena tener una fase intermedia en la cual se abstraigan los problemas relevantes usando las especificaciones de los requerimientos iniciales:

$$R \rightarrow P \rightarrow S$$

Donde P representa el problema real para el cual se busca una solución de software. En el ejemplo de recuperación de fallas, P podría representar el problema de diseñar un algoritmo de recuperación de fallas transparente.
Si vemos el modelo S con más detalle. Básicamente, se puede definir S como un conjunto de abstracciones y relaciones que cumplen con un rol específico.

$$S = (SA, SR)$$

Aquí SA representa el conjunto de abstracciones (sa_1, sa_2, sa_3, ... sa_n) y SR representa el conjunto de relaciones entre estas abstracciones(sr_1, sr_2, sr_3, ..., sr_n). En teoría, para un problema P dado, existen muchas soluciones posibles. La calidad de las soluciones individuales es proporcional al grado en el cual estas alcanzan las siguientes propiedades:

1. Propiedad de Eficacia
   La solución S requiere cumplir la meta y las restricciones del problema P. Esto es S debe ser eficaz, y tan válido para el contexto que fue definido por P.
2. Propiedad de Forma Canónica
   La solución S debe incluir las abstracciones suficientes y necesarias que permitan una solución eficaz. Más aún, esta no debe incluir abstracciones irrelevantes y/o redundantes. En otras palabras, S debe ser genérica y concisa.

La propiedad de eficacia asegura que se busca la solución correcta para el problema adecuado. La propiedad de forma canónica inherentemente reduce la complejidad innecesaria.
Dado este modelo de desarrollo de software se define un ámbito como: una abstracción de una solución canónica que es eficaz para un problema dado.
Esta definición implica que los ámbitos no son absolutos sino relativos con respecto al problema considerado. Lo que puede ser un ámbito para un problema puede no serlo para otro problema distinto.
Ya que un ámbito esta definido como una abstracción entonces este representa un conjunto de posibles instancias de ese ámbito. Esto implica que se pueden derivar diferentes alternativas de un modelo de solución dado. Para explicar más explícitamente el conjunto de alternativas de un modelo de solución se presenta el siguiente conjunto de definiciones junto con la *figura 1.2.2.1*

Espacio de diseño: una representación multidimensional que representa el conjunto de soluciones alternativas para un problema dado.
Dimensiones: representaciones ortogonales de ámbitos abarcando un espacio de diseño de una abstracción de solución.
Instancia: un elemento en una dimensión.
Alternativa: una combinación de una selección de instancias de ámbitos en un modelo de solución. Las alternativas son representadas como puntos en el espacio de diseño. Un punto es un conjunto de elementos instanciados de las dimensiones individuales.

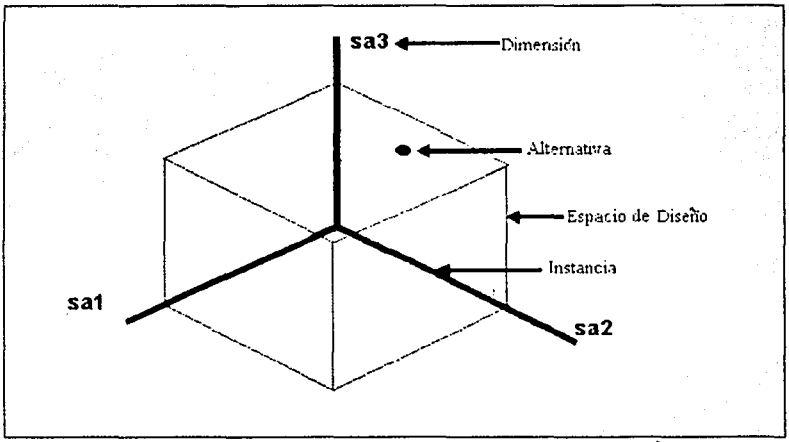


Figura 1 2 2.1 . Una visualización de un espacio de diseño en 3D para una abstracción de solución SA = (sa_1, sa_2, sa_3)

1.2.3 Expresando Ámbitos

Para implementar la fase de diseño se tienen que expresar los ámbitos en un lenguaje determinado. Para dar ciertos factores de calidad, como adaptabilidad y reutilización se requiere que el lenguaje adoptado siga lo siguiente:

- Represente ámbitos como entidades de clase base.
- Proporcione operadores de integración.

La primera propiedad de clase base significa que los ámbitos expresados pueden ser representados y manipulados independientemente, la segunda que el lenguaje debe proporcionar un conjunto de operadores para integrar las entidades de clase base.

La propiedad de clase base de los ámbitos ayuda a incrementar los factores de calidad ya que los ámbitos se localizan y se representan de forma separada. Esto significa que dada una solución con un conjunto de abstracciones definida por SA = (sa_1, sa_2, sa_3, ..., sa_n), se

puede definir un modelo de abstracción AI (Abstracciones de Implementación) en un lenguaje de forma que :

$$AI = (a_{i1}, a_{i2}, a_{i3}, ..., a_{in})$$

Donde hay un mapeo de SA a AI, esto quiere decir que, cada abstracción en SA, puede ser representada de manera separada por una abstracción en AI. Tipos similares de mapeos podrían ser requeridos por el conjunto de relaciones semánticas definidas por RI (Relaciones de Implementación).

Todos estos requerimientos juntos llevan a un modelo de implementación I = (AI, RI), que forma un modelo con una estructura común de la abstracción de solución. Se obtiene tal solución cuando la estructura de los ámbitos en el modelo de solución se mantiene en la transformación.

Para cada problema técnico individual se puede generar una solución ideal que requiera un lenguaje específico en el cual todos los ámbitos pueden ser fácilmente representados y la estructura es conservada de manera óptima. En la práctica, existe un conjunto inmenso de problemas técnicos que requieren diferentes modelos de solución con diferentes ámbitos. Dado que el tiempo y los recursos son restringidos en los proyectos de software, generalmente esto no es una opción y usualmente en su lugar se utiliza un lenguaje de propósito general ya existente.

Para un problema dado, algunos lenguajes son mejores en la descomposición en ámbitos, y conservan la estructura del modelo de solución, mientras que otros lenguajes son menos efectivos en esta cuestión. En el caso de que el lenguaje no sea lo suficiente expresivo para ya sea (1) representar los ámbitos de forma separada y/o (2) proporcionar los apropiados operadores de integración, se dice que la transformación del modelo de solución al modelo de implementación es anómalo y esta situación se denota como anomalía de la descomposición en ámbitos.

1.2.4 Anomalía de la Descomposición en Ámbitos

Esta anomalía se presenta en la situación en la cual un modelo de solución es transformado hacia un modelo de implementación y no se conserva la estructura de los ámbitos del modelo de solución. Básicamente para un lenguaje y un problema dados, la anomalía de la descomposición en ámbitos puede ocurrir en dos formas :

- Ámbitos que no se pueden separar

Los ámbitos del problema no pueden ser representados como entidades de clase base en el lenguaje adoptado. Aquí los ámbitos originales del modelo de solución son esparcidos sobre muchas implementaciones de ámbitos. Generando así los ámbitos diseminados[5].
Se puede representar esto como : $sa_i \rightarrow \{ia_j, ia_k, ia_l, ...\}$, donde el ámbito sa_i en el modelo de solución del dominio esta representado por más de una entidad de clase base $\{ia_j, ia_k, ia_l, ...\}$.

- Falta de operadores de integración adecuados

Los operadores de integración definidos por el lenguaje no pueden manipular apropiadamente las entidades de clase base que representan los ámbitos separados. A esto se le llama anomalía de composición o integración.
Si por ejemplo tenemos que $sa_3 = sa_2 + sa_1$ donde los ámbitos sa_1 y sa_2 se componen conjuntamente y producen la abstracción de la solución sa_3 usando el operador de

[5] se puede encontrar en la literatura como cross-cutting concerns y se verá a detalle en las siguientes secciones

integración +. Habrá una anomalía de manipulación si no se puede definir un operador de integración en el lenguaje de implementación para realizar $ia_3 = ia_2 + ia_1$.

1.3 Generaciones de Lenguajes de Programación

En las primeras fases del desarrollo de los lenguajes de programación se tenía un código en el que no había separación de conceptos, los datos y la funcionalidad se integraban sin una línea divisoria clara, a lo que se le conoce comúnmente como *código spaghetti*, ya que se tenía una gran maraña entre datos y funcionalidad que recuerda a la que se forma cuando se come un plato de esta pasta italiana.
En la siguiente etapa se aplicó la llamada *descomposición funcional*, aplicando el principio fundamental de *Dijkstra*, visto anteriormente, identificando las partes más manejables como funciones que se definen en el dominio del problema. La principal ventaja que proporciona esta descomposición es la facilidad de integración de nuevas funciones, aunque también tiene grandes inconvenientes, como son el hecho de que las funciones quedan algunas veces poco claras debido a la utilización de datos compartidos, y el que los datos quedan esparcidos por todo el código, con lo cual, normalmente el integrar un nuevo tipo de datos implica que se tengan que modificar varias funciones para adecuarlas a las necesidades. Intentando resolver estas desventajas con respecto a los datos, se dio otro paso en el desarrollo de los sistemas de software. La *Programación Orientada a Objetos (POO)* ha supuesto uno de los avances más importantes de los últimos años en la ingeniería de software para construir sistemas complejos utilizando el mismo principio de descomposición, ya que el modelo de objetos se ajusta mejor a los problemas del dominio real que la descomposición funcional. La ventaja que tiene es que es fácil la integración de nuevos datos.
Uno de los principales inconvenientes con el que nos encontramos al aplicar estas descomposiciones ya tradicionales es que muchas veces se tienen ejecuciones ineficientes debido a que las unidades de descomposición no siempre van acompañadas de un buen tratamiento de aspectos tales como el manejo de memoria, la coordinación, la distribución, las restricciones de tiempo real, la sincronización, la distribución, el manejo de errores, la optimización de la memoria, el manejo de la seguridad, etc. Mientras que las descomposiciones funcional y orientada a objetos no nos plantean ningún problema con respecto al diseño y la implementación de la funcionalidad básica, estas técnicas no se comportan bien con los otros aspectos; es decir, que nos encontramos con problemas de programación en los cuales ni las técnicas funcionales, ni las orientadas a objetos son suficientes para capturar todas las decisiones de diseño que el programa debe implementar. Con las descomposiciones tradicionales no se separan bien estos otros aspectos, sino que quedan diseminados por todo el sistema enmarañando el código que implementa la funcionalidad básica, y va en contra de la claridad del mismo. Se puede afirmar entonces que las técnicas tradicionales no soportan bien la descomposición en ámbitos para aspectos distintos de la funcionalidad básica de un sistema, y que esta situación claramente tiene un impacto negativo en la calidad del software.
La *programación orientada a aspectos (POA)* es una nueva metodología de programación que aspira a soportar la descomposición en ámbitos para los aspectos antes mencionados.

intenta separar los componentes y los aspectos unos de otros, proporcionando mecanismos que hagan posible abstraerlos e integrarlos para formar todo el sistema En definitiva, lo que se persigue es implementar una aplicación de forma eficiente y fácil de entender.
Por último, vista la evolución de los lenguajes de programación, y comprobando que las formas de descomponer los sistemas han conducido a nuevas generaciones de sistemas, cabe plantearse la siguiente pregunta ¿estaremos en la etapa naciente de una nueva generación de sistemas de software?

1.4 Orígenes de la Programación Orientada a Aspectos

El concepto de programación orientada a aspectos fue introducido por *Gregor Kiczales*[6] y su grupo, aunque el equipo *Demeter*[7] había estado utilizando ideas orientadas a aspectos antes incluso de que se estableciera el término.
No fue hasta 1995 cuando se publicó la primera definición temprana del concepto de aspecto, realizada por el grupo *Demeter*, la cual se verá más adelante.
Gracias a la colaboración de *Cristina Lopes* y *Karl J. Lieberherr* junto con *Gregor Kiczales* y su grupo se introdujo el término de programación orientada a aspectos ya como tal.
Entre los objetivos que se ha propuesto la programación orientada a aspectos están principalmente el de la descomposición en ámbitos y el de minimizar las dependencias entre ellos. Con el primer objetivo se consigue que cada cosa esté en su sitio, es decir, que cada decisión se tome en un lugar concreto, con el segundo se tiene una pérdida del acoplamiento entre los distintos elementos.
De la consecución de estos objetivos se pueden obtener las siguientes ventajas:

- Un código menos enmarañado, más natural y más reducido.
- Una mayor facilidad para razonar sobre los ámbitos, ya que están separados y tienen una dependencia mínima.
- Más facilidad para depurar y hacer modificaciones en el código.
- Se consigue que un conjunto grande de modificaciones en la definición de un ámbito tenga un impacto mínimo en los otros.
- Se tiene un código más reutilizable y que se puede acoplar y desacoplar cuando sea necesario.

La POA proporciona métodos y técnicas para descomponer problemas dentro de un número de componentes funcionales así como un número de aspectos los cuales se diseminan por los componentes funcionales y luego integra estos componentes y aspectos para obtener las implementaciones de los sistemas.
Para alcanzar todo lo anterior: se necesitan métodos concretos y técnicas. *K. Czarnecki* establece que existen tres factores principales:

- ¿Cuales son esas cuestiones importantes que necesitamos separar?, lo que estamos buscando son conjuntos de ámbitos que puedan ser reutilizados para ser usados en la descomposición de problemas; algunos ámbitos serán más específicos a una aplicación,

[6] es el principal científico en Xerox Palo Alto Research Center, sus intereses de investigación son la arquitectura de software, los lenguajes de programación e ingeniería de software; su trabajo en técnicas para capturar mejor la estructura de los sistemas de software complejos incluyen los protocolos de meta-objetos, implementación abierta y más recientemente la POA. En particular el es uno de los diseñadores de CLOS (Common-Lisp Object System) y co-autor del libro "The Art of The Metaobject Protocol".
[7] es un proyecto de investigación en Northeastern University encabezado por el Dr. Karl Lieberherr.

por ejemplo productos de finanzas o la configuración de los servicios de red: otros ámbitos serán más generales por ejemplo la sincronización: algunos de los ámbitos (tanto los de aplicación específica como los generales) se convertirán eventualmente en aspectos.

- ¿Qué otros mecanismos aparte de los convencionales se pueden utilizar para realizar la descomposición?. si todos los aspectos no pueden ser encapsulados y manipulados usando mecanismos convencionales, se necesitan encontrar otros mecanismos de manipulación. La característica de tales mecanismos será alcanzar un acoplamiento amplio entre descripciones parciales. Estaría bien además que esos mecanismos soportaran una adaptabilidad no invasiva por ejemplo por integración y transformación más que por un cambio al código original.
- ¿Cómo capturamos los aspectos en sí mismos?. los aspectos pueden expresarse usando algunos medios apropiados dentro de los lenguajes. en los casos más simples, se pueden usar librerías de clases convencionales: en otros casos usando lenguajes especializados o extensiones de los lenguajes ya existentes. Cada enfoque tiene sus ventajas y desventajas.

1.4.1 Ámbitos Diseminados

Un gran problema, según *Kiczales* y otros investigadores, es el de la existencia de ámbitos diseminados. Estos son ámbitos que no pueden ser restringidos o encapsulados dentro de una forma modular. Los ámbitos diseminados destruyen la modularidad por la cual tanto se lucha en los programas *OO*, introducen código relacionado e inclusive duplicado dentro de uno o más módulos.
Los ámbitos diseminados existen en los programas, y su representación puede causar problemas con el mantenimiento y la eficiencia. En algunas situaciones, rediseñar el sistema puede transformar un ámbito diseminado en un objeto, así el ámbito diseminado ya no se esparce más en el código, sino que es claramente encapsulado en un módulo. Uno puede estar tentado en pensar que este método de rehacer o rediseñar los módulos podría eliminar este problema completamente. *Kiczales* [9]afirma, aunque no lo prueba, que existen algunas situaciones que invariablemente no importa como rediseñes el sistema, siempre existirán algunos ámbitos diseminados. Aunque no existe una prueba formal que muestre esto, parece intuitivo que es verdad.

1.4.2 ¿Qué es un Aspecto?

Una de las primeras definiciones que aparecieron del concepto de aspecto fue publicada en 1995, y se describía de la siguiente manera: *Un aspecto es una unidad que se define en términos de información parcial de otras unidades*.
La definición de aspecto ha evolucionado a lo largo del tiempo, pero con la que se trabaja actualmente es la siguiente: "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 programa o de código es una unidad modular del programa que aparece en otras unidades modulares del programa" (*traducción propia de la definición de G. Kiczales*).
De manera más informal podemos decir que los aspectos son la unidad básica de la *POA*, y pueden definirse como las partes de una aplicación que describen las cuestiones claves

relacionadas con la forma de realizar algo. También pueden verse como los elementos que se diseminan por todo el código y que son difíciles de describir localmente con respecto a otros componentes.
Se puede diferenciar entre un componente y un aspecto viendo al primero como aquella propiedad que se puede encapsular claramente en un procedimiento, mientras que un aspecto no se puede encapsular en un procedimiento con los lenguajes tradicionales.
Los aspectos no suelen ser unidades de descomposición funcional del sistema, sino propiedades del sistema.
Algunos ejemplos de aspectos son, los patrones de acceso a memoria, la sincronización de procesos concurrentes, el manejo de errores, etc.
En la *Figura 1.5.3.1*, se muestra un programa como un todo formado por un conjunto de aspectos más un modelo de objetos. Con el modelo de objetos se recoge la funcionalidad básica, mientras que el resto, los aspectos, recogen características de actuación y otras no relacionadas con la funcionalidad esencial del mismo.
Teniendo esto en cuenta, podemos realizar una comparación de la apariencia que presenta un programa tradicional con lo que muestra un programa orientado a aspectos *Figura 1.5.3.2*.

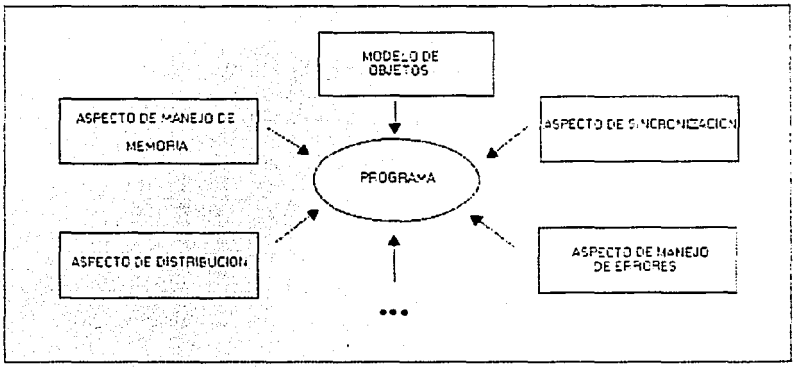


Figura 1.5.3.1 Estructura de un Programa Orientado a Aspectos

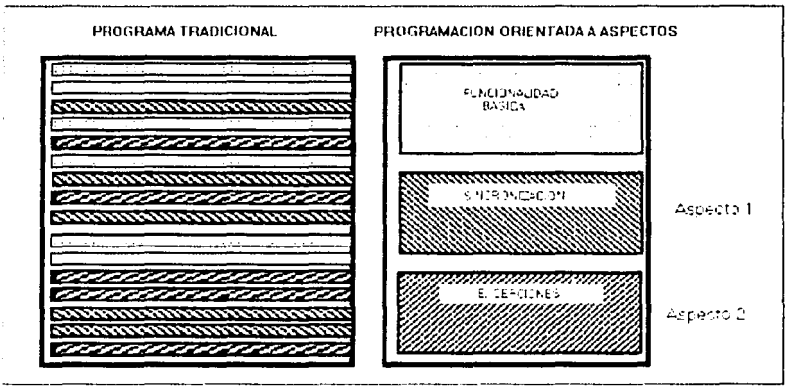


Figura 1.5.3.2 Comparacion entre la forma de un programa tradicional con uno orientado a aspectos

Si el rectangulo de apariencia más gruesa representa el código de un programa, se puede apreciar que la estructura del programa tradicional esta formada por una serie de rectangulos horizontales. Cada rectangulo esta relleno utilizando una apariencia distinta(por la figura que tiene dentro), y cada apariencia representa una funcionalidad.
El programa orientado a aspectos, sin embargo, esta formado por tres bloques compactos, cada uno de los cuales representa un aspecto dentro del código.
Como se puede observar, en la versión tradicional estos mismos bloques de funcionalidad quedan repartidos por todo el código, mientras que en la version orientada a aspectos tenemos un programa más compacto y modularizado, teniendo cada aspecto su propia área de competencia.

1.4.3 Código Enmarañado

Si los ámbitos diseminados no son manejados apropiadamente estos eventualmente causaran estragos en el mantenimiento del código.
Si uno de estos ámbitos necesita ser cambiado en algún punto del ciclo de vida del sistema, se tendrán que modificar muchos módulos y algunas veces todos.
Diseñar en base a la descomposición funcional frecuentemente lleva a programas que no satisfacen adecuadamente uno o más aspectos. Estos programas son arreglados a mano invadiendo el código original, para satisfacer los aspectos y al hacer esto el código frecuentemente se vuelve muy complejo. Tal y como el uso indiscriminado del *goto* causaba código spaghetti, como *Knuth* [P1] lo describió, en terminología de *Kiczales y Lopes* "the re-working of the code to satisfy aspects creates tangled code ...", lo que se entiende como "el modificar el codigo para satisfacer aspectos crea codigo enmarañado..."
Los sistemas de software modernos son tan complejos que muchos de los problemas que deben ser programados se relacionan con otras partes del sistema en formas sofisticadas. Los mecanismos de integración existentes que proporcionan los lenguajes de programación no representan adecuadamente estos problemas en el programa

K. Czarnecki [9] establece que el programador puede escoger una de dos implementaciones: puede escoger aquella que es clara, fácil de modificar y que representa bien el diseño o puede escoger aquella que mejor satisface uno o más aspectos del programa. Con el segundo tipo de implementación, se obtiene un código enmarañado porque se intenta direccionar ámbitos diseminados sin soporte para representar estos ámbitos en una forma modular. La mayoría de los lenguajes proporcionan mecanismos para manipular unidades de funcionalidad de manera conjunta pero no proporcionan soporte para unidades funcionales relacionadas con aspectos.

El proceso de arreglar o adecuar manualmente es el responsable de que se genere código enmarañado. La solución basada en la modificación del código original, según *Czarnecki* no es siempre lo ideal, el hecho de que simplemente se transforme el problema en un nuevo componente, introduce complejidad extra y posiblemente afecte al rendimiento; aunque es una solución, en general, sería ingenuo pensar que se pueden deshacer todos los aspectos simplemente con la reconstrucción de los modelos. Al adecuar algunos problemas siempre causara que otros problemas, distintos, surjan. Los sistemas reales siempre tendrán algunas diseminaciones inherentes que al reconstruir no nos podremos deshacer de ellas. Más que tratar de barrer el problema bajo la alfombra, se debe proporcionar tecnología adecuada para tratar con los ámbitos diseminados.

1.5 Una Filosofía de Diseño en la POA

La *POA* ignora muchos de los principios que son centrales para la *POO*.
La *POA* introduce un nuevo tipo de módulo (el aspecto) y este nuevo tipo de módulo es muy diferente a un objeto, ya que tiene la habilidad de romper todas las reglas de los objetos. *Highley, Lack, y Myers* [P1] proponen una filosofía de aspectos que permite a un aspecto atravesar la jerarquía del objeto manteniendo la separación de los módulos. Existen cuatro puntos en esta filosofía :

1. Un objeto es algo.
2. Un aspecto no nada más es algo, es algo acerca de algo.
3. Los objetos no dependen de los aspectos.
4. Los aspectos representan alguna característica o propiedad de los objetos, pero no tienen control sobre estos.

Estas definiciones parecen ser evidentes pero no están por demás.

1. *Un objeto es algo*

   Un objeto existe en sí mismo (es algo). ¿Cómo se puede determinar a un objeto en particular?, un programador debe ser capaz de buscar una clase en el código y determinar lo que el objeto es (las variables que forman o constituyen al objeto) y lo que el objeto puede hacer (los métodos del objeto). Algo de lo que es y de lo que puede hacer podría estar basado en una superclase o clase padre. Un objeto puede además estar asociado con un número de aspectos. Esas relaciones están determinadas por los aspectos y no por los objetos. Si el objeto no esta asociado con algún aspecto, no importa este sigue siendo un objeto menos detallado, pero seguiría siendo un objeto. Un objeto es una entidad en sí mismo.

2. *Un aspecto no es algo. Es algo acerca de algo*

   Un aspecto se utiliza para modularizar limpiamente un ámbito diseminado. Este ámbito, por definición se disemina por un número de componentes diferentes. En la *POO* estos componentes son llamados objetos. Si un aspecto no esta asociado con ninguna clase,

entonces su ámbito se disemina por ninguna clase y por lo tanto el aspecto no tiene significado en ese contexto: sin embargo, no tiene sentido hablar de utilizar un aspecto sin tener en mente las clases en se disemina. Tiene sentido discutir lo que un aspecto es capaz de proporcionar, pero el aspecto no tiene funcionalidad a menos que este de hecho aplicado a una clase. Estos no son unidades funcionales en sí mismas y no deben ser tratadas como tales.

3. *Los objetos no dependen de los aspectos*
   Un aspecto no debe cambiar las interfaces de las clases que toca. Deberá sólo aumentar las implementaciones de esas interfaces. Gracias a que sólo afecta las implementaciones de las clases, y no cambia las interfaces, se mantiene la encapsulación. Las clases conservan sus interfaces originales como cajas negras, aunque lo que esta dentro de las cajas si se podría cambiar. La inclusión de un aspecto en un programa debe, y de hecho lo hace, afectar el comportamiento de los objetos en el programa.
4. *Los aspectos representan alguna caracteristica o propiedad de los objetos, pero no tienen control sobre estos*
   Los aspectos pueden violar información escondida en ciertas formas ya que pueden saber cosas acerca de un objeto que están escondidas para otros objetos, pero no deben inmiscuirse con la representación interna de un objeto más de lo que pueden otros objetos. A los aspectos se les debe permitir tener esta vista especial de muchos objetos, pero deben ser restringidos para manipular objetos en la misma forma que otros objetos lo hacen a través de sus funciones miembro disponibles.

Esta filosofía nace de un intento de generalizar los principios de diseño que han sido aceptados de manera general. En particular se quiere que los aspectos sean capaces de proporcionar abstracción y automatización, mientras se cumplan los principios de información escondida y de una interfaz constante.

Para entender mejor esta filosofía, que mejor que un cuento para ejemplificarla. Salgámonos por un momento del reino de las ciencias de la computación y viajemos hacia un mundo diferente ...

En este mundo, existe una especie de duendes jorobados (objetos) que están sordos y tienen sus propias casas (módulos) donde ellos habitan. En estas casas, se reciben ciertos mensajes que los duendes utilizan para comunicarse entre sí, porque también están mudos y es por eso que nunca se enteran de la existencia de ciertos dragones (aspectos) que recién acaban de crearlos los dioses de ese mundo y los están poniendo a prueba para ver si pueden coexistir con los duendes, de hecho la creación de estos dragones fue un premio a la buena conducta de los duendes, para cuidarlos y hacerles la vida más fácil: cada casa tiene un número diferente de puerta y los mensajes sólo son recibidos en un buzón de correo por puerta. Siendo jorobados, no pueden mirar nunca hacia arriba y siendo sordos, tampoco escuchan a los dragones volar por encima de ellos. Tampoco son concientes de que algunas de las casas a las que ellos también mandan mensajes son propiedad de esos dragones. Los duendes además no saben que los techos de sus casas están hechos de vidrio y se puede ver hacia dentro desde arriba.

Los dragones al volar miran a los duendes dejando mensajes en los buzones de correo y ocasionalmente un dragón puede volar bajo para sacar una parte del correo del buzón. Al dragón no se le permite descender dentro de la casa, pero si de disfrutar de su vista especial dentro de la casa a través del techo de vidrio. Los dragones tienen distintas actividades de

las cuales ellos pueden escoger (ámbitos), pueden por ejemplo pintar una casa (manejo de excepciones).

Los dragones además tienen un código de honor, pueden leer el correo de alguien pero no robarlo y siempre lo regresan en el buzón correcto.

De vez en cuando, los dragones van al vecindario de a lado al Festival Nacional de los Dragones, y mientras los dragones no están, los duendes siguen haciendo lo suyo como siempre, ellos notan que las casas ya no cambian de color(ya que los dragones no las están pintando), pero eso no afecta sus vidas en lo más mínimo. Cuando los dragones regresan hacen su mejor esfuerzo para que los duendes no choquen entre sí, ya que mientras los dragones están en el festival, chocan más seguido.

A pesar de la falta de color y de los choques, los mensajes de los duendes se siguen entregando y recibiendo como siempre.

Esta tierra descabellada de duendes y dragones ilustra la filosofía de las relaciones entre los objetos y los aspectos. Los dragones siguen tratando de encontrar como relacionarse mejor con los duendes para demostrarles a los dioses que con ellos la vida de los duendes cambia trascendentalmente para bien y que si bien sin ellos la vida continua como siempre con ellos se acerca a la felicidad y así termine su tiempo de prueba y establecerse para siempre junto con los duendes.

1.6 Elementos de la POA

Con todo lo visto en las secciones anteriores, se puede decir que con las clases se implementa la funcionalidad principal de una aplicación (como por ejemplo, el control de un almacén), mientras que con los aspectos se capturan conceptos técnicos tales como la persistencia, el manejo de errores, la sincronización o la comunicación de procesos. Estos conceptos no son totalmente independientes, y está claro que hay una relación entre los componentes y los aspectos, y que por lo tanto, el código de los componentes y de estas nuevas unidades de programación tienen que interactuar de alguna manera. Para que ambos (aspectos y componentes) se puedan mezclar, deben tener algunos puntos comunes, que son los que se conocen como *puntos de enlace*[8], y debe haber algún modo de integrarlos.

Los *puntos de enlace* son una clase especial de interfaz entre los aspectos y los módulos del lenguaje de componentes. Son los lugares del código en los que se puede insertar comportamiento adicional y este comportamiento se especifica en los aspectos.

El encargado de realizar este proceso de mezcla se conoce como *integrador*[9] o también se le puede llamar tejedor, mezclador, acoplador o montador. Este integrador se encarga de acoplar los diferentes mecanismos de abstracción e integración que aparecen en los lenguajes de aspectos y componentes ayudándose de los puntos de enlace.

Para tener un programa orientado a aspectos necesitamos definir los siguientes elementos:

- Un lenguaje para definir la funcionalidad básica. Este lenguaje se conoce como *lenguaje base*. Suele ser un lenguaje de propósito general, tal como *C++* o *Java*.
- Uno o varios lenguajes de aspectos. El lenguaje de aspectos define la forma de los aspectos – por ejemplo, los aspectos de *AspectJ* se programan de forma muy parecida a las clases.

[8] en la literatura se encuentran como join points
[9] que viene del término en inglés *weaver*

- Un integrador de aspectos. El cual se encargará de combinar los lenguajes. El proceso de acoplamiento se puede retrasar para hacerse en tiempo de ejecución, o hacerse en tiempo de compilación.

En las aplicaciones tradicionales, bastaba con un compilador o intérprete que tradujera nuestro programa escrito en un lenguaje de alto nivel a un código directamente entendible por la máquina. En las aplicaciones orientadas a aspectos, además del compilador, debemos de tener el integrador, que nos combina el código que implementa la funcionalidad básica, con los distintos módulos que implementan los aspectos, pudiendo estar cada aspecto codificado en un lenguaje distinto.

La *POA* empezó como un marco experimental de un lenguaje, llamado *D* [P2], creado como parte de la tesis doctoral de *Cristina Videira Lopes*. El lenguaje *D* tiene tres componentes, un lenguaje para definir componentes (como *Java* o *C*) y dos lenguajes especiales para describir aspectos de dominio: *COOL* y *RIDL*.

Lopes trabajó con un dominio el cual tenía como ámbitos diseminados a la sincronización y a la comunicación. Estos ámbitos diseminados fueron modelados como aspectos a través de dos lenguajes de aspectos. *COOL* (modelo de coordinación de hilos de ejecución) y *RIDL* (modelo de acceso remoto). El lenguaje de definición de componentes fue *Jcore*, el cual era un subconjunto de *Java*.

Los dos aspectos fueron escritos en sus correspondientes lenguajes de aspectos y los otros componentes en *Jcore*. Un preprocesador combinaba los tres programas y producía un programa completo en *Java* a través de la integración. El proceso de integración es similar al proceso de optimizar a mano el cual crea código enmarañado. Los lenguajes de aspectos y el integrador simplemente permiten al programador especificar como se debe hacer la optimización, y donde se debe modificar el código; sin embargo el programador no necesita preocuparse con el código integrado, así como no se preocupa por los resultados intermedios de un compilador convencional. El código integrado nunca se modifica. Las modificaciones se hacen en los programas originales(*Jcore*, *RIDL*, y *COOL*) y luego estos programas se vuelven a integrar y a recompilar para crear un programa ejecutable.

1.7 Integrando Clases y Aspectos

Los aspectos describen extensiones al comportamiento de los objetos. Hacen referencia a las clases de los objetos y definen en qué punto se colocan estas extensiones.

Las clases y los aspectos se pueden integrar de dos formas distintas: de manera estática o bien de manera dinámica[9]

- Integración Estática

La integración estática implica modificar el código fuente de una clase insertando sentencias en estos puntos de enlace. Es decir, que el código del aspecto se introduce en el de la clase. Un ejemplo de este tipo de integrador es el integrador de aspectos de *AspectJ*.

La principal ventaja de esta forma de integración es que se evita que el nivel de abstracción que se introduce con la programación orientada a aspectos propicie un impacto negativo en el rendimiento de la aplicación. Pero, por el contrario, es bastante difícil identificar los aspectos en el código una vez que éste ya se ha integrado, lo cual implica que

si se desea adaptar o reemplazar los aspectos de forma dinámica, en tiempo de ejecución, nos encontremos con un problema de eficiencia, e incluso imposible de resolver a veces.

- Integración Dinámica.

Una precondición o requisito para que se pueda realizar una integración dinámica es que los aspectos existan de forma explícita tanto en tiempo de compilación como en tiempo de ejecución; para conseguir esto, tanto los aspectos como las estructuras integradas se deben modelar como objetos y deben mantenerse en el ejecutable. Dada una interfaz, el integrador es capaz de añadir, adaptar y borrar aspectos de forma dinámica, si así se desea, durante la ejecución. Un ejemplo de este tipo es el integrador de *HyperJ*, que no modifica el código fuente de las clases mientras se integran los aspectos. En su lugar utiliza la herencia para añadir el código específico del aspecto a sus clases.
La forma en la que trabaja este integrador se muestra en la *Figura 1.8.1*, que enseña la estructura de clases UML resultante de integrar dos aspectos, sincronización y localización, a la clase *Rectángulo*. La idea que se aplica es que cada aspecto se representa por su propia subclase *Rectángulo*. Los métodos a los que les afectan los aspectos se redefinen en las subclases utilizando los *métodos integrados*. Un método integrado envuelve la invocación de la implementación del método heredada. La clase vacía *clase integrada* finaliza la cadena de subclases.
Las subclases generadas por el integrador de aspectos no tienen ningún efecto en la ejecución del programa. Si se le envía un mensaje a un objeto *Rectángulo*, se buscará el correspondiente método de la clase del objeto y ejecutará la implementación que tenga. Para hacer que el código integrado tenga efecto, el integrador cambia la clase de todos los objetos *Rectángulo* a la clase integrada. También instala un mecanismo para que esto ocurra con todos los objetos *rectángulo* que se añadan en el futuro.
Este integrador también tiene en cuenta el orden en el que se integran los aspectos. Esto lo resuelve asignando una prioridad al aspecto. El aspecto que tenga asignado un número menor es el que se integra primero, y por lo tanto, aparecerá antes en la jerarquía de herencia. Esta prioridad se ha representado en la estructura UML de las clases como un número entre paréntesis después de la clase estereotipada.
El principal inconveniente bajo este enfoque es el rendimiento y que se utiliza más memoria con la generación de todas estas subclases.

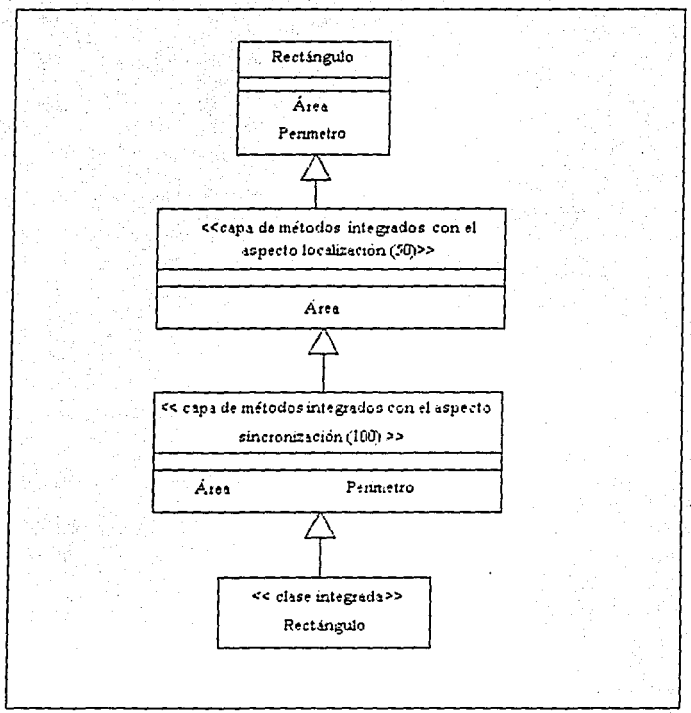


Figura 1.8.1 Estructura de clases resultado de integrar dos aspectos y una clase con el integrador de HyperJ

1.8 Los Aspectos en el Diseño

¿Porque utilizar la *POA*?, ¿porque alguien debería utilizar la *POA* en vez de la *POO*?, uno puede modelar todo como objetos, y también se puede además modelar todo como funciones si así se quiere. El diseño de software se ha movido, aunque lentamente, hacia los objetos por una muy buena razón: pensar en objetos es más fácil y más natural que pensar sólo en funciones. De manera similar, cuando tratamos de construir o crear sistemas que explotan ámbitos diseminados, pensar en aspectos es más fácil y más natural que sólo pensar en objetos.

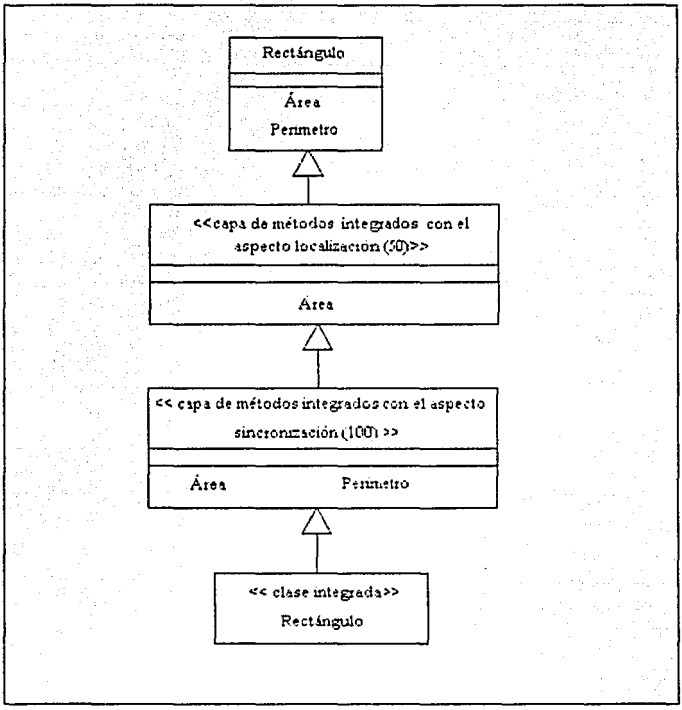


Figura 1.8.1 Estructura de clases resultado de integrar dos aspectos y una clase con el integrador de HyperJ

1.8 Los Aspectos en el Diseño

¿Porque utilizar la *POA*?, ¿porque alguien debería utilizar la *POA* en vez de la *POO*?, uno puede modelar todo como objetos, y también se puede además modelar todo como funciones si así se quiere. El diseño de software se ha movido, aunque lentamente, hacia los objetos por una muy buena razón: pensar en objetos es más fácil y más natural que pensar sólo en funciones. De manera similar, cuando tratamos de construir o crear sistemas que explotan ámbitos diseminados, pensar en aspectos es más fácil y más natural que sólo pensar en objetos.

En sus primeras fases, la orientación a aspectos se centró principalmente en el nivel de implementación y codificación, pero en los últimos tiempos cada vez hay más gente involucrada y surgen más trabajos para llevar la descomposición en ámbitos a nivel de diseño

Esta descomposición se hace necesaria, sobre todo, cuando las aplicaciones necesitan un alto grado adaptabilidad o que se puedan reutilizar. Los trabajos surgidos hasta el momento proponen utilizar *UML* como lenguaje de modelado, ampliando su semántica con los mecanismos que el propio lenguaje unificado tiene para tales efectos y consiguiendo así representar el diseño funcional del objeto separado del diseño no funcional del mismo, o lo que es lo mismo, representar la funcionalidad básica separada de los aspectos.

Las ventajas que se obtienen al capturar los aspectos ya desde la fase de diseño son claras:

- Facilita la creación de documentación y el entendimiento del sistema. El tener los aspectos como constructores de diseño permite que los desarrolladores los reconozcan en las primeras etapas del proceso de desarrollo, teniendo así una visión a un nivel más alto y ayudando a que los diseñadores de aspectos y los principiantes puedan entender y documentar los modelos de aspectos de un modo más intuitivo, pudiendo tal vez incluso utilizar herramientas *CASE* para tener representado el modelado de forma visual.
- Facilita la reutilización de los aspectos. La ventaja anterior influye en la reutilización de la información de los aspectos. Al saber cómo se diseña un aspecto y cómo afecta a otras clases, es fácil imaginar formas sofisticadas de usar aspectos como por ejemplo con las herramientas *CASE*, lo que incrementaría la reutilización del diseño junto con los aspectos.

La propuesta que realiza *Junichi Suzuki y Yoshikazu Yamamoto* [P3] es la extensión del meta-modelo de *UML* para tener en cuenta a los aspectos en la fase de diseño y consiste en lo siguiente:

Añadir nuevos elementos al meta-modelo para el aspecto y la clase "integrada", y de reutilizar un elemento ya existente para la relación clase-aspecto.

- Aspecto.

El aspecto se añade como un nuevo constructor derivado del elemento Clasificador (*Figura 1.9.1*) que describe características estructurales y de comportamiento.

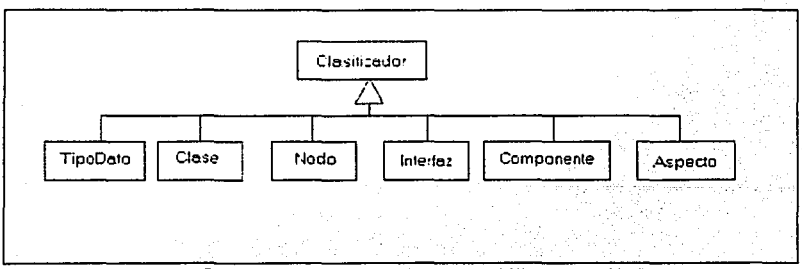


Figura 1.9.1 El Aspecto como un elemento del meta-modelo UML derivado de Clasificador

Un aspecto asi modelado puede tener atributos, operaciones y relaciones. Los atributos son utilizados por su conjunto de definiciones integradas. Las operaciones se consideran como sus declaraciones integradas. Las relaciones incluyen la generalización, la asociación y la dependencia. En el caso de que el integrador de aspectos utilice cierto tipo de integración, como por ejemplo colocar un aviso integrado en *Aspect*, éste se especifica como una restricción para la correspondiente declaración integrada.

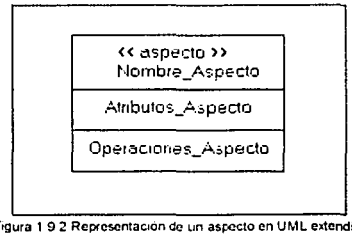


Figura 1.9.2 Representacion de un aspecto en UML extendido

El aspecto se representa como un rectángulo de clase con el estereotipo <<aspecto>> (*Figura 1.9.2*). En la sección de la lista de operaciones se muestran las declaraciones integradas. Cada integración se muestra como una operación con el estereotipo <<integrada>>. Una declaración integrada representa un identificador cuyos elementos (clases, métodos y atributos) están afectados por el aspecto.

- Relación aspecto-clase.

En el meta-modelo de *UML* se definen tres relaciones básicas, que se derivan del elemento Relación: Asociación, Generalización y Dependencia.

La relación entre un aspecto y las clases que afecta es un tipo de relación de dependencia. La relación de dependencia modela aquellos casos en que la implementación o el funcionamiento de un elemento necesita la presencia de otro u otros elementos.

Del elemento Dependencia se derivan otros cuatro que son: Abstracción, Ligadura, Permiso y Uso (*Figura 1.9.3*). La relación aspecto-clase se clasifica como una relación de dependencia del tipo abstracción.

Una relación de dependencia de tipo abstracción relaciona dos elementos que se refieren al mismo concepto pero desde diferentes puntos de vista, o aplicando diferentes niveles de abstracción. El meta-modelo *UML* define también cuatro estereotipos para la dependencia de abstracción: derivación, realización, refinamiento y traza.

Con el estereotipo <<realiza>> se especifica una relación entre un elemento o elementos del modelo de especificación y un elemento o elementos del modelo que lo implementa. El elemento del modelo de implementación tiene que soportar la declaración del elemento del modelo de especificación.

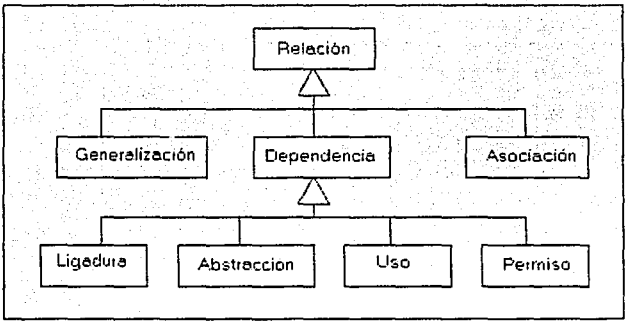


Figura 1.9.3. Tipos de relaciones del meta-modelo de UML

La relación aspecto-clase se indica en la dependencia de abstracción con el estereotipo realización (<<realiza>>). Esta relación se representa en la *Figura 1.9.4*.

- Clase integrada

Al utilizar un integrador de aspectos, el código de las clases y los aspectos se integran y se genera como resultado una clase acoplada. La estructura de la clase integrada depende del integrador de aspectos y del lenguaje de programación que se haya utilizado. Por ejemplo, *AspectJ* reemplaza la clase original por la clase integrada, mientras que *HyperJ* genera una clase integrada que se deriva de la clase original.

La solución que proponen *Suzuki y Yamamoto* para modelar esto es introducir el estereotipo <<clase integrada>> en el elemento *Clase* para representar una clase integrada. Se recomienda que en la clase integrada se especifique con una etiqueta la clase y el aspecto que se hayan utilizado para generarla.

En la *Figura 1.9.5* se representa la estructura de las clases integradas, utilizando tanto el integrador de *AspectJ* como el integrador de *HyperJ*.

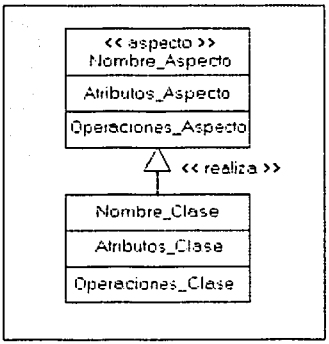


Figura 1.9.4 Notación de la relación aspecto-clase

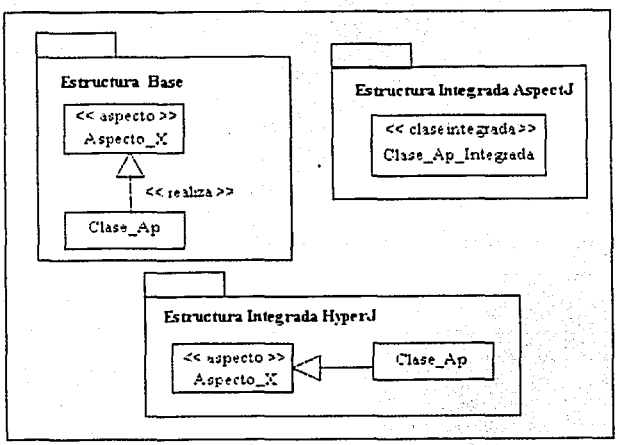


Figura 1.9.5 Estructura de las clases integradas

CAPITULO II

Enfoques de Descomposición OA

Existen actualmente en desarrollo varias instancias de la *OA*, cada una con un propósito específico pero a fin de cuentas todas ellas utilizan el mismo razonamiento para poder mejorar la calidad del software: la descomposición en ámbitos de una manera más adecuada y más natural que las convencionales, las cuales traen consecuencias negativas y después combinar automáticamente estas descripciones que están separadas en un ejecutable final. Este capítulo se encarga de ver tres de los enfoques más representativos, los cuales extienden el modelo de programación *OO* para poder cumplir su cometido. Cada uno de estos enfoques proponen mecanismos concretos de descomposición e integración de módulos.

Algunos ejemplos de mecanismos de integración soportados por los lenguajes convencionales son: las llamadas a funciones, la parametrización y la herencia. Sin embargo no todos los aspectos relevantes que ocurren en la práctica pueden ser adecuadamente integrados usando estos recursos y por lo tanto se necesitan implementar nuevos mecanismos. Algunos de estos nuevos mecanismos de integración son: filtros de mensajes en el modelo filtros de integración, reglas de integración de aspectos en la programación subjetiva y los hiperespacios en el modelo de descomposición multidimensional.

2.1 Filtros de Integración

Los filtros de integración es una técnica modular y ortogonal de programación orientada a aspectos. Modular significa que los filtros son independientes del código base. Ortogonal significa que el significado de un filtro es independiente de los demás filtros.

Este enfoque utiliza el concepto de filtro de integración para poder encapsular aspectos. Imaginemos un fotógrafo aficionado que pretende llegar a ser profesional y para empezar quiere tomar fotos de animales dentro de un zoológico, para esto necesita los elementos necesarios para hacerlo, esto es, una cámara, para lo cual escoge la más barata que encuentra, al empezar a tomar sus primeras fotos descubre que no salen como el hubiera querido ya que la distancia en la que se encuentra, no es propicia para unas buenas tomas. El fotógrafo regresa a la tienda donde compró la cámara y comenta su problema, el encargado de la tienda modestamente le dice a este que esa tienda se encarga de construir cámaras para todas las necesidades y que con gusto le vende una que responde al problema de poder tomar fotos a una distancia muy corta, el fotógrafo no muy contento por el desembolso de una cantidad considerable de dinero, acepta la cámara ofrecida y se dispone ahora si a tomar unas mejores fotos; al llegar a la jaula donde se encuentran las aves exóticas que quiere tomar, se da cuenta que efectivamente las fotos salen mejor, pero la luz no es la indicada, no quedando satisfecho del todo con las tomas vuelve a regresar a la tienda de cámaras, donde después de plantear su nuevo problema el encargado le indica que

precisamente tiene otra cámara que se ajusta a esa necesidad y que gustoso le facilitará a cambio de otra buena cantidad de dinero.
Cada vez que se presenta un problema se tiene que construir una cámara nueva ocasionando que esto sea una solución muy cara. Este mismo escenario se presenta con el software actual teniendo soluciones a la medida, y esto no es nada económico, además se debe asegurar que las soluciones sean capaces de trabajar bien conjuntamente.
El fotógrafo se pone a pensar que debe de haber otra forma para no tener que estar comprando cámaras para cada problema que se le presente, así que decide ir a otra tienda y pregunta si no hay algún aditamento que pueda utilizar con su primera y austera cámara, el encargado de esta otra tienda le muestra una serie de lentes que se pueden usar como extensiones en su cámara, esta extensión modular es una solución económica, además le muestra un filtro de color que también se le puede poner como extensión y le indica que cualquier problema que tenga, como por ejemplo intensificar imágenes, siempre se pueden añadir extensiones adicionales; ya que estas extensiones son independientes entre sí. Esto es lo que propone el enfoque filtros de integración donde cada extensión debe direccionar un ámbito y tener un significado perfectamente definido.
Resumiendo, las características de los filtros de integración son:

- Extensiones modulares
- Extensiones independientes
- Soluciones que pueden añadirse o quitarse en cualquier momento
- Significado de los filtros perfectamente definido

Estas características responden a las deficiencias de los métodos, lenguajes y herramientas *OO* que surgen cuando los requerimientos evolucionan y no son adecuados para direccionar ciertos aspectos de las aplicaciones, lo cual en el mejor de los casos requiere un número considerable de redefiniciones; cuando el software incorpora ámbitos diseminados este no evoluciona fácilmente con las técnicas convencionales.
En lugar de proporcionar soluciones dedicadas al modificar las descripciones de las clases, los filtros de integración ofrecen *extensiones modulares* a las abstracciones de las clases. En el caso de que se tenga que expresar más de un ámbito, se pueden utilizar varios filtros, donde cada filtro transforma los mensajes que involucra a un determinado objeto.
El modelo propone mejorar, modificar y ampliar a un objeto convencional con tan sólo la manipulación de los mensajes que este involucra. La conjunción de los filtros define el comportamiento del objeto. Estos filtros son capaces de monitorear todos los mensajes que entran y salen, así como del objeto en sí mismo y son capaces de modificar el envío de mensajes o redirigirlos a otros objetos.
Para que sea posible que un filtro evalúe y manipule un mensaje se debe aplicar alguna forma de reflexión sobre el mensaje.
Al extender el modelo de objetos convencional con filtros de integración, los objetos extendidos se componen por un kernel y una capa de interfaz. El objeto kernel implementa el comportamiento específico del objeto y puede verse como un objeto clásico en el modelo convencional *OO*. La capa de interfaz encierra el kernel y maneja los mensajes que entran y los que salen.
La *figura 2.1.1* muestra la representación de un objeto con filtros, donde esta clara la separación entre los métodos (funcionalidad) y las condiciones (estado).

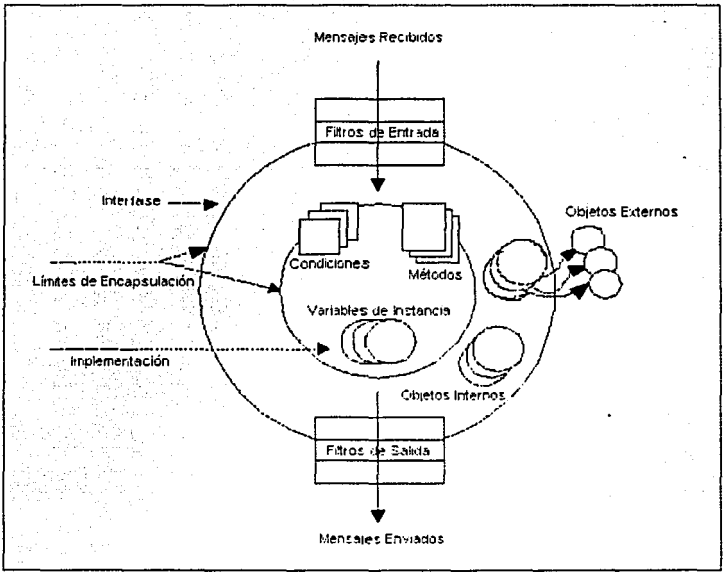


Figura 2.1.1 Un objeto en el modelo Filtros de Integración

La parte de los filtros describe la interfaz del objeto. La parte de la implementación del objeto la describe el kernel. El kernel implementa un comportamiento específico del objeto y mantiene su estado. Como se ve en la *figura 2.1.2* el kernel contiene tres componentes: métodos, condiciones y variables de instancia. Los nombres de los métodos y las condiciones son visibles en el límite de la encapsulación del kernel, mientras que las variables de instancia están completamente encapsulados.

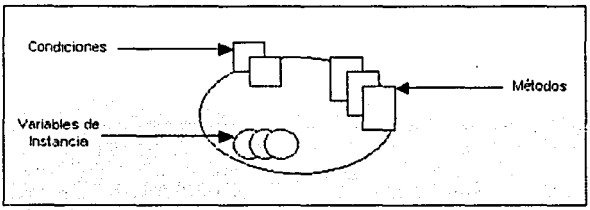


Figura 2.1.2 Kernel

Una variable de instancia mantiene el estado de un objeto. Esta es una instancia de una clase especifica la cual se crea cuando el objeto se crea.
El comportamiento de un objeto se implementa por sus métodos. Un método define un número de acciones que son realizadas en reacción a la invocación de ese método.
Una condición da información acerca del estado actual del objeto, es un tipo especial de método que no toma parámetros y regresa un valor *booleano*.
Los objetos internos son objetos completamente encapsulados y son usados para componer el comportamiento del objeto por ejemplo un mensaje recibido podría ser delegado a un objeto interno en lugar de al kernel.
Los objetos externos son objetos que existen fuera del objeto, como objetos globales. Estos pueden ser usados para compartir datos entre objetos. Similar a los objetos internos, además se usan para componer el comportamiento del objeto. En la *figura 2.1.2* no se muestra la estructura de los objetos internos y externos y de las variables de instancia, pero ellos tiene una estructura igual, teniendo un kernel y una capa de interfaz.

2.1.1 El Significado del Envío de Mensajes

Un objeto puede requerir un servicio de otro objeto al mandar un mensaje a este. El envío de mensajes en el modelo de objetos *filtros de integración* sigue el modelo requerimiento/respuesta: después de que un objeto manda un mensaje (requerimiento) a un objeto servidor, el objeto cliente espera hasta que recibe un valor de regreso (respuesta) del objeto servidor.
El objeto servidor es responsable de atender el requerimiento: este puede decidir si maneja él mismo el requerimiento (al ejecutar algún método) o delegar el requerimiento a otro objeto o inclusive rechazarlo.
En el modelo de objetos filtros de integración el procesamiento de un mensaje involucra dos pasos:

- Primero, el mensaje pasa por el conjunto de filtros. El mensaje al salir del objeto emisor tiene que pasar a través de los filtros de salida del objeto emisor: luego tiene que pasar por los filtros de entrada del objeto receptor. El objeto receptor puede decidir (por medio de un filtro *dispatch*) delegar el mensaje a otro objeto donde tendrá que pasar por los filtros de entrada del objeto delegado.
- Eventualmente, se invoca un método. Se invoca un método si existe un filtro *dispatch* que direcciona el mensaje a un objeto interno.

Los dos pasos anteriores se ilustran en el diagrama de la *figura 2.1.3*

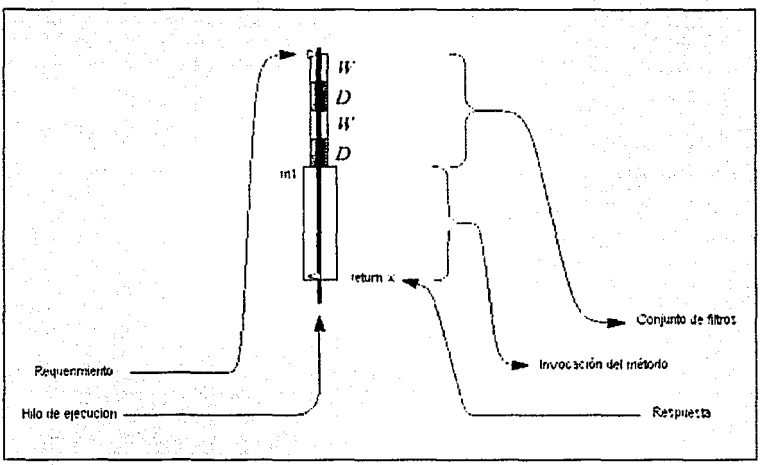


Figura 2.1 3 Procesamiento de un mensaje

El inicio de la ejecución del mensaje empieza en la parte superior del diagrama. El mensaje pasa a través de los filtros, mostrados por rectángulos donde se indican cuatro filtros: dos filtros *wait* (W) y dos filtros *dispatch* (D). El último filtro invoca un método, *m1* en este caso, el cual esta representado por el ultimo rectángulo. La ejecución del método termina cuando el método invocado regresa un resultado (*return x*). Al mensaje que sé esta ejecutando se le denomina mensaje activo.

Después de que ha terminado la ejecución de un mensaje, el siguiente mensaje se ejecuta en una forma similar; se filtra e invoca un método. La invocación del método, el rectángulo más grande en la *figura 2.1.3*, consiste de una secuencia de ejecuciones de mensajes. Si se pudiera ver dentro de este, se podrían ver varias ejecuciones de mensajes una detrás de la otra, cada una de ellas consistiendo también de un filtrado y una invocación a un método[10]. A una secuencia de tales ejecuciones en un mensaje se le llama hilo de ejecución[11].

En Sina[12] se puede tener acceso a una variable llamada *message*, la cual representa el mensaje activo, por lo tanto se refiere al mensaje que se esta filtrando en ese momento y con eso se puede accesar a los atributos de un mensaje activo, entre los cuales están: el método, sus argumentos, el emisor del mensaje, el objeto que recibió primero el mensaje, y el objeto que recibió el mensaje en ese momento preciso.

En Sina puede existir al mismo tiempo cualquier número de hilos de ejecución, de esta forma los mensajes y los métodos se pueden ejecutar concurrentemente. Ocasionando que

[10] uno se podría preguntar, cuando se termina esto. Hay algunos metodos que no mandan otros mensajes: ellos implementan operaciones primitivas como por ejemplo sumar dos numeros, entonces allí terminaría.
[11] en la literatura se encuentra como *thread*.
[12] lenguaje nativo de los filtros de integracion, el cual se verá mas adelante

más de un hilo de ejecución pueda estar activo (ejecutandose concurrentemente) dentro de un mismo objeto. esto quiere decir que hay dos o más métodos que han sido invocados y se están ejecutando. Aunque, si no se pretende esto, el objeto puede definir exclusión mutua con la ayuda de un filtro de entrada *wait*[13].

Existen dos ocasiones en las cuales se crea e inicializa un hilo de ejecución:

- antes de la invocación de un método de regreso anticipado[14] (tiene más sentencias a ejecutar después de alguna sentencia *return*)
- cuando un mensaje se manda a un objeto de comunicación abstracta[15], antes de la conversión del mensaje por un filtro *meta*.

Un hilo de ejecución termina cuando el último mensaje de este hilo ya se haya ejecutado.

El lado izquierdo de la *figura 2.1.4* muestra un método de regreso normal (no anticipado) se ejecuta en el mismo hilo de ejecución: el mensaje *a.m1* se filtra y se invoca *m1*. Después de que el método m1 regreso algo (*return x*), el próximo mensaje *b.m2* se filtra y se invoca *m2*. El diagrama de en medio muestra la ejecución de un método de regreso anticipado *m3*. El mensaje *a.m3* se filtra de manera normal. Después de pasar por el último filtro, un filtro *dispatch*, se crea un nuevo hilo de ejecución para la ejecución del método *m3*: El hilo original se bloquea hasta que el método *m3* regrese algo (*return x*). El mensaje ahora tiene una respuesta y el hilo bloqueado (la linea punteada) continúa ejecutando el próximo mensaje, *b.m4*. Después de que el método regresó algo, el resto de *m3* (los mensajes que siguen después de la sentencia *return*) se ejecutan concurrentemente con el hilo original y por lo tanto con el mensaje *b.m4*. El nuevo hilo termina cuando el último mensaje en el método de regreso anticipado *m3* se haya ejecutado.

También se crea un nuevo hilo antes de la conversión de un mensaje por un filtro *meta*(el lado derecho del diagrama de la *figura 2.1.4*). La conversión y desconversión de mensajes se explicará en breve. Por ahora, podemos ver que el mensaje *act.mAct* se ejecuta en un nuevo hilo de ejecución concurrentemente con el método *m5* en el hilo de ejecución original.

La *conversión* de un mensaje es el proceso de transformar un mensaje activo en un objeto. El proceso opuesto se llama *desconversión* y lo que hace es regresar este mensaje convertido otra vez a un mensaje activo. La *conversión* suspende la ejecución de un mensaje, mientras que la *desconversión* lo reactiva.

El proceso de *conversión* se lleva a cabo en un filtro *meta*: si un filtro *meta* acepta un mensaje entonces lo convertirá a un objeto. Luego, los filtros *meta* ofrecen el mensaje ya convertido a un *objeto de comunicación abstracta*. El objeto de comunicación abstracta es un objeto interno o externo arbitrario, o sea, una instancia de alguna clase.

---

[13] el compilador de Sina inserta por defecto a cada clase un filtro de entrada *wait* que realiza la mutua exclusión. El programador puede redefinir esto, y así definir un esquema de sincronización más complejo para su clase o dejarla sin sincronización.

[14] en un método de regreso anticipado se crea un nuevo hilo de ejecución y las sentencias del cuerpo del método se ejecutan en este nuevo hilo de ejecución. El método que invocó junto con su hilo de ejecución se bloquean hasta que el método invocado regrese un objeto (return x). El hilo que invoco entonces continúa su ejecución: se ejecutan las sentencias que faltaban en el método invocador, se ejecutan concurrentemente con las sentencias siguientes a la sentencia return en el método invocado. El hilo de ejecución en este último método finaliza cuando se termine de ejecutar la última sentencia de este.

[15] la explicación de un TCA y la conversión del mensaje se mostrará posteriormente.

Un mensaje convertido será reactivado si este recibe uno de los mensajes de desconversión: *continue*, *reply* o *send*. El objeto de comunicación abstracta es responsable de mandar uno de estos mensajes al mensaje convertido.

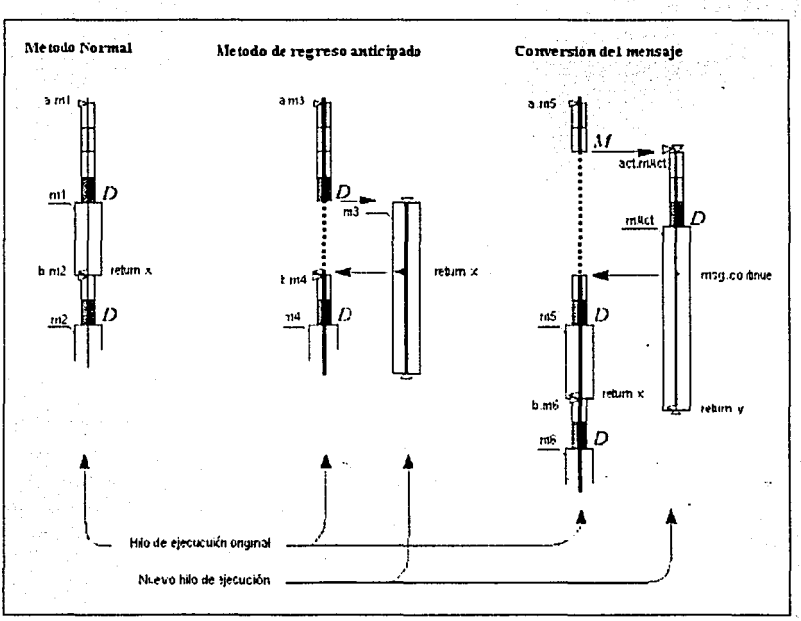


Figura 2.1.4 hilo de ejecucion sencillo (izquierda) y dos hilos concurrentes nuevos (en medio y derecha

2.1.2 El Principio de Filtrado de Mensajes

Los filtros de integración se especifican en dos conjuntos ordenados; un conjunto de filtros de entrada y un conjunto de filtros de salida. Cada especificación de los filtros esta compuesta de un número de elementos de cada uno de los filtros, los cuales especifican el comportamiento de filtración del filtro. Se explica el proceso de filtrado, centrándose sólo en filtros de entrada pero los filtros de salida se comportan en la misma forma.

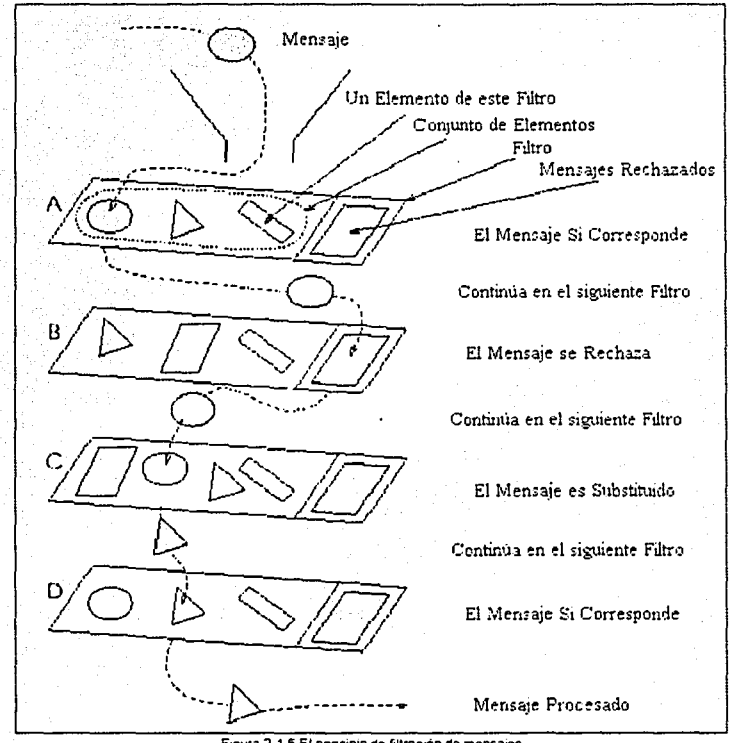


Figura 2.1.5 El principio de filtración de mensajes

En la *figura 2.1.5* se muestra la recepción de un mensaje por un objeto. El primer mensaje tiene que pasar por el conjunto de filtros de entrada del objeto. El conjunto de filtros de entrada describe cuando se acepta o se rechaza el mensaje. Cada filtro en el conjunto tiene su propio conjunto de elementos el cual describe una parte del manejo de aceptación y rechazo del conjunto de filtros de un mensaje. Este conjunto consiste de un número de elementos. Cada elemento del filtro describe una parte del manejo de aceptación y rechazo de su filtro. Los elementos del filtro se evalúan de izquierda a derecha. El filtro compara el patrón de sus elementos con el patrón del mensaje. El patrón significa el objeto en el cual el mensaje esta pasando por sus filtros y el método a llamar.
El mensaje en la *figura 2.1.5* se filtra primero por el filtro *A*. El patrón del mensaje es comparado con el patrón de los elementos del filtro. En este ejemplo, el filtro acepta el mensaje porque el primer elemento del filtro si corresponde. El mensaje enseguida se filtra

por el filtro *B* donde ninguno de sus elementos se corresponde con el patrón del mensaje. El mensaje entonces se rechaza por este filtro y pasa al siguiente filtro *C* el cual lo acepta pero lo substituye con otro patrón, el último filtro *D* acepta el mensaje substituido y lo dirige al objeto destino.
Cada filtro puede tener diferente comportamiento en lo que se refiere a la aceptación o rechazo de un mensaje
En general cada conjunto de filtros contiene un filtro que causa que el mensaje sea ejecutado si es que se corresponde, si el mensaje no puede corresponder en el último filtro, se lanza un error para denotar que ya no hay más filtros que pueden manejar ese mensaje.

2.1.3 Tipos de Filtros Actuales

Los filtros que están disponibles (aunque ya se están desarrollando otros más) son: *dispatch, send, wait, error* y *meta.*

- El filtro *Dispatch* (para substituciones sobre los mensajes, sólo como filtro de entrada): si acepta el mensaje, el nuevo objeto receptor del mensaje y el nuevo método se sustituyen si es que estos están definidos; si el receptor del mensaje es un objeto interno, el método que se invoca es el mismo método del mensaje; de lo contrario el mensaje se ofrecerá para filtrarse a los filtros de entrada del receptor del mensaje. Si se rechaza el mensaje continúa en el próximo filtro.
- El filtro *Error* (para substituciones sobre los mensajes, filtro de entrada/salida): si acepta el mensaje, el nuevo objeto receptor y el nuevo método se sustituyen si es que estos están definidos; luego el mensaje continúa en el siguiente filtro. Si se rechaza se lanza el error "*Reject in ErrorFilter X::f of M*", donde *X* es la clase en la cual esta definido el filtro *error f* y *M* es una representación del mensaje.
- El filtro *Send* (para substituciones sobre los mensajes, sólo filtro de salida): si acepta el mensaje, el nuevo objeto receptor y el nuevo método se sustituyen si es que estos están definidos; el mensaje se ofrece a los filtros de entrada del receptor del mensaje o se ofrece a los filtros de salida del objeto encapsulado. Si se rechaza el mensaje continúa en el próximo filtro. Si una clase no declara algún filtro de salida, el compilador inserta automáticamente un filtro *send.*
- El filtro *Wait* (para substituciones desechadas, filtro de entrada/salida): si se acepta el mensaje, este continúa en el próximo filtro. No se realiza ninguna substitución. Si se rechaza el mensaje, este se bloquea, esto es, se suspende la ejecución del hilo de ejecución correspondiente hasta que la condición sobre la cual el mensaje se bloqueo sea verdadera; luego el mensaje continúa en el próximo filtro.
- El filtro *Meta* (en mensajes para objetos de comunicación abstracta, filtro de entrada/salida): si se acepta el mensaje, se suspende la ejecución del hilo correspondiente; el mensaje aceptado se transforma y se vuelve una instancia de la clase *SinaMessage*; se crea un nuevo hilo el cual manda un mensaje con el nuevo método y el mensaje transformado como argumento para el nuevo objetivo, este mensaje no pasa a través de los filtros de salida del objeto pero es ofrecido a los filtros de entrada del nuevo objeto receptor directamente, el hilo emisor permanece suspendido hasta que el mensaje transformado sea desconvertido(si recibe un mensaje *send, reply, fire* o *continue*); si el mensaje ahora activo tiene una respuesta (si la forma transformada fue reactivada con un mensaje *reply*) no seguirá en el

siguiente filtro pero regresará la respuesta al emisor, de lo contrario será filtrado por el siguiente filtro. Si se rechaza el mensaje, este continúa en el próximo filtro.

En la parte de la interfaz de clase se pueden declarar filtros de entrada y salida conteniendo un conjunto arbitrario de todos estos filtros. De esta forma, el determinado comportamiento de un objeto se forma al seleccionar la combinación apropiada de filtros.

2.1.4 Filtros de Integración en Sina

Como *Sina* es el lenguaje nativo del modelo filtros de integración el traslado del modelo a la implementación es inmediato, la capa de interfaz se traslada a un bloque ***interface*** y el kernel por el bloque ***implementation*** de la definición de una clase. Se mostrará como ejemplo la implementación de una clase *Stack* [P4]. La clase *Stack* proporciona operaciones como push, pop, top y size. Se lanza un error cuando se quiera realizar una operación Pop en un Stack vacío, o cuando se requiera el elemento Top también de un Stack vacío.

```
class Stack interface
  comment
    'Esta clase representa una pila que contiene elementos de un tipo arbitrario.';
  externals
    // Esta clase no tiene objetos externos, aunque estos se podrían declarar como
    // unExt : NombreClase;
  internals
    default : SinaObject;
  conditions
    stackNotEmpty;
  methods
    push(object : Any) returns nil;  // Almacena el  objeto argumento.
    pop returns nil;                 // Elimina el elemento top
    top returns Any;                 // Regresa el elemento top
    size returns SmallInteger;       // Regresa el número de elementos
  inputfilters
    stackEmpty : Error = { stackNotEmpty => {pop, top}, true ~> {pop, top} };
    invoke : Dispatch = { true=>inner.*, true=>default.* };
  outputfilters
    send : Send = { true=>* };
end;                                 // Interfaz de la clase Stack
```

Código 2.1.1 Interfaz de la clase Stack en Sina/st

El *código 2.1.1* muestra la definición de la interfaz de la clase Stack en *Sina*.

La sección que inicia con la palabra reservada ***comment*** define el comentario de la interfaz de la clase, también se aceptan los // y /* */ clásicos de *C* y *C++*.

Los objetos externos junto con los internos se declaran como instancias de alguna clase. Ya que los objetos externos están fuera del objeto, se comprueba si estos existen y donde se localizan cuando el objeto se crea. La clase Stack no tiene objetos externos.

Cuando se crea una instancia de la clase, sus objetos internos se crean. Ya que cada objeto interno se declara como una instancia de alguna clase, sus objetos internos, si es que los hubiera también se crearían y así sucesivamente en cascada.

El objeto interno *default* de la clase Stack se declara como una instancia de la clase *SinaObject*. Esta clase proporciona operaciones por defecto, como *printing*, *testing* y *comparison*.

En la sección ***conditions***, se especifican los nombres de las condiciones. Las condiciones pueden ser usadas en un filtro para probar si se acepta o se rechaza un mensaje. La clase *Stack* sólo tiene una condición, *stackNotEmpty*, la cual indica si el stack esta vacío o no.
La sección ***methods*** define los métodos que estarán disponibles para otros objetos. Se especifican los parámetros y el tipo del valor que regresan. Además de los métodos en la interfaz, una clase puede tener métodos privados los cuales sólo se pueden accesar por la clase en sí. Los métodos privados son declarados en la parte de implementación.
En la sección ***inputfilters***, se declara un número arbitrario de filtros. Una declaración define el nombre del filtro, el tipo y el inicializador.
El inicializador es la parte entre llaves { }, y especifica que mensajes y bajo que condiciones se aceptaran por ese filtro. El inicializador esta compuesto de un número de elementos de filtro, cada uno de los cuales especifica una condición. Los elementos de cada filtro son evaluados de izquierda a derecha, si el mensaje no es aceptado por un elemento del filtro, por ejemplo cuando la condición es inválida, entonces el mensaje podría todavía ser aceptado por el siguiente elemento del filtro.
Por ejemplo el inicializador del *Stack* del primer filtro de entrada, es un filtro *error* con el nombre *stackEmpty*, el cual especifica que el filtro acepta el mensaje con el método pop o top cuando el stack no este vacío. Esto se expresa por el primer elemento del filtro *stackNotEmpty=>{pop,top}*, que podría leerse como si el stack no esta vacío entonces acepta el mensaje con *(pop o top)*.
El segundo elemento del filtro *true~>{pop,top}* se podría interpretar como si *true* entonces acepta el mensaje sin *pop* o *top*, lo cual es lo mismo que acepta el mensaje sin *pop* y sin *top*. Este elemento del filtro especifica que el filtro acepta mensajes con cualquier método excepto por *pop* y *top*. Como resultado, el filtro *error* rechaza mensajes con el método *pop* o *top* cuando el stack esta vacío, causando la generación de un error y aceptando los mensajes en los otros casos.
Los mensajes que ya pasaron por el filtro *error*, se filtran en el siguiente filtro, el filtro *dispatch* con el nombre *invoke*. Este filtro acepta los mensajes que son soportados por el kernel (el elemento del *filtro true=>inner.**) o que son soportados por el objeto interno *default (true=>default.*)*. Por definición los mensajes soportados por el kernel son los métodos de la interfaz, los cuales son *push*, *pop*, *top* y *size* en este caso. Cuando el filtro *dispatch* acepta un mensaje este invoca el correspondiente método del kernel u ofrece el mensaje al objeto default. Este objeto filtra el mensaje y eventualmente invocara un método. Si el objeto *default* además soporta un método *size*, entonces por la evaluación de los elementos del filtro de izquierda a derecha, de todos modos se invocará el método *size* del kernel.
De lo anterior se puede concluir que una instancia de la clase *Stack* soporta el mensaje *push*, *pop*, *top*, *size* y todos los mensajes que están definidos por la clase *SinaObject*. El mensaje *pop* o *top* mandado a un stack vacío genera un error.
Al igual que en los filtros de entrada, también se pueden declarar un número arbitrario de filtros en la sección ***outputfilters***. El filtro *send* manda todos los mensajes(el elemento del filtro *true=>**) mandados por una instancia de *Stack* hacia el receptor del mensaje.
El *código 2.1.2* muestra la implementación de la clase *Stack* en *Sina*

```
class Stack implementation
  comment
    'El atributo "stackPtr" apunta a la última localidad de la
     pila. El atributo "storage" contiene los elementos de la pila';
  instvars
    stackPtr : SmallInteger;
    storage : Array;
  conditions
    stackNotEmpty
    comment 'Este estado es true cuando la pila contiene elementos';
    begin return stackPtr > 0; end;
  initial
    comment 'Inicia con una pila vacía';
    begin stackPtr := 0; end;
  methods
    push(object : Any)
      comment 'Introduce el objeto argumento dentro de la pila';
      begin
        stackPtr := stackPtr + 1;
        storage.at:put:(stackPtr, object)
      end;
    top
      comment 'Regresa el elemento top(el último) de la pila';
      temps object : Any;
      begin
        object := storage.at:(stackPtr);
        return object
      end;
    pop
      comment 'Elimina el elemento top de la pila';
      begin
        stackPtr := stackPtr - 1;
      end;
    size
      comment 'Regresa el numero de elementos de la pila';
      begin return stackPtr
      end;
end;
```

Código 2.1.2 Implementación de la clase Stack en Sina/st

Las variables de instancia de la clase se definen en la sección ***instvars***. Como los objetos internos y externos, estas se declaran como una instancia de alguna clase. Las variables de instancia de un objeto se crean después de que se creen los objetos internos. La variable *storage* contiene los elementos del stack, mientras que *stackPtr* apunta a la ultima localidad utilizada en este arreglo.
La implementación de las condiciones se definen en la sección ***conditions***. Una condición debe regresar un valor booleano. En la condición *stackNotEmpty*, el estado del stack se determina con la expresión *stackPtr > 0*.
El método ***initial*** puede ser usado para inicializar la instancia, como un típico constructor. Este se invoca cuando se crea una instancia, después de que se hayan creado los objetos internos y las variables de instancia. La clase Stack lo usa para inicializar *stackPtr*.
En la sección ***methods***, se define la implementación de los métodos de la interfaz. Las implementaciones de los métodos privados también se pueden encontrar aquí.

Las variables locales de un método y alguna condición se definen después de la palabra reservada ***temps***. El método *top*, por ejemplo, usa una variable local llamada *object* para regresar el elemento *top* del stack.
El *código 2.1.3* muestra un programa principal para probar la clase Stack en Sina

```
main
  comment 'Aplicación de la clase Stack';
  externals
    Transcript : TextCollector;
  temps
    stack : Stack;
  begin
    stack.push('1st element');
    stack.push(2.0d0);
    stack.push('3rd element');
    while stack.size > 0
    begin
      Transcript.print:(stack.top);
      Transcript.cr;
      stack.pop
    end;
    Transcript.flush
  end
```

Código 2.1.3 Main utilizando la clase Stack en Sina/st

El método ***main*** se usa para iniciar una aplicación. En general, el método *main* es muy pequeño, pero se puede usar para probar la definición de clases nuevas.
El método ***main*** tiene la misma estructura de la implementación de un método. La única diferencia es que los objetos globales a los que se refiere ***main*** deben ser declarados en la sección ***externals***. El objeto externo *Transcript* en el *código 2.1.3* es un objeto global de *Smalltalk* el cual puede ser usado para desplegar mensajes. La variable *stack* es local a ***main*** y se declara en la sección ***temps***. En el *código 2.1.3* se introducen al stack tres elementos, luego se despliegan en la pantalla y por último se eliminan del stack.
La descomposición en ámbitos diseminados se logra al definir una clase de filtro para cada ámbito. Cada tipo de filtro es responsable de manejar su ámbito asociado. Este mecanismo de filtros da a los programadores la oportunidad de atrapar los mensajes recibidos y mandados por sus objetos y de realizar ciertas acciones antes de que el código de el método sea ejecutado. El código resultante es por lo tanto separado dentro de un ámbito de propósito especial (en el filtro) y el ámbito básico o base (en el método).

2.1.5 Deficiencias en la Tecnología OO y Soluciones OA

El modelo de *OO* clásico proporciona características que son muy útiles para una amplia gama de aplicaciones, con las cuales se obtiene un software robusto, reutilizable y extensible. Sin embargo, los métodos, lenguajes y herramientas *OO* actuales tienen un número de deficiencias que las hacen menos adecuadas para ciertas categorías de aplicaciones.
El proyecto TRESE (Twente Research & Education on Software Engineering) ha llevado a cabo varias investigaciones para identificar las deficiencias de la tecnología *OO* actual. Encontrando que los problemas de modelado se deben al hecho de que los modelos *OO* no son capaces de expresar adecuadamente ciertos aspectos de las aplicaciones. Además

TRESE propone ciertas soluciones a estas deficiencias aplicando los conceptos del modelo filtros de integración :

- **Herencia y delegación dinámica** (*todos los dominios de aplicaciones*), por ejemplo, si se tiene una clase *Correo* y se extiende con los tipos *Audio* y *Video*. Cuando una clase se extiende se crea una subclase. Cada extensión causará la creación de una nueva subclase. Las subclases de la clase *Correo* serían *AudioCorreo* y *VideoCorreo*. Ambas subclases representan una especialización de la superclase o clase base. Otro tipo de integración es hacer que la clase *Correo* herede de las clases *Audio* y *Video*. Esto crearía una clase que contendría todas las extensiones, en lugar de una subclase por cada extensión. El problema que surge es el de como decidir que tipo de medio (*Audio* o *Video*) deberá ser visible en la interfaz. Los objetos tienen que heredar irremediablemente de su clase base: por lo que se requiere una herencia dinámica. Este problema se puede resolver usando un enfoque *OA* con un filtro *Dispatch* y herencia asociativa junto con la delegación. La herencia asociativa y la delegación asociativa[16] se logran cuando las condiciones (las cuales reflejan el contexto o el estado del receptor) se usan para escoger de entre varios objetos destino. Debido a que se evalúan dinámicamente las condiciones que representan el contexto o el estado del receptor, la estructura de la herencia y la delegación cambia apropiadamente. Las implementaciones se pueden llevar a cabo con inserción de comportamiento. La inserción de comportamiento esta definido como remplazar o extender el comportamiento de un objeto dinámicamente (en tiempo de ejecución) con un nuevo comportamiento. En el modelo de objetos filtros de integración esto se puede lograr remplazando un objeto, lo cual es el objetivo del filtro *Dispatch*, con otro objeto. Al añadir un filtro *Dispatch* a un objeto se puede proporcionar herencia y/o delegación dinámica. El filtro *Dispatch* mandará los mensajes recibidos a diferentes objetos destino dependiendo de los resultados de los métodos de condición. Si el objeto destino está encapsulado dentro del objeto en sí, la herencia es simulada. Si el objeto destino está fuera del límite de la encapsulación del objeto receptor, entonces esto se vuelve un mecanismo de delegación. Debido a que el filtro *Dispatch* puede mandar mensajes a diferentes objetos, la herencia y delegación múltiple se soporta satisfactoriamente. Los métodos de condición capturan el estado del objeto, el cual puede cambiar en tiempo de ejecución. Los resultados de los métodos de condición afectan además el proceso de atención del mensaje y por lo tanto la herencia y la delegación pueden cambiar dinámicamente.
- **Declaración excesiva de clases** (*todos los dominios de aplicaciones*), las declaraciones de clase excesivas se pueden presentar cuando se necesitan todas las combinaciones de algunas clases y declaradas como clases separadas. Ya que cada combinación posible debe ser expresada explícitamente en la jerarquía de herencia, creando una jerarquía inmanejable. Por ejemplo en una aplicación gráfica, la cual necesita dibujar puntos en la pantalla. La clase *Punto* representa tal despliegue. La clase *PuntoHistoria* hereda de *Punto* manteniendo una lista que contiene la secuencia de la localización de los puntos que ha tenido. La clase *PuntoLimitado* tiene restringidas las coordenadas de despliegue y además hereda de *Punto*. La clase *PuntoHistoriaLimitado* combina las características de *PuntoHistoria* y *PuntoLimitado* y por lo tanto hereda de estas dos clases. Supóngase que ahora se requiere dibujar líneas, y para ello se necesita definir la clase

[16] al utilizar la herencia asociativa y la delegación asociativa , el objeto cliente puede afectar la jerarquía de herencia así como la estructura de delegación a algún contexto.

LineaContinua la cual hereda de *Punto*. Luego *LineaPunteada* hereda de *LineaContinua* pero dibuja líneas punteadas en lugar de líneas continuas. Si todas las combinaciones de estas clases son significativas entonces se necesitan declarar todas las posibles combinaciones como clases separadas: *Punto*, *LineaContinuaHistoria*, *LineaContinuaLimitado*, *LineaContinuaHistoriaLimitado*, *LineaPunteadaHistoria*, *LineaPunteadaLimitado*, *LineaPunteadaHistoriaLimitado*. La jerarquía de herencia crecerá más si se suman más clases a la jerarquía. Por ejemplo, si se crea una clase *Punto3D* como una subclase de *Punto* entonces se duplicará el número de declaraciones de las clases. Para resolver esto se puede usar un filtro *Dispatch* y la herencia asociativa para evitar las declaraciones excesivas de clases. Un cliente puede escoger la combinación precisa de superclases de las que puede escoger al poner las condiciones apropiadas cuando se creen las instancias.

- **Herencia vs sincronización**(*concurrencia y sincronización, interfaces de usuario*), los conflictos entre la herencia y la sincronización pueden aparecer cuando una clase que implementa la sincronización se extiende en una subclase al introducir nuevos métodos, redefinir otros, o al añadir restricciones de sincronización. Cuando se necesitan redefiniciones adicionales, surge un problema el cual es conocido como anomalía de la herencia. Por ejemplo cuando el código de sincronización se integra con el código de aplicación(dentro de un método) cambiar la sincronización es imposible sin afectar el código de la aplicación. Otra fuente de las anomalías de la herencia viene de las especificaciones de sincronización que no pueden ser descompuestas en algunos lenguajes. Es por lo tanto difícil sumar una nueva restricción de sincronización o definir una parte de la especificación de la sincronización en una subclase. El acceso a un objeto puede estar sincronizado al retardar los mensajes condicionalmente. Si el objeto no esta listo para aceptar un mensaje; el mensaje debe ser retardado o frenado hasta que el objeto este listo. La especificación de la sincronización puede ser integrada con la implementación de los métodos. Esto hace difícil cambiar, extender o reutilizar las especificaciones. En este problema se puede usar un filtro *Wait* para especificar las restricciones de sincronización y con esto eliminar las anomalías de la herencia. La condición de restricción de sincronización se especifica por los métodos de condición como una abstracción del estado del objeto. Un filtro Wait define un mapeo desde estas condiciones hacia los mensajes a los cuales se aplica la restricción de sincronización. La reutilización y la extensión de la sincronización es posible sin redefiniciones innecesarias.
- **Interfaces múltiples** (*todos los dominios de aplicaciones*), no todas las operaciones de un objeto son necesariamente del interés de otros objetos que utilizan sus servicios. Por lo tanto es deseable definir interfaces para un objeto, diferenciando entre clientes, esto es, entre los emisores de los mensajes. Por ejemplo, un buzón de correo público debería hacer la distinción entre el cartero y los demás, ya que a todo mundo se le permite poner una carta dentro de este, pero sólo al cartero se le permite vaciar el buzón. La encapsulación es un requerimiento básico por los modelos *OO*. No obstante, muchas aplicaciones requieren que el conjunto de métodos visibles sean variados en la interfaz. Por ejemplo, dependiendo del objeto cliente, el objeto servidor podría querer cambiar el conjunto de métodos en su interfaz. La mayoría de los modelos *OO* convencionales no pueden cambiar este conjunto dinámicamente. A esto se le llama el problema de interfaces múltiples. En *Smalltalk*, las interfaces múltiples sólo se pueden realizar insertando chequeos explícitos en todos los métodos del objeto. La integración

resultante de ámbitos causa problemas cuando se trata de reutilizar y extender los objetos con interfaces múltiples. Las interfaces múltiples se pueden especificar usando un filtro *Dispatch* o un filtro *Error* con condiciones basadas en el contexto. Estas condiciones se usan para diferenciar entre un cliente (o grupos de clientes).

- **Falta de reflexión**(*todos los dominios, pero típicamente en sistemas de control, sistemas distribuidos e interfaces de usuario*). Recientemente existe una cantidad de trabajo considerable en la reflexión *OO*, por ejemplo, en programación concurrente, en la estructuración de sistemas operativos, en el diseño de compiladores y en la programación de tiempo real. Los métodos convencionales *OO* no proporcionan soporte para el desarrollo de sistemas reflexivos o sólo proporcionan un soporte limitado para la reflexión. Como la reflexión es un concepto base del modelo filtros de integración esta se implementa directamente con un filtro *Meta* y la comunicación entre objetos con los objetos de comunicación abstracta.

2.2 Programación Subjetiva

De acuerdo a la idea tradicional de la orientación a objetos, un objeto encapsula tanto su estado como su comportamiento. Lo que se busca con esto es que el diseñador de una clase defina e implemente intrínsecamente las propiedades y el comportamiento base del objeto y para que los usuarios del objeto puedan utilizarlo tienen que hacerlo mediante las operaciones públicas o interfaz de dicho objeto.
Esta idea clásica es inadecuada para tratar con situaciones en las que vistas diferentes de objetos compartidos se usan en diferentes partes del sistema, por diferentes usuarios o en diferentes tiempos. Por ejemplo:

- La construcción de grupos de aplicaciones grandes, que evolucionan y que manipulan objetos compartidos. Las nuevas aplicaciones generan la necesidad de tener propiedades y comportamientos nuevos. Distintas aplicaciones pueden inclusive necesitar clasificar los mismos objetos de manera diferente.
- Vistas Múltiples. Por ejemplo un arreglo de datos se puede desplegar como un histograma o una gráfica de pastel. Estos despliegues diferentes pueden ser tomados como diferentes puntos de vista sobre el mismo objeto. Las rutinas de despliegue pueden estar directamente asociadas con el arreglo de objetos e inclusive tener acceso al estado encapsulado. Estas pueden no ser consideradas propiedades internas de los arreglos y deberia ser posible escribirlas como aplicaciones separadas.
- Versiones. A los diferentes estados asociados a algún objeto o aplicación puede ser visto como subjetividad. Los usuarios o las aplicaciones pueden encontrar conveniente ir pasando de uno a otro estado.

2.2.1 Subjetividad

Albert Einstein dijo alguna vez que "todo es relativo", y también se aplica a la computación, ya que no existe una sola perspectiva desde la cual se puedan ver todos los aspectos de un objeto; dependiendo de la perspectiva que se tome, algunos aspectos pueden estar completamente escondidos, mientras que otros pueden parecer distorsionados. Esta simple observación tiene importancia para la reutilización del software. Por ejemplo en una aplicación que maneje libros; un libro podría ser un objeto con atributos como: nombre del autor, título, tema, editorial, etc. Estos serían atributos naturales si la aplicación necesitara recuperar los libros sobre la base de quien es el autor, contenido o título; sin embargo no serían apropiados, si la aplicación mantuviera información de las existencias y los tomos para una bodega (donde el autor y el tema son irrelevantes), o si la aplicación registrará los materiales usados en la fabricación de los mismos(donde también el tema y el título son irrelevantes). Claramente, los datos y las operaciones que son encapsulados por un objeto variarán de aplicación a aplicación. Esto impacta a la reutilización del software: los objetos escritos para un aplicación pueden no ser reutilizables en otra aplicación porque sus vistas o perspectivas son incompatibles, aunque ambas aplicaciones traten con el mismo objeto en la vida real.
El principio de subjetividad afirma que no hay interfaz sencilla que pueda describir adecuadamente un objeto, los objetos son descritos por una familia de interfaces

relacionadas. La interfaz apropiada para un objeto es dependiente de la aplicación, es decir, es subjetiva.

Como resultado de la no concordancia entre la objetividad de los lenguajes de programación actuales y la subjetividad de los dominios de los problemas, se debe crear un mecanismo adecuado cada vez que aumente un problema que incluya perspectivas múltiples. Esto sugiere que sería provechoso soportar la subjetividad como principio fundamental de los sistemas *OO* al insertarla dentro de los lenguajes de programación.

2.2.2 Enfoques de Diseño y Desarrollo de Software Orientados a la Subjetividad

En la programación subjetiva se utiliza una nueva unidad de software llamada "subject", aunque aquí se referirá a esta como aspecto. Una buena modularización del software consiste de colecciones de módulos separados y que la mayoría de los cambios que se tienen que hacer deben realizarse por medio de añadir o reconfigurar estos módulos, en vez de que se tenga que modificar el código que ya se tenía hecho.

Cuando un enfoque orientado a la subjetividad se usa en el desarrollo de software, muchas actividades que previamente requerían modificaciones de programas ya existentes se pueden realizar por medio de cierta integración de módulos.

Los siguientes modelos de diseño[P5] difieren en términos de cuando y porque se realiza la descomposición y consecuentemente en que es lo que se va a descomponer. La habilidad de usar estas y otras técnicas similares es probablemente el principal beneficio de la programación subjetiva.

- *Basado en Características:* las aplicaciones complejas comúnmente proporcionan múltiples características. Las configuraciones diferentes de la aplicación para diferentes usuarios pueden requerir la inserción de características diferentes o escogidas de entre las implementaciones alternativas de una característica. Las aplicaciones estructuradas de acuerdo a los requerimientos tienden a tener *aspectos* que corresponden a características. Las diferentes configuraciones se pueden obtener fácilmente como integraciones diferentes de los aspectos de características. Una importante característica de una aplicación es su interfaz de usuario. Mucho del trabajo de investigación que se ha hecho en el pasado ha sido sobre la separación de la interfaz de la aplicación. La programación subjetiva proporciona una forma natural para hacer esto: la interfaz de usuario es un aspecto (o colección de aspectos) que se integra con los módulos que proporcionan el procesamiento base. Se pueden proporcionar diferentes interfaces de usuario y la que se desea puede ser usada en la integración. Cuando se requieran nuevas características conforme el sistema evolucione: se puede hacer por medio de aspectos de extensión, manteniendo la correspondencia entre las características y los aspectos conforme el sistema crezca. Por ejemplo, un procedimiento a seguir puede ser: descomponer un problema en características separadas, tal vez con una característica base distinguida que contenga funcionalidad común, diseñar e implementar cada una de estas características de manera separada, luego integrar las implementaciones de las características para producir el sistema completo.
- *Basado en los Requerimientos:* el software se escribe para cumplir ciertos requerimientos, y se modifica para cumplir con nuevos requerimientos. Típicamente, el

código escrito para cumplir un conjunto particular de requerimientos se esparce por todo el sistema, intercalado con el código escrito para cumplir otros requerimientos. El mantener la correspondencia entre los requerimientos y el código es un problema clásico en la ingeniería de software. La programación subjetiva soporta una metodología de desarrollo en la cual un sólo aspecto se diseña y codifica para sólo un conjunto de requerimientos; se integran todos los aspectos para producir un sistema que cumpla todos los requerimientos. La estructura del sistema es una integración de aspectos y por lo tanto directamente refleja los conjuntos de requerimientos. Este enfoque tiene mucho valor durante el desarrollo de la aplicación inicial y todavía más durante el mantenimiento. A medida que se identifiquen nuevos requerimientos, se pueden escribir nuevos aspectos para soportarlos y después los nuevos aspectos se integran con la aplicación original.

- *por Etapas:* en este modelo lo que se hace es: definir el problema y diseñar el sistema, después descomponer el diseño dentro de etapas de implementación, cada una de las cuales extiende su predecesor; implementar estas etapas sucesivamente extendiendo el sistema a través de la integración. Por ejemplo, un compilador orientado a la subjetividad cuya implementación esta dividida dentro de dos partes: los constructores básicos de C++ y un soporte de *templates* (plantillas); aquí se deben implementar las características base como etapa uno, después de que el compilador este completo, se implementa la etapa dos y se integra con la etapa uno para producir un compilador más completo.
- *Extensiones no planeadas:* Cualquier aplicación exitosa debe evolucionar durante su ciclo de vida para alcanzar las nuevas necesidades del usuario. No hay un desarrollador por muy astuto que este sea que pueda anticiparse, cuando esta en el diseño original, de la naturaleza de las futuras extensiones. Un buen diseñador se anticipará a algunos tipos de extensiones futuras y por lo tanto construirá los puntos de enlace que permitirán a esas extensiones ser añadidas cuando sea necesario. Pero llegará el tiempo en que se requerirá una extensión no anticipada, para la cual no se hayan construido puntos de enlace. Con la tecnología convencional, tal extensión sólo puede ser hecha editando y modificando el código interno del sistema. Una aplicación orientada a la subjetividad siempre puede ser extendida por medio de alguna integración. Si los objetos de la aplicación que necesitan extensión fueron inicialmente diseñados como objetos expuestos en una etiqueta del aspecto, la extensión puede ser escrita como un aspecto separado y luego ser añadido en la integración. La integración puede llevarse a cabo sin accesar al código fuente original de la aplicación. El conocimiento del código fuente puede serle útil al programador al escribir la extensión, pero es posible escribir las extensiones con sólo la referencia a las etiquetas del aspecto y a la documentación. Por ejemplo, un navegador de páginas web al cual se le debe añadir funcionalidad para extensiones del protocolo *HTTP*. Se implementa el navegador, luego se diseñan los componentes adicionales para la nueva funcionalidad asociada con la extensión del protocolo, se implementa este diseño como una extensión y se integra con el navegador inicial.
- *Integración no planeada* Existe actualmente una colección creciente de aplicaciones ya disponibles y con esto una frustración también creciente por parte de los usuarios debido a que las aplicaciones no trabajan bien conjuntamente. Los esfuerzos de integración actuales involucran un gran trabajo de programación detallada y la modificación de las aplicaciones. Las aplicaciones orientadas a la subjetividad

mantienen la promesa de ser capaces de integrarse en formas no anticipadas por los creadores originales. Para que esto resulte, los creadores deben haber planeado que sus aplicaciones serán integradas, pero no necesitan anticiparse a que tipos de otras aplicaciones estarían involucradas, ni que puntos de enlace serían necesarios para la interacción entre aspectos, ni haber alcanzado un acuerdo con alguien acerca de los detalles de la aplicación. Las diferencias entre las aplicaciones, inevitablemente en este mundo donde se crean aplicaciones separadas sin cuidado, con un control centralizado, pueden ser potencialmente resueltas por las reglas de integración. El desarrollador realiza la integración, estudia las etiquetas de la aplicación, la documentación y escribe una regla de integración para enlazar conjuntamente las aplicaciones apropiadamente. Por ejemplo, un navegador de documentos de hipertexto y una facilidad para descargar archivos remotos, desarrollados separadamente. Estos pueden ser integrados para permitir ligas remotas en el hipertexto. El código puente asocia las ligas de hipertexto con peticiones para descargar archivos.

2.2.3 Programación Orientada a la Subjectividad

El enfoque *Subject Oriented Programming* (SOP) fue propuesto por *Harrison* y *Ossher* de IBM Watson Research Center como una extensión al paradigma *OO* para direccionar el problema de manejar diferentes perspectivas subjetivas sobre los objetos que se modelan. Por ejemplo, el objeto libro del departamento de mercadotecnia de una editorial incluiría atributos como área, sinopsis, etc, mientras que el departamento de fabricación estaría interesado en diferentes características como el tipo de papel, tipo de cubierta, etc. La utilización contextual de un objeto no es la única razón para diferentes perspectivas. El enfoque además busca direccionar el problema de integrar sistemas que fueron desarrollados de manera independiente, por ejemplo las dos aplicaciones diferentes para una editorial. En este caso, tenemos que tratar con perspectivas de diferentes equipos de desarrollo sobre los mismos objetos (por ejemplo el objeto libro).
La meta es llegar a ser capaz de sumar extensiones imprevistas a un sistema existente en una forma no invasiva. Por lo tanto las perspectivas subjetivas pueden corresponder a diferentes usuarios finales, contextos y desarrolladores.
Cada perspectiva se encuentra en un aspecto. Un aspecto, en este enfoque, es una colección de clases y/o fragmentos de estas relacionados por medio de la herencia y otras relaciones pertenecientes al aspecto. Por lo tanto, un aspecto es un modelo de objetos completo o parcial. Los aspectos pueden ser integrados usando sus propias reglas de integración. Existen tres tipos de reglas de integración :

- reglas de correspondencia
- reglas de combinación
- reglas de correspondencia y combinación

Las reglas de correspondencia especifican precisamente eso, la correspondencia, si es que existe alguna, entre clases, métodos y atributos de un objeto pertenecientes a diferentes aspectos. Por ejemplo, se podría usar una regla de correspondencia para expresar que el libro en la aplicación del departamento de mercadotecnia es el mismo libro en la aplicación del departamento de fabricación(inclusive si las clases correspondientes tuvieran diferentes nombres).

Además se pueden tener reglas de correspondencia estableciendo la misma entre los métodos y atributos de estas clases. Alternativamente, es posible usar una regla de correspondencia especial la cual establecería la correspondencia de las dos clases y la de todos sus miembros que tengan nombres iguales. Por supuesto, se puede rescribir esta correspondencia automática para algunos miembros al definir reglas de correspondencia de miembro adicionales. También, se pueden usar reglas de combinación para especificar como se van a combinar las dos clases. La clase resultante incluirá los métodos y atributos independientes y los correspondientes serán combinados de acuerdo a las reglas de combinación. Por ejemplo, un método de algún aspecto podría rescribir los métodos correspondientes de los otros aspectos o todos los métodos correspondientes podrían ser ejecutados en algún orden especificado. Cuando se combinan dos aspectos desarrollados de manera independiente, se desarrollara un tercero el cual incluye el código necesario para combinar los otros dos. Para conveniencia, además existen reglas de correspondencia y combinación las cuales proporcionan un atajo para especificar las reglas de correspondencia y combinación al mismo tiempo.

Una aplicación o un conjunto de aplicaciones orientadas a la subjetividad, se construye al integrar una colección de aspectos. Cada aspecto, en este enfoque, es en sí mismo un programa *OO*, aunque la mayoría de las veces será uno incompleto. La aplicación consiste de una colección de clases (agrupadas en una clasificación jerárquica, de manera estándar) y además está escrita en un lenguaje *OO* estándar.

Las clases en un aspecto, definen una vista subjetiva de una colección de objetos apropiados para un propósito particular. Por ejemplo[P6], si se tienen dos sistemas: uno de transportación y el otro de envío de paquetes. La aplicación de envío define una vista particular de intercambios: objetos donde se pueden empaquetar cosas o elementos, con dimensiones, capacidad y propiedades similares. La aplicación además define otros objetos apropiados como cajas. Todas estas definiciones crean el "*Aspecto Envío*". Por el otro lado, la aplicación transportación, define una vista diferente del objeto intercambio: como un objeto que puede viajar de acuerdo a una ruta establecida, con propiedades tales como el rango que puede viajar con un sólo tanque de gasolina. Además define otros objetos apropiados para la transportación, como las ciudades. Estas definiciones crean el "*Aspecto Transportación*".

Los esquemas de estos dos ejemplos se presentan en el *código 2.2.1*. Se utiliza sintaxis de *C++*. Cuando se componen estos dos *aspectos*, los detalles de las clases correspondientes se combinan para producir el *aspecto* compuesto mostrado en el *código 2.2.2*. Este *aspecto* combinado, claramente satisface las necesidades de ambas aplicaciones.

```
// Aspecto Envio
class Intercambio  {
  public :
    float capacidad() {...};
    // Otras operaciones públicas
  private
    float longitud, ancho, altura;
    // Otras operaciones y datos internos
};

class Caja  {
  public :
    float capacidad(){...};
    void empaquetar(ListaElem &elem) {...};
```

```
    // Aspecto Transportación
class Intercambio  {
  public :
    void poner_Ruta(ruta &r) {...};
    float rango(){...};
    // Otras operaciones públicas
  private :
    ruta &RutaPlaneada;
    float CapacidadTanqueGas;
    // Otras operaciones y datos internos
};

class Ciudad  {
  public :
```

```
  // ...                                    char *nombre() {...};
};                                          // ...
                                          };
```

Código 2.2.1 Esquemas de los Aspectos Envio y Transportacion

```
class Intercambio  {
  public :
    float capacidad() {...};
    void poner_Ruta(ruta & r) {...};
    float rango(){...};
    // Otras operaciones públicas
  private
    float longitud, ancho, altura;
    ruta &RutaPlaneada;
    float CapacidadTanqueGas;
    // Otras operaciones y datos internos
};

class Caja  {
  public :
    float capacidad(){...};
    void empaquetar(ListaElem &elem) {...};
     // ...
};

class Ciudad  {
  public :
    char *nombre() {...};
    // ...
};
```

Código 2.2.2 Clase Compuesta

Por supuesto, cada definición de clase combinada, como la de la clase Intercambio en el *código 2.2.2* ya compuesta, podría escribirse directamente, en vez de ser derivada en el curso de integrar dos aspectos. Aunque, esto llevaría al problema de la negociación entre los desarrolladores de aplicaciones y los propietarios de las clases. La definición de la clase combinada es una pieza de código, la cual debe ser programada y mantenida en cualquier enfoque razonable por una persona, y la cual contiene detalles de ambas aplicaciones. La ventaja de este enfoque es que los dos aspectos pueden ser escritos y mantenidos de manera separada y después ser integrados.

La integración se realiza en código objeto: por lo tanto, el código fuente del aspecto combinado mostrado en el *código 2.2.2*, no se produce como tal. En lugar de eso cada aspecto se compila de manera separada para producir un aspecto binario.

El aspecto binario consiste de una etiqueta que proporciona información acerca de este, y el código binario producido por el compilador. El integrador orientado a la subjetividad usa información de las etiquetas para unir los aspectos. No examina o modifica el código binario de los aspectos individuales.

Una etiqueta de un *aspecto* binario consiste de tres partes :

1. Un esquema, que define las clases que el aspecto usa y/o proporciona, y su clasificación jerárquica y atributos desde el punto de vista de este aspecto.

2. Interfaces, que definen las operaciones(funciones genéricas) que el aspecto usa y/o proporciona.
3. Especificaciones de estructura, que definen detalles de como sería integrado este aspecto desde aspectos secundarios y como las operaciones son mapeadas hacia puntos de enlace(de métodos) en el código binario del aspecto para las diferentes clases definidas en el esquema. El código de los métodos en si mismo no aparece en la etiqueta.

Un desarrollador de aspectos puede seleccionar los detalles que serán expuestos en la interfaz y las partes del esquema de la etiqueta, de ese modo controla que detalles se esconden totalmente en el aspecto y que detalles pueden ser compartidos por otros aspectos.

A diferencia del ejemplo anterior que es muy sencillo, los aspectos integrados pueden definir un número de diferentes clases, operaciones y atributos, algunos de los cuales pueden ser compartidos por los aspectos y algunos otros no. La regla de integración debe especificar como se corresponden las clases, operaciones y atributos en los diferentes aspectos. Por ejemplo una aplicación de mantenimiento de vehículos se puede escribir en términos de una clase *Vehiculo*. Cuando esta se integra con el aspecto *Transportación*, visto anteriormente, la regla de integración establecerá que *Intercambio* en el *aspecto Transportación* corresponde a *Vehiculo* en el aspecto *Mantenimiento*. Esta regla simple dará la funcionalidad de mantenimiento de vehículos a los objetos *Intercambio*.

Las reglas de integración pueden especificar correspondencia en términos simples y genéricos (como la opción que se tiene por defecto basado en la correspondencia de los nombres) o en detalle. Las especificaciones detalladas de la correspondencia de la operación pueden incluir transformación de parámetros. Las especificaciones detalladas de la correspondencia de los atributos identifica explícitamente que atributos se comparten por múltiples *aspectos*.

Las reglas de integración pueden además especificar una variedad de formas para combinar las clases correspondientes. Por ejemplo, si las clases correspondientes en dos aspectos, proporcionan métodos para las operaciones que se corresponden, varias opciones están disponibles, incluyendo:

- Ejecutar ambos métodos donde quiera que se llame a la operación . Esta opción esta por defecto, ya que esto efectivamente combina la funcionalidad de los dos aspectos. Hay varias sub opciones para manejar los valores regresados.
- Ejecutar uno especifico de los dos métodos. Esto permite a un aspecto esconder al otro.

Se esta además explorando la integración dinámica, en la cual al estar corriendo los aspectos se pueden componer sin interrumpir su ejecución. Esto permitiría que se extendieran las aplicaciones que ya están corriendo o que empezaran a cooperar con otras aplicaciones según el usuario lo requiera.

2.3 Descomposición Multidimensional

La descomposición en ámbitos, como se vio anteriormente, es fundamental en la ingeniería de software. En su forma más general se refiere a la habilidad de identificar, encapsular, modularizar y manipular sólo aquellas partes del software que son relevantes para un concepto particular, meta o propósito. Los ámbitos son la motivación primaria para organizar y descomponer el software en partes manejables y comprensibles. Muchos tipos de ámbitos diferentes pueden ser relevantes para diferentes desarrolladores en diferentes roles, o en diferentes etapas en el ciclo de vida del software; por ejemplo, el tipo de ámbito que prevalece en la *POO* es la clase: cada ámbito de este tipo es un tipo de datos definido y encapsulado por una clase. Las características como el manejo de excepciones y la sincronización son también ámbitos comunes como también lo son el control de la concurrencia, la distribución, los roles, los puntos de vista, las variantes y las configuraciones. A un tipo de ámbito(por ejemplo el ámbito clase) se le llama *dimensión de ámbito*. La descomposición en ámbitos involucra la descomposición del software de acuerdo a una o más dimensiones de ámbitos. Lograr una limpia separación de ámbitos puede llevar a:

- reducir la complejidad del software y a mejorar la claridad del mismo.
- permitir la identificación eficiente de aspectos específicos a lo largo de todo el ciclo de vida del software.
- limitar el impacto de cambio, facilitando la evolución, la adaptación no invasiva y personalización.
- facilitar la reutilización.
- simplificar la integración de componentes.

Estas metas tan importantes, no se han alcanzado en la práctica. Principalmente porque el conjunto de ámbitos relevantes varía con el tiempo y es sensible al contexto; por ejemplo diferentes actividades de desarrollo, diferentes etapas en el ciclo de vida del software, diferentes desarrolladores, etc, y frecuentemente involucran tipos muy diferentes de ámbitos y por consecuencia múltiples dimensiones. La descomposición en una dimensión puede estimular algunas metas y actividades, mientras que puede impedir otras; por lo tanto cualquier criterio de descomposición e integración será apropiado para algunos contextos y requerimientos, pero no para todos. Por ejemplo, la descomposición por clases, en los sistemas *OO* facilita enormemente la evolución de los detalles de estructuras de datos debido a que se encapsulan dentro de clases sencillas(o en algunas relacionadas), pero impide la adición o evolución de características, porque estas típicamente incluyen métodos y variables de instancia en múltiples clases. Más aún, múltiples dimensiones de ámbitos pueden ser relevantes simultáneamente, y estas pueden traslaparse e interactuar. Por lo tanto, la modularización de acuerdo a diferentes dimensiones de ámbitos se necesita para diferentes propósitos: algunas veces por clase, algunas por característica y algunas otras por puntos de vista, aspectos, roles u otro criterio.

2.3.1 Tiranía de la Descomposición Dominante

Los ingenieros de software deben ser capaces de identificar, encapsular, modularizar y manipular múltiples dimensiones de ámbitos simultáneamente, y deben ser capaces de

introducir nuevos ámbitos y dimensiones en cualquier punto durante el ciclo de vida del software, sin sufrir de los efectos de la modificación invasiva y rediseño.
Los lenguajes y las metodologías modernas sufren de un problema que se conoce como *la tiranía de la descomposición dominante*: en la cual se permite la descomposición y encapsulación de sólo un tipo de ámbito a la vez. Ejemplos de descomposiciones tiránicas son las clases (en lenguajes *OO*), funciones (en lenguajes funcionales) y reglas (en sistemas basados en reglas). Por ejemplo es imposible encapsular y manipular características en el paradigma *OO*, u objetos en sistemas basados en reglas. Por lo tanto es imposible obtener los beneficios de diferentes dimensiones de descomposición a través del ciclo de vida del software. Los desarrolladores están forzados a casarse con una sola dimensión. Cuando el software se descompone en módulos basados en una sola dimensión de ámbito dominante, el software que direcciona otros ámbitos no se puede localizar fácilmente: se disemina a través de muchos módulos, y dentro de la mayoría de estos, se enmarañan con software que direcciona otros ámbitos.
Se cree que la tiranía de la descomposición dominante es la causa más significativa, hasta ahora, para no alcanzar muchos de los beneficios esperados de la descomposición en ámbitos.

2.3.2 Rompiendo la Tiranía

Se utiliza el término descomposición multidimensional para denotar la descomposición en ámbitos que comprende:

- Dimensiones de ámbitos múltiples y arbitrarias.
- Separación en esas dimensiones simultáneamente. Ninguna dimensión debe excluir la descomposición en otra dimensión.
- La habilidad de manejar nuevos ámbitos, y nuevas dimensiones de ámbitos, dinámicamente, conforme estas surjan a lo largo del ciclo de vida del software.
- Ámbitos interactuando y traslapandose.

El contar con un soporte completo para la descomposición multidimensional abre la puerta a la tan demandada remodularización, permitiendo a un desarrollador escoger en cualquier momento la mejor modularización basada en cualquiera o todos los ámbitos, para la tarea de desarrollo en cuestión.
La descomposición multidimensional representa un conjunto de metas muy ambiciosas. Ellas se aplican sin importar el lenguaje de desarrollo de software o paradigma. Un enfoque que esta en evolución, desarrollado por *IBM*, son los hiperespacios: los cuales cuentan con una herramienta llamada *HyperJ*, la cual proporciona soporte para la descomposición multidimensional para *Java*.

2.3.3 Hiperespacios

Los hiperespacios permiten la identificación explícita de cualquier dimensión y ámbito de importancia, en cualquier etapa del ciclo de vida del software, además permite la encapsulación de los ámbitos involucrados, la identificación y manejo de las relaciones entre estos y la integración de los mismos.

2.3.3.1 Espacio de Ámbitos

Una unidad, en los hiperespacios es un constructor sintáctico. Una unidad puede ser por ejemplo una declaración, sentencia, clase, interfaz, especificación de requerimiento, o cualquier otra entidad coherente que pueda ser descrita en un lenguaje determinado. Se distinguen unidades primitivas que son tratadas como atómicas de unidades compuestas, las cuales agrupan unidades conjuntamente. Por ejemplo un método, una variable de instancia, o un requerimiento de alguna tarea determinada, pueden ser tratados como unidades primitivas; mientras que una clase, o un paquete pueden ser tratados como unidades compuestas.
Un espacio de ámbitos abarca todas las unidades como un conjunto de librerías de componentes o una familia de productos.
El trabajo del espacio de ámbitos es organizar las unidades para separar todos los ámbitos importantes y para describir varios tipos de interrelaciones entre ellos y para indicar como se pueden construir e integrar los componentes de las unidades que direccionan estos ámbitos.

2.3.3.2 Identificación de Ámbitos

Un hiperespacio es un espacio de ámbitos especialmente estructurado para soportar la descomposición multidimensional. Su primera característica distinguible es que sus unidades están organizadas en una matriz multidimensional. Cada eje representa una dimensión de ámbito y cada punto en el eje un ámbito en esa dimensión. Esto hace explícitas todas las dimensiones de interés, los ámbitos que pertenecen a cada dimensión y que ámbitos se afectan por que unidades. Las coordenadas de una unidad indican todos los ámbitos que afecta: la estructura clarifica que cada unidad afecta exactamente un ámbito en cada dimensión.
Cada dimensión puede por lo tanto ser vista como una partición del conjunto de unidades, es decir, una descomposición de software particular. Cada ámbito sencillo dentro de una dimensión define un hiperplano que contiene todas las unidades que afectan a ese ámbito. La estructura de matriz permite un tratamiento uniforme de todos los tipos de ámbitos y permite a los desarrolladores recorrer a través de la matriz de acuerdo a cualquier ámbito deseado.

2.3.3.2.1 Unidades

Hasta la fecha IBM ha trabajado con unidades a nivel de declaraciones (por ejemplo: métodos, funciones, clases) en vez de sentencias o expresiones. *HyperJ* trata a las funciones y variables de instancia de Java como unidades primitivas, y a las interfaces, clases y paquetes como unidades compuestas.

2.3.3.2.2 Especificaciones de Ámbitos

Las especificaciones de ámbitos en los hiperespacios sirven para identificar las dimensiones y sus ámbitos, y para especificar las coordenadas de cada unidad dentro de la matriz. Un enfoque simple, usado en *HyperJ*, es a través de un conjunto de mapeos de ámbitos de la forma

x : dimensión.ámbito

donde *x* es el nombre de una unidad o colección de unidades(por ejemplo un directorio o paquete) o un patrón representando muchas unidades o colección de unidades.

2.3.3.3 Encapsulación de Ámbitos

La matriz de ámbitos identifica ámbitos y organiza las unidades de acuerdo a dimensiones y ámbitos. Permite muchos conjuntos útiles de unidades a ser identificados basados en los ámbitos que afectan, por ejemplo todas las unidades pertenecientes a un ámbito, o a todos los ámbitos(áreas de traslape), o a un ámbito pero no a otro. No obstante, la matriz, en sí misma, no soporta la encapsulación de ámbitos ya que los conjuntos de unidades no pueden simplemente ser tratados como módulos sin algún mecanismo adicional. En los hiperespacios, ese mecanismo es el hipercorte el cual es un conjunto de ámbitos que están perfectamente especificados. Este enfoque por lo tanto promueve una configuración flexible y la reutilización de los hipercortes, y es crucial para lograr un impacto limitado cuando surjan cambios.
Ya que cualquier conjunto de unidades puede llegar a ser un hipercorte, los ámbitos arbitrarios se pueden encapsular usando hipercortes. Por lo tanto, cualquier limitación del lenguaje y/o cualquier ámbito extra siempre es posible crearlo en un hipercorte que contenga aquellas unidades pertenecientes a ese ámbito.

2.3.3.4 Integración de Ámbitos

Los hipercortes son bloques de construcción; y estos pueden ser integrados para formar bloques de construcción más grandes y eventualmente, sistemas completos. Un tipo de relación de unión entre unidades es la correspondencia. La correspondencia es una relación sensible al contexto; ocurre dentro del contexto de la integración de un componente o algún sistema en particular, una misma unidad puede estar asociada, por ejemplo, con diferentes unidades de implementación en diferentes sistemas. En un hiperespacio, este contexto de integración es un hipermódulo.
Un hipermódulo comprende un conjunto de hipercortes integrados y un conjunto de relaciones de integración, las cuales especifican como los hipercortes se relacionan entre sí, y como deben ser integrados. La correspondencia es una relación de integración importante, la cual indica que unidades específicas dentro de los diferentes hipercortes van a ser integrados entre sí. No obstante, se necesitan los detalles adicionales para especificar como ocurre la integración. Por ejemplo si dos métodos se corresponden, uno debe reescribir al otro en el sistema integrado o si los dos se van a ejecutar, si los dos en que orden y como debe ser manejado el valor de regreso, si los tipos de sus parámetros no corresponden, que transformaciones se necesitan para reconciliarlos. Las relaciones de integración en *HyperJ* extienden las reglas de integración de la programación orientada a la subjetividad.
Ya que los hipercortes no dependen entre sí directamente, el cambiar una definición o implementación es no invasivo; estos sólo requieren la redefinición de las relaciones de integración. La correspondencia por lo tanto proporciona una gran flexibilidad y soporta directamente la substitución.

2.3.4 HyperJ

HyperJ permite la descomposición e integración de ámbitos multidimensional en *Java* estándar: facilitando así la adaptación, descomposición e integración modular y además la remodularización de los componentes en *Java* y además se hace de manera no invasiva.

Con *HyperJ* los desarrolladores pueden descomponer un programa en la forma en que ellos mejor consideren. Pueden crear nuevos módulos separados, encapsulando estos ámbitos desde cero, sin modificar el resto del programa o interferir con el trabajo de otros desarrolladores, además pueden extraer tales módulos de programas ya existentes. Se pueden integrar algunos o todos estos módulos para producir programas ejecutables en maquinas virtuales de *Java* estándar. Pueden inclusive crear múltiples sistemas de descomposición simultáneamente por ejemplo por objeto, por característica, etc., y pueden añadir nuevas descomposiciones en cualquier etapa del ciclo de vida del desarrollo del software. *HyperJ* ayuda a manejar las interacciones a través de las diferentes descomposiciones.

Hyperj proporciona una capacidad de integración poderosa, la cual puede ser usada para combinar ámbitos separados selectivamente en un programa o componente. Por ejemplo se puede usar para crear una versión de un sistema de software que contenga algunas características, pero no otras, inclusive si el sistema original no fue escrito con esas características separadas, también puede ser usado para extender o adaptar algún componente.

HyperJ se puede utilizar en cualquier etapa del ciclo de vida de software. Cuando se usa durante el diseño o implementación de componentes del sistema, *HyperJ* permite a los desarrolladores diseñar el sistema o componentes separando todos los ámbitos de importancia desde el inicio. Cuando se usa durante la integración del sistema, el mecanismo de integración de *HyperJ* se puede usar para integrar el software desarrollado de manera separada, incluyendo los componentes reutilizables, y para personalizar y adaptar el software como sea necesario para usarlo en un contexto particular. Cuando se usa durante la evolución del sistema, el mecanismo de descomposición de *HyperJ* permite a los desarrolladores centrarse en sólo esas piezas del sistema que son relevantes sólo para la evolución, y sus mecanismos de integración y adaptación hacen muchas formas posibles de evolución sin cambios al código existente. Cuando se usa durante la reingeniería, introduce nuevas descomposiciones sin cambios en el código original.

Actualmente *HyperJ* soporta las siguientes unidades: paquetes, interfaces, clases. Soporta una matriz de ámbitos de estas unidades, y la habilidad de hacer hipercortes de conjuntos de unidades y luego integrar estos hipercortes en hipermódulos. Esto genera archivos de clase *Java* para todos los hipermódulos producidos. Estos pueden ser ejecutados, si es que están completos o usados como bloques de construcción para un desarrollo futuro.

Al programar *OO* frecuentemente surgen los siguientes problemas:

- Todas las subclases de una clase ya existente deben extenderse con algún comportamiento nuevo.
- Los objetos creados por código ya existente deben extenderse con algún comportamiento nuevo.

Considérese, como un ejemplo sencillo[P7], un sistema de pagos con una clase central llamada *Factura*. Las facturas se entregan llamando al método *entregar()* en la clase *Factura*. Las instancias de *Factura* se crean en múltiples lugares del sistema y existen además múltiples subclases de *Factura*. Considérense también dos usuarios de la clase *Factura*, una subclase *FacturaEspecial* y un cliente *ClienteFactura*:

```
package sistemadecuentas;

class Factura {
  void entregar() { ... }
  ...
}

class FacturaEspecial extends Factura {
  ...
}

class ClienteFactura {
  Factura factura = new Factura();
  void foo() {
    factura.entregar();
  }
}
```

Código 2.3.4.1 Clases Factura, FacturaEspecial y ClienteFactura del paquete sistemadecuentas

Ahora supongamos que esta aplicación evoluciona y necesita que las facturas se entreguen vía e-mail antes de la entrega normal. Ninguna de las técnicas convencionales pueden ayudar a resolver este problema ya que los clientes existentes hacen referencia a la clase ya existente en lugar de a las nuevas clases a implementar. Así que los sistemas *OO* sólo dejan la opción de modificar ya sea la clase *Factura* o los clientes de esta. Y ninguna de estas soluciones es recomendable; porque es de la idea general que los cambios a un sistema deben ser incrementales, en vez de invasivos, porque los cambios invasivos requieren acceso al código fuente y vuelve más complejo el proceso de mantenimiento y documentación.

HyperJ soluciona este problema al extender el sistema con una extensión al mismo sin tocar el código original; creando una nueva clase llamada *extension.Factura* la cual implementa la funcionalidad del envío vía e-mail. La clase *Factura* original junto con sus clientes se unen con la nueva clase en un nuevo hipermódulo (se omitieron las especificaciones de los ámbitos y de los hipercortes, sin embargo en el siguiente capítulo se mostrará un ejemplo donde se especifican todos los componentes que aquí faltaron, por razones de simpleza).

```
package extension;

public class Factura {
  public void mandaMail() {
    ...
  }
  public void entregar() {
    mandaMail();
  }
}
```

Código 2.3.4.2 Paquete extension

```
hypermodule SistemaExtendido
  hyperslices:
    Feature.sistemadecuentas,
    Feature.extension;
  relationships:
    mergeByName;
end hypermodule;
```

Código 2.3.4.3 Archivo de integración de los paquetes sistemadecuentas y extension

Después de realizar la integración, el paquete *SistemaExtendido* contiene las clases *Factura*, *FacturaEspecial* y *ClienteFactura*. Lo interesante es que *FacturaEspecial* y *ClienteFactura* se refieren a *SistemaExtendido.Factura*. Esto significa que ellas usan *SistemaExtendido.Factura* en cualquier lado donde antes usaban *sistemadecuentas.Factura*. En lugar de cambiar la clase *Factura* original, se crean nuevos clientes que se refieren a la clase extendida *Factura*. A este tipo de transformación se la llama migración del cliente.

CAPITULO III

Lenguajes de Aspectos

En este último capítulo se da un panorama general de los tipos de lenguajes *OA* y finaliza este trabajo con el seguimiento de la implementación de una pila. en las formas convencionales y en las *OA*.

3.1 Enfoques de Lenguajes de Aspectos

Se han implementado dos formas para aplicar los conceptos de la orientación a aspectos en los lenguajes de programación: los lenguajes de aspectos de dominio específico y los lenguajes de aspectos de propósito general.

3.1.1 Lenguajes de Aspectos de Dominio Específico vs Propósito General.

Los lenguajes de aspectos de dominio específico sólo soportan uno o algunos de los aspectos (para los que fueron diseñados) que se pueden presentar en los sistemas informáticos. Los lenguajes de aspectos de dominio específico tienen o deberían tener (por ser específico de ese dominio) un nivel de abstracción mayor que el lenguaje base y, por lo tanto, expresan los conceptos del dominio específico en un nivel de representación más alto.
Estos lenguajes normalmente imponen restricciones en la utilización del lenguaje base. Esto se hace para garantizar que los conceptos del dominio específico se programen utilizando el lenguaje diseñado para este fin y así evitar interferencias entre los dos. Se pretende evitar que los aspectos se programen en ambos lenguajes lo cual podría ocasionar algún tipo de conflicto. Como ejemplos de lenguajes de dominio específico están *COOL*, que trata el aspecto de sincronización, y *RIDL*, para el aspecto de distribución[P2].

Los lenguajes de aspectos de propósito general se diseñaron para ser utilizados con cualquier clase de aspecto, no solamente con aspectos específicos. Por lo tanto, no pueden imponer restricciones en el lenguaje base. Principalmente soportan la definición separada de los aspectos proporcionando unidades de aspectos. Normalmente tienen el mismo nivel de abstracción que el lenguaje base y también el mismo conjunto de instrucciones, ya que debería ser posible expresar cualquier código en las unidades de aspectos. Un ejemplo de este tipo de lenguajes es *AspectJ*, que utiliza el lenguaje de programación *Java* como base, y las instrucciones de los aspectos también se escriben en el mismo lenguaje.

A los programadores siempre les será más fácil aprender un lenguaje de propósito general, que tener que estudiar varios lenguajes distintos de propósito específico, uno para tratar cada uno de los aspectos del sistema.

3.1.2 El problema de los Lenguajes Base

Para el diseño de los lenguajes de aspectos hay dos alternativas relativas al lenguaje base. Una sería el diseñar un nuevo lenguaje base junto con el lenguaje de aspectos y la otra sería tomar un lenguaje ya existente como base, es decir, un lenguaje de propósito general. Esta última opción tiene la ventaja de que se tiene que trabajar menos en el diseño e implementación de lenguajes para los entornos orientados a aspectos, y se pueden utilizar lenguajes ya bien conocidos (*C++*, *Java*, *Smalltalk*), y el programador solamente tendrá que aprender el lenguaje de aspectos, pero no el lenguaje para la funcionalidad básica, que todavía constituye la mayor parte de los programas orientados a aspectos.
Hasta ahora los lenguajes de dominio específico y los de propósito general se han diseñado para utilizarse con lenguajes base existentes. *COOL* y *AspectJ* utilizan *Java* como base. Sin embargo, aparte de las ventajas ya mencionadas, esta decisión también propicia algunos problemas.
Con respecto a los lenguajes de dominio específico, puede tener bastante importancia el hecho de escoger un lenguaje base. Se tiene que tener en cuenta que los puntos de enlace solamente pueden ser los que se identifiquen en el lenguaje base. Así que no se es completamente libre para diseñar esos puntos de enlace. Segundo, si se necesitara separar las funcionalidades, el lenguaje base debe restringirse después de que se hayan separado los aspectos. Esta es la parte más difícil, ya que se tienen que quitar elementos de un sistema complejo(el lenguaje base). Si el diseño de un lenguaje de programación es una tarea difícil y compleja, aún lo es más el hacerle cambios a un lenguaje, que no fue diseñado para tal propósito.
Los lenguajes de aspectos de propósito general son menos difíciles de implementar por encima de un lenguaje de programación existente, ya que no necesitan restringir el lenguaje base. Aparte de esto, la situación es la misma que con los lenguajes de dominio específicos, es decir, en el diseño de los puntos de enlace, se tiene que limitar a los que pueden definirse en el lenguaje base.

3.2 Un lenguaje de Dominio Específico: COOL

COOL es un lenguaje de dominio específico creado por *Xerox* cuya finalidad es la sincronización de hilos de ejecución concurrentes. El lenguaje base que utiliza es una versión restringida de *Java*, ya que se tuvieron que eliminar los métodos *wait*, *notify* y *notifyAll*, y la palabra reservada *synchronized* para evitar que se produzcan situaciones de duplicidad al intentar sincronizar los hilos en el aspecto y en la clase.
Un programa en *COOL* está formado por un conjunto de módulos coordinadores.
En cada coordinador se define una estrategia de sincronización, en la cual pueden intervenir varias clases. Aunque estén asociados con las clases, los coordinadores no son clases. La unidad mínima de sincronización que se define en *COOL* es un método.

Los coordinadores no se pueden instanciar directamente, sino que se asocian con las instancias de las clases a las que coordinan en tiempo de instanciación. Esta relación permanece durante toda la vida de los objetos y tiene un protocolo perfectamente definido.

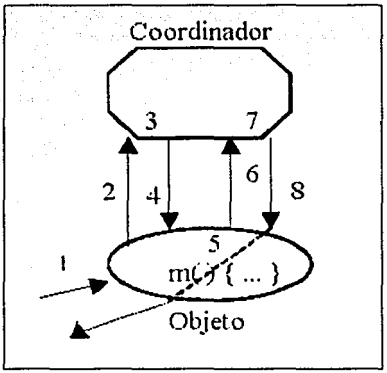


Figura 3.2.1 Protocolo entre un objeto y su coordinador en COOL

En la *figura 3.2.1* [P2]:

1. Dentro del hilo de ejecución H se invoca al método m del objeto (por ejemplo Objeto.m())
2. La petición se envía primero al coordinador del objeto.
3. El coordinador comprueba las restricciones de exclusión y las precondiciones para el método m. Si se da alguna restricción o no se cumple una precondición, se suspende H. Cuando todas las restricciones se cumplan, H puede ejecutar el método m, pero, antes, el coordinador ejecuta el bloque *on entry* asociado con el método m.
4. Se envía la petición al objeto.
5. El hilo H ejecuta el metodo m en el objeto.
6. Se le envía el valor de regreso del método al coordinador.
7. El coordinador ejecuta su bloque *on exit* para el método m.
8. Finalmente se regresa el valor de la invocación de m.

Los coordinadores se escriben sabiendo perfectamente las clases a las que coordinan, sin embargo, las clases no tienen conocimiento de los coordinadores.

Antes de explicar como esta constituido un coordinador, es necesario mostrar el concepto de suspensión con guardia: cuando un hilo H quiere ejecutar un método *M* que tiene una precondición, definida en una cláusula *requires*, y se cumplen las restricciones de exclusión puede suceder lo siguiente.

1. Si la condición definida en *requires* se cumple, entonces el método *M* puede ser ejecutado por H.
2. Si no, el hilo H no puede ejecutar *M*, y se suspende. El hilo permanecerá suspendido hasta que se cumpla la precondición. Cuando esto ocurra, se le notificará a H, y si la restricción de exclusión aún se mantiene, H podrá ejecutar *M*, pero si no se mantiene, se volverá a suspender H.

Las sentencias *on_entry* y *on_exit* se ejecutan cuando un hilo tiene permiso para ejecutar un método. Justo antes de ejecutarlo, se ejecutan las sentencias incluidas en el bloque *on_entry*, y justo después las del bloque *on_exit*.

El cuerpo de un coordinador puede esta formado por:

- Variables de condición.

Las variables de condición se utilizan para controlar el estado de sincronización, con el propósito de utilizarlas en la suspensión con guardia y en la notificación de los hilos. Se declaran utilizando la palabra reservada *condition*.

- Variables comunes.

Las variables comunes mantienen la parte del estado del coordinador que no conduce directamente a la suspensión con guardia y a la notificación de hilos, pero que pueden afectar al estado de sincronización y se declaran igual que en *Java*.

- Un conjunto de métodos auto excluyentes.

En esta sección se identifican los métodos que solamente pueden ser ejecutados por un hilo a la vez y se identifican con la palabra reservada *selfex*.

- Varios conjuntos de métodos de exclusión mutua.

Un conjunto de exclusión mutua identifica una serie de métodos que no se pueden ejecutar concurrentemente por distintos hilos. Es decir, que la ejecución de un hilo H de uno de los métodos del conjunto evita que otro hilo H' ejecute cualquier otro método del mismo conjunto. El conjunto de exclusión mutua se declara con *mutex*.

- Manejadores de métodos.

Estos manejadores se encargan de la suspensión con guardia y de la notificación de los hilos utilizando el estilo de las precondiciones utilizando la cláusula *requires* (a modo de precondición) y las sentencias *on_entry* y *on_exit*.

3.3 Un Lenguaje de Propósito General: AspectJ

AspectJ es una extensión orientada a aspectos de propósito general al lenguaje de programación *Java*. *AspectJ* extiende *Java* con una nueva clase de módulo llamado *aspect*. Los aspectos atraviesan las clases, las interfaces y a otros aspectos, además mejoran la descomposición en ámbitos haciendo posible localizar limpiamente los conceptos de diseño.

AspectJ presenta cuatro nuevos constructores: *pointcuts*, *advice*, *introduction* y *aspect*. Los *poincuts y advice* afectan dinámicamente el flujo del programa e *introduction* afecta estáticamente la jerarquía de clases.

Los *pointCuts* seleccionan ciertos puntos de enlace y ciertos valores en ellos; *advice* define código que se ejecuta cuando se alcanza un *poinCut*; *introduction* modifica la estructura estática del programa: los miembros de sus clases y la relación entre clases.

Un *aspect* es una unidad de modularidad para los ámbitos diseminados y esta definida en términos de *pointcuts*, *advice* e *introduction*.

Un punto de enlace es un elemento crítico en el diseño de cualquier lenguaje *OA*. Proporciona el marco común de referencia que hace posible definir la estructura dinámica de los ámbitos diseminados.

Aspectj proporciona muchos tipos de puntos de enlace por ejemplo las llamadas a métodos abarcan las acciones de un objeto al recibir una llamada a uno de sus métodos: incluye todas las acciones que comprende la llamada al método, iniciando después de que todos los argumentos se evalúan e incluye el regreso normal o abrupto.
El contexto dinámico de la llamada a un método podría incluir muchos otros puntos de enlace: todos los puntos de enlace que ocurren cuando se esta ejecutando la llamada a un método y a cualquier otro método que este llame.

Los *pointcuts* capturan colecciones de eventos en la ejecución de un programa: no definen acciones, sino que describen eventos.

En *AspectJ* los aspectos son constructores que trabajan al atravesar la modularidad de las clases de forma limpia y esta cuidadosamente diseñada, por lo que, un aspecto puede afectar la implementación de varios métodos en varias clases.
En general, un aspecto en *AspectJ* está formado por una serie de elementos:

- *PointCuts*. Está formado por una parte izquierda y una parte derecha, separadas ambas por dos puntos. En la parte izquierda se define el nombre del *PointCut* y el contexto del mismo. La parte derecha define los eventos asociados. Los *pointcuts* se utilizan para definir el código de los aspectos utilizando *advices*. A los descriptores de eventos de la parte derecha de la definición del *pointcut* se les llama designadores. Un designador puede ser: un método, un constructor o un manejador de excepciones. Se pueden componer utilizando los operadores: o ("|"), y ("&") y no ("!"), también se pueden utilizar caracteres comodines en la descripción de los eventos.
- *Introduction*. Se utilizan para introducir elementos completamente nuevos en las clases dadas. Entre estos elementos están: un nuevo método a la clase, un nuevo constructor, un atributo, varios de los elementos anteriores al mismo tiempo y varios de los elementos anteriores en varias clases.
- *Advices*. Definen partes de la implementación del aspecto que se ejecutan en puntos bien definidos. Estos puntos pueden venir dados ya sea por *pointcuts* con nombre o por *pointcuts* anónimos. El cuerpo de un *advice* puede ejecutarse en distintos puntos del código:
    - *before*.- Se ejecuta justo antes de que lo hagan las acciones asociadas con los eventos del *PointCut*.
    - *after*.- Se ejecuta justo después de que lo hayan hecho las acciones asociadas con los eventos del *PointCut*.
    - *catch*.- Se ejecuta cuando durante la ejecución de las acciones asociadas con los eventos definidos en el *PointCut* se lanza una excepción del tipo definido en la propia cláusula *catch*.
    - *finally*.- Se ejecuta justo después de la ejecución de las acciones asociadas con los eventos del *PointCut*, incluso aunque se haya producido una excepción durante la misma.
    - *around*.- Atrapa la ejecución de los métodos designados por el evento. La acción original asociada con el mismo se puede invocar utilizando *thisJoinPoint.runNext*().

El aspecto es una unidad modular de la implementación de un ámbito diseminado. Se define como una clase y sólo los aspectos pueden tener *advices* de esta forma los efectos de

los ámbitos diseminados son localizables y los aspectos desarrollados se pueden eliminar fácilmente de los bloques de producción.

3.4 La Implementación de un Stack

Una vez que se han presentado los dos enfoques existentes a la hora de implementar los conceptos de programación orientada a aspectos sobre los lenguajes de programación, se verá con un ejemplo práctico [9] cómo se reflejan estas características en el código de las aplicaciones.
A continuación se mostrara el desarrollo de una estructura de datos simple, un stack o pila, pasando desde de la implementación en un código convencional enmarañado hasta su implementación en *Aspectj*, pasando por diferentes técnicas de programación: herencia, patrón de estrategias, descomposición multidimensional, filtros de integración, lenguaje de aspectos de dominio especifico y lenguaje de aspectos de propósito general; para lograr la separación y tratamiento de ámbitos diseminados, mostrando así, la necesidad que tienen las técnicas actuales, de la programación orientada a aspectos.

3.4.1 Descripción de la Implementación.

Un stack es una estructura de datos *FILO* (*First Input Last Output*: el primero que entra es el último en salir) y la mejor manera de entenderla es haciendo una analogía con una pila común de platos. Se pretende implementar un stack con un límite en lo que se refiere a su tamaño, y además se necesita sincronizar su acceso en un ambiente de múltiples hilos de ejecución, es decir, de varias tareas ejecutándose al mismo tiempo. En otras palabras habrá clientes que corran en diferentes hilos y tomen y/o inserten elementos en este stack, y debido a que el stack es un recurso compartido se debe sincronizar. Por otro lado también seria interesante que este stack se pudiera utilizar en un ambiente de sólo un hilo de ejecución y poder utilizar una u otra opción.
El aspecto de sincronización debe establecer ciertas restricciones como: no se deben tomar o insertar elementos al stack al mismo tiempo; si se quiere insertar se debe esperar si el stack esta lleno; y si se quiere tomar y el stack esta vacío se debe esperar hasta que ya no lo este.
Una observación importante es que mientras el stack se puede usar en un ambiente con un solo hilo, el aspecto de sincronización no. Este aspecto no hace nada útil por si solo sino que necesita ser usado por una estructura de datos especifica. Este ejemplo muestra el tipo de asimetría entre los componentes y sus aspectos que es común en la práctica. A pesar de lo anterior, un aspecto de sincronización es una pieza de código reutilizable
La primera implementación se realiza en *Java* en una forma convencional, donde el código de sincronización no esta separado del código de la funcionalidad base, es decir, se tiene un código enmarañado. El código en *Java* se muestra en el *código 3.4.1.1*.
Las sentencias de sincronización se remarcaron para mostrar la maraña y la diseminación de código que se obtiene.

```
public class Stack {

   private int max_top;
   private int under_max_top;
   private int top;
   private Object [] elements;

   public Stack(int size) {
      elements = new Object[size];
      top = -1;
      max_top = size-1;
      under_max_top = max_top-1;
   }

   public synchronized void push(Object element)  {
      while (top == max_top) {
         try {
            wait ();
         }
         catch (InterruptedException e) {};
      }
      System.out.println("push: top = " + top);
      elements[++top] = element;
      if (top==0) notifyAll(); // ya no esta vacio
   }

   public synchronized Object pop()   {
      while (top==-1) {
         try {
            wait ();
         }
         catch (InterruptedException e) {};
      }
      System.out.println("pop : top = " + top);
      Object return_val = elements[top--];
      if (top==under_max_top) notifyAll(); // ya no esta lleno
      return return_val;
   }
}
```

Código 3.4.1.1 Implementación enmarañada de un stack sincronizado en Java

El problema con la implementación en el *código 3.4.1.1* es que integra el código de sincronización con el código base. Las desventajas que surgen son:

- La sincronización esta estrechamente codificada: Ya que la sincronización no esta parametrizada, no se puede cambiar sin tener que modificar manualmente la clase *Stack*. Un componente reutilizable tiene que trabajar en diferentes contextos, además de que se podrían requerir diferentes estrategias de sincronización también para diferentes contextos. En el caso más simple, el stack que estamos viendo podría necesitarse en un ambiente de un solo hilo de ejecución pero el código de sincronización introduce líneas de código innecesarias y todavía peor causaría un estancamiento cuando se trate de meter un elemento en un stack lleno o sacar un elemento de un stack vacío ya que hace esperar a la ejecución con esas condiciones.
- Maraña de aspectos: El código base se integra con el código de la sincronización, lo que hace más difícil tratar con cada uno de los aspectos de manera separada. Esto

causa problemas de mantenimiento y una adaptabilidad y reutilización imposible de alcanzar. Otros componentes más complejos podrían requerir algunas variables de estado extra para utilizarse en otras condiciones de sincronización. En el ejemplo del *código 3.4.1.1* se ve como el aspecto de sincronización atraviesa o invade los métodos del stack. En general, la situación puede ser mucho peor ya que si se necesitara coordinar varios objetos, el código de sincronización podría atravesar todos o algunos de estos objetos en diferentes formas.

3.4.2 Separando la Sincronización Usando la Herencia

Una forma estándar de separar el código de sincronización del código base es usar un mecanismo común de los lenguajes *OO*: la herencia. La implementación usual involucra poner la sincronización dentro de una subclase. La subclase envuelve los métodos de la superclase la cual tiene que ser sincronizada con el código de sincronización apropiado. La implementación de la clase *sincronizacion* aplica esta técnica llamada *wrapper*. La implementación del stack con sus aspectos de funcionalidad y sincronización ya separados se muestra en el *código 3.4.2.1*.

```
public class StackErrores {

        protected int top;
        protected Object [] elements;
        protected int s_size;

        public StackErrores(int size)  {
                elements = new Object[size];
                top = -1;
                s_size = size;
        }

        public void push(Object element) {
          try  {
            if (top == s_size - 1) throw new ExcepcionOverFlow("El stack esta lleno no se puede
añadir más");
           }
           catch (ExcepcionOverFlow e)  {
             String s = e.getMessage();
             System.out.println(s);
             return;
           }
           System.out.println("push: top = " + top);
           elements[++top] = element;
        }

        public Object pop() {
           try  {
             if (top == -1) throw new ExcepcionUnderFlow("El stack esta vacío no se pueden
sacar más elementos");
            }
            catch (ExcepcionUnderFlow e)  {
              Object val = null;
              String s = e.getMessage();
              System.out.println(s);
```

```
            return val;
        }
        System.out.println("pop : top = " + top);
        return elements[top--];
    }
}

public class sincronizacion extends StackErrores {
    private int max_top;
    private int under_max_top;

    public sincronizacion(int size)  {
        super(size);
        max_top = size-1;
        under_max_top = max_top-1;
    }

    public synchronized void push(Object element) {
        while (top == max_top)  {
            try  {
                wait ();
            } catch (InterruptedException e) {};
        }
        super.push(element);
        if (top == 0) notifyAll(); // ya no esta vacio
    }

    public synchronized Object pop()  {
        Object return_val = null;
        while (top == -1)  {
            try  {
                wait ();
            }  catch (InterruptedException e) {};
        }
        return_val = super.pop();
        if (top == under_max_top) notifyAll(); // ya no esta lleno
        return return_val;
    }
}
```

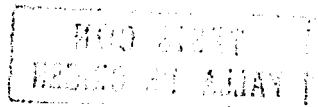
Código 3.4.2.1 Implementación de un stack y un wrapper de sincronización usando herencia parametrizada

El stack del *código 3.4.2.1* contiene un manejador de errores el cual se necesita si además se quiere usar el stack en un ambiente de un solo hilo, ya que al querer meter un elemento a un stack lleno o sacar de uno vació se debe generar una excepción a diferencia de en uno de múltiples hilos en donde no se genera un error sino que el hilo en cuestión se espera hasta tener las condiciones propicias. Una de las ventajas de esta solución con un *wrapper* es que se puede reutilizar el módulo de sincronización para diferentes implementaciones de stacks.

La implementación del stack en el *código 3.4.2.1* todavía tiene una deficiencia mayor: si se envuelve el stack en un *wrapper* de sincronización para usarse en un ambiente de múltiples hilos, la implementación base del stack checa los errores, aunque ya no sea necesario puesto que eso mismo se checa en el módulo de sincronización, es decir, esta repetido dos veces y además los errores en un ambiente de un solo hilo no se aplican de la misma manera que en uno de múltiples hilos. Una solución a este problema es separar el aspecto de chequeo de errores de la implementación base del stack usando otro *wrapper*

basado en la herencia, de la misma forma como se hizo con el aspecto de sincronización. Esto se muestra en el *código 3.4.2.2.*

```
public class StackSimple  {
        protected int top;
        protected Object [] elements;
        protected int s_size;

        public StackSimple(int size)  {
           elements = new Object[size];
           top = -1;
           s_size = size;
        }

        public void push(Object element) {
           System.out.println("push: top = " + top);
           elements[++top] = element;
        }

        public Object pop() {
           System.out.println("pop : top = " + top);
           return elements[top--];
        }
}

public class wrapperErrorStack extends StackSimple  {

  public wrapperErrorStack(int size) {
    super(size);
  }

  public void push (Object element) {
    try  {
      if (top == s_size - 1) throw new
        ExcepcionOverFlow("El stack esta lleno por lo que no se pueden añadir más
elementos");
    }
    catch (ExcepcionOverFlow e)  {
      String s = e.getMessage();
      System.out.println(s);
      return;
    }
    super.push(element);
  }

  public Object pop() {
    Object return_val = null;
    try  {
      if (top == -1) throw new
        ExcepcionUnderFlow("El stack esta vacío por lo que no se pueden sacar más
elementos");
    }
    catch (ExcepcionUnderFlow e)  {
      String s = e.getMessage();
      System.out.println(s);
      return null;
    }
    return_val = super.pop();
```

```
    return return_val;
  }
}
```

Código 3.4.2.2 Implementación de un stack sin chequeo de errores y un wrapper para el stack usando herencia parametrizada.

La clase *wrapperErrorStack* tiene el mismo significado que la clase *StackErrores* del *código 3.4.2.1*. Ahora ya se puede usar la clase *wrapperErrorStack* en una aplicación de un solo hilo y la clase *sincronizacion* del *código 3.4.2.1* en una aplicación de múltiples hilos, ambas clases heredarían la clase *StackSimple* del *código 3.4.2.2*. Todavía se podría parametrizar más el aspecto de sincronización y el de los errores proporcionando diferentes tipos de sincronización o diferentes comprobaciones de errores, pero con esto se ejemplifica perfectamente esta técnica de programación.

Pero añadir los aspectos de sincronización y manejo de errores mediante *wrappers*, como en la clase *sincronizacion* y la clase *wrapperErrorStack*, no siempre es posible. Algunas veces se necesitara invocar a un sincronizador o a un manejador de errores en algún lado dentro de un método. En este caso se puede parametrizar el método con un sincronizador o un manejador de errores usando una técnica de programación llamada patrón de estrategias[12].

3.4.3 Patrón de Estrategias

Una patrón de estrategias define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables: permite variar los algoritmos independientemente de sus clientes que los usan y permite cambiar el mecanismo interno.

El algoritmo que el objeto usa es altamente variable y puede ser usado donde quiera que se necesite.

Los puntos fuertes del patrón de estrategias son:

- Algoritmos diferentes se pueden aplicar en diferentes tiempos.
- El mismo algoritmo se puede aplicar a diferentes objetos.
- El algoritmo puede usar datos que el objeto no necesite saber de estos.

Lo que se hace es encapsular los distintos algoritmos al definir objetos para ellos (denominados objetos_estrategia). Los objetos que usan el algoritmo mantienen una referencia al objeto_estrategia y se cambian los algoritmos al cambiar los objetos_estrategia.

Las consecuencias de todo esto son:

- Las familias de los objetos_estrategia pueden cubrir un amplio rango de intercambios.
- El objeto_estrategia puede encapsular estructuras de datos con algoritmos específicos.
- Los distintos objetos_estrategia pueden usar la herencia para reutilizar código común.
- Un objeto puede cambiar el patrón de estrategias en tiempo de ejecución, por ejemplo para adecuarse a determinado resultado en determinado momento o para

mostrar diferentes estados. Un estado deshabilitado se puede implementar por un objeto_estrategia que no hace nada. Esto elimina sentencias condicionales en el objeto.

Se debe utilizar un patrón de estrategias cuando:

- Muchas clases relacionadas difieren sólo en su comportamiento. El patrón de estrategias proporcionan una forma para configurar una clase con uno o más comportamientos.
- Se necesitan diferentes variantes de un algoritmo.
- Un algoritmo usa datos que el cliente no debe conocer
- Se necesita evitar exponer estructuras de datos con algoritmos específicos.
- Una clase define muchos comportamientos y esto propicia tener muchas sentencias de condición múltiple en sus operaciones.

```
public interface Algoritmo_Interfaz{

  public void Ejecucion_Algoritmo();

}

class Algoritmo_1 implements Algoritmo_Interfaz {

  public void Ejecucion_Algoritmo() { imprime(); }

  public void imprime() { System.out.println(" La ejecución en Algoritmo_1."); }

}

class Algoritmo_2 implements Algoritmo_Interfaz {

  public void Ejecucion_Algoritmo() { imprime(); }

  public void imprime() { System.out.println(" La ejecución en Algoritmo_2."); }

}

class Contexto  {

  private Algoritmo_Interfaz algo_contexto;

  public Contexto(Algoritmo_Interfaz algo){ algo_contexto = algo; }

  public void ContextoInterface()  {
    algo_contexto.Ejecucion_Algoritmo();
  }

}
```

Código 3.4.3.1 Ejemplo de la implementación del patrón de estrategias.

En el *código 3.4.3.1 Algoritmo_Interfaz* es una interfaz común a todos los algoritmos de las clases *Algoritmo_1* y *Algoritmo_2*. La clase *Contexto* esta configurada con diferentes objetos ya sea de la clase *Algoritmo_1* o de la clase *Algoritmo_2* ya que mantiene una referencia a la clase de tipo *Algoritmo_Interfaz*. Para seguir con el ejemplo del stack: en el

método *Ejecucion_Algoritmo* se podrían haber implementado diferentes manejadores de errores o diferentes formas de sincronización.

Desafortunadamente, esta técnica contamina el código que implementa la funcionalidad base. Algunas veces se podrá solucionar este problema separando los métodos y clases, por ejemplo la sección del código que necesite sincronizarse se puede mover dentro de un método por separado.

3.4.4 Separando el Aspecto de Sincronización Usando Hiperespacios

Usando el enfoque de descomposición multidimensional, se puede separar el código que implementa la funcionalidad base(una clase *stack* en el paquete *Stack*) del código que implementa la sincronización (una clase *stack* en el paquete *Sincronizacion*) encapsulándolos en dos hipercortes.

La implementación de la clase *Stack.stack* se muestra en el *código 3.4.4.1*.

```
package Ejemplo.Stack;

public class stack  {
        protected int top;
        protected Object [] elementos;
        protected int s_size;

        public stack (int size)  {
           elementos = new Object [size];
           top = -1;
           s_size = size;
        }

        public void push(Object elemento) {
          System.out.println("push: top antes de meter = " + top);
          elementos[++top] = elemento;
        }

        public Object pop() {
           System.out.println("pop : top antes de sacar = " + top);
           return elementos[top--];
        }
}
```

Código 3.4.4.1 Clase Stack.stack

El *código 3.4.4.2* describe la clase *Sincronizacion.stack* que sincroniza a la clase *Stack.stack*.

```
package Ejemplo.Sincronizacion;

public class stack  {
        private int max_top;
        private int under_max_top;

        public stack(int size)  {
           max_top = size - 1;
           under_max_top = max_top - 1;
        }
```

```
    public synchronized void push(Object elemento) {
      while (top == max_top) {
        try {
          wait ();
        }
        catch (InterruptedException e) {};
      }
      push_base(elemento);
      if (top == 0) notifyAll();
    }

    public synchronized Object pop() {
      while (top == -1) {
        try {
          wait ();
        }
        catch (InterruptedException e) {};
      }
      Object return_val = pop_base();
      if (top == under_max_top) notifyAll();
      return return_val;
    }

  public void push_base(Object elemento){}

  public Object pop_base(){return null;}

  protected int top;
}
```

Código 3.4.4.2 Clase Sincronizacion.stack

Sincronizacion.stack es muy similar en funcionalidad a la clase *sincronizacion* del *código 3.4.2.1*. Las diferencias entre *Sincronizacion.stack* y la clase *sincronizacion* son:

1. *Sincronizacion.stack* no tiene superclase.
2. *Sincronizacion.stack* tiene dos declaraciones de métodos adicionales *push_base* y *pop_base*: las implementaciones para estos metodos se proporcionan en la integración.
3. *push* y *pop* de *Sincronizacion.stack* llaman a *push_base* y *pop_base* respectivamente.

La integración en *HyperJ* se realiza escribiendo ciertas reglas de integración las cuales especifican la correspondencia de paquetes, clases y miembros y además como combinarlos. Antes de mostrar las reglas de integración para este ejemplo, es mejor primero describirlas y luego hacer la traducción en *HyperJ*:

1. Las clases *Stack.stack* y *Sincronizacion.stack* se deben integrar dentro de un nuevo paquete llamado *StackCompleto* uniendo sus miembros(clases), operaciones, atributos, y asegurándose de que si se tienen dos miembros con el mismo nombre, estos se deben corresponder. Por ejemplo *Stack.stack* y *Sincronizacion.stack* corresponden y deben ser unidas dentro de *StackCompleto.stack*. Además, cuando se unan los atributos correspondientes por ejemplo *Stack.stack.top* y

Sincronizacion.stack.top, sólo se debe incluir una copia en la clase compuesta, por ejemplo *StackCompleto.stack.top*.

2. La correspondencia por nombre del requerimiento anterior de los dos *push* y los dos *pop* se debe cambiar y en su lugar se quiere que *Sincronizacion.stack.push_base()* y *Stack.stack.push()* se correspondan y sean combinadas dentro de *StackCompleto.stack.push_base()*. La operación combinada debe usar la implementación de *Stack.stack.push()*. Se quiere que lo mismo pase con *Sincronizacion.stack.pop_base()* y *Stack.stack.pop()*. Como resultado, *push()* y *pop()* de la clase sincronizadora llamaran a *push()* y *pop()* de la clase *Stack.stack*, como sus métodos internos(ver el próximo requerimiento). Esto tiene el efecto de envolver los métodos *push()* y *pop()* de *Stack.stack* dentro del código de sincronización.
3. Si un método se llama dentro de una de las clases ya compuestas, se quiere que se refiera a la clase compuesta. Por ejemplo, la llamada a *push_base* en *Sincronizacion.stack.push*, una vez ya compuesta en *StackCompleto.stack.push*, debe llamar al nuevo método *push_base*, por ejemplo *StackCompleto.stack.push_base*. Lo mismo se aplica con la llamada a *pop_base* en *Sincronizacion.stack.pop*.
4. Los constructores de ambas clases a integrar también se deben integrar. De este requerimiento se encarga automáticamente el sistema de *HyperJ*.

```
-hyperspace
   hyperspace HyperEspacioStack
   composable class Ejemplo.Stack.*;
   composable class Ejemplo.Sincronizacion.*;

-concerns
   package Ejemplo.Stack : Feature.Kernel
   package Ejemplo.Sincronizacion : Feature.Sincronizacion

-hypermodules
   hypermodule StackCompleto
     hyperslices:
       Feature.Kernel,
       Feature.Sincronizacion;
     relationships:
       mergeByName;
       merge operation Feature.Sincronizacion.push_base,
                       Feature.Kernel.push;
       merge operation Feature.Sincronizacion.pop_base,
                       Feature.Kernel.pop;

   end hypermodule;
```

Figure 3.4.4.3 Reglas para la integración de las características Kernel y Sincronización (stack.opt)

La relación *mergeByName* en el *código 3.4.4.3* es una regla de correspondencia y combinación, la cual implementa los requerimientos 1, 3 y 4. Esto implica la correspondencia por nombres y la combinación de los paquetes *Stack* y *Sincronizacion*. Además especifica que el nombre del paquete resultante es *StackCompleto*. Las dos relaciones siguientes (*merge*) son reglas de correspondencia. Ellas implementan el

requerimiento 2. Ya que el alcance de estas dos reglas es de métodos sencillos y tal alcance es más pequeño que el alcance de las reglas de combinación, por lo tanto las reglas *merge* tienen precedencia sobre la regla *mergeByName*.

Una vez que se corre *HyperJ* sobre los archivos *Stack.stack.java*, *Sincronizacion.stack.java* y *algun_cliente.java* junto con el archivo *stack.opt* el código cliente en *algun_cliente.java* automáticamente hace referencia a *StackCompleto.stack* en cualquier lugar en el que se haga referencia a *Stack.stack*. Dando como resultado que la integración en la descomposición multidimensional es no invasiva con respecto al código del stack y al código del cliente del mismo.

Al compilar este programa el compilador lanza el siguiente mensaje:

```
WARNING:  Attempt to specify merge for anything other than hyperslices, which is
not currently supported; specification ignored.
```

Lo que se interpreta como que la opción de unir dos funciones fuera de un hipercorte no esta soportada hasta la fecha aunque sí hace la integración de todos modos, pero ignora la última especificación, sin embargo así es como debe funcionar y además una vez que este soportada esta función se tendrá que añadir un método para especificar que valor de regreso es el que se va a utilizar en el método ya integrado[17] por *pop* y *pop_base* por ejemplo utilizando la especificación:

```
set summary function for action TipodeUnidad NombredelaUnidad to
FuncionCreadaParaDevolverElValorAdecuado;
```

donde el método podría ser:

```
public static int pop_valor ( int[] nValues ) {
   return nValues[0];
}
```

y ya con esto se obtiene la integración que se requiere.

Este ejemplo ilustra dos tipos de reglas de integración disponibles en *la* descomposición multidimensional. Hay otras reglas que permiten un control más fino sobre la integración de los hipercortes.

3.4.5 El Stack en Filtros de Integración

En el capitulo anterior en el *código 2.1.3* se mostró la implementación del stack con dos filtros *error* y *dispatch*, es decir, el stack en un ambiente de un solo hilo de ejecución. Ahora para implementarlo en un ambiente de múltiples hilos existe ya como tal un filtro encargado de la sincronización: el filtro *Wait*, la implementación en *Sina* se muestra en el *código 3.4.5.1*.

[17] esta función si esta soportada actualmente

```
class SyncStack interface
        comment hereda de la clase OrderredCollection, y añade una restricción de sincronización:
                un mensaje pop en un stack vacío se bloquea hasta que haya un elemento en el stack.
        internals
                coll : OrderedCollection; // instancia de la superclase
        methods
                pop returns Element; // obtiene y elimina un elemento referenciado por top
                push(Element) returns Nil; // añade un nuevo elemento al stack
                size returns SmallInteger;      // Regresa el número de elementos
        conditions
                NonEmpty; // es true cuando al menos haya un elemento en el stack
        inputfilters
                sync:Wait={NonEmpty=>pop, True=>*\pop }; // restricciones de sincronización
                disp:Dispatch={ coll.* }; // proporciona todos los métodos de OrderedCollection
end;
```

Código 3.4.5.1 Implementación de la sincronización para el Stack en Sina

Si el stack esta vacío, la condición *NonEmpty* será falsa, y por lo tanto se bloqueara cualquier petición al método *pop* hasta que se cumpla la condición. Esto se expresa por el primer elemento del filtro *sync*. En su segundo elemento, la expresión *"*\pop"* se usa para indicar que todos los mensajes se aceptan excluyendo el mensaje *pop*. Por lo tanto, el mensaje *push* y *size* siempre pasaran este filtro, ya que están asociados con la condición *True*. La expresión *"disp:Dispatch={coll.*};"* significa que *disp* delega todos los mensajes recibidos al objeto *coll*, el cual es una instancia de la clase *OrderedCollection*. Aquí, el mensaje recibido se manipula al remplazar la identidad del receptor con la identidad de *coll*, esto es, el filtro *dispatch* implementa la delegación .
Si un mensaje se delega a un objeto interfaz (aquí *coll*) el objeto encapsulado(aquí instancia de *SyncStack*), hereda el comportamiento de ese objeto (aquí instancia de *OrderedColection*) a través del mismo mecanismo de delegación.

Cuando un mensaje se acepta por una instancia del filtro *wait*, el mensaje pasa al próximo filtro. Si la evaluación no es exitosa, el mensaje permanece en una cola en espera hasta que se cumpla la condición de uno de los elementos del filtro. Los requerimientos a los métodos de un objeto se pueden sincronizar al asociar mensajes con condiciones especificas que implementan las condiciones de sincronización especificas. Cuando un mensaje se acepta por una instancia de un filtro *dispatch* el mensaje se delega a un objeto especifico y si la evaluación no es exitosa el mensaje pasa el próximo filtro.

Para ilustrar las características de reutilización y extensibilidad del enfoque filtros de integración se muestran los siguientes ejemplos.
El *código 3.4.5.2* define la clase *Pop2Stack*, la cual hereda de la clase *SyncStack* e introduce un nuevo método *pop2*. El método *pop2* toma dos elementos del stack a la vez, en lugar de uno nada más: esto requiere sincronización adicional. La clase *Pop2Stack* sincroniza e implementa el método *pop2* al llamar dos veces a *pop* y combina los resultados en un objeto *pair*. El filtro de sincronización *pop2Sync* se encarga de que ningún otro mensaje *pop* se pueda ejecutar mientras que un *pop2* se este ejecutando: esto asegura que dos elementos que sean tomados por *pop2* sean elementos subsecuentes del stack.

```
class Pop2Stack interface
        internals
                superStack:SyncStack;
        methods
                pop2 returns Pair;
        conditions
                MasdeUno;
```

```
        inputfilters
                pop2Sync:Wait = { MasdeUno=>pop2, True=>*\pop2 };
                disp:Dispatch = { superStack.*, inner.pop2 };
end;

class Pop2Stack implementation
        conditions
          MasdeUno
                begin
                        return superStack.size>1;
                end; // regresa true cuando hayan dos o más elementos en el stack
        methods
                pop2 // regresa una instancia de la clase Pair conteniendo 2 elementos
                objects p:Pair; // declara un objecto temporal de la clase Pair
                begin
                        p.putFirst(superStack.pop);
                        p.putSecond(superStack.pop);
                        return p;
                end;
end;
```

Código 3.4.5.2 Definición de la clase Pop2Stack

El próximo ejemplo demuestra la integración de objetos dentro de un nuevo objeto. Para mostrar esto, primero la clase *Bloqueando*, en el *código 3.4.5.3*, proporciona dos métodos, *bloquear* y *desbloquear*: el primero protege al objeto causando que ningún método excepto *desbloquear* sea aceptado y el segundo libera al objeto causando que todos los métodos sean aceptables otra vez. El estado del objeto se almacena en una variable de instancia booleana llamada *liberar*.

```
class Bloqueando interface
        methods
                bloquear returns Nil;
                desbloquear returns Nil;
        conditions
                Bloqueado;
        inputfilters
                 locksync:Wait={ True=>desbloquear, Bloqueado=>* };
end;

class Bloqueando implementation
        instvars
                liberar:Boolean;
        conditions
                Bloqueado
                begin
                        return liberar;
                end;
        methods
                bloquear
                 begin
                        liberar:=false;
                end;
                desbloquear
                begin
                        liberar:=true;
                end;
end;
```

Código 3.4.5.3 Definición de la clase Bloqueando

El método *desbloquear* siempre esta disponible, independientemente del estado del objeto. Los otros métodos del objeto sólo están disponibles cuando el objeto esta en el estado de *Bloqueado*. Debido a la utilización de '*', la especificación seguirá siendo valida cuando se añadan nuevos métodos (asumiendo que estos también tienen que ser bloqueados).
La clase *BloqueandoStack*, *código 3.4.5.4*, es una integración de la clase *Bloqueando* y la clase *SyncStack*. Por lo tanto las instancias de las dos clases se proporcionan como objetos internos. Debido a que la sincronización que esta definida por estas dos clases se necesita combinar, sus filtros de sincronización se usan directamente por la clase *BloqueandoStack*. Las restricciones impuestas por las dos clases tienen que ser satisfechas para dejar que los mensajes sean aceptados.

```
class BloqueandoStack interface
        internals
                superStack : SyncStack;
                bloquador : Bloqueando;
        inputfilters
                bloqueador.lockSync;
                superStack.sync;
                disp:Dispatch={ superStack.*, bloqueador.* };
end;
```

Código 3.4.5.4 Interfaz de la clase BloqueandoStack

3.4.6 La Implementación del Stack en Cool

Antes de mostrar la implementación en código de *Cool*, es conveniente que se determinen los requerimientos de esta estructura de datos para así mostrar como un lenguaje de propósito especifico orientado a aspectos puede traducir de forma más entendible y más directa (lenguaje de más alto nivel) al codificarlas con conceptos propios de ese aspecto especifico.

1. *push* es auto exclusivo.
2. *pop* también es auto exclusivo.
3. *push y pop* son mutuamente exclusivos.
4. *push* sólo puede ejecutarse si el stack no esta lleno y
5. *pop* sólo puede ejecutarse si el stack no esta vacío.

El stack esta lleno si top == s_size - 1 y esta vacío si top == -1. Todas estas declaraciones se pueden representar directamente en *Cool* como un coordinador. El *código 3.4.6.1* muestra la implementación en *Java* del Stack junto con el coordinador del Stack en *Cool*.

```
//en un archivo JCore separado por ejemplo Stack.jcore

public class Stack  {
  private int s_size;
  private int top;
  private Object [] elements;

  public Stack(int size)  {
    elements = new Object[size];
```

```
    top = -1;
    s_size = size;
  }

  public void push(Object element)  {
    elements[++top] = element;
  }

  public Object pop()  {
    return elements[top--];
  }
}

//en un archivo Cool separado como Stacksync.cool   // 1
coordinator Stack  {                                // 2
  selfex push, pop;                                 // 3
  mutex {push, pop};                                // 4
  condition full=false, empty=true;                 // 5
                                                    // 6
  guard push:                                       // 7
  requires !full;                                   // 8
  onexit  {                                         // 9
    if (empty) empty=false;                         // 10
    if (top==s_size-1) full=true;                   // 11
  }                                                 // 12
  guard pop:                                        // 13
  requires !empty;                                  // 14
  onexit  {                                         // 15
    if (full) full=false;                           // 16
    if (top==-1) empty=true;                        // 17
  }                                                 // 18
}                                                   // 19
```

Código 3.4.6.1 Implementación en Java del stack y la implementación en Cool de un coordinador para el stack

El nombre del coordinador aparece en la línea 2 en el *código 3.4.6.1* es el mismo nombre de la clase que sincroniza, en este caso la clase *Stack*. En la línea 3 se establece que *push y pop* son auto exclusivos(lo que correspondería a los requerimientos 1 y 2). La línea 4 establece que *push y pop* son mutuamente exclusivos (lo que corresponde al requerimiento 3). Los estados del Stack se modelan por las variables de condición *full* y *empty* declaradas en la línea 5. Las líneas 7 y 8 dicen que *push* sólo puede ejecutarse si el stack no esta lleno(requerimiento 4). La línea 9 inicia la definición de un bloque de código *on-exit* que se ejecuta inmediatamente después de la ejecución de *push* de la clase Stack. El bloque *on-exit* de *push*(líneas de la 9 a la 12) y el bloque *on-exit* de pop(líneas 15 a la 18) definen el significado de los estados *full y empty*. Estos bloques accesan las variables del Stack *s_size y top*. En general, un acceso de sólo lectura de las variables del objeto coordinado es posible, pero no se está permitido modificarlos. Finalmente, la línea 14 establece que *pop* sólo puede ejecutarse si el stack no esta vacio(requerimiento 5).

Este ejemplo demuestra que las restricciones del stack pueden ser traducidas directamente en código *Cool*, haciéndolo más claro que todas las técnicas utilizadas anteriormente.

El coordinador del Stack en el *código 3.4.6.1* es un ejemplo de un coordinador por instancia, esto es, que cada instancia de la clase *Stack* tendrá su propia instancia coordinador; pero además *Cool* soporta coordinadores por clase, los cuales además se pueden compartir entre un número de clases.

3.4.7 La implementación del Stack en AspectJ

La implementación en *Aspectj* del stack utiliza la clase *StackSimple* en el *código 3.4.2.2*, la integración se hace con esta clase, algún cliente de esta y el aspecto *Sincronizacion_Pila* el cual se muestra en el *código 3.4.7.1*.

```
aspect Sincronizacion_Pila  {

  pointcut insertar(StackSimple pila) : target(pila) && call(void push(Object));
  pointcut extraer(StackSimple pila) : target(pila) && call(Object pop());

  before(StackSimple pila) : insertar(pila)  {
    System.out.println("antes de insertar");
    antesdeinsertar(pila);
  }

  protected synchronized void antesdeinsertar(StackSimple pila)  {
    while (pila.top == pila.s_size - 1)  {
      try  {
        wait();
      } catch(InterruptedException e)  {}
    }
  }

  after (StackSimple pila) : insertar(pila)  {
    despuesdeinsertar(pila);
    System.out.println("despues de insertar");
  }

  protected synchronized void despuesdeinsertar(StackSimple pila)  {
    if (pila.top == 0) notifyAll();
  }

  before(StackSimple pila) : extraer(pila)  {
    System.out.println("antes de extraer");
    antesdeextraer(pila);
  }

  protected synchronized void antesdeextraer(StackSimple pila)  {
    while (pila.top == - 1)  {
      try  {
        wait();
      } catch(InterruptedException e)  {}
    }
  }

  after (StackSimple pila) : extraer(pila)  {
    despuesdeextraer(pila);
  }

  protected synchronized void despuesdeextraer(StackSimple pila)  {
    if (pila.top == pila.s_size - 2) notifyAll();
  }
}
```

Código 3.4.7.1 Aspecto de Sincronización para el Stack

El aspecto tiene el nombre de *Sincronizacion_Pila* como se hace de manera convencional para declarar una clase común en *Java*, enseguida se declaran los *pointcuts insertar* y *extraer* por ejemplo en:

```
pointcut insertar(StackSimple pila) : target(pila) && call(void push(Object))
```

insertar es el nombre de este *pointcut* que recibe como parámetro un objeto de tipo *StackSimple* la sentencia `target(Patron_de_Tipo)` selecciona los objetos que sean una instancia del `Patron_de_Tipo` en este caso una instancia de la clase *StackSimple* y la siguiente `call(void push(Object)` toma todas las llamadas al método `push(Object)`: es decir en conjunción atrapa las llamadas a *push* de un objeto de tipo *StackSimple*: el otro *poincut* es similar pero con el método *pop*.

La sentencia `before(StackSimple pila) : insertar(pila)` se ejecuta antes de los eventos contenidos en el *pointcut insertar*(es decir, antes de que se ejecute *push*) mandando el control al método *antesdeinsertar*, el cual comprueba que el stack no este lleno, si lo esta el hilo de ejecución en cuestión se pone en espera con la sentencia *wait*, si no esta lleno, se ejecuta el método *push*: cuando termina *push* la sentencia *after (StackSimple pila) : insertar(pila)* se ejecuta inmediatamente despues entregando el control al método *despuesdeinsertar*, y lo que hace es revisar si antes de insertar el stack estaba vacío, si el stack estaba vacío, posiblemente algún hilo de ejecución estaba en espera de que alguien insertara elementos al stack para realizar una operación *pop* o alguna otra, entonces como se acaba de insertar un elemento se debe avisar a los otros hilos de ejecución de que ya no esta vacío el stack con la sentencia *notifyAll* para que los hilos de ejecución que estaban esperando sepan que ya pueden ejecutar sus operaciones.

Lo mismo pasa con el *pointcut extraer*, sólo que aquí antes se comprueba que el stack no esta vacío, si lo esta se espera el hilo de ejecución: si no esta vacío se ejecuta *pop* y después de realizar la operación *pop* se comprueba si el stack estuvo lleno antes de extraer, si es así, se notifica a los otros hilos de ejecución de que ya no esta lleno puesto que se acaba de sacar un elemento, despertando así a los hilos de ejecución que esperaban para realizar una operación *push*.

Conclusiones

Al término de este trabajo de investigación, me siento satisfecho del resultado obtenido, sobre todo porque al principio, debo reconocer, me parecía complejo, denso e interminable, pero valió la pena por los beneficios que obtuve y sin lugar a dudas fue un gran ejercicio y una gran oportunidad de seguir aprendiendo.

Al principio, no sabia de que iba a tratar mi investigación y como en todos los grandes descubrimientos fue por mero accidente: cierto día leyendo revistas en una tienda de prestigio encontré una que me llamo la atención de sobre manera por el título que presentaba en la portada: "Ten emerging technologies that will change the world", y donde ponían a la *POA* como la gran panacea informática; y después de haber concluido este trabajo, me atrevo a decir que definitivamente llegara a ser una realidad en un tiempo no muy lejano como lo fue en su momento la *POO*. Los desarrollos en *POA* terminarán por marcar el inicio de un cambio fundamental en la forma en que se diseña y se escribe el software, debido a que se ataca de frente el gran obstáculo de los ámbitos diseminados, que ni la *OO* ni los lenguajes procedurales pueden tratar adecuadamente. La *POA* es una vía prometedora de la ingeniería de software para reducir los costos de desarrollo, evolución y mantenimiento de los programas informáticos, además aumenta las oportunidades de reutilización definiendo una nueva manera de estructurar las aplicaciones; sin embargo, al ser un área nueva hay muy pocos lenguajes disponibles para experimentar y en algunos como *Sina* donde no se han hecho las actualizaciones necesarias ocasiona que no se pueda instalar adecuadamente debido a que el entorno para el que se desarrollo ha sufrido muchos cambios y hace que sea muy difícil la instalación de este o las características que tienen algunos lenguajes no están completas o no están soportadas como en *HyperJ*.

Al ir investigando me di cuenta que existen actualmente muchos esfuerzos aparte de los que se mostraron en este trabajo y cada uno aporta puntos interesantes a esta tecnología y a pesar de que hay mucha gente trabajando en ella, como con cualquier otro nuevo paradigma de programación, tomará algo de tiempo antes de que esta tecnología alcance todo su potencial.

Una tarea que fue muy difícil fue la recopilación de información porque así como en los ámbitos diseminados, la información también esta diseminada por lo que tuve que aplicar también aquí el principio de descomposición en ámbitos de Dijkstra para ir estructurando el trabajo. Mi investigación se baso fundamentalmente en Internet sobre todo en la búsqueda de publicaciones y charlas de investigadores que acudían a los eventos: *ECOOP*, *OOPSLA* y *ICSE*.

Otra de las dificultades que me encontré es que toda la información esta en idioma ingles, con muchas palabras técnicas, las cuales traduje lo mejor posible, lo cual me obligo a mejorar considerablemente mi lectura y aumento notablemente mi vocabulario que se restringía a las traducciones de mis canciones favoritas.

Al estar depurando la información me di cuenta que existen mucha gente investigando cosas muy interesantes, hay un universo interminable de cosas que investigar y es muy fácil perderse. El motivo por lo que este trabajo fue una tesina es porque la *OA* es todo un mundo dentro del universo de la Informática y como todo universo esta en constante

expansión: estoy convencido que el estudiante de *MAC* encontrara interesante este trabajo no nada más como una novedad sino como fuente de futuras tesis como por ejemplo: nuevos lenguajes *OA*, la combinación de varios enfoques en uno mejorado, el diseño de diferentes filtros de integración, la creación de formas inteligentes de integración, etc.

Las aplicaciones de esta tecnología son inmensas ya que se puede implementar en prácticamente cualquier programa: al tener el poder de modularizar todo, incrementa la claridad de los programas en general. En *Acatlán* por ejemplo he visto que tienen un sistema diferente para servicios escolares, centro de computo, biblioteca, e idiomas y entonces aplicando la *OA* se podrían integrar todos esos sistemas en uno solo, sin modificación alguna a los sistemas actuales. En la *UNAM* y en especifico en *DGSCA* en mi estancia como becario en súper computo me di cuenta que tienen algunos programas, generalmente en Fortran, que necesitan de una interfaz grafica la cual se podría diseñar e implementar utilizando la tecnologia *OA* y así tener interfaces graficas genéricas y reutilizables. En *México*, se puede emplear la tecnología *OA* en los servicios que se dan en línea puesto que se esta dando un gran revuelo de Internet, ahora los impuestos se pagan por esta vía, un servicio en línea puede involucrar cientos sino es que miles de clientes requiriendo el mismo servicio, donde cada cliente puede tener diferentes preferencias (posiblemente conflictivas) acerca de como el servicio debe ser personalizado a sus necesidades, por lo que esas múltiples vistas especificas de cada cliente deben ser tomadas simultáneamente en cuenta durante el proceso de personalización de tal sistema. Los clientes de un servicio en línea deben tener la opción de personalizar este servicio para el uso de su propio contexto (sin afectar el comportamiento del servicio que se entrega a otros clientes de ese mismo servicio). Otra aplicación es el código de los sistemas operativos el cual es evidentemente complejo, con muchos subsistemas de millones de líneas de código y con muchas interacciones entre ellos. Los programadores no pueden tratar claramente con todas las partes del sistema operativo que tienen una estructura compleja. La *POA* puede mejorar la modularidad de los sistemas operativos y por lo tanto reducir la complejidad y fragilidad asociada con la implementación del sistema.

Indudablemente la formación que me dio la carrera de *MAC* (una muy buena lógica, una base matemática sólida y unos muy buenos fundamentos computacionales) me permitió la comprensión de este tema; donde cada una de las materias aportaron lo suyo a esa formación donde yo destaco mucho la lógica que tenemos nosotros los de *MAC* y a pesar de ser parte de una de las generaciones del primer plan de estudios, por la versatilidad de este, no le pide nada al nuevo; por todo lo anterior considero que el estudiante de *MAC* esta listo para nuevos paradigmas, puesto que el traslado es casi inmediato.

Quisiera destacar el uso de Internet porque sin este recurso no hubiera podido investigar absolutamente nada sobre el tema, ya que en *México* siempre nos mandan lo que ya no esta vigente en otros lados y siempre tendremos un retraso con respecto a otros países. Definitivamente Internet es el parte aguas, podemos decir que la vida se puede calificar como antes y después de Internet; por medio de este me pude comunicar con los investigadores que por cierto muy amablemente me contestaban a todo lo que les preguntaba.

Finalmente, termino con la analogía del capitulo primero, los dragones esperan que los dioses de este mundo bizarro terminen su tiempo de prueba y los dejen establecerse y coexistir con los duendes definitivamente.

Apéndice

Terminología Orientada a Objetos

Un lenguaje de programación se considera orientado a objetos si incluye tres conceptos base: identidad, clase y herencia.

Los ***objetos***[18] son entidades autónomas que responden a mensajes. Un ***mensaje*** es un requerimiento a un objeto para que este realice o ejecute alguna de sus operaciones[19]. Un mensaje tiene la forma *objeto.método (parámetros)*. Las ***operaciones*** de un objeto son un conjunto de funciones, cada una con un significado propio proporcionado y aplicable a ese objeto. Cada objeto proporciona una ***encapsulación***, para que la implementación de las operaciones y las estructuras de datos internos de cada objeto estén escondidas de sus usuarios. Dentro de un objeto las estructuras de datos internas están definidas como variables locales. Las variables locales de un objeto sólo son accesibles a través de sus operaciones o interfaz. La encapsulación es útil porque esconde los detalles innecesarios de implementación y ya que la interfaz es independiente de la implementación, la re-implementación o modificación de un objeto no tendrá ningún efecto en los otros objetos del sistema. En enfoques puramente *OO*, como en *Smalltalk*, todo se considera un objeto y por lo tanto, la encapsulación se presenta en todos los niveles.

Una ***clase*** es una abstracción de un concepto y lo que hace es agrupar objetos de acuerdo a un comportamiento y forma común. Estos objetos individuales de una clase además llamados instancias, sólo pueden mantener propiedades específicas en la forma de valores únicos de sus variables locales. Las clases pueden ser vistas como plantillas. Ellas pueden heredar variables y métodos de instancia de otras clases. Por ejemplo, la clase *delfín* puede heredar de la clase *animal*.

La ***identidad*** se refiere a características de distinción de un objeto que denoten la existencia separada del mismo, aunque el objeto pueda tener los mismos valores que otro(s) objeto(s); en otras palabras, dos objetos son distintos aunque tengan los mismos valores en todos sus atributos (como color y velocidad), por ejemplo las personas que son gemelas aunque tengan la misma apariencia siguen siendo personas diferentes.

Las ***variables de instancia***, llamadas también datos miembro, variables o atributos, representan los datos internos del objeto y pueden tener diferentes valores, por ejemplo el atributo color puede ser verde. Estas describen el ***estado del objeto***. Estos atributos pueden ser cambiados por las operaciones, por ejemplo la operación *pinta(rojo)*, cambiaría el atributo color de verde a rojo.

Las ***variables de clase*** son compartidas por todas las instancias de la clase. Estas se usan para compartir cierto estado o para propósitos de administración de la clase[20].

Un ***constructor*** es una operación especial utilizada para inicializar el objeto después de su creación, por ejemplo, asignar sus variables de instancia con algún valor por defecto.

---

[18] conocidos también como *target* u objetivo

[19] también llamadas métodos o funciones miembro o de igual forma como selector.

[20] por ejemplo para saber cuantos objetos se han creado de la clase en cuestión en determinado momento o para tener una referencia a todas las instancias de la clase.

La ***herencia*** es una organización de clases, por medio de la cual una clase puede adoptar las operaciones y/o atributos de sus superclases y también puede otorgar tanto sus operaciones como atributos a sus subclases. Esta relación de herencia sirve para la reutilización de las clases. Las clases además pueden heredar de múltiples padres, proporcionando una reutilización adicional, aunque no todos los lenguajes lo permiten, como *Java*. Por ejemplo una clase *empleado* puede heredar de la clase *persona* ya que un empleado también es una persona.

Como una alternativa a la herencia, existe una técnica llamada ***delegación***. En lugar de heredar entre clases, la delegación hereda entre objetos. La delegación es un mecanismo que permite a los objetos delegar algún requerimiento de sus usuarios a uno o más objetos designados. La delegación es independiente del concepto de clase, y es adoptado frecuentemente por lenguajes que no implementan clases. Básicamente, los mecanismos de herencia y delegación tienen características similares en que ambos pueden implementar estructuras de software modulares que comparten operaciones. Los diseñadores de lenguajes basados en la delegación afirman que la delegación es más poderosa que la herencia porque puede soportar la evolución dinámica de los sistemas ya que las delegaciones se pueden configurar en tiempo de ejecución. Con la delegación el objeto delegado es parte de la identidad extendida del objeto delegador.

La ***redefinición*** es el proceso de adecuar un método de una superclase en una subclase para propósitos de esta última.

La ***definición de un objeto*** son las operaciones visibles desde fuera del objeto y esta compuesta por las operaciones de la clase y superclases a las que pertenece.

La ***definición de una operación*** es el número de parámetros y en el caso de un lenguaje que sea muy exigente con los tipos, el tipo de regreso dando a la operación un único identificador.

La ***definición de una clase*** es el conjunto de operaciones accesibles por los objetos de otras clases. Y esta compuesta por sus propias operaciones y aquellas de las superclases.

La ***información de reflexión*** es aquella información que un lenguaje de programación es capaz de presentar de sí mismo[21]. Ejemplos de información de reflexión son la definición de un objeto, el tipo de un objeto, la información de sus superclases, etc. Así como con la herencia múltiple, sólo ciertos lenguajes son capaces de hacer esto, mientras que otros puedan tener algo o inclusive nada para poder soportar esta característica.

Una ***clase abstracta*** es una clase que esta parcialmente implementada. Las class abstractas pueden ser vistas como una conveniencia de diseño. Los métodos abstractos son métodos que no tienen implementación; se parecen mucho a las interfaces, y ni las clases abstractas ni las interfaces pueden ser instanciadas.

El ***polimorfismo*** es la habilidad de tomar varias formas. Esto significa que cada objeto, de una clase base común, puede tener diferente comportamiento; permitiendo que diferentes objetos respondan al mismo mensaje, es decir, que cada objeto responda al mismo mensaje en una forma apropiada para el objeto.

[21] también llamada meta información.

Glosario

Ámbito
Es un concepto particular, meta o propósito y que con el advenimiento de los aspectos su implementación no solo se restringe a técnicas convencionales como las clases o procedimientos sino a configuraciones, características, versiones, etc. Es equivalente a una clase en la terminología OO pero en su forma más genérica.

Ámbito Diseminado
Es el ámbito que no puede ser encapsulado dentro de una forma modular y por ende se distribuye en más de un módulo.

Aspecto
Se puede hablar de aspectos en la fase de diseño y en la fase de implementación en los sistemas de software.
Los aspectos en el diseño no están muy claros. Tienden a ser unidades que atraviesan varios componentes en el diseño de acuerdo a alguna división de trabajo e interactúan con los demás componentes de acuerdo a algunas interfaces definidas. La diferencia con un objeto es la naturaleza de diseminación del aspecto: esto quiere decir que los aspectos afectan a otros componentes, mientras que los objetos no lo hacen, solo se limitan a sí mismos.
Los aspectos en la implementación, es algo muy concreto, pero su definición precisa depende del lenguaje de aspectos. Ya que la programación orientada a aspectos es relativamente nueva, no hay muchos lenguajes de aspectos; sin embargo, en todos los lenguajes actuales, un aspecto es una unidad que encapsula estado(variables), comportamiento (métodos) y comportamiento que se añade a otras unidades(acciones).

AspectJ
Es una extensión a Java orientada a aspectos de propósito general. AspectJ extiende Java con un nuevo tipo de módulo, llamado aspect. Integra archivos fuente en donde cada archivo puede contener una clase o un aspecto.

Código Enmarañado
Es el código resultado de tener que implementar en un sistema informático uno o varios ámbitos diseminados.

Clase Base
Propiedad que permite expresar, representar y manipular unidades de software de manera aislada o independiente y perfectamente localizable.

Compilador
Es un programa que traduce otro programa escrito en un lenguaje a un programa equivalente en otro lenguaje.

COOL (Coordination-Oriented Language)
Lenguaje orientado a aspectos de dominio especifico y se encarga de la coordinación de hilos de ejecución.

ECOOP (European Conference on Object-Oriented Programming)
Conferencia europea sobre programación orientada a objetos, donde en los últimos años ha dado un énfasis a la programación orientada a aspectos.

Filtro de Integración
Es una extensión al modelo convencional de objetos, para diseñar, modularizar e implementar un aspecto.

Hilo de Ejecución (Thread)
Una aplicación esta constituida de uno o más procesos. A su vez, cada proceso puede estar formado por uno o más hilos de ejecución. Cada proceso tiene su propio espacio de direcciones virtuales, datos y código. Además puede tener asignados determinados recursos del sistema, como archivos, puertos de e/s, etc.

Hipercorte
Mecanismo que ayuda a encapsular ámbitos para poder tratarlos como módulos.

Hiperespacios
Enfoque orientado a aspectos de IBM que implementa la descomposición multidimensional.

Hipermódulo
Mecanismo que permite la especificación de las reglas de integración.

Hiperplano
Contiene todas las unidades de software que afectan a un determinado ámbito.

HyperJ
Lenguaje orientado a aspectos de propósito general que permite implementar el enfoque de descomposición multidimensional.

ICSE (International Conference on Software Engineering)
Conferencia internacional sobre ingeniería de software.

Integrador
Herramienta que permite combinar los aspectos con el código que implementa la funcionalidad base.

Interprete
Programa que traduce una expresión o declaración de algún lenguaje, calcula y luego imprime o utiliza de otro modo su resultado, su velocidad de ejecución es lenta pero los errores de ejecución así como los de sintaxis se detectan a medida que se encuentra cada

declaración, eliminando así cualquier duda acerca de donde reside el problema: los resultados son línea por línea.

Kernel
En el enfoque filtros de integración es como un objeto convencional y se utiliza el término para distinguirlo de un objeto con filtros de integración.

Lenguaje Base
Es el lenguaje que se encarga de implementar la funcionalidad esencial de un sistema y no incluye a los aspectos.

Lenguaje de Aspectos
Es el lenguaje que se encarga de implementar los aspectos y puede ser de propósito general o de dominio especifico.

Lenguaje de Bajo Nivel
Un lenguaje se considera de bajo nivel si uno puede manipular directamente la memoria de acceso aleatorio (RAM) de una computadora utilizando instrucciones del lenguaje, el caso de C; en los lenguajes de alto nivel se pierde el control preciso sobre cuales localidades de memoria contendrán los datos.

OOPSLA (Object-Oriented Programming, Systems, Languages and Applications)
Conferencia que trata de la Programación Orientada a Objetos, Sistemas, Lenguajes y Aplicaciones.

Patrón de Estrategias
Es una forma conveniente para reutilizar código orientado a objetos entre proyectos y programadores: consiste de un número de algoritmos relacionados encapsulados en una clase manejadora llamada contexto: el cliente del programa puede seleccionar alguno de estos algoritmos o en algunos casos el contexto podría seleccionar el mejor para ese cliente, la meta es intercambiar fácilmente entre algoritmos sin ninguna sentencia condicional.

Reglas de Integración
Son las indicaciones de como y en que forma se realizara la integración; la cual puede ser estática en tiempo de compilación o dinámica en tiempo de ejecución.

RIDL (Remote Interface Data transfer Language)
Lenguaje de transferencia de datos de interfaz remota.

Sina
Lenguaje de programación residente en Smalltalk VisualWorks versión 1.0 que implementa el modelo de objetos filtros de integración: nombrado así en memoria del filosofo y físico medieval Ibn Sina quien también era conocido con el nombre de Latin Avicenna.

Stack
Nombre en idioma inglés que se le da a la estructura de datos llamada pila, la cual tiene la propiedad FILO (First Input Last Output), el primero que entra es el último en salir.

TRERE (Twente Research & Education on Software Engineering)
Es un proyecto de investigación para identificar las deficiencias de la tecnología orientada a objetos.

UML (Unified Modelling Language)
Es la combinación y recopilación de las mejores características de diseño y análisis orientadas a objetos de Grady Booch, James Rumbaugh e Ivar Jacobson en un método unificado.

Bibliografía

[1] *Thomas S. Khun*, The Structures of Scientific Revolution,University of Chicago Press, Chicago.

[2] *Appleby Doris, Vandekopple Julius J.* Lenguajes de Programación, McGraw-Hill Segunda Edición.

[3] *J. Mcarthy*, LISP 1.5 Programming Manual Cambridge MA. MIT Press.

[4] *A. Colmeraure, P. Roussel*, The Birth of PROLOG . ACM SIgPlan Notices, Vol. 28, No. 3, March 1993.

[5] *Fred Brooks*, The Mythical Man-Month, Addison-Wesley.

[6] *O.J. Dahl, B. Myhrhaug, K. Nygaard*, Simula 67 Common Base Language, Norwegian Computer Center 1970.

[7] *Edsger Wybe Dijkstra*, A Discipline of Programming, Prentice Hall PTR

[8] *Roger S. Pressman*, Ingeniería del Software, McGraw-Hill Quinta Edición.

[9] *Krzysztof Czarnecki, Ulrich Eisenecker*, Generative Programming: Methods, Techniques, and Applications, Addison-Wesley.

[10] *Karl Lieberherr*, Adaptive Object-Oriented Software The Demeter Method, PWS Publishing Company.

[11] *Clemens Szyperski*, Component Software: Beyond Object-Oriented Programming, Addison Wesley.

[12] *E. Gamma, R. Helm, R. Johnson, and J. Vlissides*, Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995

[13] Ecoop '97 - Object-Oriented Programming : 11th European Conference Jyvaskyla, Finland, June 9-13, 1997 : Proceedings (Lecture Notes in Computer Science), Mehmet Aksit , Satoshi Matsuoka (Editor). http://trese.cs.utwente.nl/aop-ecoop97/position.html

[14] Ecoop '98-Object-Oriented Programming : 12th European Conference, Brussels, Belgium, July 20-24, 1998 : Proceedings (Lecture Notes in Computer Science), Eric Jul (Editor). http://www.fzi.de/ECOOP98-W03/

[15] Ecoop'99 - Object-Oriented Programming : 13th European Conference, Lisbon, Portugal, June 14-18, 1999 Proceedings (Lecture Notes in Computer Science), Rachid Guerraoui (Editor). http://ecoop99.di.fc.ul.pt/

[16] Ecoop 2000 - Object-Oriented Programming : 14th European Conference, Sophia Antipolis and Cannes, France, June 12-16, 2000 : Proceedings (Lecture Notes in Computer Science), Ecoop 2000 (Editor) Sophia-Antipolis, France, Cannes. http://ecoop2000.unice.fr/

[P1] T.J. Higley, Michael Lack, Perry Myers. A critical analysis of a new programming paradigm. http://citeseer.nj.nec.com/highley99aspect.html

[P2] Cristina Isabel Videira Lopes. D: A language framework for distributed programming. http://www2.parc.com/csl/groups/sda/publications/papers/PARC-AOP-D97/for-web.pdf

[P3] Junichi Suzuki, Yoshikazu Yamamoto. Extending UML with Aspects: Aspect Support in the Design Phase. http://www.ics.uci.edu/~jsuzuki/pub/ecoop99.pdf

[P4] Piet S. Koopmans. On the Definition and Implementation of the Sina/st Language. http://trese.cs.utwente.nl/publications/paperinfo/koopmans.thesis.pi.abs.htm

[P5] Matthew Kaplan, Harold Ossher, William Harrison, Vincent Kruskal. Subject-oriented design and the Watson subject compiler. http://www.research.ibm.com/sop/papers/position96.htm

[P6] Harold Ossher, William Harrison. Supporting descentralized development of objects. http://www.research.ibm.com/people/o/ossher/

[P7] Klaus Ostermann, Günter Kniesel. Independent Extensibility. http://trese.cs.utwente.nl/Workshops/adc2000/papers/Ostermann.pdf

URL's

Aspect-Oriented Programming, Xerox, 2002
http://aspectj.org

HyperJ: Multi-Dimensional Separation of Concerns for Java, IBM Research, 2002
http://researchweb.watson.ibm.com/hyperspace/HyperJ/HyperJ.htm

Aspect-Oriented Research on Composition Filters homepage, TRESE, 2002
http://trese.cs.utwente.nl/composition_filters/

Programación Subjetiva, IBM Research, 2002
http://www.research.ibm.com/sop/

Aspect-Oriented Software Development, Concur Group, 2002
http://www.iit.edu/~concur/

Design Support for Aspect-oriented Software Development, TecComm, 2002
http://www.teccomm.les.inf.puc-rio.br/socagents/yaam.htm