Universidad de Buenos Aires

Facultad de Ciencias Exactas y Naturales

Departamento de Computación

Tesis de Licenciatura

Análisis del Despacho Dinámico de Mensajes en Lenguajes Orientados a Objetos

Autor:

Wilkinson, Hernán

LU: 228/91

hawilki@yahoo.com

Director:

Lic. Máximo Prieto

2000-2002

Resumen

Programas escritos con lenguajes de objetos tienen grandes diferencias con los programas escritos en lenguajes procedurales. Una de las principales diferencias proviene de la utilización de una herramienta de programación denominada "polimorfismo".

El polimorfismo facilita la creación y utilización de las abstracciones que modelan el problema que se está resolviendo con el sistema, permitiendo obtener un mejor diseño del mismo, con los consiguientes beneficios que ello implica, como facilidad de mantenimiento, reuso de código, etc.

El polimorfismo se hace "realidad" a través de la implementación del algoritmo de despacho de mensajes, también denominado "method lookup", que cada lenguaje de programación ofrece. Es este algoritmo el encargado de decidir a partir de un mensaje y un objeto, que método (código) se debe ejecutar.

Este trabajo realiza una recopilación de las distintas implementaciones del algoritmo de "method lookup", presenta un framework destinado a estudiar dichas implementaciones y ofrece un informe final de varios tests ejecutados sobre distintas implementaciones de dicho algoritmo.

La investigación se concentró en implementaciones del algoritmo para lenguajes de objetos con variables no tipadas que utilizan despacho de mensajes dinámico. Lenguajes que entran en esta categoría son por ejemplo Smalltalk y Self.

El framework de "method lookup" está implementado en Squeak y permite modificar dinámicamente el algoritmo que está siendo utilizado por la máquina virtual.

La utilizacion del framework permitió investigar nuevas implementaciones del algoritmo de method lookup no presentes en la bibliografía utilizada, las cuales fueron testeadas y documentadas debidamente.

Dedicado a:

Mis tres amores: Mónica, Sol y Michelle.
Mis abuelos Grampa y Popa.
Mis Padres.

Indice

Indice de figuras

Indice de Tablas

1. Introducción

Mejorar la facilidad de desarrollo es uno de los continuos objetivos que se encuentran en los distintos trabajos de investigación del área de Ciencias de la Computación. Uno de los principales motivos por los cuales se busca realizar este objetivo se debe al costo asociado que tiene el desarrollo de software. Bajar la complejidad de desarrollo implica minimizar los costos asociados a dicha tarea.

Se han realizado varios avances en el tiempo de vida de nuestra profesión que permitieron bajar la complejidad del desarrollo, como la posibilidad de agrupar código común en funciones o procedimientos (Programación Estructurada), la posibilidad de agrupar comportamiento y abstraerlo por medio de objetos (Programación Orientada a Objetos), agrupar soluciones a problemas comunes y recurrentes en patrones de diseño (Design Patterns) y últimamente la posibilidad de separar la programación que resuelve los problemas funcionales de los no funcionales por medio de la metaprogramación y programación orientada a aspectos.

El objetivo de minimizar la complejidad del desarrollo puede ser logrado tomando distintas acciones, como por ejemplo crear lenguajes de mayor capacidad expresiva o simplemente acelerando tiempos de compilación (Esta última opción facilita el proceso de desarrollo pero no la complejidad intrínseca del mismo, o sea el diseño de la solución). Cualquiera sea la acción tomada para lograr este objetivo, en definitiva la solución termina depositando mayor responsabilidad y capacidad de acción en la "computadora" que en el "humano".

Es indudable que la programación orientada a objetos permite atacar el problema de la complejidad de desarrollo de una manera fácil y productiva (No formará parte del trabajo demostrar esta premisa, simplemente la asumimos como verdadera basados en la experiencia. Ver [Taylor] para una explicación no formal del tema). Sin embargo esta tecnología tuvo poca aceptación durante el tiempo de su aparición debido a los recursos que necesitaba para poder ser utilizada. La "performance" fue siempre un problema, y lamentablemente lo sigue siendo para muchos casos, más allá de los recursos que se posean.

Uno de los grandes motivos del consumo de recursos en los lenguajes de objetos es el polimorfismo[1]. El polimorfismo permite delegar en la "computadora" en tiempo de ejecución la decisión de qué "método" de un objeto ejecutar basados en el mensaje[2] que le fue enviado. La gran ventaja de esta característica es que permite que un programa puede modificar su comportamiento en tiempo de ejecución (flexibilidad).

Como es de suponer, el polimorfismo tiene su costo en performance. Este costo está íntimamente relacionado con el algoritmo denominado "despacho de mensajes", que es en definitiva la implementación que permite obtener polimorfismo. Por ende, la

[1] Polimorfismo: Ver definición en 3.1

[2] Enviar un mensaje en lenguajes de objetos es similar a decir que se realiza una llamada a función en lenguajes procedurales, con la característica polimorfica intrínseca de los lenguajes de objetos.

implementación del algoritmo de despacho de mensajes en lenguajes de objetos es un punto crucial dentro de la performance general del sistema.

Debido al costo de "performance" que implica tener polimorfismo (entre otros motivos) se tuvieron que tomar decisiones de implementación de la teoría de objetos que entre otras cosas, llevaron a la separación de lo que se denominan "lenguajes de objetos puros" y "lenguaje de objetos híbridos" (los primeros más cerca del paradigma, los segundos más pragmáticos dentro de las restricciones tecnológicas). Como es de suponer, una de las diferencias entre los distintos lenguajes es cómo resuelven el despacho de mensajes. Algunos decidieron hacerlo de manera dinámica, otros de manera más estática.

Es claro a nuestro entender que cuanto más tarde se realice el "*binding*" entre una variable y el objeto que referencia, más nos acercaremos al "*verdadero*"[3] polimorfismo, obteniendo mayor flexibilidad. Posponer el "*binding*" hasta último momento implica realizar el despacho de mensajes dinámicamente, o sea resolverlo en tiempo de ejecución.

A continuación se detalla el motivo de esta investigación y sus objetivos.

1.1 Determinación del Problema

La finalidad de este trabajo de investigación es estudiar el costo del despacho dinámico de mensajes, proponer técnicas de testing y profiling, y nuevos algoritmos para resolver dicho problema.

El estudio se centrará en el despacho dinámico de mensajes en lenguajes orientados a objetos. Dentro de este tema hay varios motivos de investigación, sin embargo se decidió acotar el trabajo y definirlo por medio de objetivos.

Los objetivos que se propone cumplir esta tesis son:

- Repasar la investigación realizada hasta el momento sobre el tema que nos concierne.
- Crear un framework que permita utilizar distintos algoritmos de despacho de mensajes, proveyendo facilidades de testing y profiling.
- Proponer un conjunto nuevo de algoritmos y probarlos con el framework realizado en el punto anterior.
- Ver si se puede lograr aumentar la performance general del sistema utilizando los algoritmos del punto anterior.

Debido a los objetivos propuestos, es necesario realizar una investigación no solamente teórica sino también práctica, lo cual implica la utilización de un lenguaje de objetos lo suficientemente flexible como para poder modelar las distintas realidades o implementaciones de los algoritmos que se quieran probar. Dicho lenguaje es Smalltalk.

[3] La expresión "polimorfismo más verdadero" significa que ofrece mayor flexibilidad al lenguaje.

Veremos ahora el por qué de esta decisión y más adelante detallaremos cada uno de los objetivos que se desean cumplir.

1.1.1 Objetos y Smalltalk. Un poco de historia

La programación orientada a objetos es una técnica que posee varios años de experiencia e investigación. El primer lenguaje que se puede denominar como "orientado a objetos" fue Simula-67, el cual tenía dentro de su estructura el concepto de *clase*.

Sin embargo, el trabajo más importante y revolucionario dentro de esta materia fue la realizada por el grupo de trabajo de Xerox PARC[4] (Palo Alto Research Center) compuesto por personas de altísimo nivel como Alan Kay, Dan Ingalls y Adele Goldberg (entre otros). Este grupo creó el primer ambiente de objetos, el cual fue denominado Smalltalk y estuvo acompañado por un conjunto de innovaciones técnicas muy importantes, como por ejemplo el "Graphical User Interface", el "mouse", el concepto de "todo es un objeto", la utilización de un "garbage collector", entre otros.

El mismo implementa el concepto minimalista del paradigma orientado a objetos con gran éxito. En Smalltalk todo es un objeto, y todos los objetos se comunican por medio de mensajes. Esto es cierto aún para los tipos de datos más básicos como por ejemplo un Entero (*Integer*) o las estructuras de control de flujo como el *if...then...else*, *do...while*, etc.

Por ejemplo, el código de la figura 1 indica que se le está enviando el mensaje "+" al objeto "1", pasándole como parámetro (o colaborador) el objeto "2".

```
1 + 2
```

Figura 1: Ejemplo de código Smalltalk

Smalltalk fue creciendo con el pasar de la década del 70 hasta llegar a lo que hoy conocemos como Smalltalk-80, la versión del lenguaje que se dio a conocer públicamente y que fue definido por medio del denominado "Blue Book"[5] [Goldberg1].

Smalltalk-80 fue la base para todos los distintos "sabores" de Smalltalk que luego fueron apareciendo en el mercado, como Smalltalk/V, VisualWorks, VisualSmalltalk, IBM Smalltalk (o VisualAge Smalltalk), etc.

En el mismo se basa también Squeak, un nuevo Smalltalk creado por las mismas personas que inicialmente trabajaron en PARC, que se adecua a las nuevas situaciones tecnológicas. (Ver [Ingalls et al])

[4] Xerox PARC: Se decía en su momento que Xerox PARC tenía a los mejores investigadores del área. El grupo de investigación de Smalltalk se dividió. Una parte fundó ParcPlace y otra migró a Apple. Ahora parte del grupo se encuentra en Disney Imaginary llevando adelante el proyecto Squeak.

[5] "Blue Book" es la manera de llamar al libro [Goldberg1] debido a su tapa, la cual es consta de un dibujo con fondo azul.

Una de las finalidades de Squeak es la de demostrar que todo puede ser implementado en Smalltalk, por lo que la Máquina Virtual (*Virtual Machine* o *VM*) está escrita en dicho lenguaje (aunque por razones de performance se creó un traductor de Smalltalk a C para tener una VM en lenguaje nativo).

Tener la VM escrita en Smalltalk permite realizar una gran cantidad de pruebas que serían imposibles en arquitecturas cerradas. Los trabajos de investigación realizados en esta tesis se basan fuertemente en esta facilidad y utiliza Squeak como implementación de Smalltalk para realizar las implementaciones y pruebas necesarias.

En conclusión el uso de Squeak como herramienta de desarrollo para esta investigación se basa en los siguientes puntos:

- Es un implementación completamente abierta
- El código de la VM esta escrito en Smalltalk
- Debido a los puntos anteriores Squeak facilita la implementación de distintos algoritmos de despacho de mensajes.

1.1.2 Repasar la historia sobre la investigación realizada sobre el despacho de mensajes

Este objetivo tiene por finalidad analizar los trabajos realizados hasta el momento sobre el problema del despacho de mensajes y categorizarlos para poder distinguir con mayor facilidad cada solución.

Dentro de este análisis se encontrarán los algoritmos más comunes como DTS (Dynamic Table Search), GLC (Global Lookup Cache) como así también los más complicados como Tablas compactadas de índices de selectores (Compact Selector-Indexed Dispatch Table, CT).

1.1.3 Framework de Despacho de Mensaje con Testing y Profiling

Para poder realizar testeos sobre nuevos algoritmos y ver los recursos que consumen, se creará un framework de despacho de mensajes que ofrecerá posibilidades de realizar testing y profiling dentro de la VM de Smalltalk. Este framework permitirá con facilidad probar nuevos algoritmos de despacho de mensajes y a la vez ver como se comportan con respecto de la performance, los recursos, etc.

El objetivo de este framework será facilitar el desarrollo y prueba de algoritmos de despacho de mensajes.

1.1.4 Nuevos algoritmos de Despacho de Mensajes

Basándonos en el framework del punto 1.1.3 y el estudio realizado sobre los distintos algoritmos del punto 1.1.2, se propondrán nuevos algoritmos de despacho de mensajes.

Estos algoritmos serán testeados usando el framework para ver su comportamiento general y compararlos con otros algoritmos.

1.1.5 Mejorar la performance del sistema mejorando la performance del algoritmo de method lookup

A partir del framework obtenido en el punto 1.1.3, se buscará disminuir el tiempo de despacho de mensajes para lograr una mejora de la performance del sistema en general.

¿Por qué la performance?. Porque debido a la característica dinámica del despacho de mensajes en lenguajes de objetos, el tiempo de ejecución del algoritmo de "*dispatching*" impacta fuertemente en la performance final del sistema.

Una creencia común dentro de la comunidad de sistemas es que los programas orientados a objetos tienen una frecuencia más alta de llamadas a "procedimientos" que los lenguajes tradicionales. Esta es un suposición obvia, más aún cuando se utilizan lenguajes denominados "puros" (ver punto 2.2.5), debido a que toda acción es ejecutada por medio del envío de mensajes, desde acciones aritméticas hasta el control de flujo del programa.

En Smalltalk-80 se realiza un envío de mensaje cada 6.67 bytecodes (Ver [Goldberg2] para definición del bytecode de Smalltalk) según [Falcone]. La frecuencia es aún más alta en el Berkeley Smalltalk, la cual llega a un promedio de un mensaje enviado cada 5.26 bytecodes [Vitek2].

Claramente esto demuestra que el costo de la búsqueda de mensajes es muy importante en la performance del sistema, como demuestra David Ungar en el Proyecto SOAR[6], el cual concluye que el 23% del tiempo de ejecución del programa se consume en el algoritmo de búsqueda de métodos [Ungar].

Cabe aclarar que esta no es una característica propietaria de los lenguajes "puros", es el paradigma orientado a objetos el que incita a crear código más modular, lo cual implica una mayor frecuencia de llamadas. Esto es cierto también para aquellos lenguajes denominados "híbridos" (ver punto 2.2.5) y queda demostrado en [Vitek2], donde se puede observar que el promedio de instrucciones en C++ es de 31.3 por método contra 197.5 en C.

Otra característica de los lenguajes orientados a objetos es que la cantidad de estructuras de control que alteran el flujo de ejecución (estructuras *if...then...else* y *switch...case*) disminuye drásticamente debido al rasgo polimórfico de dichos lenguajes. Esto fortalece más aún la necesidad de acelerar la ejecución del algoritmo de búsqueda de los métodos.

[6] SOAR: Smalltalk On A Risc. Proyecto de la universidad de Berkeley bajo el mando de David Ungar, en el cual se probó la posiblidad de ejecutar Smalltalk con un conjuto reducido de instrucciones de procesador y se documenta la técnica de Garbage Collection denominada "Generation Scavenging". David Ungar fue quien luego tuviese participación activa en el proyecto Self y creador de la técnica PIC.

1.2 Organización general del trabajo

El trabajo está dividido en varias capítulos. El segundo capítulo realiza una breve introducción a los conceptos del paradigma de objetos. El tercer capítulo ahonda la definición de polimorfismo y demuestra la influencia de distintas características de los lenguajes en el algoritmo de búsqueda de métodos.

El cuarto capítulo presenta todos los trabajos de investigación realizados para acelerar el envío de mensajes. En ellos veremos tanto técnicas estáticas como dinámicas, como así también técnicas orientadas a eliminar el envío del mensaje completamente.

El capítulo cinco muestra en detalle el framework de despacho de mensajes implementados. El seis muestra varias implementaciones de algoritmos de despacho de mensajes junto a estadísticas de performance y el siete hace lo propio con algoritmos de caching.

Por último en los capítulos ocho y nueve se tratan posibles trabajos que se pueden realizar en el futuro y las conclusiones finales.

El capítulo 10 muestra las referencias bibliográficas del trabajo, el 11 una pequeña reseña a la notación utilizada para mostrar el diseño de los modelos presentados en el trabajo y el 12 todo el código implementado, ordenado por paquete y clase.

2. Conceptos y Características de la Programación Orientada a Objetos

Este capítulo introduce el concepto de la programación orientada a objetos junto con sus características que diferencian a los lenguajes basados en objetos de otros lenguajes. Su carácter de introducción hace que no sea completo, sin embargo presenta las bases necesarias para el trabajo de investigación propuesto.

2.1 Conceptos

Los conceptos aquí volcados forman parte de el capítulo uno del "Blue Book" [Goldberg1], los cuales son utilizados como base del proyecto de investigación y de la definición del paradigma de programación con objetos.

2.1.1 Objetos y mensajes

Un *objeto* representa un componente o elemento dentro de un sistema de software. Ejemplos de objetos son el número uno, un rectángulo, un compilador.

Un objeto está representado por una parte de memoria privada y un conjunto de operaciones a las cuales el objeto sabe responder (*interface*). Dicha interface depende del tipo de componente que el objeto representa. Objetos que representan números sabrán responder a operaciones aritméticas como la suma, resta, multiplicación y división. Objetos que representan un compilador sabrán responder a mensajes que dado un texto (código fuente) produce código que puede ser ejecutado por una computadora (código ejecutable).

Un *mensaje* es un pedido que se realiza a un objeto para que ejecute alguna de sus operaciones. Un mensaje especifica qué operación se debe realizar, no cómo debe ser realizada.

El objeto que recibe el mensaje (*receiver*) decide qué método debe ser ejecutado, y la única manera de interactuar entre los objetos es por medio de mensajes.

Una propiedad crucial de los objetos es que su *memoria privada* sólo puede ser manipulada y accedida por medio de sus métodos. Esta propiedad implementa el concepto de "*information hiding*"[7].

Information hiding contribuye a minimizar el acoplamiento, puesto que sólo se puede acceder a los datos de un objeto por medio de mensajes. Esto significa que el acoplamiento se realiza ahora por medio de mensajes y no por los datos que un objeto posee. Si se modifica la representación de un dato de un objeto, los colaboradores de dicho objeto no deberían ser impactados debido a que acceden a él por medio de mensajes.

[7] Information Hiding: El término fue introducido por primear vez por D. Parmas. Ver [Kiczalez] y [Parmas]

Un claro ejemplo de estos conceptos son los objetos que representan los números. Dependiendo de la representación numérica con la que se desea trabajar, existen objetos que representas enteros, fracciones, números reales, etc., los cuales son representados de distinta manera. Sin embargo, todos estos objetos responden al mismo conjunto de operaciones aritméticas, como la suma, aunque cada uno de ellos implementa de una manera distinta la ejecución de dicha operación.

Los objetos que responden al mismo conjunto de mensajes son denominados "*objetos polimórficos*", los cuales pueden ser intercambiados sin modificar el resultado final ni romper la secuencia de ejecución del programa.

2.1.2 Clases e Instancias

Una *clase* describe la implementación de un conjunto de objetos. Cada objeto individual descripto por una clase se denomina *instancia de dicha clase*.

Una clase describe la forma en que la memoria privada de una instancia es utilizada y la manera en que se ejecutan los mensajes a los cuales el objeto responde.

La memoria privada de un objeto está definida por lo que se denominan *variables de instancia* y la manera en que el objeto responde a los mensajes está definido por un conjunto de *métodos*.

Todas las instancias de una clase utilizan el mismo conjunto de métodos para describir sus operaciones, sin embargo cada instancia tiene su conjunto privado de variables de instancias.

Cada clase tiene un nombre, el cual la identifica unívocamente en todo el sistema, y cada variable de instancia esta compuesta por la dupla {nombre, valor}. El valor de cada variable de instancia es una referencia a un objeto.

Cada método en una clase especifica cómo se debe ejecutar una operación solicitada por medio de un mensaje. Un método puede modificar la memoria interna del objeto receptor, enviar mensajes a otros objetos o a si mismo y siempre devuelve un objeto resultado de la ejecución del método.

2.1.3 Herencia y Polimorfismo

Una manera de implementar polimorfismo es por medio de *clases y herencia*, que es el mecanismo utilizado en los lenguajes orientados a objetos más tradicionales (Smalltalk, C++, Java).

Sin embargo, hay un conjunto de lenguajes que implementan polimorfismo por medio de *delegación*, siendo el más conocido *Self*. Para una discusión más completa sobre las distintas posibilidades de implementación de polimorfismo y técnicas para compartir código (*code sharing*) ver "*The Treaty of Orlando*" [Stein et. al].

En lenguajes de objetos hay dos implementaciones de herencia.

- Herencia Múltiple (Multiple inheritance)
- Herencia Simple (Subclassing)

Herencia múltiple permite que una clase “herede” el comportamiento de más de una superclase. Aunque inicialmente parece una gran ventaja, debido a que permite “compartir” código de más de una sola clase, la experiencia ha demostrado que es una herramienta peligrosa, difícil de implementar y sustituible por herencia simple.

Por el contrario, subclassing es un mecanismo estrictamente jerárquico lo cual simplifica su utilización e implementación del paradigma en los lenguajes de objetos.

Smalltalk provee subclassing como mecanismo de sharing, y en él se basa su algoritmo de *búsqueda de método* (method lookup).

Una subclase especifica que todas sus instancias van a ser iguales a las instancias de su *superclase* salvo por aquellas diferencias que son explícitamente determinadas. Una subclase es a la vez una clase, por lo que puede tener subclases de sí misma.

Una subclase provee un nuevo nombre para sí pero hereda todas las declaraciones de variable y método de su superclase. Nuevas variables y métodos pueden ser agregados, como también estos últimos pueden ser *sobre-escritos*.

Al sobre-escribir un método se está especificando que una subclase va a reaccionar de una manera distinta al comportamiento especificado en la superclase.

Un concepto que generalmente se confunde con subclassing es el de *subtyping*. El tipo de un objeto esta dado por el conjunto de operaciones que puede realizar (su interface) no por la manera en que dichas operaciones están implementadas.

En Smalltalk subclasificar implica no solo compartir código por medio de una nueva clase sino también crear un sub-tipo de la superclase. En otros lenguajes como Java, los conceptos están más separados. En Java existe el concepto de *Interface* el cual es utilizado para definir tipos, y por separado existe su implementación, que es la clase. De esta manera se puede tener multiple-subtyping pero con herencia simple (que es el mecanismo utilizado para compartir código).

2.1.4 Tipo

Es importante definir correctamente la noción de Tipo en el paradigma de objetos puesto que esto trae muchas confusiones.

En el paradigma de objetos, un Tipo define el conjunto de mensajes que un objeto sabe responder y la definición de cada mensaje está compuesto por el nombre, el tipo y nombre de los colaboradores externos que recibe (parámetros) y el tipo de resultado que otorga[8].

[8] Java agrega a la definición de cada mensaje las excepciones que puede producir.

La diferencia que existe entre tipo y clase, es que la clase además de definir un tipo, define su implementación. Por lo tanto, un tipo solo define "qué" puede hacer un objeto y la clase define "qué" puede hacer y "cómo" lo hace.

Una subclase siempre define como mínimo el mismo tipo de la superclase, por definición de herencia, aunque también puede definir un nuevo tipo si implementa mensajes no definidos en la superclase.

En Smalltalk no existen elementos del lenguaje que permitan diferenciar tipos de clases, simplemente se trabaja con clases. Para definir tipos, generalmente se utilizan clases abstractas (clases con definición de mensajes pero sin implementación[9]) sin embargo esto impide que una clase pueda implementar varios tipos definidos en distintas clases abstractas debido a que Smalltalk utiliza herencia simple.

En Java se realiza la distinción entre tipos y clases. Los tipos son definidos por la construcción sintáctica denominada "Interface" y las clases por medio de la construcción sintáctica "Class". Las clases pueden implementar varias "interfaces", lo que equivale a decir que implementan varios "tipos". A pesar de esta diferencia entre tipo y clase que hace Java, se sigue manteniendo la característica que una clase define también un tipo.

Dada esta definición de tipo, es importante aclarar que se entiende entonces cuando se dice "Smalltalk es no tipado" o "Java es fuertemente tipado". Como pudimos ver, estos comentarios no se deben a la posibilidad de crear tipos puesto que las clases son tratadas como tipos, sino a como se definen y utilizan las variables.

En un lenguaje no tipado (o dinámicamente tipado), las variables no tienen un tipo asociado al nombre de la misma. Simplemente su propósito es "nombrar" a un objeto que en algún momento estarán referenciando sin imponer ninguna restricción sobre la clase de la cual son instancia o el tipo que se espera que implemente.

Por el contrario, los lenguajes denominados tipados son aquellos que asocian al nombre de una variable un tipo. Esto indica que esa variable solo puede referenciar a objetos que implementen ese tipo y en caso de no hacerlo estarían rompiendo el "sistema de tipos".

Según el comportamiento que tiene el lenguaje cuando se no se cumple con el sistema de tipos, es que se los denomina "Fuertemente Tipados" (Strong Typed) o "Debilmente Tipados" (Weak Typed).

Lenguajes fuertemente tipados no aceptan bajo ninguna cirscunstancia que una variable referencie a un objeto que no conforme con el tipo que tiene asociada la variable. Ejemplos de estos lenguajes son Java y Eiffel. Lenguages debilmente tipados permiten que las variables referencien objetos que no conforman con el tipo que tienen definido, sin embargo generalmente avisan sobre estos hechos en tiempo de compilación (siempre y cuando logren detectarlo).

[9] En rigor de verdad, los mensajes abstractos de las clases abstractas envían el mensaje "subclassResponsibility" a "self" para indicar que deben ser implementados por las subclases.

Más adelante veremos como influye al algoritmo de Despacho de Mensajes que el lenguaje utilice variables tipadas o no.

2.2 Características

De los conceptos volcados en los puntos anteriores, se puede concluir que los lenguajes orientado a objetos poseen ciertas características que los identifican. Estas características se derivan de dichos conceptos y no son más que definiciones que agrupan dichos conceptos.

Las mismas se detallan a continuación.

2.2.1 Abstracción

Es por medio de la abstracción que el hombre logra dividir y caracterizar la complejidad.

Una abstracción denota la características esenciales de un ente, las cuales lo distinguen de otros tipos de entes y proveen límites conceptuales definidos, aunque estos pueden depender del punto de vista del creador de la abstracción.

Los objetos son los que representan nuestras abstracciones en los lenguajes de objetos, y las clases (en aquellos lenguajes que las utilicen) son las encargadas de agruparlos, o como se denomina comúnmente, "*clasificarlos*". Las clases reúnen el comportamiento de varios objetos y representan conceptos claros e indivisibles dentro de un problema.

En otros lenguajes, como Self, se utilizan "*prototipos*" para reutilizar comportamiento común, aunque cada objeto puede tener comportamiento propio por más que utilicen el mismo prototipo (Ver [Ungar et al3] y [Smith et al.2]). Esto permite mayor facilidad de abstracción y especialización por objeto (y no por clase como en Smalltalk, por ejemplo).

Como dijimos con anterioridad, lo importante de una abstracción es determinar su comportamiento y cómo se comunica con el exterior (por medio de su protocolo o interface). Bertrand Meyer lleva al extremo este concepto de tal manera que propone implementar en los lenguajes orientados a objetos lo que se denomina "programación por contrato", lo cual él implementa en su lenguaje Eiffel. (Ver [Meyer]).

En la programación por contrato, un método define las pre-condiciones y post-condiciones que se deben cumplir para que la ejecución del método sea exitoso, y las clases definen lo que se denomina la invariante, condición que deben cumplir todas sus instancias en cada instante.

En cada abstracción es importante lograr lo que se denomina "*the principle of least astonishment*", el cual determina que una abstracción debe capturar el comportamiento

completo de un objeto, no más y no menos, y que no debe ofrecer sorpresas o efectos secundarios que vayan más allá del alcance de la abstracción. (Ver [Booch]).

Lograr una buena abstracción es una de las tareas más difíciles en un ambiente de objetos. La tarea de nombrar las abstracciones (aka, darle nombre a una clase) es también difícil pero importante. Es por ello que generalmente el ciclo de vida propuesto en el desarrollo con lenguajes de objetos es iterativo, evolutivo y por medio de prototipos funcionales, los cuales irán evolucionando y estabilizándose a medida que las abstracciones elegidas reflejen con mayor exactitud el problema.

2.2.2 Encapsulamiento

Encapsulamiento es el proceso por el cual la representación e implementación de un objeto se "*esconde*" del mundo exterior. Es este concepto el que permite disminuir el acoplamiento entre distintos objetos.

Es conveniente a esta altura definir correctamente "information hiding" (nombrado con anterioridad) y "encapsulamiento". Encapsular es ubicar en un mismo lugar (el objeto) la representación de sus "datos" y las operaciones necesarias para acceder a los mismos. "Information Hiding" es la característica que impide que un objeto acceda directamente a los "datos" de otro. Dada esta diferencia, se puede inferir que "encapsulamiento" implica "information hiding" como así también ubicar el comportamiento de un objeto en un mismo lugar.

Las abstracciones definen cómo se debe comportar un objeto, y el encapsulamiento es el que permite que la manera en la cual está implementado ese comportamiento sea independiente del objeto que lo requiere.

El encapsulamiento permite detener el denominado "*ripple effect*"[10], o sea que un cambio en la estructura interna de un objeto implique la modificación de sus dependientes. También permite que se pueda modificar la representación de un objeto (su memoria privada) sin que ello impacte en los objetos que lo utilizan.

Esta característica es lograda al 100% en lenguajes como Smalltalk[11] o Self, sin embargo en C++, el encapsulamiento puede ser quebrado fácilmente y no logra limitar el efecto onda, debido a que un cambio en la representación de una clase, implica la recompilación de todas sus subclases.

[10] Ripple effect: Effecto onda. Significa que un cambio en un elemento implique la modificación de un inumerable cantidad de dependientes o no dependientes. Este efecto a veces pasa a ser incontrolable, más aún cuando existen referecias o dependencias circulares.

[11] Cabe aclarar que también en Smalltalk existen maneras de romper el encapsulamiento y acceder desde un objeto a las variables de instancia de otro. Esta carácterística permite la existencia de herramientas para inspeccionar los objetos como los "inspectores" y "debuggers", como así también que se pueda realizar metaprogramación. Sin embargo no es algo común ni recomendado utilizar estas facilidades fuera del contexto mencionado.

2.2.3 Modularidad

Modularidad es el concepto por el cual se pueden agrupar varias abstracciones en lo que se denomina un “módulo”, los cuales determinan límites e interfaces claras dentro de un sistema complejo.

Inicialmente Smalltalk no tenía soporte para esta característica, sin embargo en implementaciones más recientes el concepto fue utilizado para poder “categorizar” los mensajes de una clase. Actualmente Squeak, no solo ofrece la posibilidad de categorizar los mensajes de una clase, sino también agrupar varias clases en una categoría.

En Java los módulos son denominados paquetes (Ver [Gosling]). Los nombres de los paquetes son antepuestos a los nombres de las clases para evitar conflictos entre nombres. A su vez son utilizados como medio de seguridad. En C++ se implementó lo que se denomina “*namespaces*” con un resultado muy negativo.

2.2.4 Reuso

Reuso es un concepto muy amplio. Se puede hablar de reuso de diseño (ver [GOF] y [GOF2]), reuso de modelos conceptuales (Ver [Fowler]) , etc. Sin embargo en este punto nos referiremos al reuso de código.

Se puede obtener reuso de código de varias maneras. Por ejemplo por medio de un buen framework que permita ser instanciando para un problema particular indicándole simplemente los “parámetros” específicos para esa solución, o a través de librería de clases las cuales pueden ser utilizadas para resolver problemas recurrentes, etc.

Además de estas “agrupaciones” de objetos, los lenguajes generalmente ofrecen mecanismos más simples que permiten reusar código, como la jerarquización de clases o la prototipación.

El tener clases agrupadas en jerarquías permite que subclasses puedan reusar el código de la superclase, y así recursivamente. Ver 2.1.3 para una exposición sobre el tema.

La jerarquización es la opción elegida por lenguajes como Smalltalk, Java y C++. La prototipación no tuvo éxito comercial y sólo algunos lenguajes de investigación la utilizan, como Self, Cecil, etc. (Ver [Ungar et al3], [Smith et al.2] y [Chambers]).

El reuso permite disminuir el tiempo y costo de desarrollo de sistemas, de allí que sea una característica muy importante. Sin embargo, lograr un buen reuso es muy difícil y se llega sólo a partir de buenas abstracciones.

2.2.5 Lenguajes Puros vs. Lenguajes Híbridos

La programación orientada a objetos posee varios lenguajes que se dicen compatibles con el paradigma. Sin embargo, se pueden encontrar grandes diferencias entre ellos debido a la oposición entre el “*idealismo*” y “*pragmatismo*”, lo cual a su vez produce

grandes diferencias de opiniones, que terminan siendo discusiones más filosóficas que técnicas.

Los lenguajes ideales son aquellos denominados “lenguajes puros” (*pure object-oriented*) los cuales se basan en la teoría minimalista que implica que todo es un objeto y toda acción es ejecutada por medio de envío de mensajes. Los lenguajes que mejor implementan esta teoría son Smalltalk y Self. Esta investigación se basa ampliamente en dichos lenguajes y la teoría “ideal” que los mismos implementan.

Lenguajes más pragmáticos son C++ y Java. Debido a ello tienen que realizar ciertos balances de implementación que terminan caracterizándolos como lenguajes “híbridos”. Esto es, ni puros ni estructurados. Ejemplos clásicos dentro de C++ es la posibilidad de tener funciones o rutinas (al estilo C) que no dependen de ningún objeto, o tipos de datos básicos como int, long, etc. los cuales son utilizados por ejemplo en Java.

Por supuesto que estas determinaciones dentro del lenguaje tiene sus explicaciones. Björn Stroustrup explica en [Stroustrup2] los motivos de esas decisiones. Por ejemplo, la posibilidad de llamar a funciones C dentro de C++ permite que sistemas nuevos utilicen las librerías creadas para sistemas hechos en C, de lo contrario todo el código debería ser reescrito implicando un alto costo de desarrollo. También esta decisión se debe a que C++ no poseía una librería estándar, error que Stroustrup admite y trató de solucionar en la creación del estándar de C++ propuesto por ANSI, sin embargo ya era demasiado tarde puesto que cada vendor creador de compiladores C++ tenía intereses creados en sus librerías de desarrollo. Por ello, utilizar C++ con el compilador de Microsoft no es lo mismo que hacerlo con el de Borland, por ejemplo.

Gosling en [Gosling] justifica la necesidad de tener tipos básicos de datos en Java como int, long, double, etc. que no se comporten como objetos, por cuestiones de performance. Aunque tener estos tipos de datos permita acelerar la ejecución del código, es tal el problema que le crea al programador por no adherir por completo al paradigma, que termina siendo una desventaja en vez de una ventaja. Este balance hecho en Java también deja de lado la flexibilidad. Ejemplos clásicos de Smalltalk, como calcular el factorial de 100, los cuales son ejecutados rápidamente y con exactitud total, son muy difíciles de lograr en Java y necesitan de una codificación especial.

Este balance de performance vs. adhesión total al paradigma es algo que vamos a ir viendo desaparecer a medida que los MIPS[12] vayan siendo más baratos, puesto que lo que no se puede abaratar es el costo de la educación de la gente y el proceso de desarrollo de los sistemas.

Las diferencias mencionadas entre ambos tipos de lenguajes hacen que el código y las técnicas utilizadas para la resolución de problemas sean muy distintas (aunque ahora se este empezando a encarar soluciones genéricas más a nivel diseño, ver [GOF]). Estas diferencias hacen también que en el momento de optimizar el código, se tengan que aplicar distintas metodologías. Esta investigación se concentrará en los lenguajes denominados puros, más exactamente en Smalltalk, para limitar el rango de acción.

[12] MIPS: Millones de Instrucciones por Segundo

2.2.6 Mundo cerrado vs. Mundo Abierto

Otra diferencia entre los distintos tipos de lenguajes orientados a objetos es cuan abiertos son.

En lenguajes como Smalltalk, el programador tiene todo el sistema al alcance de sus manos. Desde el código de las librerías o frameworks hasta la manera en que ejecuta la Virtual Machine. Esta característica hace que la implementación de problemas como lógicas difusas, nuevas estructuras de control, etc. sean fácilmente agregadas al sistema de programación. Es un "mundo abierto", donde el programador es el que tiene el control y no la computadora.

En el caso de lenguajes como C++ y Java, el mundo en el que trabajan es más "cerrado". No todo el código está disponible, no es posible modificar la sintaxis del lenguaje, etc. Esta decisión se debe por supuesto a un balance que se hace entre flexibilidad y performance, eligiendo estos lenguajes la última.

Performance es una característica que vende en los lenguajes de desarrollo. Flexibilidad es una característica necesaria en los lenguajes de investigación. Lamentablemente performance sigue siendo un punto muy importante dentro de los problemas no funcionales del desarrollo de software y es uno de los puntos que este trabajo investigará. Sin embargo, es nuestra creencia que la importancia de dicha restricción irá disminuyendo a medida que los MIPS sean más baratos, motivo por el cual creemos que lenguajes como Smalltalk tienen el futuro asegurado.

Cabe aclarar que en Smalltalk existen ciertos "trucos" que hacen que el mismo sea utilizable para desarrollos comerciales. Uno podría pensar que estos "trucos" atentan contra el principio de "mundo abierto", sin embargo están documentados y generalmente los distintos Smalltalk permiten que sean modificados.

Un ejemplos de estos "trucos" es el conjunto de mensajes cuyo código es agregado directamente en el método llamador. Estos mensajes son por ejemplo `ifTrue:`, `ifFalse:`, `whileTrue:`, etc (Ver [Goldberg2]). Por lo tanto hacer modificaciones al método `ifTrue:` de la clase True, no afectaría en nada la ejecución del sistema.

También existen ciertas clases que son tratadas de manera particular, como la clase SmallInteger, la cual en realidad no es representada con un espacio de memoria propio, sino por el valor del puntero que lo referenciaría (Ver [Goldberg2]). De esta manera, si se utiliza el número 10 en el método `m1` y en el método `m2`, ambos métodos poseerán la representación de dicho número en su bytecode y no una referencia a una zona de memoria que contenga el número 10. Esto permite disminuir la cantidad de memoria necesaria para los enteros (la cual no es poca), la necesidad de crear un objeto por cada número, lo cual es un proceso caro (teniendo en cuenta que después puede generar basura que debe ser recolectada por el garbage collector) y la necesidad de mantener referencias únicas (esto es que variables que referencien al número 10, lo hagan apuntando a la misma zona de memoria).

Más allá de estos “trucos” de los cuales se debe valer Smalltalk para obtener una performance considerable, es interesante cómo todo esto está oculto al programador y el principio minimalista del paradigma es respetado a rajatabla.

2.3 Conclusiones

Como pudimos ver en el transcurso de este capítulo, la programación con objetos tiene varios conceptos interesantes como también características importantes que facilitan el desarrollo.

También se hizo hincapié en la importancia del polimorfismo y cómo afecta la flexibilidad de un sistema. En el capitulo que viene se verá más en detalle qué significa y cómo se implementa el polimorfismo.

3. Polimorfismo

Este capitulo se dedica enteramente al polimorfismo. Se verá más en detalle cómo puede ser implementado y qué características de los lenguajes lo pueden afectar.

3.1 Definición

La palabra "Polimorfismo" proviene del griego *Poly* (Muchas) *Morph* (Formas), "muchas formas".

Como vimos en el capítulo anterior, los objetos definen su comportamiento a través de los mensajes que saben responder. A un conjunto de mensajes, se lo denomina protocolo. Un protocolo puede estar compuesto, por ejemplo por un único mensaje o por todos los mensajes de un objeto.

En la teoría de objetos se dice que dos objetos son polimórficos si saben responder a un mismo conjunto de mensajes o protocolo. Por lo tanto si tengo dos objetos **A** y **B**, y ambos responden al mensaje *m1*, se dice que **A** y **B** son polimórficos con respecto al mensaje *m1*.

En rigor de verdad, un objeto no puede tomar distintas formas, sino que son las variables las que poseen la capacidad de referenciar distintos "tipos" de objetos. Por lo tanto el resultado de enviar un mensaje puede variar según el objeto destinatario, el cual es "nombrado" por una variable (u obtenido por medio de enviar un mensaje a un objeto, el cual en definitiva es nombrado por una variable).

3.2 Llamada a funciones en lenguajes procedurales

En los lenguajes procedurales, una llamada a función es traducida simplemente a la instrucción de código máquina denominada "*call*", luego de haber preparado el contexto de la nueva función, trabajo que se realiza generalmente modificando la pila de la máquina.

Por ejemplo, si usamos el lenguaje C, una llamada del estilo:

```
funcion1 ( 10, "Hola" );
```

Figura 2: Llamada a función en lenguaje 'C'

Es traducida al pseudo lenguaje máquina de la arquitectura x86 de la Figura 3.

```
Push Direccion de “Hola”
Push 10
Call funcion1
```

Figura 3: Implementación assembler de llamado a función

Como se puede observar, el costo de la llamada a la función es simplemente el de la ejecución de la instrucción call (asumiendo que el pasaje de parámetros es similar en lenguajes procedurales y orientados a objetos), la cual depende de la cantidad de ciclos que consume el procesador para ejecutarla. La instrucción call es una instrucción ampliamente soportada por los procesadores y arquitecturas actuales y su costo está ampliamente aceptado debido a la ventaja que produce poder realizar llamadas a funciones, lo cual se traduce en una mayor modularidad del código.

En este tipo de llamadas, el binding del label “*funcion1*” y su dirección en memoria se realiza en tiempo de link-edición (o compilación dependiendo de donde se encuentre la función a llamar). Dicho binding no se realiza en tiempo de ejecución, lo cual significa una mayor rapidez de ejecución del programa.

Sin embargo, esta situación, que parece ser ventajosa, termina siendo una desventaja debido a su rigidez. G. Kiczales asegura que “*cuanto más tarde se realice el binding, más flexible será el programa*” [Kiczalez]. Al realizar el binding en tiempo de compilación, no hay manera de modificar el comportamiento del programa en tiempo de ejecución, como tampoco se pueden realizar llamadas dependiendo de la estructura de datos que se está manipulando.

Supongamos cómo se comportaría una librería de Interface Gráfica escrita en un lenguaje procedural, digamos C, que contiene los siguientes componentes: Button, List y EntryField, los cuales pueden ser parte de formularios o ventanas, que por cuestiones de simplicidad llamaremos Window.

Debido a que una Window puede contener componentes gráficos, utilizará una estructura para manejar la colección de datos correspondiente, como por ejemplo un vector.

La función más común que la librería debe realizar con estos componentes es poder dibujarlos en la pantalla. Para ello la misma poseería una función por cada componente. (Ver Figura 4: Definición de funciones para dibujar componentes).

```
void drawButton ( Button aButton )
void drawList ( List aList )
void drawEntryField ( EntryField aEntryField )
void drawWindow ( Window aWindow )
```

Figura 4: Definición de funciones para dibujar componentes

Debido a que una Window puede contener componentes, la función drawWindow () tendrá que recorrer su lista de componentes y pedirle a cada uno de ellos que se dibuje. (Ver Figura 5: Implementación de la función drawWindow)

Como se puede observar este código es proclive a tener errores y además es claramente ineficiente.

Es un código proclive a tener errores debido a la gran cantidad de sentencias "if" que el mismo posee. Es muy fácil cometer errores en códigos con muchas condiciones. Estas estructuras de control son una gran fuente de los errores producidos en la programación de sistemas.

Otro motivo de problemas con este código se debe a que es poco flexible al cambio. Si se llega a agregar un nuevo tipo de componente, habrá que modificar todas aquellas funciones en la cual se realiza este tipo de "*case*" para determinar a que función llamar, lo cual es altamente costoso en lo que respecta al tiempo de modificación y por supuesto, testeo, puesto que se estarían tocando infinidad de funciones. (Un buen testeo realizaría un regression test sobre todas aquellas funciones modificadas.)

El código además es ineficiente debido a la gran cantidad de condiciones "*if*" que se están realizando, la cual crece a medida que se incrementa la cantidad de componentes. Este es un costo muy grande en librerías gráficas que están compuestas por casi un centenar de componentes.

```
void drawWindow ( Window aWindow ) {
int i;

for ( i=0;i<aWindow.components.size;i++ )
{
    if ( aWindow.components[i] es un Button )
         drawButton ((Button) aWindow.components[i]);
    else if ( aWindow.components[i] es una List )
         drawList ((List) aWindow.components[i]):
    else if ( aWindow.components[i] es EntryField )
         drawEntryField
((EntryField)aWindow.components[i]);
     else
         // Error!!!
         abort (); }
}
```

Figura 5: Implementación de la función drawWindow

3.3 Llamadas polimórficas

El mismo problema planteado en el punto anterior, es fácilmente resuelto en la programación orientada a objetos debido a la característica que se denomina polimorfismo, en el cual se basan gran parte de los beneficios de la programación con objetos.

El polimorfismo es una herramienta de programación muy potente que permite invocar una operación sobre un objeto, especificando el efecto deseado, sin tener que preocuparse acerca de los detalles de implementación que cada objeto diferente tiene.

Siguiendo el ejemplo del punto anterior, el mismo problema sería resuelto creando un método "*draw*" para todos los componentes, el cual como su nombre lo indica, se encargaría de dibujar en pantalla el componente que recibe dicho mensaje.

Por lo tanto, el código para dibujar una Window, sería simplificado en el presentado en la Figura 6: Método draw de la clase Window.

Como se puede observar, el código es mucho más simple, consistente y fácil de entender, lo cual mejora su mantenibilidad. A la vez es más eficaz debido a que se eliminaron todas las cláusulas de comparación necesarias de los lenguajes procedurales.

Con polimorfismo, el programador está liberado de tener que realizar la correcta llamada a función dependiendo del componente o tipo de dato (*binding*). El soporte de run-time se encarga de distinguir los distintos componentes e invocar la correcta operación *draw*.

```
Window>>draw ()

self components do: [ :aComponent | aComponent draw ].
```

Figura 6: Método draw de la clase Window

Esto permite realizar una rápida prototipación y permite que el código sea mucho más fácil de mantener. Por ejemplo, si un nuevo componente es agregado al sistema, solo es necesario implementar el método *draw* para dicho componente y todo el resto del código manipulará el nuevo componente como si siempre hubiese existido.

El polimorfismo ayuda a minimizar la cantidad de estructuras de control dedicadas a decidir el control del flujo como son el if y el case (no es extraño que la estructura case no exista en la especificación de Smalltalk-80[Goldberg1]). Al minimizar este tipo de estructuras se obtiene un código más simple de implementar y mantener.

Por supuesto todos estos beneficios que el polimorfismo ofrece no se logran utilizar tan fácilmente. Como gran parte de nuestra profesión, realizar la decisión correcta sobre cómo aplicar polimorfismo a los objetos de un sistema es más una cuestión de experiencia y buena capacidad de diseño que un proceso que pueda ser estipulado paso a paso.

El mal uso de polimorfismo puede llevar también a resultados desastrosos, como cualquier herramienta que es mal utilizada.

Esta gran facilidad que nos ofrece el polimorfismo es el resultado de la delegación de responsabilidades que hace el programador sobre la computadora. Dicha responsabilidad es el *binding* de la operación que se quiere realizar. En sistemas orientados a objetos este binding se realiza en tiempo de ejecución, de allí que las llamadas polimórficas sean más lentas que una llamadas a funciones de un lenguaje procedural. Sin embargo como ya veremos, el tiempo adicional en el que se incurre es despreciable con respecto de la gran flexibilidad que nos ofrece, como así también debido a la disminución en el uso de estructuras if y case.

La manera en que se resuelve el binding en tiempo de ejecución depende de cada lenguaje y cada implementación. Hay lenguajes en los cuales, por su naturaleza tipada, la llamada a un método polimórfico simplemente significa una indirección más con respecto al código presentado en el punto 3.2 (Por ejemplo C++ o Java). En otros lenguajes, el mecanismo de determinación del código a ejecutar (o *message dispatch*) consume unos ciclos más de procesador (como Smalltalk o Self).

3.4 Simulación de polimorfismo con lenguajes procedurales

Por supuesto que el polimorfismo no es exclusividad de los lenguajes orientados objetos o imposible de lograr con lenguajes procedurales. Simplemente con lenguajes orientados a objetos el polimorfismo es una característica intrínseca.

Con lenguajes procedurales que soporten punteros a funciones, el polimorfismo se puede emular. Es más, los primeros compiladores de C++ creados por B. Stroustrup creaban código C, el cual era luego compilado para obtener el código ejecutable. (Ver [Stroustrup1] y [Stroustrup2]).

Tratar de emular esta facilidad no es algo sencillo, y el código producido no es muy limpio lo cual perjudica su mantenibilidad. Además, se pueden cometer errores de tipos que pueden llevar a que el programa aborte su ejecución.

Otra manera de resolver esta situación es creando una única función que reciba como parámetro el tipo de datos sobre el cual se desea ejecutar la función. Luego, esta función dependiendo del tipo de dato pasado como parámetro eligiría qué código ejecutar.

Para seguir con el ejemplo, está función quedaría de la siguiente manera:

```
draw ( Component aComponent )
{
   switch ( aComponent.type )
     case Button:
          drawButton ( (Button) aComponent );
          break;
     case List:
          drawList ( (List) aComponent );
          break;
     case TextField:
          drawTextField ( (TextField) aComponent );
          break;
     case Window:
          drawWindow ( (Window) aComponent );
          break;
     default:
          abort ( “Componente inválido” );
}
```

Figura 7: Implementación de la función draw en lenguaje procedural

La ventaja que tiene esta metodología es que no se estaría trabajando con punteros a funciones.

Aunque esta implementación remueve la necesidad de tener que realizar una comparación cada vez que se desee realizar un “*draw*”, el programador debe resolver manualmente la implementación que simula la llamada polimórfica, por lo que tiene los mismos problemas de mantenimiento que se mencionaron anteriormente.

3.5 Envío de mensajes en lenguajes Orientados a Objetos

Los lenguajes orientados a objetos permiten la definición de los tipos de datos y sus operaciones en un solo lugar, que comúnmente se denomina clase o prototipo (Para una definición más completa de estos elementos ver [Goldberg1] para clase y [Stein et. al], [Ungar et al3],[Smith et al.2] para prototipos).

En lenguajes orientados a objetos como Smalltalk, la resolución de llamadas a métodos se resuelve por medio de lo que se denomina "envío de mensajes" (*message send*) y la resolución del código a ejecutar dado un mensaje se denomina "despacho de mensajes" (*message dispatch*).

Una de las técnicas más simples y utilizadas de despacho de mensajes es la denominada Búsqueda en Tablas de Despacho (Dispatch Table Search, DTS de ahora en adelante).

En esta técnica cada clase almacena la definición de los mensajes que implementa y tiene un referencia a su superclase. Cuando se envía un mensaje a un objeto, el mismo le pide a la clase que busque dentro de su diccionario de tuplas *{mensaje,método}* la implementación correspondiente para dicho mensaje. En caso que la clase no posea dicha implementación, le pide a su superclase que realice la búsqueda en su propio diccionario, siguiendo así la cadena de herencia hasta llegar a la raíz de la jerarquía[13]. En caso de no encontrarse el mensaje que se está buscando se reportará un error, indicando el motivo. Dicho error en el caso de Smalltalk es resuelto por el envío de otro mensaje denominado: *#doesNotUnderstand*, al cual todos los objetos saben responder y cuyo comportamiento por omisión es mostrar una pantalla denominada MNU (Message Not Understood), la cual permite abrir la ventana de depuración de código (*debugger*). (Ver [Goldberg1],[Vitek2] y [Driesen2] para una explicación más exhaustiva del algoritmo).

Este algoritmo es muy simple de implementar, pero es completamente ineficiente. Además, cuanto más profundidad posee una rama dentro del árbol de jerarquía, más lenta puede llegar a ser la resolución del método a ejecutar. Por ello, muchas técnicas han tratado de disminuir el tiempo utilizado por este algoritmo.

3.5.1 Influencia del Sistema de Tipos del Lenguaje

La flexibilidad del polimorfismo y el costo de la resolución de llamadas polimórficas se ve muy afectado por el sistema de tipos utilizado por el lenguaje.

Lenguajes con un fuerte o débil chequeo de tipos (*Strong o Weak Typed*) pueden resolver el binding de las variables con respecto a su tipo en tiempo de compilación. Esto reduce drásticamente el costo de la resolución de llamadas polimórficas.

[13] Hay implementaciones de Smalltalk que difieren un poco en como referencian los diccionarios de mensajes, como el Smalltalk Express, sin embargo el algoritmo sigue siendo similar.

En el ejemplo de la Figura 8, el compilador puede determinar de antemano si realmente objetos del tipo Component saben responder al mensaje draw. En caso negativo, genera un error en tiempo de compilación. En el caso que el mensaje sea válido el proceso de búsqueda del código correspondiente al mensaje draw conlleva una indirección más que las llamadas a funciones procedurales, como veremos más adelante.

```
Component aComponent;
.
.
aComponent = new List ();
aComponent.draw ();
```

Figura 8: Lenguajes tipados y polimorfismo

En lenguajes tipados dinámicamente, como Smalltalk, las variables no tienen asociado un tipo de dato con el cual van a trabajar exclusivamente (no confundir con el tipo de los objetos, los objetos siguen definiendo un tipo de dato, el cual queda determinado por su interface), por lo tanto los métodos que pueden ser ejecutados por los objetos referenciados por dichas variables (las cuales, en definitiva, lo único que hacen es otorgarle un nombre a un objeto) deben ser buscados en tiempo de ejecución a partir del mensaje enviado y el objeto que recibe el mensaje.

```
| aComponent |
aComponent := List new.
aComponent draw.
```

Figura 9: Lenguajes no tipados y polimorfismo

En el ejemplo de la Figura 9, el soporte de ejecución debe buscar la implementación del mensaje draw para la clase de la cual es instancia el objeto que está siendo referenciado por la variable aComponent (en este ejemplo una List), y en caso de no encontrarlo reportar un error.

Lenguajes tipados estáticamente hacen que sea más simple la implementación de la búsqueda del código a ejecutar, dado que la cantidad de mensajes que se pueden enviar al objeto referenciado por la variable está determinado por el tipo que tiene asociado dicha variable (Ver definición de tipos para más detalle).

En lenguajes tipados dinámicamente, no hay relación entre las variables y los mensajes que se le asocian en una expresión, puesto que dicha variable puede estar referenciando cualquier objeto. Por lo tanto, el conjunto de mensajes sobre los que hay que realizar la búsqueda es mucho más amplio que para los lenguajes tipados. Además puede suceder que dicho mensaje no esté implementado en el objeto que la variable está referenciando.

Justamente la característica dinámica de los lenguajes tipados dinámicamente hace que el lenguaje sea más flexible que los tipados estáticamente, facilitando así el desarrollo de sistemas.

Es aquí donde se presenta uno de los dilemas más recurrentes dentro del desarrollo de software: ¿Dejamos que el problema lo resuelva la computadora o hacemos que el programador lo resuelva?. La respuesta debería ser simple y deberíamos inclinarnos por la primera. Sin embargo, el tiempo de respuesta de los sistemas termina siendo un factor importante en todos los proyectos, por lo que hay veces en que hay que sacrificar flexibilidad y facilidad de implementación por rigidez y mayor costo de implementación debido a la performance final.

3.5.2 Influencia de Herencia Múltiple

La herencia múltiple permite que una clase herede comportamiento de dos o más superclases. Como veremos a continuación, esto conlleva varios problemas para la implementación del algoritmo de despacho de mensajes.

El poseer más de una superclase hace que el algoritmo despacho de mensajes pueda encontrar el método a ejecutar de un mensaje en más de una rama de la jerarquía de herencia. Por lo tanto, ¿por qué rama empieza la búsqueda?, o ¿qué hace en el caso de que haya dos implementaciones distintas del mismo mensaje en distintas ramas de la jerarquía?

Para ejemplificar este problema utilizaremos una herencia de clases de tal manera que se produzca lo que se denomina “jerarquía diamante”. Ver Figura 10.

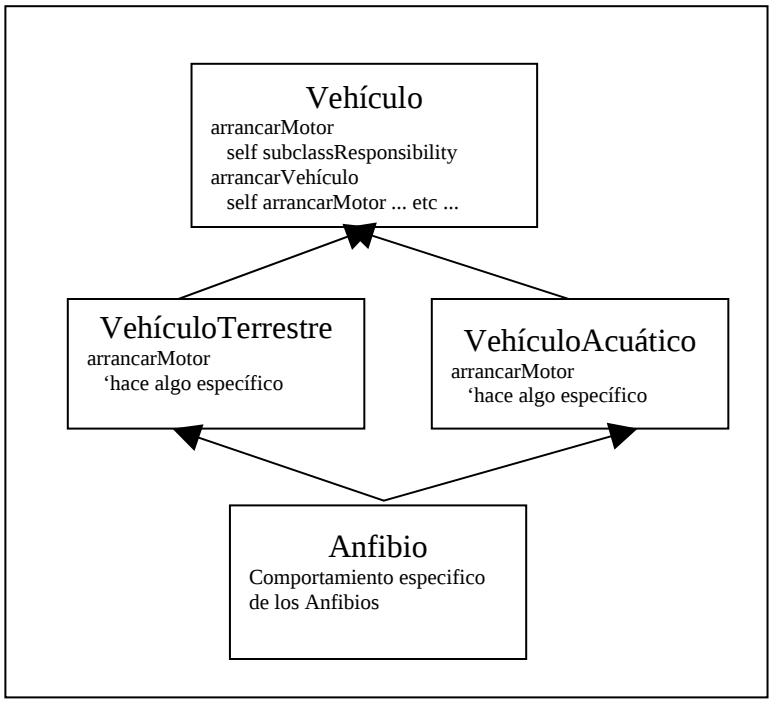


Figura 10: Jerarquía diamante

Supongamos que en dicha jerarquía, la clase Vehículo define el mensaje "arrancarMotor" de tal manera que debe ser implementado por las subclases y "arrancarVehículo" que se encarga de indicarle a cada uno de los componentes del vehiculo que arranque, entre ellos el motor, enviando el mensaje "arrancarMotor". VehículoTerrestre y VehículoAcuático implementan de manera particular el mensaje "arrancarMotor".

Ahora supongamos que una instancia de la clase Anfibio recibe el mensaje "arrancarVehículo". Según la implementación propuesta por Vehículo, este envía el mensaje "arrancarMotor", y es aquí donde se produce el dilema, ¿cuál sería el método "arrancarMotor" a ejecutar?, ¿el implementado en VehículoTerrestre o en VehículoAcuático?. El compilador no lo puede determinar sin que se le provea una regla para resolver esta inconsistencia.

En realidad el motor a arrancar del Anfibio dependerá de si este está en la tierra o en el agua, una característica completamente dinámica, lo que hace que sea imposible que el compilador pueda resolver el problema "estáticamente" puesto que no tiene información sobre el estado de la ejecución.

Las técnicas utilizadas para resolver el problema de la "jerarquía diamante" varían según el lenguaje. En CLOS dependerá de cómo se declaró la clase Anfibio (Ver [Kiczalez et. al],[Paepcke] y [Kiczalez et al2]). Si en la lista de superclases aparece primero VehículoTerrestre y luego de VehículoAcuático, por default ejecutará el método "arrancar" de VehículoTerrestre sin embargo es posible redefinir este comportamiento[14]. En el caso de C++, el problema se resuelve indicando que el mensaje arrancar se envía por ejemplo, al VehículoTerrestre de la manera que se muestra en la Figura 11. (Sin embargo como se puede ver en el ejemplo, esto implica redefinir el mensaje "arrancarVehículo" en Anfibio, no se puede reutilizar el definido en Vehículo)

```
void Anfibio::arrancarVehículo ()
{
     .
     .
     VehículoTerrestre::arrancar (); // Se explicita el
método a ejecutar
     .
     .
}
```

Figura 11: Resolución de conflicto de múltiple herencia en C++

Tanto Java como Smalltalk no permiten utilizar herencia múltiple a nivel clases, sin embargo Java permite utilizar "herencia múltiple" a nivel tipos.

Como vimos en el capítulo anterior, Java utiliza Interfaces para definir tipos. Estos tipos describen comportamiento pero no proveen implementación. Al no tener las interface

[14] CLOS tiene un sistema de metaclases muy potente, denominado "Meta Object Protocolo (MOP)", el cual permite redefinir este algoritmo dependiendo de la clase (entre otras cosas). Por lo tanto, para el ejemplo propuesto el programador podría decidir que para los Anfibios siempre se arranca el motor del VehiculoAcuatico.

una implementación asociada, el problema del “diamante” no existe. Por ejemplo, si una interface denominada “Interface3” hereda (en Java se diría “extiende”) de otras dos denominadas “Interface1” e “Interface2” las cuales definen un mismo mensaje “m1”, lo único que esta definición de tipos indica es que aquellas clases que implementen la interface “Interface3” deben implementar el mensaje “m1”. La implementación provista para este mensaje “m1” será utilizanda tanto si se ve a dicho objeto por medio de variables del tipo “Interface1”, “Interface2” o “Interface3”.

En conclusión, la herencia múltiple disminuye la eficiencia de la resolución de llamadas polimórficas, complica el algoritmo de despacho de mensajes y hace que las estructuras que soporten el despacho de mensajes para herencia múltiple sean de mayor tamaño que para herencia simple. Para una mayor detalle Ver [Vitek2],[Vitek et al.1] y [Driesen2].

3.5.3 Despacho Simple vs. Despacho Múltiple de mensajes

En la mayoría de los lenguajes orientados a objetos, el tipo del objeto que recibe el mensaje (*receiver*) es el que “permite decidir” el código a ejecutar.

Sin embargo lenguajes como CECIL[Chambers], CLOS[Paepcke],[Kiczalez et. al], C++ permiten que el tipo de algún colaborador argumento del mensaje, refine la selección del código a ejecutar. Esta característica se denomina Despacho Múltiple.

Utilizando el ejemplo de los componentes visuales y la ventana, se puede mostrar su utilidad. Supongamos ahora que la operación *draw* se puede realizar sobre distintos medios, como por ejemplo la pantalla y la impresora. Supongamos también que el dispositivo donde se desea realizar la operación de dibujo es pasado como colaborador al método *draw*. (Ver Figura 12 para una definición de dichos métodos en el lenguaje C++.)

Este tipo de facilidad se resuelve en C++ de manera estática y se denomina sobrecarga de parámetros (*parameter overloading*)[Stroustrup1].

```
Class Component
{
public:
     void draw ( Screen aScreen );
     void draw ( Printer aPrinter );
}

void Componente::draw ( Screen aScreen ) { Algún código }
void Componente::draw ( Printer aPrinter ) { Algún código }

void main ()
{
     Printer p = new Printer ();
     Button b = new Button ();
     b.draw ( p );  // Ejecutaría el código correspondiente
a dibujar el botón en la impresora
}
```

Figura 12: Double dispatch en C++

Este rasgo que poseen algunos lenguajes puede ser emulado con una técnica denominada "despacho doble" (*double dispatching*) (Ver [Ingalls2],[GOF],[GOF2], [Vlissides]) y es muy utilizado por ejemplo en Smalltalk para resolver los mensajes "+","-", etc. de las clase Number (Ver [Goldberg1],[GOF2]).

En Smalltalk podríamos por lo tanto utilizar la técnica de *double dispatch* para resolver el problema planteado en párrafos anteriores, según muestra la Figura 13.

Sin embargo *Double dispatch* es más difícil de mantener que *multiple dispatch* y además es más lento (puesto que se está enviando un mensaje más que multiple dispatch). Por ello uno de los objetivos de este trabajo es investigar que cambios son necesarios hacer al algoritmo de despacho de mensajes para soportar *multiple dispatch* puesto que el mismo es importante para implementar subjetividad.

Según [Callegari et al] subjetividad es la habilidad que tiene un objeto de comportarse de diferentes maneras al momento de recibir un mensaje, de acuerdo con los estímulos o factores (internos o externos) que influyen sobre él en dicho momento. Esta habilidad permite diseñar objetos que se asemejan más al comportamiento de los objetos del mundo real, disminuyendo así el gap semántico entre el modelo computacional y la realidad.

No es intención de este trabajo profundizar sobre subjetividad sino ver como influye el algoritmo de despacho de mensaje en su implementación. Para más información sobre subjetividad ver [Callegari et al].

Para más información sobre despacho múltiple de mensajes, ver [Driesen].

```
Button>>draw: aMedia
     aMedia drawButton: self.

List>>draw: aMedia
     aMedia drawList: self.

Screen>>drawButton: aButton
     algún código

Screen>>drawList: aList
     algún código

Printer>>drawButton: aButton
     algún código

Printer>>drawList: aList
     algún código
```

Figura 13: Implementación de Double Dispatch en Smalltalk

3.6 Conclusión

En este capítulo pudimos ver la diferencia en flexibilidad, implementación y performance entre llamadas polimórficas y llamadas no polimórficas. También vimos cómo la implementación del algoritmo de búsqueda de mensaje es influenciado por características del lenguaje, como la herencia, el tipado, etc.

4. Técnicas de implementación de llamadas polimórficas

En esta sección se presentan las distintas optimizaciones realizadas para la resolución de las llamadas polimórficas, como así también su influencia en tiempo y memoria.

Las mismas están separadas en tres categorías: Técnicas Generales, Técnicas Dinámicas y Técnicas Estáticas.

Las Técnicas Generales son ineficientes. DTS es ineficiente por el tiempo que consume y STI por la gran cantidad de memoria que necesita. Sin embargo son utilizadas, como por ejemplo DTS que es utilizada en Smalltalk como algoritmo de resguardo como veremos más adelante.

Las técnicas estáticas calculan todas las estructuras de búsqueda en tiempo de compilación o linkedición y no permiten realizar cambios en tiempo de ejecución. Estas técnicas utilizan sólo la información obtenida de la compilación de código fuente de los programas. No se utiliza información adicional como comportamiento en ejecución, etc.

Las técnicas dinámicas por el contrario pueden computar alguna información en tiempo de compilación, pero modifican dinámicamente en tiempo de ejecución sus estructuras de búsqueda. Debido a esta característica, las técnicas dinámicas pueden explotar información que se obtiene en tiempo de ejecución para mejorar los algoritmos y la performance como veremos más adelante.

A continuación podemos ver una tabla donde se muestran todas la técnicas:

	Abreviatura	Nombre Completo
Técnicas Dinámicas	GLC	Global Lookup Cache
	IC	Inline Cache
	PIC	Polymorphic Inline Cache
Técnicas Estáticas	VTBL	Virtual Function Tables
	VTBL-MI	Virtual Function Tables- Multiple Inheritance
	SC	Selector Coloring
	RD	Row Displacement
	CT	Compact Tables
Técnicas Generales	DTS	Dispatch Table Search
	STI	Selector Table Indexing

Tabla 1: Diferentes técnicas de message dispatch

4.1 Factores que influyen en los algoritmos de despacho de mensajes

Hay varios factores que influyen en el algoritmo de despacho de mensajes. Estos factores determinan la usabilidad del algoritmo, la performance, los recursos que necesitan, etc.

Ha sucedido que por no tener en cuenta todos estos factores, soluciones que parecían óptimas terminaron siendo más lentas que la técnica que suponían reemplazar. Un claro ejemplo es el trabajo de Jan Vitek [Viteck1] que creaba un tabla compactada de selectores, a los cuales se podían acceder rápidamente por medio de un índice. Sin embargo, por no haber tenido en cuenta el tiempo de ejecución del código del prólogo[15] de los métodos más el código adicional para cada búsqueda el resultado terminó siendo más lento y ocupando más espacio de lo esperado [Vitek et al.1] .

A continuación se mencionan cuáles son dichas características para que el lector las tenga en cuenta durante la presentación de los distintos algoritmos. Cabe aclarar que algunas de estas características cobrarán importancia en algunos algoritmos y en otros serán irrelevantes.

4.1.1 Costo de ejecución

El más simple de imaginar es el costo de ejecución de la llamada. Cuanto menos ciclos de máquina represente, más rápidamente se ejecutará la llamada.

Es un punto es trivial, por lo que no necesita explicación adicional.

4.1.2 Espacio en memoria de las estructuras de búsqueda

Un punto que generalmente no se tiene en cuenta es el espacio que ocupan en memoria las estructuras de datos utilizadas para resolver el problema.

El caso más simple de búsqueda consiste en tener un array de doble entrada, utilizando las clases como filas y los selectores como columnas. Esta estructura permite encontrar rápidamente un método, dado su clase y su selector. Sin embargo, el espacio que ocupa es inadmisible, puesto que está determinado por la cantidad de clases multiplicado por la cantidad de selectores multiplicado por la cantidad de bytes que ocupa cada entrada. Esta estructura contendría elementos vacíos en más de 80-90% en lenguajes como Smalltalk (Ver [Vitek2]). Esta solución además de ocupar demasiada memoria, estresa[16] el *garbage collector*, lo cual en definitiva termina degradando la performance general del sistema.

[15] Se denomina “prólogo” de un método al código que el compilador agrega al principio del mismo para realizar tratamientos especiales que están fuera de la funcionalidad del método. Por ejemplo, código para reservar espacio en la pila de ejecución para variables locales.

[16] El término “estresa el gabage collector” se utiliza para indicar que el tiempo utilizado por el garbage collector para liberar memoria se ve incrementado debido a la gran cantidad de objetos que debe recorrer para identificar cuales pueden ser borrados.

4.1.3 Espacio del código producido para la búsqueda

Hay algoritmos de búsqueda que se basan en una única "función" como punto de entrada del proceso de enviar un mensaje. Un ejemplo es la implementación de Squeak, el cual siempre llama al método *findNewMethodInClass:* de la máquina virtual.

Existen otras técnicas que se basan en producir código de búsqueda dependiendo de cada selector. Este código es agregado como parte del envío del mensaje. Por ejemplo, si el compilador determina que el mensaje *m1* posee una única implementación, podría llamar a un método de búsqueda especializado para dicha característica.

Como se comentó anteriormente, el no tener en cuenta este factor hizo fracasar las expectativas del trabajo realizado en [Vitek et al.2].

4.1.4 Código del prólogo del método

Hay soluciones que utilizan un prólogo dentro de cada método que se encarga de verificar si la clase en la cual se está ejecutando es igual a la que se estaba esperando, en caso negativo se realiza una nueva búsqueda, en caso positivo se continua con la ejecución del método.

Para estos casos, también hay que tener en cuenta el espacio y tiempo de ejecución necesario de este prólogo. No hacerlo puede llevar a conclusiones erróneas.

4.1.5 Tiempo de generación

El tiempo utilizado para la generación de las estructuras de búsqueda más el código que utilizan dichas estructuras no modifica en nada la performance final del sistema. Sin embargo es un factor importante en un ambiente de desarrollo dinámico[17], donde tiempos de más de 10 segundos para la compilación de un método son inaceptables.

Por ello es importante recalcar que por más que las técnicas generen estructuras de datos compactas y algoritmos de búsqueda eficientes, si las mismas implican tener grandes tiempos de compilación, no serán aceptadas por la comunidad que desarrolla sistemas usando ambientes de objetos[18].

[17] Un ambiente de desarrollo dinámico permite simple y rápidamente modificar las clases que se están desarrollando. En un ambiente de este estilo, los tiempos de compilación, linkedición (si existe), etc. son imperceptibles para el programador, lo cual permite tener una alto grado de productividad. Ejemplo de un ambiente dinámico es Smalltalk. Ejemplo de un ambiente no dinámico es VisualC++ o VisualBasic.

[18] Un ambiente de objetos permite manipular y trabajar con objetos reales y no con estructuras sintacticas que representan la definición de un objeto. En un ambiente de objetos, el sistema esta siempre corriendo y el programador puede inspeccionar y modificar fácilmente los objetos con los que está trabajando. El concepto de ambiente de objetos y ambiente de desarrollo dinámico estan fuertemente ligados puesto que un ambiente de objetos debe ser dinámico para permiter realizar modificaciones facilmente. Ejemplos de ambiente de objetos son Smalltalk y Self.

4.2 Técnicas Generales de llamadas polimórficas

4.2.1 Algoritmo básico del despacho de mensajes (DTS)

En lenguajes orientados a objetos la resolución de una llamada polimórfica es denominada “despacho de mensaje” o *message dispatch.*

El *message dispatch* es una función[19] que dado un nombre de mensaje (*selector*) y la clase del objeto al que se le envía el mensajes (*receiver*), busca la correcta implementación, denominada método (*method*).

El algoritmo básico utiliza una tabla por clase, donde se almacena en una estructura de búsqueda (generalmente un diccionario[20]) los pares *{selector,method}.* Cuando un objeto recibe un mensaje, el mismo le pide a su clase que le devuelva el método para dicho mensaje. Si dicho mensaje no es implementado en esa clase, se transmite la búsqueda a la superclase, y así sucesivamente hasta llegar a la raíz del árbol de jerarquía. En el caso que no se haya encontrado ninguna implementación para dicho mensaje en toda la jerarquía, se emite un error.

Para la jerarquía de clases de la Figura 14, las clases contendrían las estructuras de búsqueda especificadas en la Tabla 2.

Figura 14: Jerarquía para ejemplificar DTS

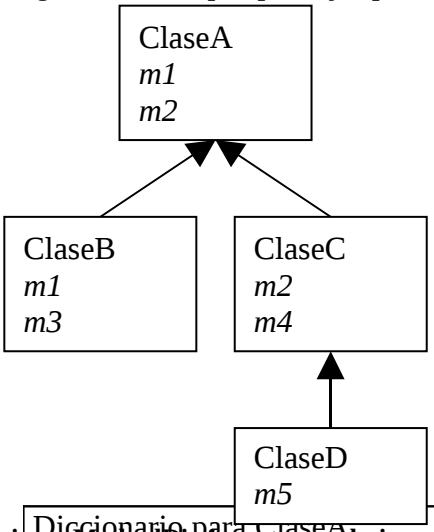


Tabla 2: Distintos MethodDictionary para la jerarquía de la Figura 14

Diccionario para ClaseA	
Selector	Método
m1	Dirección de m1
m2	Dirección de m2
Diccionario para ClaseB	
Selector	Método
m1	Dirección de m1 para B
m3	Dirección de m3
Diccionario para ClaseC	
Selector	Método
m2	Dirección de m2 para C
m4	Dirección de m4
Diccionario para ClaseD	
Selector	Método
m5	Dirección de m5

[19] Se está utilizando el termino función no para indicar una función de un lenguaje procedural, sino para indicar cierto código encargado de la resolución de un problema.

[20] En Smalltalk los diccionarios de métodos son generalmente una clase especializada que se denomina MessageDictionary

En Smalltalk, este algoritmo podría ser implementado según se muestra en la Figura 15 y la Figura 16.

```
Interpreter>>executeMethod: aSelector with: parameters for:
anObject
| method |

method := self findMethod: aSelector for: anObject.
method notNil ifTrue: [“ejecutar método…” ]
             ifFalse: [^self
executeMethod:#doesNotUndertand:
                              with: aSelector
                              for: anObject.].
```

Figura 15: Implementación del método Interpreter>>executeMethod

```
Interpreter>>findMethod: aSelector for: anObject
| class |

class := anObject class.
class notNil whileTrue: [
    method := class messageDictionary at: aSelector.
    method notNil ifTrue: [ ^method. ].
     class := class superclass. ].
^nil.
```

Figura 16: Implementación del método Interpreter>>findMethod

Los requerimientos de memoria para esta técnica son realmente mínimos. Cada clase sólo tiene las referencias a los métodos que implementa, por lo que el total de espacio ocupado por las estructuras de búsqueda será igual a la cantidad de métodos del sistema (multiplicado por supuesto por un factor que es la cantidad de bytes ocupado por entrada). Debido a esto, DTS es usado generalmente como estrategia de backup, y es invocado cuando un algoritmo de búsqueda más rápido falla, como veremos más adelante.

Los diccionarios de búsqueda utilizan distintas técnicas para acelerar la búsqueda. El más común es realizar hashing[21] sobre la clave, en este caso instancias de la clase Selector.

4.2.2 Tabla Indexada de Selectores (Selector Table Indexing, STI)

Se podría decir que indexar es una manera extrema de hashing, que privilegia velocidad sobre espacio. Suponiendo que utilizamos una arreglo para guardar referencias a objetos (en este caso a instancias de la clase Selector), la diferencia entre utilizar un índice

[21] En Smalltalk obtener un número de hashing de un objeto es algo trivial puesto que todos los objetos responde al mensaje hash. Es más, en Squeak, el número de hash es creado automáticamente cuando un objeto es creado, de esta manera no se incurre en tiempo de procesamiento adicional para calcularlo.

o un número de hashing para guardar/buscar un objeto en dicho arreglo, se basa en que un índice es un número único e irrepetible, mientras que un número de hashing puede se compartido por varios objetos.

Por lo tanto, si utilizo un índice por cada objeto, que cumple con el principio de que no puede ser repetido, puedo unívocamente encontrar dentro de un arreglo cual es la entrada correspondiente a un objeto. El problema con esta técnica es que se debe saber de antemano cual es la dimensión máxima que debe tener el arreglo. Instanciar este problema en un ambiente de objetos y un arreglo que almacene instancias de la clase Selector, implica saber cuantos selectores van a existir dentro de mi ambiente, algo imposible debido a la característica dinámica del mismo.

Si por el contrario utilizo un número de hashing, este me indica donde podría estar un objeto dentro de un arreglo, pero no lo asegura puesto que el mismo número de hashing es compartido por varios objetos. Debido a esta característica, es necesario realizar una comparación por igualdad para asegurar que se encontró el objeto requerido, y en caso negativo recorrer el arreglo hasta encontrarlo o determinar de alguna manera que no se encuentra en el arreglo (por ejemplo, encontrando una entrada vacía). Como se ve, la búsqueda es más lenta que utilizando un índice directamente.

La técnica STI es una técnica muy atractiva para implementar llamadas polimórficas debido a que el tiempo consumido por el algoritmo es constante, su ejecución posee una muy buena performance y el algoritmo es conceptualmente simple.

En un sistema de M clases y N métodos, STI tiene un array bidimensional de M*N entradas, donde cada intersección contiene la referencia al método para la clase y el selector correspondiente. En caso de que el método no exista para esa clase, contiene una referencia a la implementación del selector #doesNotUnderstand:.

El algoritmo de búsqueda por lo tanto se reduce a una simple operación de indexación en dicho array, y debido a que es tan simple no es necesario realizar una función separada sino que se puede embeber en el código llamador directamente (Ver [Vitek1], [Vitek2],[Vitek et al.1]).

Sin embargo esta técnica, por más rápida que sea, consume mucho espacio en memoria, y lo que es peor, un alto porcentaje de la misma se encuentra completamente vacía (en realidad con referencias al método #doesNotUnderstand:).

En un ambiente Smalltalk como Squeak, que posee 2018 clases y metaclases y está compuesto por 18.241 selectores, este array ocuparía un total de $2018*18.241*4^{22}$ = 143 MB, lo cual lo hace una solución inviable. La imagen actual de Squeak ronda los 5 MB. Para mayores datos sobre cuanto ocuparía dicha solución en otros Smalltalks, ver [Vitek2], [Vitek et al.1] y [Driesen3].

La dimensión que ocuparía esta solución es una explicación de por qué nunca fue utilizada.

[22] Se supone que cada referencia es de 32 bits, por ellos se multiplica la cantidad de clases y metodos por dicho coeficiente.

4.3 Técnicas dinámicas

La técnicas dinámicas se basan en “cachear” los resultados de la búsqueda realizada por algún otro algoritmo (en general DTS) para luego no tener que utilizar nuevamente tiempo de ejecución en la búsqueda de métodos ya ejecutados (o sea, de mensajes ya enviados).

Estas técnicas tienen dos características principales:

1. Se basan en estadísticas sacadas sobre la ejecución de programas típicos, por lo que su efectividad depende del comportamiento de los programas.

2. Los algoritmos difieren en cómo utilizan la cache, por ejemplo si poseen una cache global para todo el sistema, o una cache por cada lugar donde se envíe un mensaje.).

4.3.1 Cache Global de Búsqueda (Global Look-up Cache, GLC)

Las primeras implementaciones de Smalltalk se basaron en esta solución. La idea es tener una cache global en la cual se almacenan las tuplas *{clase, selector,método}*.

En el momento de enviar un mensaje, el algoritmo busca el método correspondiente en la cache global. En caso de no ser encontrado, se utiliza un algoritmo de backup (generalmente DTS) para buscarlo y luego agregarlo a la cache.

El algoritmo de búsqueda estaría compuesto por los métodos de la Figura 17, Figura 18 y Figura 19.

```
Interpreter>>findMethod: aSelector for: anObject
| class method |

class := anObject class.
method := self lookupCacheMethod: aSelector for: class.
method notNil ifTrue: [^method ].

class notNil whileTrue: [
    method := class messageDictionary find: aSelector.
    method notNil ifTrue: [
         self cacheMethod: method
              for: aSelector
              andClass: class.
         ^method. ].
     class := class superclass. ].
^nil.
```

Figura 17: Implementación Interpreter>>findMethod para el algoritmo GLC

```
Interpreter>>lookupCacheMethod: aSelector for: aClass
| hash index tupla|

“Se calcula un hash basado en algún algoritmo. En este caso
un Or exclusivo”
hash := aSelector hash xor: aClass hash.
“Se calcula la entrada del elemento para ese hash”
index := hash \\ (cache size).

tupla := cache at: index.
tupla selector=aSelector and:[ tupla class=aClass ] ifTrue:
[^tupla method.].
“De aquí en adelante se puede optar por seguir buscando
‘hacia abajo’ una cierta cantidad de veces o devolver que no
se encontró el método”
.
.
^nil.
```

Figura 18: Implementación del método Interpreter>>lookupCacheMethod

Debido a que GLC se base fuertemente en tener una cache global, este algoritmo debe tener las siguientes características:

- La búsqueda en la cache debe ser más rápida que la del algoritmo de backup.
- Se debe poder determinar rápidamente que no existe un elemento en la cache.

- El tiempo de mantenimiento de la cache debe ser mínimo.

```
Interpreter>>cacheMethod: aMethod for: aSelector andClass:
aClass
| hash index tupla|

“Se calcula un hash basado en algún algoritmo. En este caso
un Or exclusivo”
hash := aSelector hash xor: aClass hash.
“Se calcula la entrada del elemento para ese hash”
index := hash \\ (cache size).

tupla := cache at: index.
tupla selector: aSelector.
tupla class: aClass.
tupla method: aMethod.
“De aquí en adelante se puede optar por limpiar un poco la
cache. En este ejemplo no lo mostramos porque está fuera del
alcance de la investigación”.
```

Figura 19: Implementación de Interpreter>>cacheMethod

GLC empezó a ser utilizado luego de los estudios realizados en [Goldberg2] y [Krasner]. Estos estudios muestran (como lo pudimos comprobar nosotros en los resultado que se muestran más adelante), que se produce entre un 90% y 95% de cache hit sobre el total de mensajes enviados, aunque (como también se muestra más adelante) la disminución de los tiempos que se obtienen debido al alto grado de cache hit, se ve contrarrestado por el gran tiempo que insume el algoritmo de búsqueda de backup.

Squeak utiliza actualmente esta solución puesto que es la más sencilla de implementar y la que menos espacio consume. Sin embargo, esta solución es la menos adecuada de todas las técnicas dinámicas y como veremos más adelante tiene grandes falencias cuando el código es altamente polimórfico y el grado de cache hit es bajo.

Con respecto al comportamiento de la cache, Squeak opta por realizar tres pruebas dentro de la cache antes de devolver un cache miss. Esto lo realiza incrementando en uno el índice calculado en cada nueva búsqueda.

De la misma manera que verifica en tres entradas de la cache si se encuentra el método que se está buscando, cuando agrega una nueva tupla a la cache limpia las dos siguientes entradas. De esta manera el tiempo que se insume en el mantenimiento de cache es poco puesto que la misma se limpia constantemente cada vez que se agrega una nueva entrada.

Esta técnica de GLC es utilizada en Java para resolver el envío de mensajes a variables cuyo tipo esta definido por una interface y no por un objeto. Ver [Driesen2] y [32].

4.3.2 Cache en-sitio (Inline Cache, IC)

Inline Cache se basa en el hecho de que la clase del objeto receptor de un mensaje varía raramente en el punto en el que se envía dicho mensaje. Esto significa que si se envía el mensaje "*m1*" al objeto "*x*", la clase del objeto "*x*" será generalmente la misma. Los estudios realizados en [Ungar et al4],[Ungar] y [Deutsch et al.], muestran que el 90% de las veces, la clase del receiver se mantiene constante en cada envío de mensaje.

Inline Cache varía la ubicación de la cache con respecto a GLC, de global a local a cada envío de mensaje, por ello es que se denomina Inline Cache. A continuación de detalla como funciona.

Inicialmente la llamada dentro del método que se está ejecutando está apuntando a una función de búsqueda de backup (generalmente DTS), la cual devuelve el método a ejecutar. Una vez realizada la primer búsqueda el código es modificado para que apunte a la dirección del método encontrado. La próxima vez que se ejecute este envío de mensaje, el programa irá directamente a ejecutar el código del método encontrado con anterioridad.

El problema se produce cuando la clase del objeto receptor del mensaje cambia, puesto que esto implica que el valor de la cache dejó de ser válido. Para impedir que se ejecute un código indebido, todos los métodos tienen un prólogo que comparan que la clase del método que se está ejecutando coincida con la clase del objeto receptor del mensaje, que en caso positivo implica ejecutar el método. En caso negativo se llama a la rutina de búsqueda de backup y se actualiza el puntero que tenía el método llamador hacia el método llamado.

Mientras las clase del objeto receptor del mensaje no cambie, el costo de la llamada polimórfica se mantiene constante en todo momento y el único costo adicional en el que se incurre es en el del chequeo de la clase en el prólogo del método.

La eficacia de este método depende de la cantidad de hits que se produzcan dentro de la inline cache. En el peor de los casos, (100% miss) el overhead es igual al de realizar la búsqueda con el algoritmo de backup más el costo de mantener la cache.

Afortunadamente estudios realizados demuestran que el hit es bastante alto para lenguajes como Smalltalk y Self, los cuales tienen un 90% de hit. Ver [Ungar] y [Holzle et al2].

El código que implementaría este tipo de técnica se puede ver en la Figura 20.

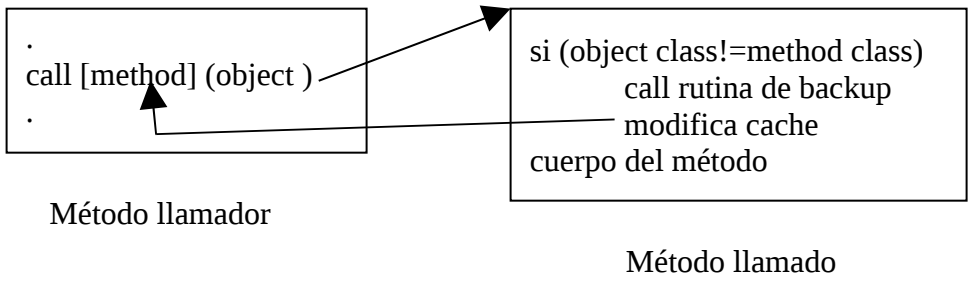


Figura 20: mplementación de IC

4.3.3 Cache Polimórfica en Sitio, (Polymorphic Inline Caching, PIC)

Polymorphic Inline Caches es una mejora realizada sobre Inline Cache. Esta técnica ha sido introducida a la comunidad por medio del proyecto Self [Holzle et al.].

El problema de IC que PIC trata de resolver se debe a que se ha comprobado que en algunos casos el sistema utiliza un 25% de su tiempo manejando cache misses [Holzle et al.], más para aquellos sistemas altamente polimórficos. Por lo tanto la idea es agrandar la cantidad de elementos que se cachean para mejorar el cache hit. El algoritmo funciona de la manera especificada en el párrafo siguiente.

Al principio PIC se comporta como IC. Inicialmente la llamada correspondiente al envío del mensaje apunta a una rutina de resguardo que será la encargada de buscar la dirección del método a ejecutar. Una vez encontrado el método, modifica dicha dirección por la del prólogo del método encontrado. Si la clase del receiver cambia, la nueva dirección del método no se sobreescribe sobre la anterior como hubiese hecho IC. Por el contrario, crea un "*stub*"[23] y hace que la llamada apunte ahora a dicho stub. El stub se encarga de buscar el método a ejecutar a partir de la clase del objeto receptor del mensaje y llamar al método correspondiente en caso de encontrarlo. Debido a que el stub discriminó la clase, el control de ejecución es transferido directamente al cuerpo del método y no a el prólogo del mismo.

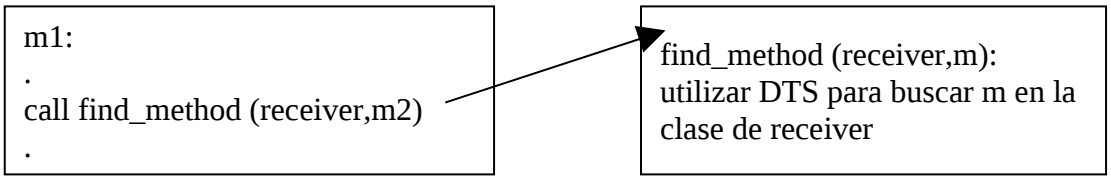
La dimensión de los stubs correspondiente a cada envío de mensaje puede crecer y adaptarse a medida que se envían mensajes a objetos instancias de distintas clases.

Ver la Figura 21 para ver la evolución del método llamador, el llamado y el stub a medida que se realizan llamadas

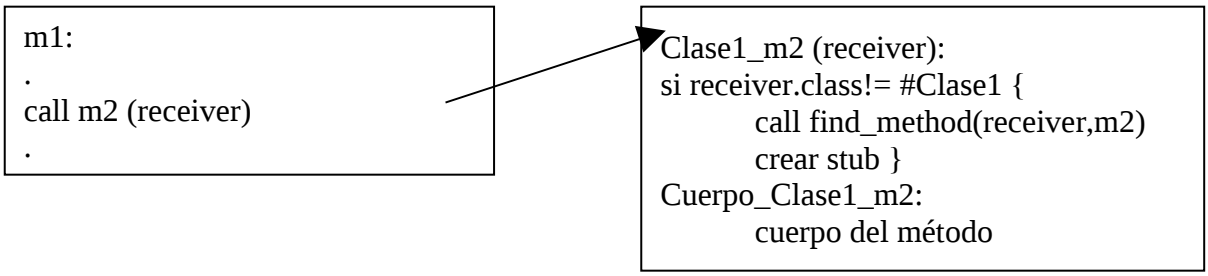
[23] Para este caso, el término "Stub" se utiliza para referenciar a una porción de código que se crea y modifica dinámicamente a medida que se envían mensajes a objetos instancias de clases distintas.

La performance de PIC baja drásticamente para aquellos lugares denominados "megamórficos" (*megamorphic*), donde existe una gran cantidad de clases candidatas para el objeto receiver. En dichos casos es mejor utilizar técnicas más conservadoras como IC.

Paso 1: La llamada todavía no se ejecutó.



Paso 2: La llamada se ejecutó y se trabaja como IC.



Paso 3: Se creo el stub debido ya que se modifico la clase del reciver.

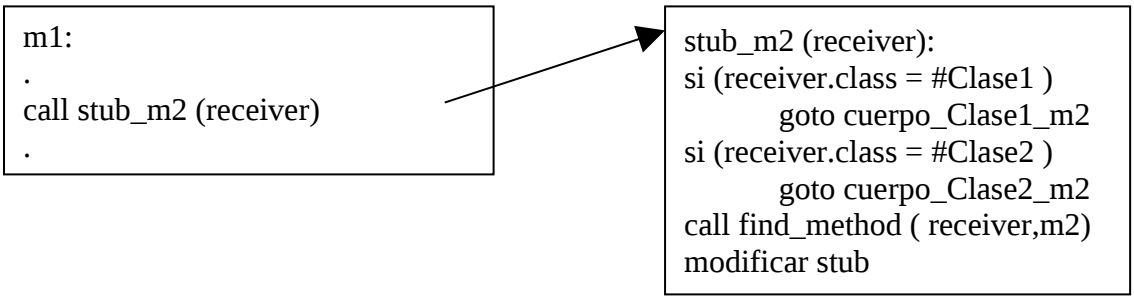


Figura 21: Evolución de la técnica PIC

4.3.4 Conclusiones sobre Técnicas Dinámicas

De las técnicas dinámicas, la que mejor performance tiene es PIC, sin embargo es la más complicada de implementar. IC tiene una performance aceptable pero no se comporta bien en lugares donde se envían mensajes a objetos que pueden ser instancias de distintas clases. Cabe destacar que ambas técnicas modifican código durante la ejecución del sistema.

GLC es la más fácil de implementar aunque los resultados de performance son los más bajos comparado con IC y PIC. Para más información ver [Vitek2] y [Driesen2].

Inline Caching es una mejora bastante significativa sobre DTS y GLC. Sus requerimientos de espacio son realmente razonables: 4 instrucciones para el método llamador y 3 instrucciones para el prólogo del método llamado [Vitek2].

Ungar realizó comparaciones de performance utilizando IC en el proyecto SOAR y obtuvo una mejora del 34% sobre GLC y un total de 74% sobre DTS [Ungar].

Un estudio realizado por Driesen, Holzle y Vitek sugieren que IC y PIC pueden llegar a mejorar la performance de la técnica VTBL (Virtual Tables) en arquitecturas de procesadores modernos, debido que a que en arquitecturas modernas, una llamada indirecta rompe el pipeline de comandos pre-procesados, mientras que un hit en la cache (o sea que la clase del receiver corresponda a la cacheada) implica un salto directo, lo cual no atasca (*stall*) el procesador [Driesen et al3].

A pesar que IC y PIC son muy buenas soluciones, tienen un punto flojo que puede llegar a producir un gran problema de performance. Imaginemos el caso que una clase con gran cantidad de subclases define un método y dicho método nunca es redefinido. En el caso de usar PIC, tendría el típico problema de llamadas megamórficas, y para el caso de IC se produciría un cache miss constantemente. Sin embargo, en ambos casos el método a ejecutar es siempre el mismo.

Una variación muy interesante que se puede lograr con PICs es la que actualmente se denominada compilación adaptiva y es presentada en [Holzle et al2]. La idea es que a partir de la información encontrada en las PICs, se modifiquen los stubs poniendo directamente el código del método a ejecutar, evitando así tener que realizar una llamada. Esto disminuiría notablemente la cantidad de llamadas a ejecutar y como consecuencia mejoraría la performance. En [Holzle et al2] se muestra que en Self la cantidad de llamadas utilizando esta técnica se reduce por un factor de 3.6 y la performance se mejora en un factor de 1.7.

Está técnica tiene asidero en que generalmente los métodos en programación orientada a objetos son relativamente cortos (3 a 6 líneas) y que hay una gran cantidad de métodos para acceder a variables de instancia (los métodos denominados accesor). Sin embargo esta técnica debe hacer un balance entre el tiempo total que llevaría modificar el stub (recompilarlo) y el tiempo que se estaría ganando por realizar un “inlining” del método a llamar. Esta técnica es utilizada actualmente en HotSpot™, la nueva virtual machine de Java implementada por Sun.

4.4 Técnicas estáticas

Las técnicas estáticas tienden a dar prioridad a la performance sobre la flexibilidad del sistema. Son técnicas que siguen la idea de STI pero que disminuyen la dimensión de la estructura utilizando métodos de compactación o información estática sobre los tipos.

Las técnicas estáticas pre-calculan las estructuras de búsqueda en tiempo de compilación y/o linkedición para disminuir el tiempo que se utilizaría en tiempo de ejecución para realizar la búsqueda de los métodos.

Generalmente el código de búsqueda obtiene la dirección del método a ejecutar a partir de la posición dentro de una tabla dado por el índice del método y realizando luego un salto indirecto a dicha posición. Una gran ventaja de estas técnicas es que el tiempo de búsqueda es usualmente constante y no depende de la "localidad" del sistema como IC o PIC.

4.4.1 Tabla de Funciones Virtuales (Virtual Function Tables, VTBL)

Las tablas virtuales fueron utilizadas por primera vez en Simula [Dahl et al.] y actualmente son el mecanismo por omisión utilizado en C++ y Java. Funciona únicamente con lenguajes tipados estáticamente.

A diferencia de STI, en VTBL las tablas de métodos son por clase y no globales. Para cada clase C, se determina la cantidad de métodos que posee desde la raíz del árbol de jerarquía, se crea un array de dicha dimensión y se numeran los métodos empezando por la clase raíz. Luego a cada objeto se le agrega un puntero a dicha tabla virtual y el código del envío del mensaje hace uso de la numeración de los métodos como índices dentro de la tabla.

La Figura 22 y su correspondiente Tabla 3 ejemplifican esta situación.

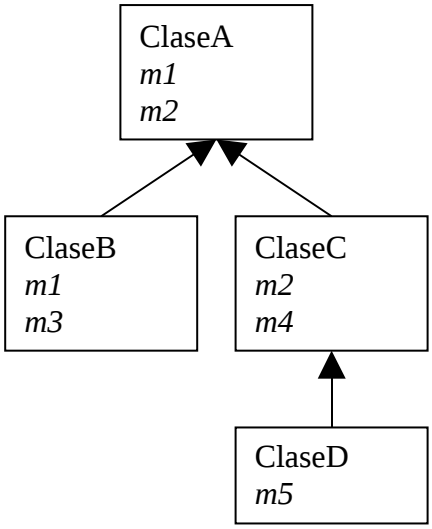


Figura 22: Jerarquía de ejemplo para VTBL

VTBL para ClaseA	
Selector	Indice
m1	0
m2	1
VTBL para ClaseB	
Selector	Indice
m1	0
m2	1
m3	2
VTBL para ClaseC	
Selector	Indice
m1	0
m2	1
m4	2
VTBL para ClaseD	
Selector	Indice
m1	0
m2	1
m4	2
m5	3

Tabla 3: Tabla resultado de la jerarquía de la Figura 22

El código para enviar un mensaje es bastante rápido y eficaz puesto que lo único que debe hacer es una indirección dentro de un tabla ya que el índice para el mensaje que se está enviando ya fue calculado en tiempo de compilación gracias a la información de los tipos provisto por el lenguaje, que es estáticamente tipado.

El código del envío del mensaje “m5” a un objeto instancia de la clase “ClaseD”, es traducida al pseudo lenguaje máquina de la Figura 23, el cual cuando es comparado con la Figura 3, se puede ver que consta de una instrucción adicional (la carga de la posición de la tabla) y una llamada indirecta en vez de una llamada directa.

```
.
Cargar DI con objeto.tabla
Call [DI+3]   ; El 3 se debe a que m5 tiene dicho índice.
.
```

Figura 23: Código máquina de una llamada por medio de VTBL

Esta técnica es muy difícil de superar a nivel performance dado su simplicidad y el poco código a ejecutar. Sin embargo tiene varias desventajas.

La principal es que solo puede ser utilizada en lenguajes con un sistema de tipos estático, con la consiguiente pérdida de flexibilidad. Otra desventaja es que genera más

información de la estrictamente necesaria. Esto se debe a que cada clase no solo contiene la dirección de los métodos que ella define sino también la de sus superclases. En un hipotética implementación de la jerarquía de Squeak con C++, se utilizaría un total de 812.229 entradas, lo que daría un total de 3127 KB (3,1 MB) en VTBL. (El código utilizado para dicho calculo está en el punto 11.11)

El espacio requerido por la VTBL no es bien tratado en algunos compiladores de C++, los cuales generaban un VTBL por cada clase que se utilizaba en cada módulo (archivo *.cpp), lo cual puede producir VTBL repetidas para las mismas clases. Esto incrementa el tamaño final del programa en dimensiones desproporcionadas [Stroustrup2]. Ejemplo de ello era el compilador Borland C++, el cual cuando trabajo con el sistema de memoria *large*[24] tiene esta desventaja.

Java no tiene este problema debido a que la definición de la clase y su implementación están dentro del mismo módulo, por lo que es en este módulo que se genera la VTBL. Ver [Gosling] y [Lindholm et al.].

Otra gran desventaja que tiene la VTBL es la de tener que re-numerar los métodos de las subclases cuando se modifica una superclase. Supongamos el ejemplo de Smalltalk en el cual todas las clases descienden de Object. En este caso si llegamos a agregar un método a la clase Object sería necesario recompilar el código de todo el sistema, un costo demasiado alto. Esto sucede en C++, donde hasta la modificación de definiciones privadas de una clase implica la recompilación de todas sus subclases. Esto rompe algunas de las reglas del paradigma de objetos, como *encapsulamiento*, la cual determina que la representación privada de un objeto no debe ser conocido ni afectar otro.

Java no posee este problema debido a que realiza linkedición dinámica cada vez que carga una clase en memoria.

4.4.2 Coloreo de Selectores (Selector Coloring, SC)

La técnica de coloreo de sectores se basa en STI y VTBL. En realidad empieza con STI y termina creando una estructura de dimensiones más pequeñas realizando un coloreo de selectores.

Esta técnica fue propuesta inicialmente por Dixon [Dixon et al.] y luego aplicada a Smalltalk por André y Royer en [Andre et al.]. La diferencia con STI y VTBL se basa en que en vez de utilizar un índice dentro de una tabla, se utiliza un color de selector. Dicho color es un número único en cada clase donde el selector es conocido. Dos selectores pueden utilizar el mismo color si no aparecen al mismo tiempo en una misma clase.

El Coloreo de Selectores permite tener una mayor compresión que STI pero no tanto como VTBL.

[24] En C++ se puede definir con el tipo de memoria que se desea trabaja. En particular, Borland C++ define los modelos Tiny, Small, Large y Huge. Las diferencias que existen entre estos modelos están fuera del alcance de esta tesis. Para más información ver el manual del Compilador C++ utilizado.

El algoritmo óptimo para colorear selectores es un problema NP-Completo, en el que asignar colores a los selectores equivale a colorear un grafo cuyos nodos representan a los selectores, y dos nodos están conectados por un arco si los dos selectores correspondientes co-ocurren en cualquier clase. Debido a que el coloreo es un problema NP-Completo, distintas heurísticas deben ser utilizadas para hacer del mismo un problema resoluble en tiempo aceptable.

Aunque la dimensión de la tabla de búsqueda obtenida por SC es mucho menor a STI, está contiene varias entradas vacías, las cuales ocupan en realidad el 43% del espacio total para Smalltalk. (Ver [Vitek2] y [Driesen2].)

Como todas las técnicas que benefician espacio como principal restricción, deben utilizar mayor tiempo de ejecución para verificar que la llamada sea correcta. En el caso de selectores que no comparten el mismo color, el código de llamada es exactamente igual al de STI. Sin embargo para aquellos selectores que comparten el mismo color, un chequeo adicional debe ser hecho para estar seguro de que se está ejecutando el método correcto.

El problema mencionado en el párrafo anterior se resuelve numerando todos los selectores de manera unívoca (además del color) y agregando un prólogo a cada método que verificará la equivalencia del número que le fue asignado con el que supuestamente se obtuvo por medio de la tabla generada con SC.

Para el ejemplo de la Figura 22, la tabla creada por STI (Tabla 4) quedaría con SC según muestra la Tabla 5.

	m1	m2	m3	m4	m5
ClaseA	0	1			
ClaseB	2	1	3		
ClaseC	0	4		5	
ClaseD	0	4		5	6

Tabla 4: Array STI para la jerarquía de la Figura 22

	m1	m2	m3 m5	m4
Color	***0***	***1***	***2***	***3***
ClaseA	0	1		
ClaseB	2	1	3	
ClaseC	0	4		5
ClaseD	0	4	6	5

Tabla 5: Tabla resultado al utilizar SC sobre la Tabla 4

Como se puede ver, los colores de m3 y m5 son los mismos (2 en este caso), por lo que el mensaje m5 debe ser desechado cuando se envía a objetos que son instancias de ClaseB, y el mensaje m3 enviado a instancias de ClaseD también debe ser rechazado. Esto se hace por medio del prólogo de los métodos, el cual compara el número asignado al selector que se está enviando con el número del selector que el método llamado implementa.

```
objeto1 := ClaseB new.
objeto1 m5.    "← Debe generar error"
```

Figura 24: Llamada inválida para SC

Para el ejemplo de la Figura 24, el mensaje m5 enviado a objeto1 debe producir un error debido a que el mismo no es válido para instancias de ClaseB, sin embargo un método es encontrado en la tabla producida por SC debido a que m5 y m3 comparten el mismo color. Justamente el método encontrado es m3, el cual es llamado, pero en su prólogo rechaza la llamada puesto que el número asignado a m3 no es igual al de m5.

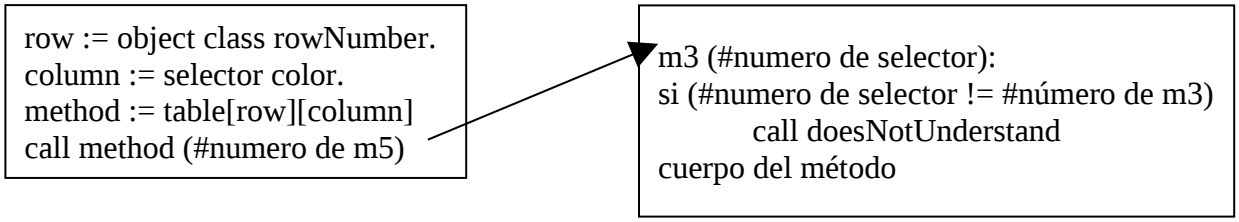


Figura 25: Rechazo de llamada en SC

La ventaja de esta técnica es que puede ser utilizada en lenguajes dinámicamente tipados como Smalltalk y Self, debido a que los números de los colores son de acceso global dentro del sistema, sin embargo tiene una gran desventaja: el tiempo necesario para computar la tabla de búsqueda.

En lenguajes como Smalltalk donde la modificación y creación de métodos es constante, habría que recalcular la tabla constantemente, lo cual haría de la programación una tarea demasiado lenta y ardua, sacándole la característica dinámica y flexible que Smalltalk provee. Debido a estas desventajas, este método no es utilizado en aplicaciones comerciales.

4.4.3 Desplazamiento de fila (Row Displacement, RD)

Row Displacement es otra técnica que trata de compactar el array producido por STI. En este caso lo que se hace es generar un array unidimensional en el cual se solapan las entradas libres para lograr una mejor compresión.

La idea es realizar una superposición del array de dos dimensiones de STI por fila de tal manera que las entradas utilizadas se superpongan sólo con aquellas que están vacías. Los offset de las filas deben ser únicos debido a que son usados luego como índices dentro del array unidimencional generado. El objetivo del algoritmo es minimizar el tamaño de array producido por STI sacando la mayor cantidad de entradas vacías posibles. Este problema es similar a minimizar tablas de parseo para parsers dirigidos por tabla (table-driven parsers). Ver [Denckery et al.].

Este problema puede ser resuelto realizando el desplazamiento sobre las clases o los selectores. Driesen, en su tesis de doctorado, demuestra que utilizando un desplazamiento sobre las clases el array resultante queda con un 33% de entradas libres[Driesen2]. Sin embargo, si el desplazamiento se realiza sobre los selectores, la cantidad de entradas libres es reducida a un 0.4%, luego de utilizar un sistema de numeración de clases apropiado. (Ver [Driesen2], Sección 4 para una explicación exhaustiva de la técnica utilizada)

Debido que el array final obtenido con RD es similar a SC, se necesita utilizar el mismo tipo de prólogo en los método que para SC.

De la tabla STI generada para la jerarquía de la Figura 22 (Tabla 6), se obtendría el array que se ve en la Tabla 7.

	ClaseA	ClaseB	ClaseC	ClaseD
m1	0	2	0	0
m2	1	1	4	4
m3		3		
m4			5	5
m5				6

Tabla 6: Tabla a utilizar para RD de la jerarquía Figura 22

m1				m2				m3	m4		m5
0	2	0	0	1	1	4	4	3	5	5	6

Tabla 7: Vector producido por RD a partir de la Tabla 6

4.4.4 Tablas compactadas de índices de selectores (Compact Selector-Indexed Dispatch Table, CT)

Este método fue propuesto por Vitek y Horspool [Vitek et al.1]. Se basa también en compactar la tabla generada por STI, con la diferencia que genera código de búsqueda según el tipo de selector.

Los selectores son divididos es dos conjuntos: Selectores Estándar y Selectores Conflictivos. Los selectores estándar son aquellos que son definidos en una clase y pueden llegar o no a estar redefinidos en alguna de las subclases de la clase que lo definió. Los selectores conflictivos son aquellos que tienen múltiples definiciones en clases que no poseen ninguna relación entre si.

Dos tablas de búsqueda son generadas. Una para selectores estándares y otra para selectores conflictivos.

Una vez determinados los selectores estándares y los conflictivos, los primeros son numerados utilizando un algoritmo de depth-first sobre la jerarquía de clases. Dos selectores estándares utilizan el mismo número siempre y cuando estén en diferentes ramas de la jerarquía. Los selectores conflictivos no pueden utilizar esta técnica de numeración, por lo que se crea una tabla especial para ellos, la cual queda indefectiblemente con entradas vacías.

Luego de realizado el paso anterior, se compactan las tablas compartiendo columnas tratando de evitar superposición en sus celdas. La cantidad de columnas a compartir influenciará el espacio final de la estructura de búsqueda como así también la performance. Para aquellas columnas que colicionen, se generan stubs similares a los de la técnica PIC, los cuales hacen una selección según el tipo de la clase.

Una vez lograda la máxima compresión deseada de las tablas, se genera un vector poniendo en secuencia las filas de las tablas comprimidas en los pasos anteriores.

Debido a la compresión realizada, se debe utilizar un prólogo similar a SC en los métodos para rechazar la ejecución de métodos que no correspondan.

A continuación se presenta un ejemplo que permite ver el funcionamiento del algoritmo. Dicho ejemplo esta sacado de [Driesen2] y se basa en la jerarquía de clases presentada en la Figura 26. La Figura 27 muestra el array final obtenido.

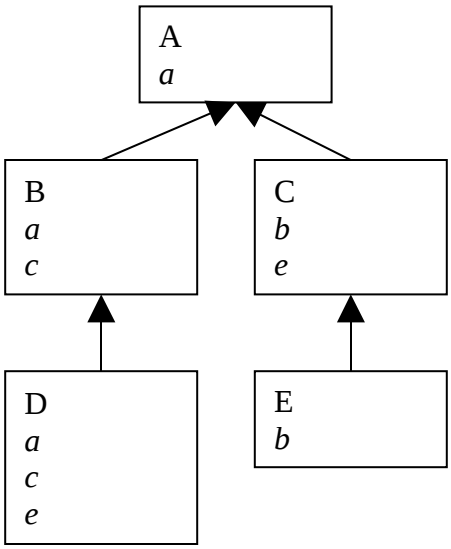


Figura 26: Jerarquía para CT

El paso (a) es el resultado de haber numerado los selectores y haber detectados cuales son conflictivos y cuales no. Como podemos observar se genera una tabla para cada uno de estos tipos de selectores.

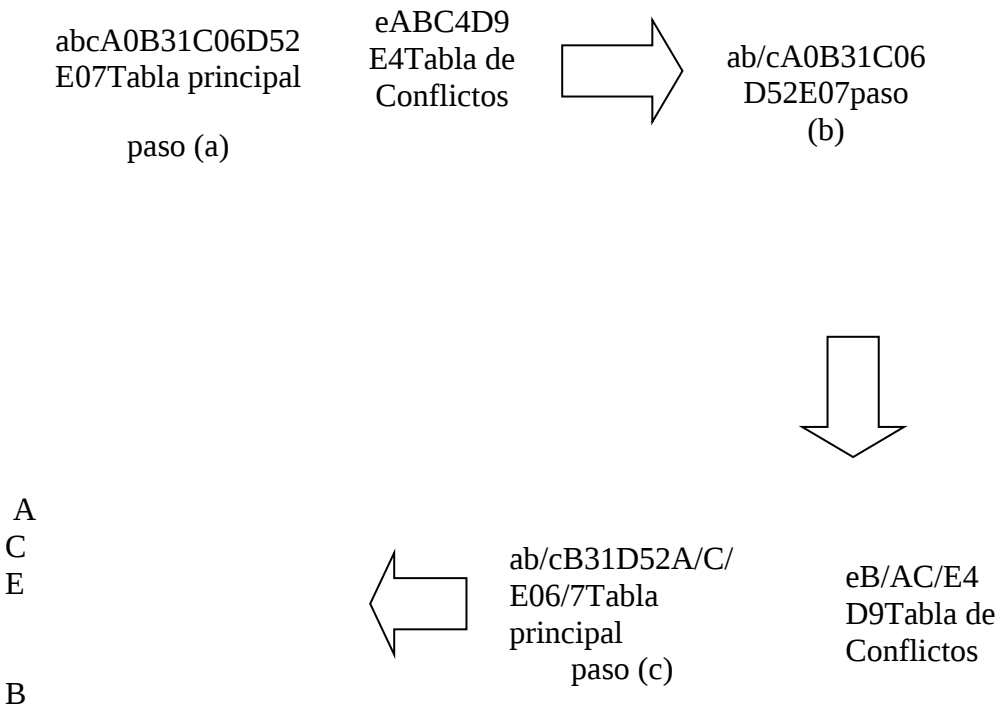


Figura 27: Pasos del algoritmo de CT

El paso (b) produce el solapamiento de los selectores c y b dentro del array, debido a que no hay entradas que se superpongan.

El paso (c) compacta la tabla de selectores conflictivos, compartiendo las entradas para las clases A,B y C,E. En la tabla principal de selectores estándares, comparte las entradas de las clases A,C y E.

Por último, en el paso (d) la tabla de selectores estándares y la tabla de selectores conflictivos son pasadas a un vector, el cual contiene las direcciones de los métodos a partir de su selector y la clase.

Este algoritmo realiza una gran compresión sobre la tabla inicial generada por STI, sin embargo genera mucho código para los prólogos de los métodos y los stubs para aquellas entradas donde los selectores se solaparon (ej. selectores b/c para las tablas A/C/E). Vitek muestra en [Vitek et al.1] que para un Smalltalk como ObjectWorks, el total de código agregado por utilizar este tipo de algoritmo sería de 1MB.

Debido al código que debe ser agregado, la cantidad de espacio final utilizado por esta técnica es mucho más de lo generado por RD y también más lento, debido a que el algoritmo de búsqueda debe tener en cuenta a selectores especiales.

En [Vitek2] y [Vitek et al.1] Vitek reconoce estos errores, los cuales fueron introducidos en el algoritmo presentado en [Vitek et al.2] y argumenta los motivos de la falla, presentando un refinamiento de la técnica que trata de resolver los problemas presentados.

Más allá de la gran compresión lograda sobre la estructura de búsqueda, el tiempo de generación utilizado por este algoritmo para llegar a los resultados finales sobrepasan los máximos aceptables para un ambiente de desarrollo dinámico como Smalltalk [Vitek et al.1].

4.4.5 Conclusiones sobre Técnicas Estáticas

Las técnicas estáticas, como pudimos observar, tienden a producir un despacho de mensajes mucho más rápido que las técnicas dinámicas. También el tiempo utilizado para la búsqueda es más fácil de determinar (tiene una cota superior) puesto que no depende del comportamiento del programa como en GLC o de la localidad del mismo como IC y PIC.

Sin embargo el tiempo que utilizan para generar las estructuras de datos que necesitan para funcionar es mucho mayor al de las téncias dinámicas. Esto, como ya mencionamos es una gran desventaja para ambientes de desarrollo como Smalltalk, donde se prioriza el dinamismo y la simplicidad sobre cualquier otro factor. Por lo tanto, no existe ambiente de desarrollo de Smalltalk que utilice algunas de las técnicas estáticas aquí mencionadas.

El mayor inconveniente que poseen estas técnicas son el tiempo de compilación necesario y la falta de flexibilidad y adaptabilidad al comportamiento del sistema.

De todas estas técnicas, la única que es utilizada a nivel comercial es la VTBL con lenguajes como C++ y Java. Pero como mencionamos en su momento, VTBL requiere que el lenguaje sea tipado estáticamente, lo cual va en contra del principio de flexibilidad de Smalltalk.

5. Framework de Despacho de Mensajes

Este capítulo muestra la implementación de un framework de caja negra que ataca la problemática del despacho de mensajes utilizando Squeak como ambiente de implementación.

Esta implementación puede ser categorizada como framework según la definición dada en [GOF] porque:

- Aborda un dominio de problema único, es este caso el despacho de mensajes.
- Dicta una arquitectura de software para las implementaciones de despacho de mensajes
- Realiza inversión de control

Este framework cumple con los siguientes objetivos:

- El algoritmo de despacho debe ser ampliamente configurable. Esto implica:
  - Poder modificar el algoritmo utilizado dinámicamente
  - Poder determinar el algoritmo a nivel sistema
  - Poder determinar el algoritmo a nivel clase
  - Si el algoritmo utiliza caching, poder configurar el algoritmo de caching
- Debe permitir correr distintos tests sobre los algoritmos
- Debe permitir realizar un profiling sobre los resultados

A continuación se muestra como está separado el framework en distintos paquetes conceptuales, luego se hace una introducción al diseño e implementación del interprete de Squeak y por último se explica en detalle la implementación del framework.

El código completo de este framework se puede ver en el Apéndice B.

5.1 Diseño de Alto Nivel

El diseño está dividido en distintos paquetes conceptuales que poseen un objetivo bien definido y están compuestos por un conjunto de clases. Los paquetes son:

- Squeak Kernel-Objects
- Squeak Interpreter
- Thesis Interpreter
- Method Lookup Strategies
- Caching Strategies
- Method Lookup Profiling
- Caching Profiling
- Profiling Support
- Benchmarks

Estos paquetes poseen ciertas dependencias que se pueden observar en Figura 28. A continuación se detalla cada uno de los paquetes[25].

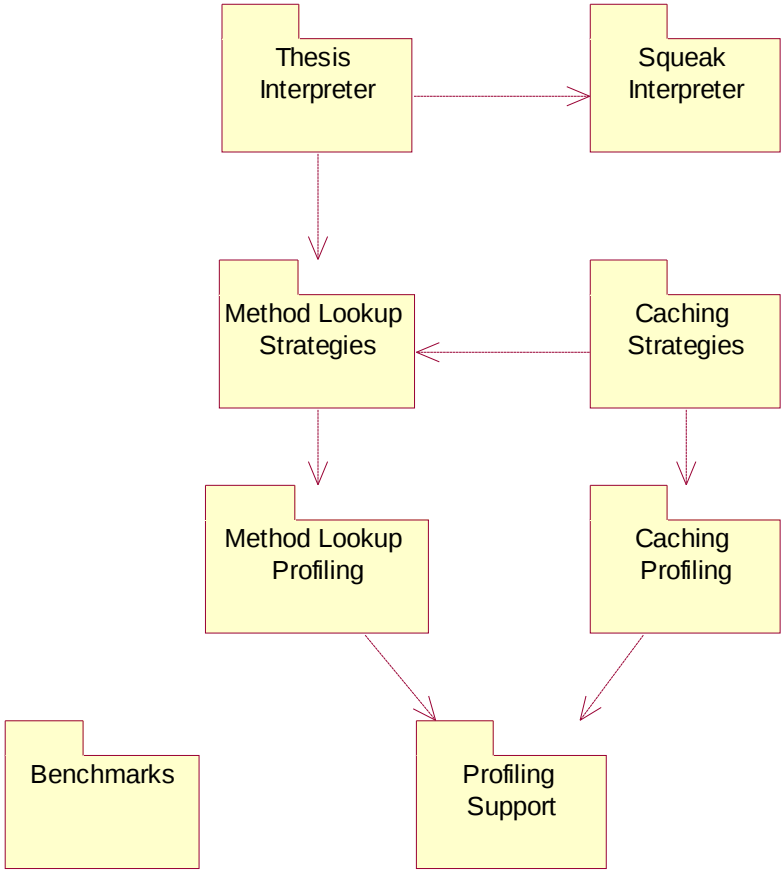


Figura 28: Paquetes del Framework y sus dependencias

5.2 Squeak Kernel-Objects

Este paquete contiene la clase Object (entre otras) que es la clase raíz de la jerarquía de herencia en Smalltalk, por lo que todos los paquetes dependen de Squeak Kernel-Objects.

No es la intención de este trabajo detallar el contenido de este paquete ni la responsabilidad de sus clases puesto que esto es conocimiento general de Smalltalk.

5.3 Squeak Interpreter

Este paquete contiene clases que cumplen un doble objetivo.

- Definir la Virtual Machine de Squeak
- Permitir correr una Virtual Machine adentro de Squeak

[25] Ver capítulo 11 para una breve reseña de la notación gráfica utilizada

Como se explica en [Ingalls et al] la VM de Squeak es generada a partir de código Smalltalk, que luego de un proceso de parsing genera código C, el cual es compilado en distintos Sistemas Operativos para obtener una VM particular por plataforma. Las clases que componen este paquete son aquellas que sufren dicho proceso de parsing.

Pero también como dijimos anteriormente, estas clases tienen la responsabilidad de poder ser ejecutadas dentro de Squeak de tal manera que simulen un VM real, objetivo que es logrado y utilizado en esta tesis para poder realizar las pruebas necesarias.

Las principales clases que componen este paquete son:

- **ObjectMemory**: Esta clase se responsabiliza del manejo de la memoria, como los distintos tipos de header de cada objeto, realizar la recolección de basura (garbage collection), alocación de espacio para nuevos objetos, etc.

- **Interpreter**: Esta clase representa la virtual machine en si. En la misma se encuentran definidos los bytecodes que la VM interpreta, el comportamiento de cada bytecode, las primitivas de la VM, como así también el algoritmo de message lookup. Esta clase es pasada luego por un generador de código C para crear la Virtual Machine real.

- **InterpreterSimulator**: Esta clase implementa las primitivas y métodos necesarios para que se pueda ejecutar el interpreter adentro de Squeak (o sea, una VM corriendo en Smalltalk, adentro de otra ejecutando código generado a partir del lenguaje C)

- **InterpreterSimulatorLSB**: Tiene las modificaciones necesarias de algunos métodos de ObjectMemory para trabajar con máquinas que representan las palabras de máquina de manera LSB (Ej. procesadores Intel x86).

Hay una par de conclusiones interesantes que se pueden sacar luego de utilizar y analizar este paquete. Una de ellas es que la jerarquía de clases no refleja mucho la realidad de la relación entre un interprete y un manejador de memoria. Según esta implementación, un interprete "**es un**" manejador de memoria debido a la relación de herencia que existe entre las clases Interpreter y ObjectMemory (Ver Figura 29). Un mejor diseño sería que el interprete colabore con un manejador de memoria, de esta manera se podría desacoplar fácilmente la ejecución de un VM con la administración del recurso de la memoria.

La pregunta es entonces, ¿por qué no se hizo así?, y la respuesta es que se eligió este tipo de implementación para permitir una traducción más rápida a código C. La realidad es que la implementación de estas clases tienen un estilo más procedural que "orientado a objetos", y esto se debe principalmente al objetivo final de tener una implementación que ejecute rápido a partir de código C.

Un ejemplo que demuestra este tipo de implementación es la resolución del algoritmo de despacho de mensaje. Uno podría suponer que debería haber un conjunto de objetos que se encarguen de este problema, sin embargo dicho algoritmo esta resuelto en un

par de métodos de las clase Interpreter de tal manera que la única solución para modificar dicho algoritmos es subclasificar o, por supuesto, modificar el código. Nosotros elegimos la primera, y en la sección siguiente se describe cómo.

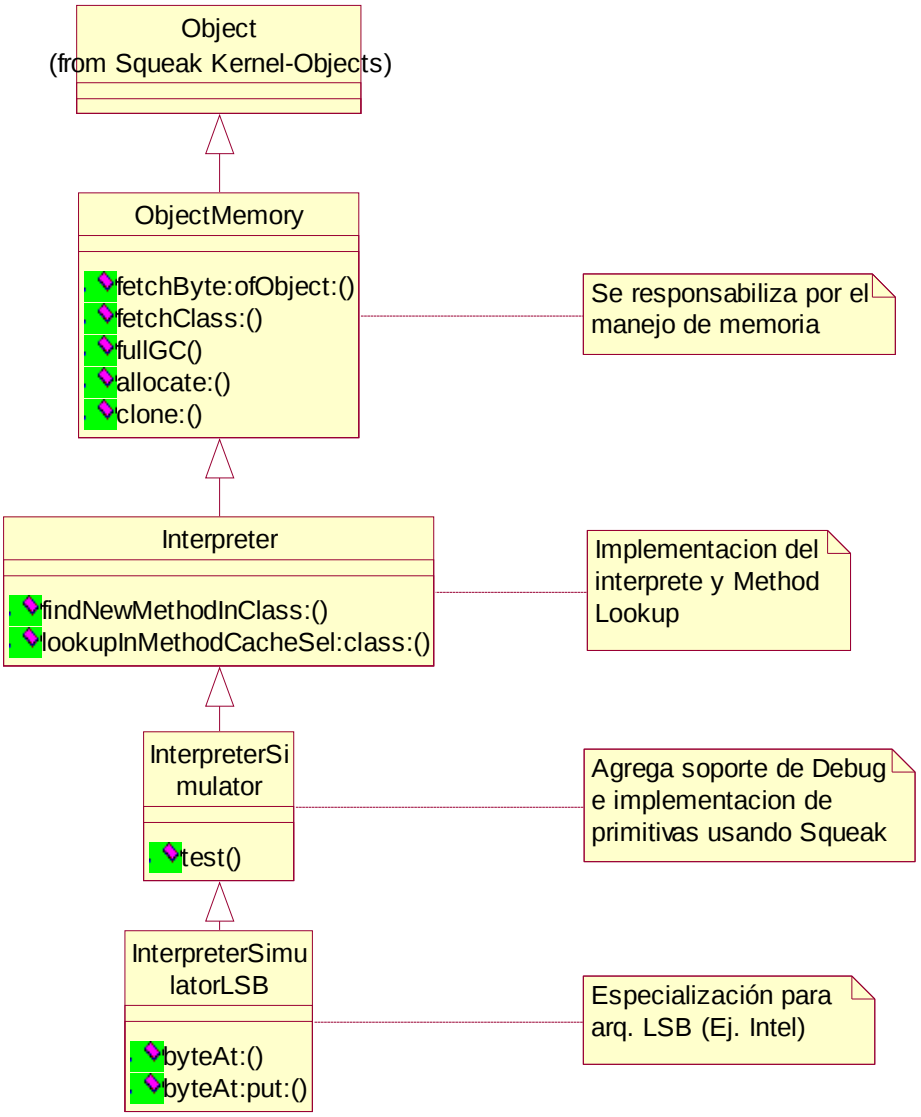


Figura 29: Diagrama de Clases del Paquete Squeak Interpreter

5.4 Thesis Interpreter

Este paquete es una especialización que se realizó de las clases del paquete Squeak Interpreter para lograr los siguientes objetivos:

- Desacoplar el interprete de bytecodes del algoritmo de message lookup

- Permitir tener más de un algoritmo de message lookup simultáneamente en un mismo interpreter
- Poder sacar estadísticas fácilmente del algoritmo de message lookup que se está utilizando

Para cumplir estos objetivos se utilizó un diseño basado en el “Strategy Pattern” (Ver [GOF]), de tal manera que el intérprete delega la búsqueda de un mensaje en la implementación de dicha estrategia. Al mismo tiempo, se delega en otro objeto la implementación de la recolección de datos para la generación de estadísticas.

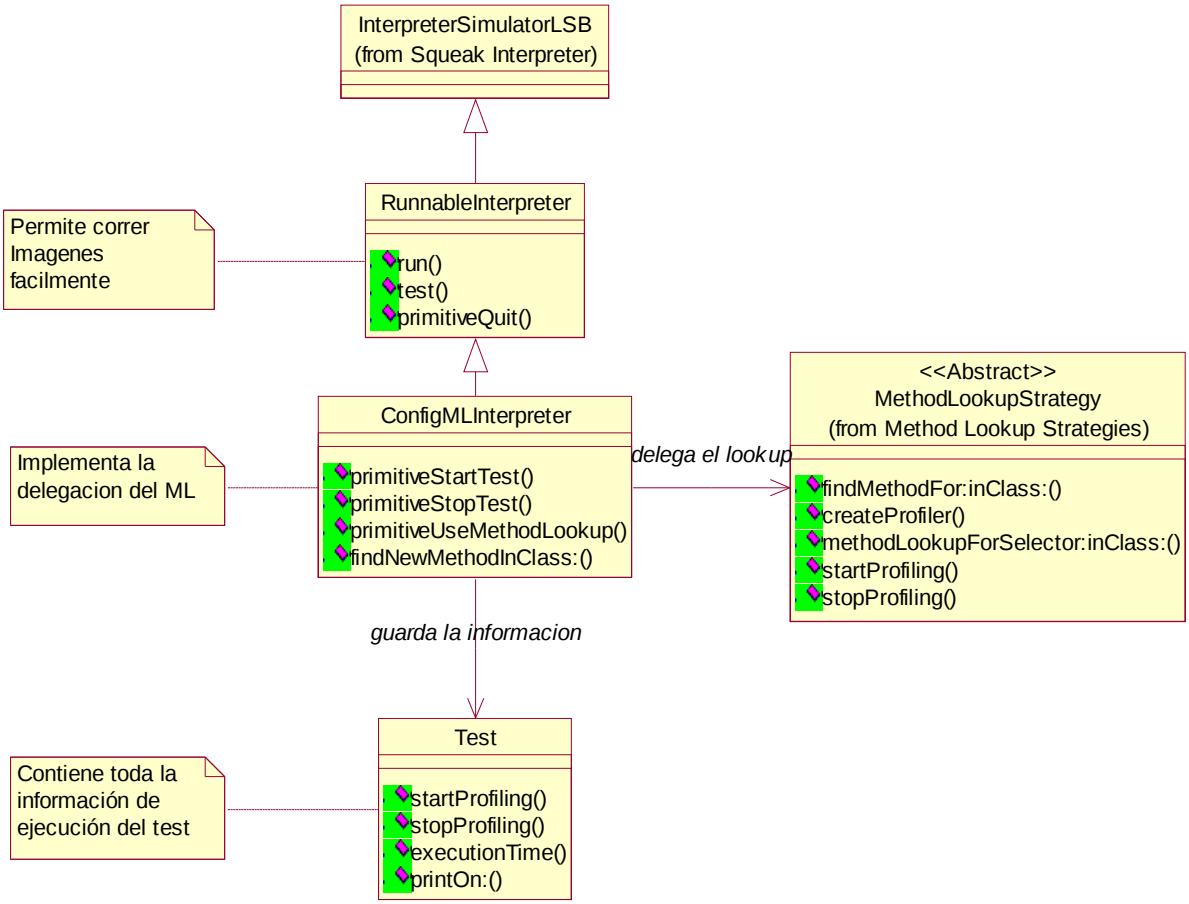


Figura 30: Diagrama de Clases del paquete Thesis Interpreter

En la Figura 30 se puede ver la jerarquía de clases de este paquete. A continuación se detalla la responsabilidad de cada clase que lo compone:

- **RunnableInterpreter**: Subclasifica InterpreterSimulatorLSB debido a que las pruebas fueron realizadas en una máquina WinTel. La responsabilidad de esta clase es permitir ejecutar fácilmente el interprete de Squeak, poder terminar dicha ejecución, etc.

- **ConfigMLInterpreter**: Esta clase es la encargada de sobre-escribir los métodos de la clase Interpreter en lo que respecta al algoritmo de message lookup, delegando en **MethodLookupStrategy** la ejecución de dicho algoritmo. Al mismo tiempo posee primitivas que permiten que la imagen que es ejecutada con este intérprete pueda configurar qué algoritmo de message lookup quiere utilizar, cuándo se comienza un test y cuándo se termina un test.

- **Test**: Para poder sacar conclusiones sobre los distintos algoritmos de message lookup que se implementaron, se corrió una secuencia determinada de tests. Esta clase abstrae dicho concepto y permite mantener la información correspondiente a la ejecución de cada test, como el tiempo total de ejecución, el nombre del test, etc.

5.5 Method Lookup Strategies

En este paquete se encuentran las clases que modelan el problema de realizar la búsqueda de un método a partir de un mensaje. Dichas clases son **MethodLookupStrategy** (clase abstracta que modela el problema) y **BasicMethodLookupStrategy** (implementación básica de una solución)

Al mismo tiempo contiene distintas implementaciones del algoritmo de message lookup, como por ejemplo la implementación de Dispatch Table Search (DTS - Ver 4.2.1).

Las principales clases que lo componen son:

- **MethodLookupStrategy**: Es la clase abstracta que define cuál debe ser el comportamiento del algoritmo de method lookup. El mensaje principal es *findMethodForSelector: aSelector inClass: aClass* que le envía *ConfigMLInterpreter* cada vez que necesita encontrar el método de un mensaje. Este mensaje es luego implementado de manera especial por cada una de sus subclases.

- **BasicMethodLookupStrategy**: Esta clase posee comportamiento común a todos los objetos que implementen el algoritmo de method lookup, como colaborar con el interpreter, colaborar con el profiler, etc. Implementa el mensaje *findMethodForSelector: aSelector inClass: aClass* utilizando el template method pattern (Ver [GOF]). Las subclases deben implementar el método *lookupMethodForSelector: aSelector inClass: aClass* donde debe estar el código correspondiente al algoritmo en sí.

En este paquete también se encuentran las implementaciones de algunos algoritmos de method lookup, como:

- **DistpatchTableSearchML**: Esta clase implementa el algoritmo básico de method lookup denominado **DTS**. La implementación de este algoritmo está detallada en 6.2.

- **HierarchyBranchML**: Esta clase implementa un algoritmo de búsqueda basado en numerar las clases y asociar conjuntos de clases con un mensaje dado. Para más detalle ver 6.4

En Figura 31 se puede ver el diagrama de clases de este paquete.

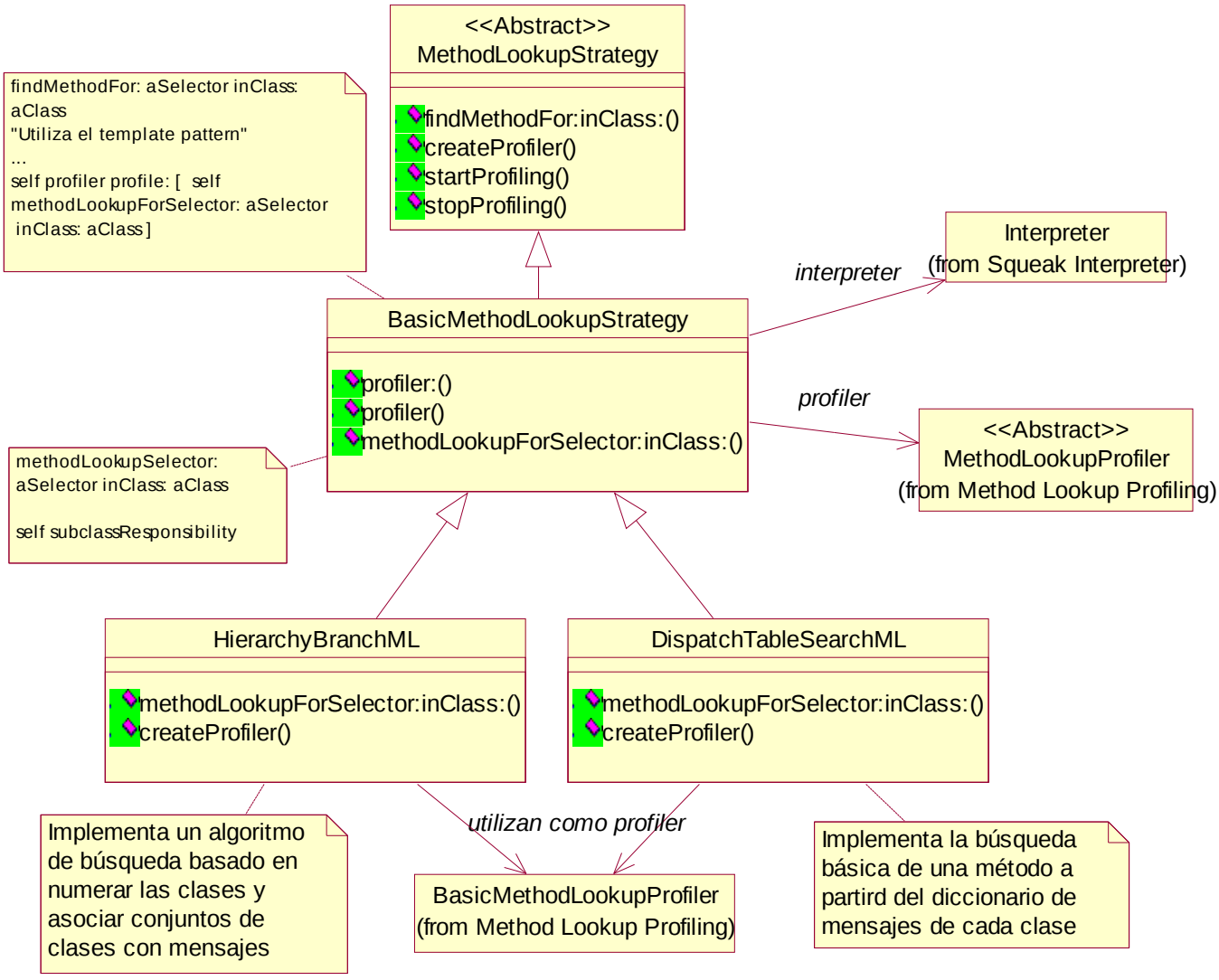


Figura 31: Diagrama de Clases del paquete Method Lookup Strategies

Como se puede observar, cada estrategia de method lookup tiene relacionado un profiler. De esta manera, cada vez que el algoritmo desea realizar alguna acción, debe indicárselo al profiler para que este pueda medir el tiempo que tarda en realizar la tarea.

Esta interacción se puede observar en la implementación del mensaje *findMethodFor: aSelector inClass: aClass*, el cual le indica al profiler que tome el tiempo de la ejecución del mensaje *methodLookupSelector: aSelector inClass: aClass*. Es este

último mensaje el que todos las estrategias de lookup deben implementar y corresponde al algoritmo de búsqueda en sí.

A continuación se presenta el diagrama de interacciones que muestra cómo colaboran estos objetos cuando se está utilizando el algoritmo de **DTS**.

5.5.1 Interacción para el Algoritmo DTS

Como se puede ver en la Figura 32 la interacción es muy sencilla. Cuando el Interprete recibe el mensaje *findNewMethodInClass: aClass* delega la búsqueda en la implementación del algoritmo de **DTS**, en este caso la clase **DispatchTableSearchML**.

Este a su vez le informa al profiler que se está realizando la búsqueda de un método en una clase para que el profiler almacene dicha información de ser necesario y luego le pide al profiler que mida el tiempo en ejecutar el algoritmo enviándole el mensaje *profileMethodLookup: aBlock.*

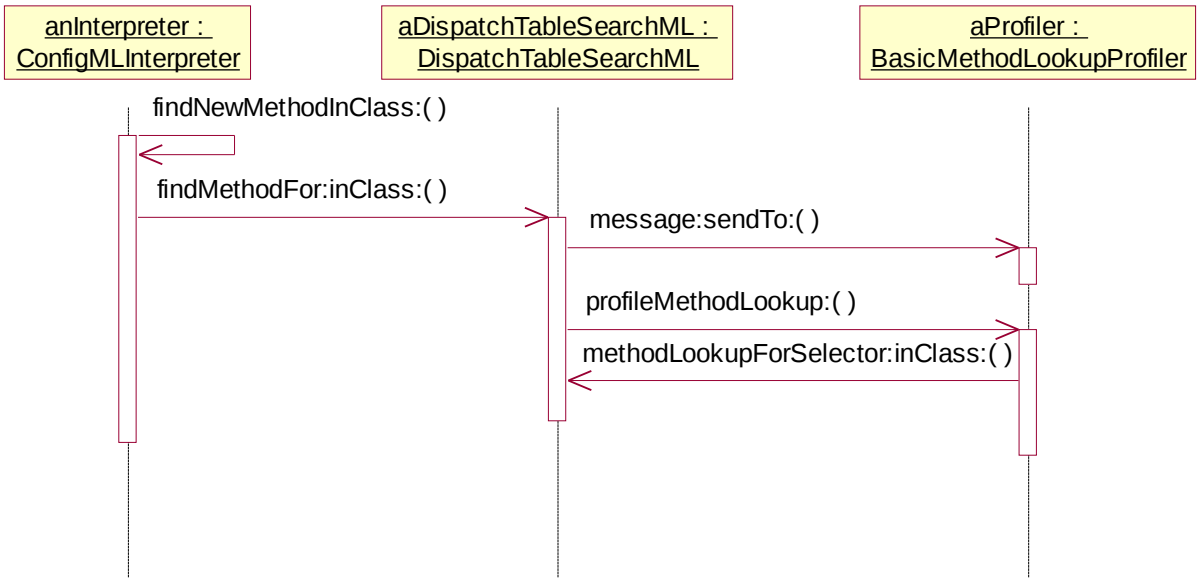


Figura 32: Diagrama de Interacciones del Algoritmo DTS

El bloque en realidad contiene un callback a la clase **DispatchTableSearchML**, puesto que envía el mensaje *methodLookupForSelector: aSelector inClass: aClass*, que es el método que realmente realiza la búsqueda.

No se muestra el diagrama de interacciones del mensaje *methodLookupForSelector: aSelector inClass: aClass* puesto que es la implementación del algoritmo **DTS** mencionado en capítulos anteriores.

5.6 Method Lookup Profiling

Este paquete contiene las clases que se encargan de realizar el profiling de las distintas estrategias de method lookup.

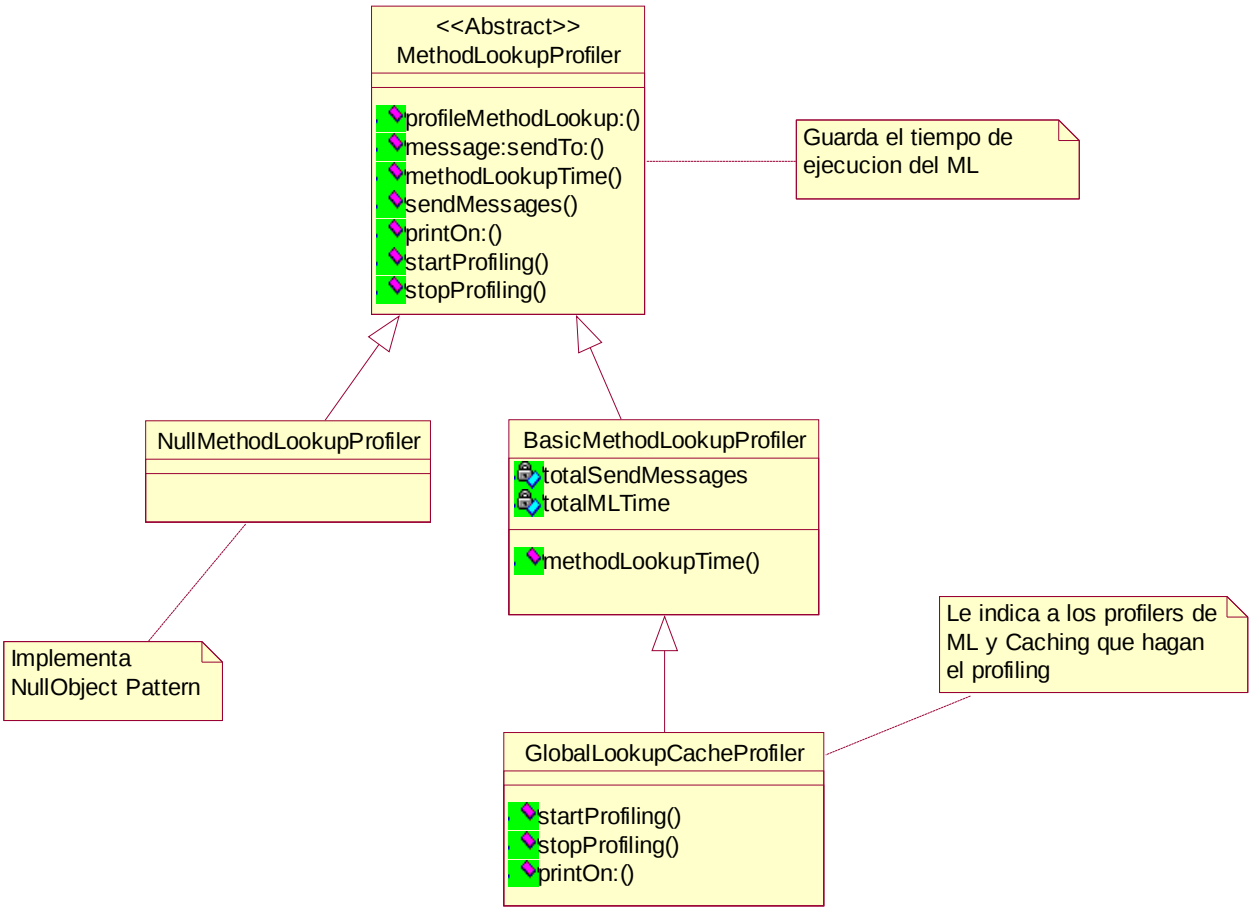


Figura 33: Diagrama de Clases del paquete Method Lookup Profiler

Las principales clases que componen este paquete son:

- **MethodLookupProfiler**: Define el comportamiento que todos los profilers deben tener. Los métodos principales son *message: aMessage sendTo: aClass* y *profileMethodLookup: aBlock*. El primero se encarga de mantener un contador que refleja la cantidad de veces que se envió un mensaje a objetos instancias de una clase. El segundo se encarga de medir el tiempo que se tarda en realizar la búsqueda.

- **NullMethodLookupProfiler**: Esta clase implementa el pattern NullObject (Ver [Martin et.al]). Es utilizado como profiler default cuando no se está corriendo ningún test. Tener este objeto permite evitar el uso de mensajes de control de flujo como *ifTrue: ifFalse:*.

- **BasicMethodLookupProfiler**: Esta clase implementa el comportamiento básico de la mayoría de los profilers. Es utilizado por distintos algoritmos como el **DTS**.

Como se menciona en el capítulo “Trabajo a Futuro”, este esquema de profiling puede ser reemplazado por otro que utilice metaprogramación, el cual permitiría obtener un menor acoplamiento entre las clases que representan los algoritmos de lookup de aquellas que realizan el profiling. No se implementó esta opción porque esta idea surgió hacia el final del trabajo de investigación y por motivos de tiempo se decidió dejarlo como trabajo a futuro.

5.6.1 Profiling de un method-lookup

Como podemos ver en la Figura 36, la interacción entre un profiler y una estrategia de method-lookup es sencilla.

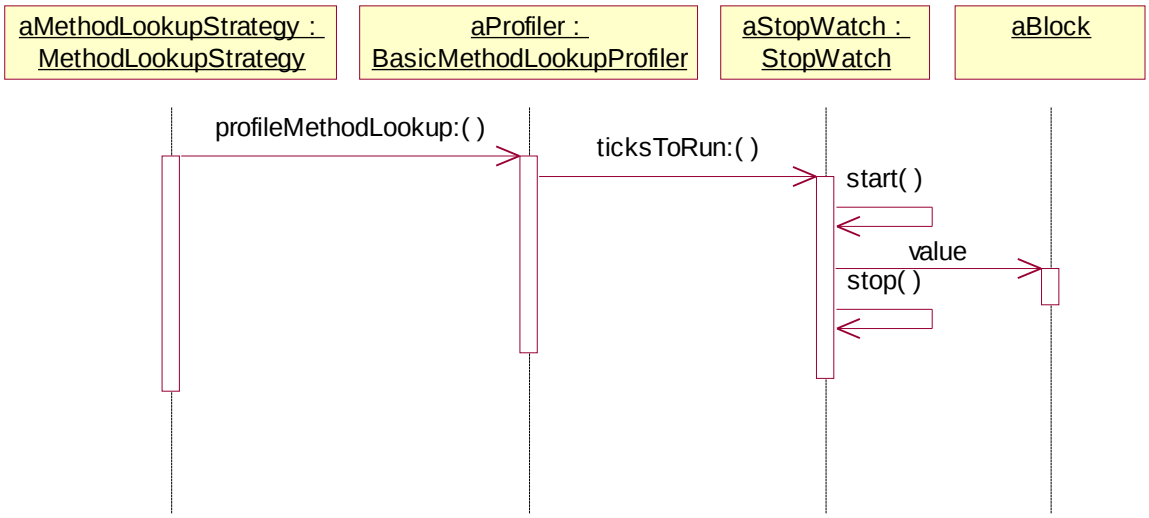


Figura 34: Diagrama de Interacciones entre un estrategia de lookup y su profiler

La estrategia envía el mensaje *profileMethodLookup: aBlock* con el bloque sobre el cual se debe medir el tiempo. Luego el profiler se encarga de inicializar un **StopWatch** (Ver 5.9) y tomar el tiempo de ejecución del bloque.

El motivo por el cual no se utiliza la implementación de Squeak para medir el tiempo de la ejecución de un bloque se debe a que Squeak utiliza como mínima unidad de tiempo a los milisegundos, una resolución muy grande para el tipo de mediciones que se desean tomar. Debido a ello se realizó una implementación propietaria que mide tiempos de ejecución a partir del reloj de tiempo real de la máquina. La resolución de dicho reloj varía por máquina, pero es muy superior al milisegundo.

5.7 Caching Strategies

Este paquete modela el problema de realizar caching con los algoritmos de method lookup. Las estrategias de caching son polimórficas con las estrategias de lookup, puesto que el hecho de tener una cache no afecta la finalidad esencial del algoritmo que es buscar la implementación de un mensaje.

La clase abstracta que define a todas las estrategias de caching se denomina **CachingStrategy** y subclasifica **BasicMethodLookupStrategy**. Al mismo tiempo hay una implementación básica denominada **BasicCachingStrategy** que contiene código común a todas las estrategias de caching. A continuación se detalla la responsabilidad de cada clase del paquete.

- **CachingStrategy**: Define el comportamiento que todas las estrategias de caching deben tener. Como es polimórfica con **MethodLookupStrategy**, es el mensaje *methodLookupSelector: aSelector inClass: aClass* el que debe ser utilizado en las diversas implementaciones. También define el protocolo para mantener la cache (*addSelector: aSelector forClass: aClass*, *flushCache*).

- **BasicCachingStrategy**: Esta clase define el comportamiento básico de la mayoría de las estrategias de caching. Se encarga también de mantener la relación con el profiler. El mensaje *methodLookupSelector: aSelector inClass: aClass* es implementado de manera similar a **BasicMethodLookupStrategy**, utilizando el method template pattern [GOF]. Esta implementación verifica si el mensaje se encuentra en la cache (enviando el mensaje *lookupInCache: aSelector forClass: aClass*), en caso negativo delega la búsqueda en el algoritmo de lookup de backup y agrega el resultado a la cache. El pseudo código de la implementación de este método se puede ver en la Figura 35.

  Esta clase fue utilizada luego para la implementación de tres estrategias de caching distintas, una que implementa una cache global, otra que implementa una cache por selector y otra específica para el algoritmo **HierarchyBranchML**.
  Aunque este algoritmo funcionó correctamente para estas tres estrategias de caching, no podemos asegurar que sirva para otras estrategias como por ejemplo la del algoritmo **PIC**.

```
BuscarMetodoPara( S: Selector, C: Clase ) → Metodo

Metodo ← BuscarEnCache ( S,C )
Si Metodo es nil entonces
        Metodo ← BuscarConMethodLookupDeBackup ( S,C )
        AgregarACache ( S,C,Metodo)

Devolver Metodo
```

Figura 35: Pseudo código de Implementación de los algoritmos que usan Cache

Este paquete contiene también distintas implementaciones de estrategias de caching, como Global Lookup Cache, Selector Cache, etc.

- **GlobalCachingStrategy**: Esta clase implementa la estrategia de caching para una cache de métodos global. Esta estrategia es la utilizada por el algoritmo **GLC**. Ver 7.1

- **HBCachingStrategy**: Esta clase implementa le estrategia de caching para el algoritmo **HierarchyBranchML.** Para más detalle ver 7.3

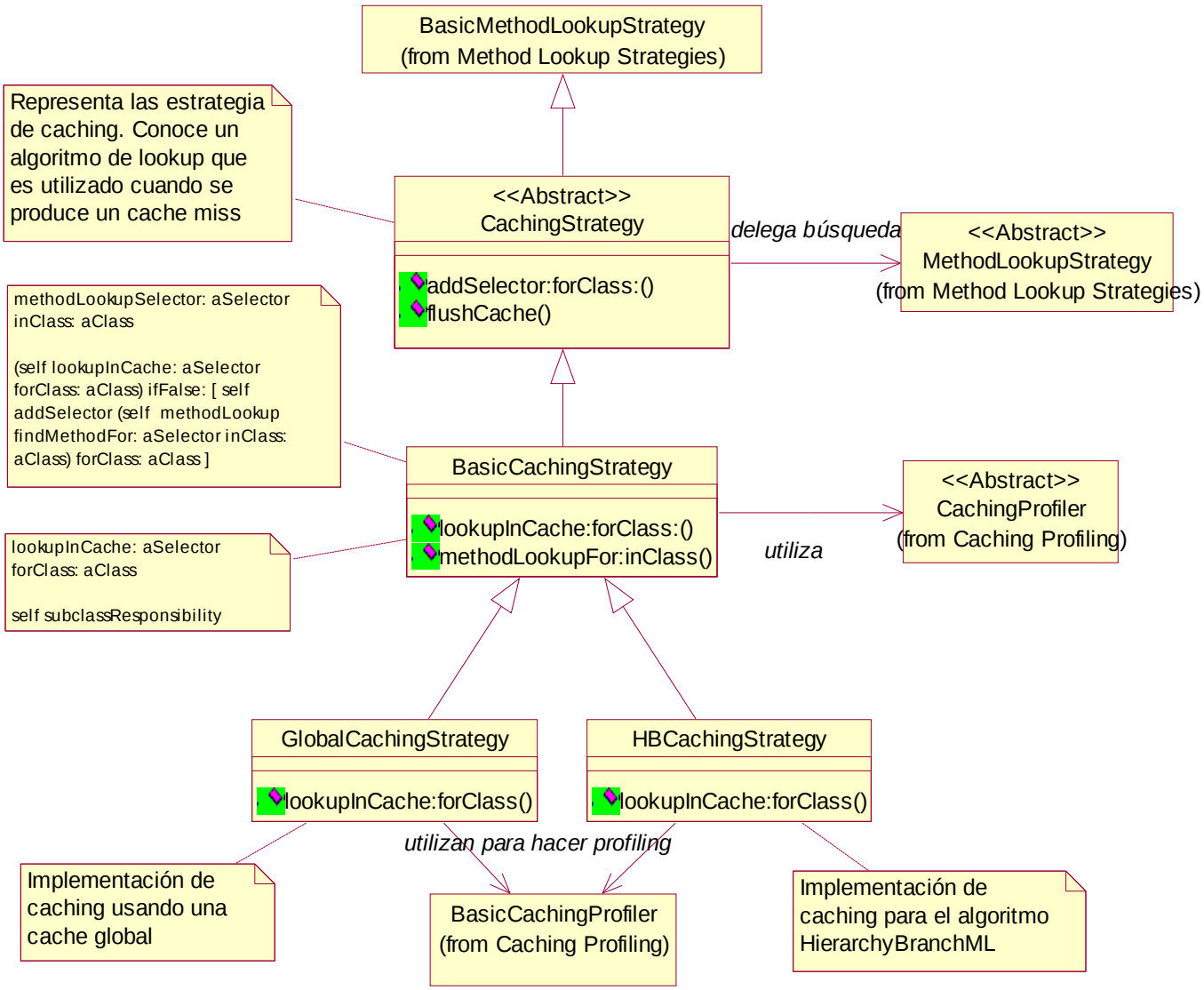


Figura 36: Diagrama de clases del paquete Caching Strategies

5.7.1 Interacción para el Algoritmo GLC

Debido a que este algoritmo es un poco más complejo que **DTS**, se puede ver que hay más colaboración entre los objetos que lo implementan.

Como se puede observar en el diagrama de la , en este caso la implementación del mensaje *methodLookupForSelector: aSelector inClass: aClass* envía el mensaje *profileCacheLookup: aBlock* al profiler para que mida el tiempo de lookup en la cache, el cual a la vez envía el mensaje *lookupInCache: aSelector forClass: aClass* a la misma estrategia de caching.

En el ejemplo está modelado el caso que se produzca un cache miss, por lo tanto la estrategia de caching informa al profiler de dicho problema enviándole el mensaje *incrementCacheMiss.*

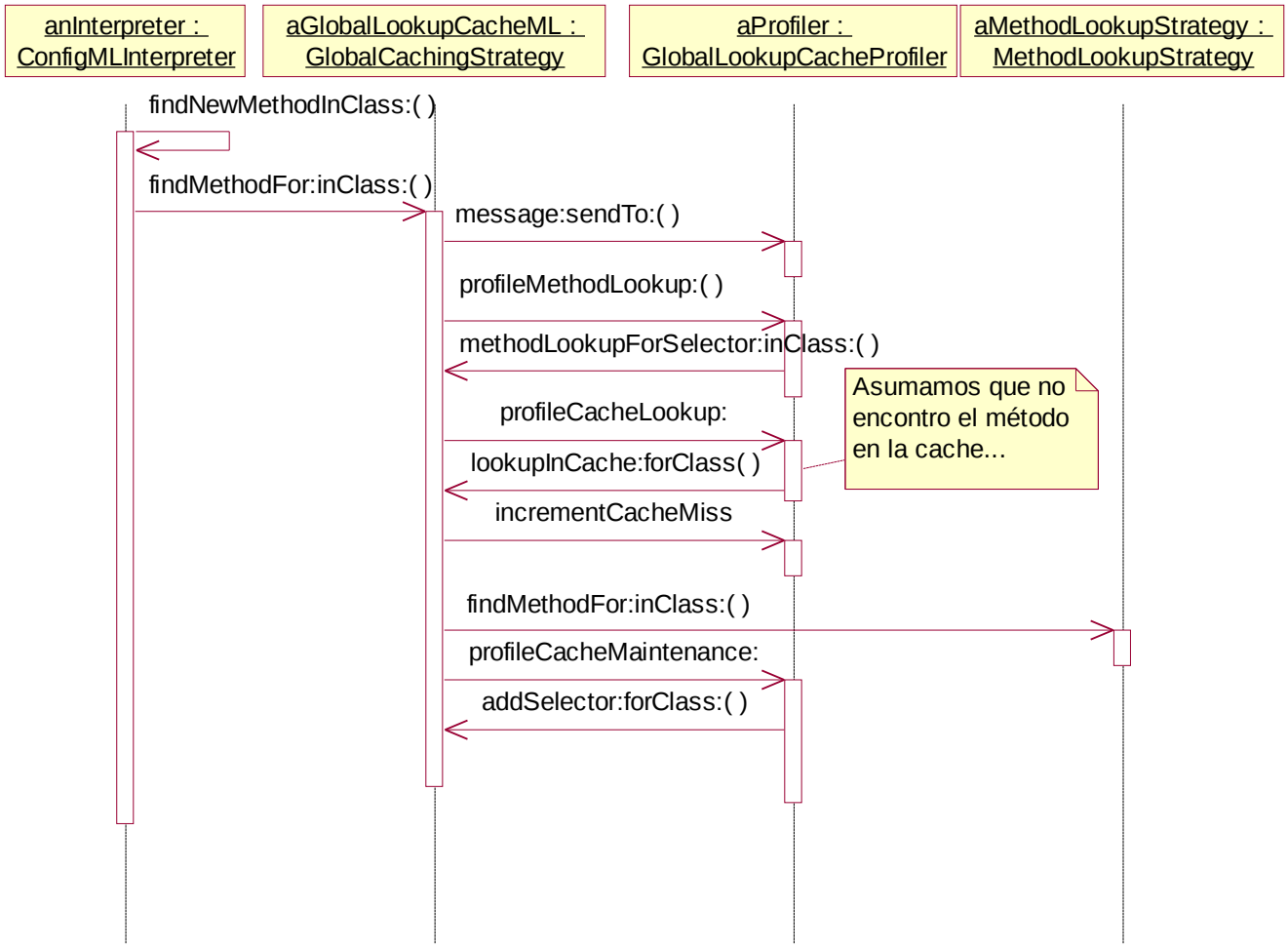


Figura 37: Diagrama de Interacción para el algoritmo GLC

Debido a que no se encontró el mensaje en la cache, el algoritmo de **GLC** utiliza un algoritmo de búsqueda de backup, en este caso **DTS**, cuya interacción esta detallada en la sección 5.10.1. Una vez encontrado el método para este mensaje, se le pide a la cache que lo agregue dentro de su estructura de búsqueda, indicándole al profiler que mida dicho

tiempo por medio del mensaje *profileCacheMaintenance: aBlock* el cual envía el mensaje *addSelector: aSelector forClass: aClass* que realmente realiza la operación de mantenimiento.

5.8 Caching Profiling

Este paquete contiene las clases que se encargan de realizar el profiling de las distintas estrategias de caching. La idea utilizada es la misma que para el paquete Method Lookup Profiling, de manera que cada vez que se desea realizar una operación relacionada con la cache, la estrategia de caching delega en el profiler la medición del tiempo que tarda dicha operación.

Estas clases son a su vez polimórficas con respecto a la clase **MethodLookupProfiler** puesto que ambas son utilizadas con el mismo fin.

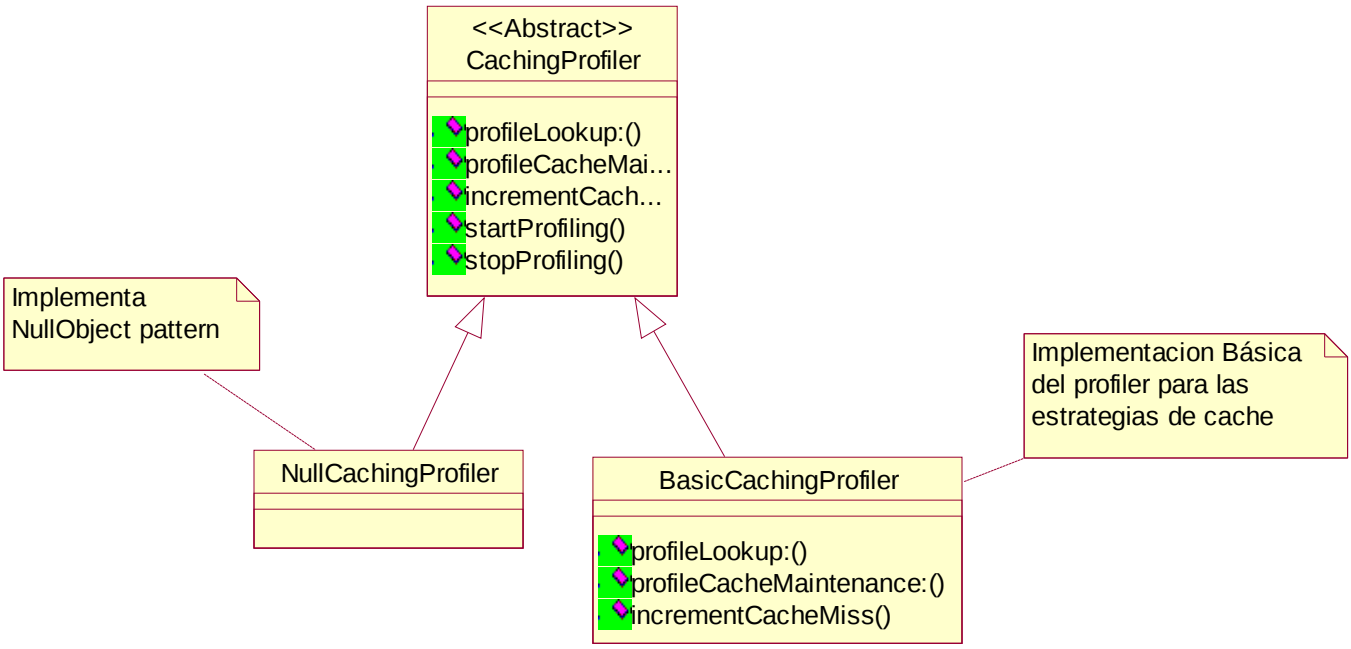


Figura 38: Diagrama de Clases del Paquete Caching Profiler

Las clases principales que componen este paquete son:

- **CachingProfiler**: Es una clase abstracta y subclasifica **BasicMethodLookupProfiler**. Define el comportamiento común para todos los profilers de las distintas estrategias de caching. Según la tarea que esté realizando la estrategia de caching, debe enviar los mensajes *profileCacheLookup: aBlock* o *profileCacheMaintenance: aBlock*. El primero debe ser utilizado cuando se realizan búsquedas en la cache y el segundo cuando

se realizan trabajos de mantenimiento como agregar elementos a la cache, realizar un flush, etc.

- **NullCachingProfiler**: Esta clase implementa el **NullObject** pattern (Ver [Martin et.al]) para el profiling de caches. Es utilizado cuando no se está realizando profiling de los algoritmos. De esta manera evitamos tener que utilizar mensajes de control de flujo como *ifTrue:ifFalse.*

- **BasicCachingProfiler**: Es la implementación básica de **CachingProfiler**.

5.9 Profiling Support

El soporte que ofrece Squeak para tomar mediciones de tiempo no es muy fino, la mínima unidad posible es el milisegundo. Debido a ello, fue necesario modificar la virtual machine agregando un par de primitivas para obtener mediciones más finas y a la vez se creó un conjunto de clases para hacer uso de dicho soporte. Estas primitivas utilizan el reloj de real-time que provee el hardware de una arquitectura Wintel que es donde se desarrolló esta tesis.

Una de las restricciones que tuvimos al escribir este paquete fue que el tiempo que se tarda en tomar las mediciones no debía influir en los tiempos totales de los algoritmos, para evitar de esta manera desvíos en las mediciones. Esto se logró en la implementación de las primitivas de mediciones utilizando un stack de longs de C, evitando de esta manera tener que crear objetos utilizando la VM de Squeak para representar dicho valores. La diferencia entre estas dos soluciones fue bastante significativa en lo que respecta al tiempo que consumen.

Las primitivas implementadas son:

- **int primitiveCurrentTicks (void)**: Devuelve un objeto que representa los ticks actuales del reloj de real-time de la máquina.

- **int primitiveTicksPerSecond (void)**: Devuelve la resolución del reloj de real-time de la máquina. Esta primitiva es llamada para inicializar el paquete de Profiling Support.

- **int primitiveSaveCurrentTicks (void)**: Pone en el stack interno los ticks actuales del reloj de real-time.

- **int primitiveElapsedTicks (void)**: Devuelve la diferencia entre los ticks actuales del reloj de real-time y el primer elemento almacenado por la primitiva primitiveSaveCurrentTicks en el stack interno.

- **int primitiveInitializeTicksStack (void)**: Inicializa el stack interno utilizado por estas primitivas.

- **int positive64BitIntegerFor (LONGLONG aLargeInteger)**: Esta primitiva convierte un tipo LONGLONG de C a un objeto de 64 bits entero positivo de Smalltalk.

Las clases de este paquete que hacen uso de esta primitivas son:

- **StopWatch**: Esta clase representa un cronómetro al cual se le indica cuándo debe empezar a contar y cuando debe terminar. Tiene mensajes adicionales para facilitar las mediciones sobre bloques, métodos, etc.

- **StopWatchTime**: Representa un tiempo devuelto por la medición realizada por un **StopWatch**. Luego, a partir de cómo se quiera ver dicho tiempo se puede convertir a segundos, milisegundos, etc. También se agregó lógica aritmética a estos objetos para poder trabajar fácilmente con ellos.

- **StopWatchTicks**: Subclasifica **StopWatchTime**. Representa los ticks utilizados por el reloj de real-time de la máquina.

- **Seconds**: Subclasifica **StopWatchTime**. Se encarga de representar los ticks del reloj de real-time en segundos.

- **Milliseconds**: Subclasifica **StopWatchTime**. Se encarga de representar los ticks del reloj de real-time en milisegundos.

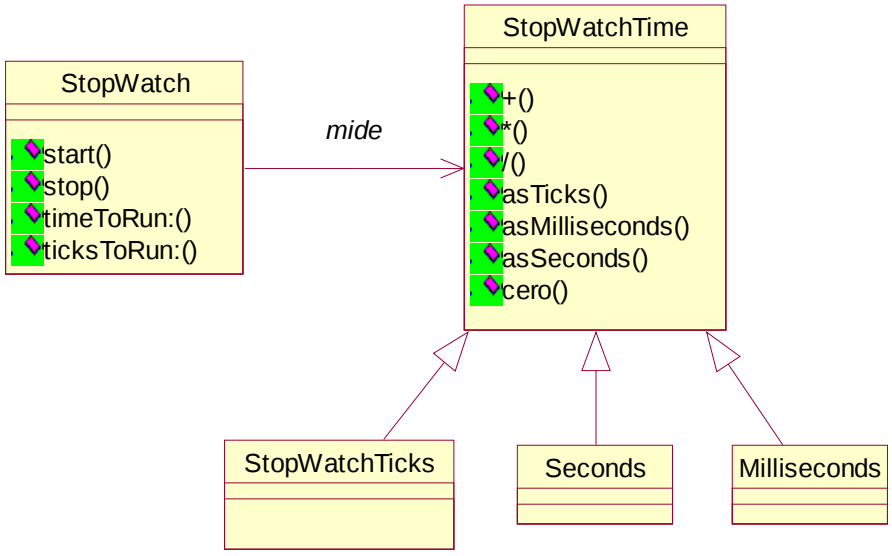


Figura 39: Diagrama de Clases del paquete Profiling Support

5.10 Benchmarks

Este paquete en realidad no forma parte del framework presentado hasta ahora, sin embargo está compuesto por clases que hacen uso de dicho framework y posee ciertos tests que son utilizados para sacar conclusiones sobre las distintas técnicas de method lookup.

Cabe destacar que este paquete se encuentra en la imagen de Squeak que es ejecutada con el interprete del framework. Por cuestiones de memoria se utilizó un imagen reducida de Squeak (la utilizada para WindowCE) para estas pruebas.

Las clases que componen este paquete son:

- **ThesisBenchmarks**: Esta clase contiene los test que se ejecutaron para probar los distintos algoritmos de method-lookup, como *testCompiler*, *testFileOut*, etc. También provee ciertos mensajes que permite realizar los test de manera más sencilla como *runAllTestUsingDTS* o *runAllTestUsingGLC*.

- **ConfigMLInterpreterProxy**: Esta clase es la encargada de comunicarse con el interprete que está ejecutando la imagen. Esta comunicación se realiza por medio de primitivas que permiten indicarle al interpreter que se está por ejecutar un test (*runPrimitiveStartTest*), que se terminó de ejecutar un test (*runPrimitiveStopTest*) o qué algoritmo de method-lookup se quiere utilizar (*runPrimitiveUseMethodLookup*).

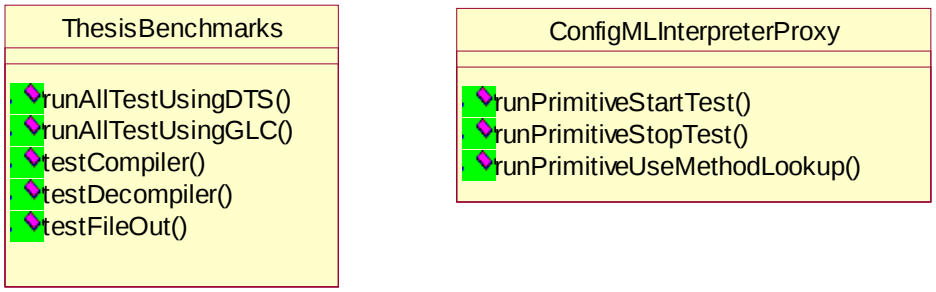


Figura 40: Diagrama de clases del paquete Bechmarks

- **DTSWithClassesProxy**: Esta clase es la encargada de crear todo el ambiente de objetos necesario para utilizar el method lookup DTSWithClasses que es explicado más adelante. (Ver 6.3)

- **FullDictionaryMethodLookup**: Esta clase tiene la responsabilidad de crear los diccionarios de mensajes completos para el algoritmo de method lookup que lleva su mismo nombre. (Ver 6.6)

- **HierarchyBranchML**: Esta clase tiene la responsabilidad de crear el grafo de objetos necesario para que el algoritmo de búsqueda con su mismo nombre funcione correctamente. (Ver 6.4)

- **HierarchyBranchWithDicML**: Idem HierarchyBranchML pero para el algoritmo derivado que lleva su mismo nombre. (Ver 6.5)

Los test utilizados son explicados más adelante en 6.1 como así también cada uno de los algoritmos de búsqueda implementados.

5.10.1 Ejecutando un benchmark

En la Figura 41 se puede observar el diagrama de interacciones entre los distintos objetos cuando se ejecuta un test.

En este caso, se está enviando el mensaje *runAllTestUsingDTS* a un objeto instancia de *Benchmark*. Este se comunica con el interpreter para indicarle que debe utilizar la estrategia de DTS de method lookup. Esto lo hace enviando el mensaje *runPrimitiveUseMethodLookup* a la clase *ConfigMLInterpreterProxy*

El interprete crea una nueva instancia de la estrategia de lookup indicada (en este caso DTS). Una vez terminada esta secuencia de colaboraciones, el benchmark le indica al interprete que se está por ejecutar un test, enviando el mensaje *runPrimitiveStartTest* al ConfigMLInterpreteProxy. El interprete ejecuta la primitiva creando una nueva instancia de la clase Test e indicándole que debe empezar a realizar profiling de lo que se esta ejecutando.

Una vez creado todo el ambiente para la ejecución de un test, el benchmark se encarga de ejecutarlo. En este caso se muestra la ejecución del test *testCompiler*.

Terminado el test, se indica al interprete por medio del mensaje *runPrimitiveStopTest* dicho suceso, el cual se encarga de mostrar los resultados por pantalla.

5.11 Conclusiones

En este capítulo se describió el framework de soporte de method lookup creado para facilitar todos los test y experimentos que se muestran en este trabajo.

La codificación de este paquete presentó varios desafíos. Por un lado se obtuvo un diseño bastante flexible y genérico dado el objetivo propuesto. Al mismo tiempo fue necesario bajar bastante de nivel de abstracción e investigar varios puntos interesantes de la ejecución de un ambiente de objetos, como la virtual machine, el formato de los objetos en memoria, el formato de las imágenes, creación de primitivas (tanto en C como en Smalltalk), etc.

De acá en adelante se podrá observar el uso intensivo que se realizó de este framework.

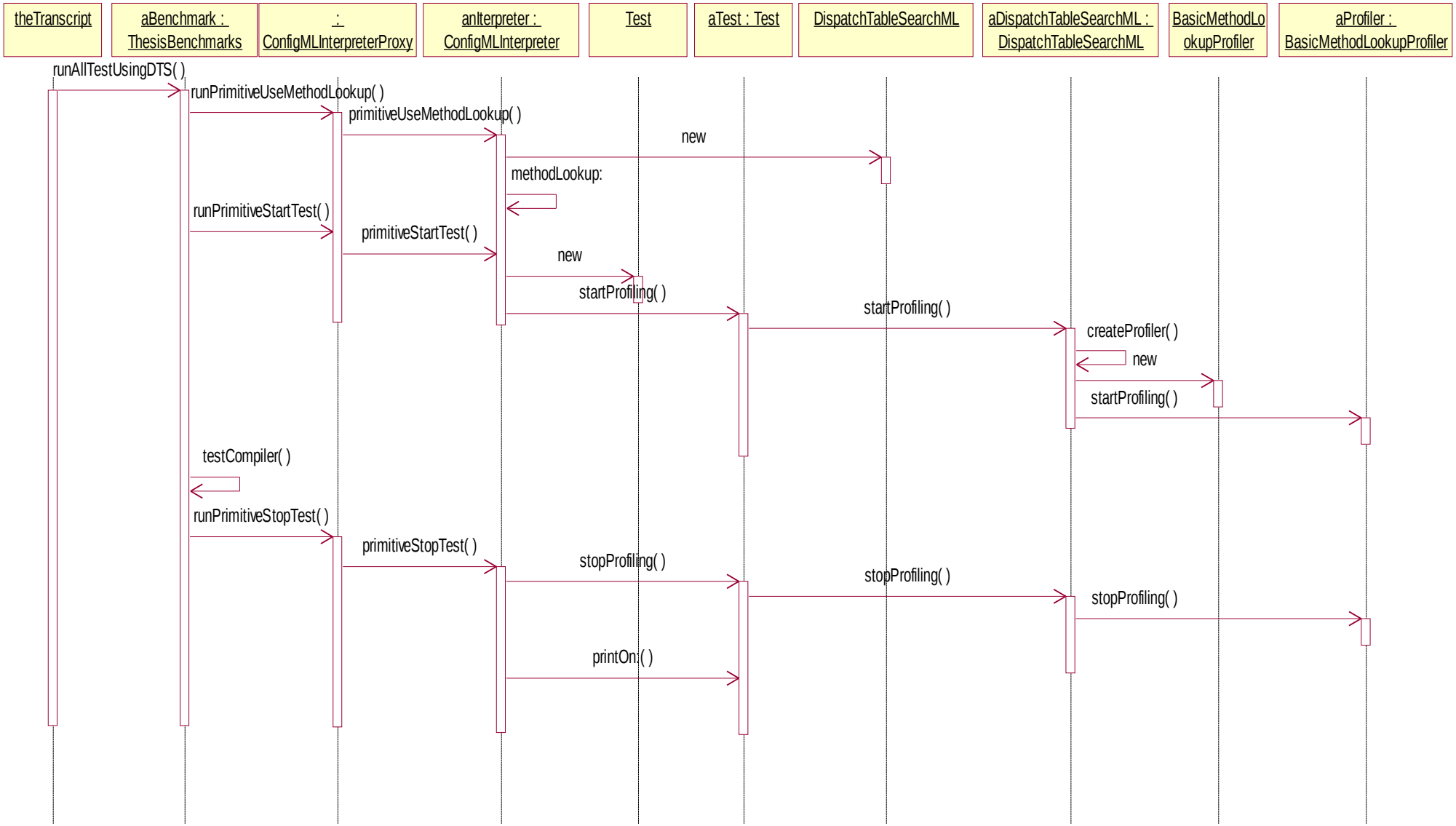


Figura 41: Interacciones de la ejecución de un test

6. Implementación de Distintos Algoritmos de Method Lookup

Este capítulo muestra la implementación de distintos algoritmos de method lookup. El objetivo que se buscó con estas distintas implementaciones fueron:

- Probar el framework comentado en el capítulo 5
- Investigar sobre técnicas de lookup no mencionadas en la literatura de investigación

Para realizar la prueba de cada algoritmo, se utilizó un conjunto de test cases los cuales son detallados en la primera sección de este capítulo. Luego se presenta cada uno de los algoritmos implementados. Cada algoritmo tiene una descripción de funcionamiento, una descripción sobre la implementación, una cuadro de estadísticas y una conclusión final. Hacia el final se muestra una tabla comparativa de todos ellos (Ver 6.7).

6.1 Test Ejecutados

Los test ejecutados son similares a los especificados en el "green book"[Krasner] y en SOAR[Ungar]. Los distintos test cases junto a una breve explicación se pueden ver en la Tabla 8. El código de los ejemplos están en el 11.10.4.

Para tomar las estadísticas de cada algoritmo, los test fueron ejecutados 3 veces en condiciones iguales de ambiente. En cada corrida se generó información relativa al tiempo de ejecución, cantidad de mensajes enviados, cache hits, cache miss, y demás datos específicos de cada algoritmo.

Nombre del test	Descripción
TestCompiler	Compila un método largo dentro de una clase
TestDecompiler	Decompila el método compilado por el test 'testCompiler'
TestFileOut	Realiza un fileOut de la clase 'AbstractSound'
TestPrintHierarchy	Imprime en el transcript la jerarquía de la clase 'Number'
TestRestoreDisplay	Redibuja toda la ventana de Squeak
TestTranscriptShow	Borra el transcript e imprime 'Hello World!'

Tabla 8: Descripción de test cases similares a los del "green book"

6.2 Dispatch Table Search

6.2.1 Descripción

Como se mencionó anteriormente, este algoritmo es el más básico y fácil de implementar. Su descripción se puede encontrar en 4.2.1

6.2.2 Implementación

Este algoritmo está implementado en la clase **DispatchTableSearchML**

6.2.3 Estadísticas

En la Tabla 9 se pueden observar los resultados de ejecutar los tests utilizando el algoritmo de DTS.

Esta tabla contiene las siguientes columnas:

- Tiempo Total: Indica el tiempo total de ejecución del test case
- Tiempo ML: Indica el tiempo insumido durante la ejecución del test case por el algoritmo de lookup
- Porcentaje: Es el porcentaje del tiempo de lookup con respecto al tiempo total de ejecución.
- Mensajes Enviados: Cantidad de mensajes enviados por el test case.

Como se puede ver, un 18,95% promedio de la ejecución de los tests es consumido por la ejecución de este algoritmo, llegando a un máximo del 22,75%.

Test	Tiempo Total	Tiempo ML	Porcentaje	Mens. Enviados
testCompiler	5376.75	1223.73	22.75	24061.33
testDecompiler	1293.75	278.10	21.49	4430.00
testFileOut	8821.46	1426.75	16.17	31398.67
testPrintHierarchy	1824.39	316.78	17.37	5838.67
testRestoreDisplay	6064.56	875.74	14.44	18046.33
testTranscriptShow	853.95	183.28	21.47	3283.33
Mínimo			14.44	
Máximo			22.75	
Promedio			18.95	

Tabla 9: Estadísticas del Algoritmo DTS

6.2.4 Conclusión

Este algoritmo tiene varias ventajas:

- Es fácil de mantener la estructura que es utilizada en la búsqueda: Cada clase tiene un diccionario de los mensajes que sabe responder asociado a su implementación. Agregar un nuevo mensaje o borrar un mensaje existente implica agregar o borrar una entrada al diccionario respectivamente. Ninguna de estas operaciones es costosa en lo que respecta al tiempo.

- El espacio que ocupa esta estructura es mínimo: Cada clase tiene solo los mensajes y la implementación de los mensajes que sus instancias saben responder.

Sin embargo, tiene la gran desventaja de ser un algoritmo lento debido a la necesidad de tener que buscar las implementaciones de los mensajes en la jerarquía de herencia de una clase. Como pudimos observar, esto implica un 18,95% promedio del tiempo total de ejecución de un algoritmo, llegando a un máximo de 22,75%.

6.3 Dispatch Table Search With Classes

6.3.1 Descripción

Dentro de la literatura que estudia los distintos algoritmos de Method lookup no pudimos encontrar investigación que realice la búsqueda de manera inversa a la de **DTS**. O sea, en vez de que cada clase tenga un diccionario de mensajes implementados, hacer que cada mensaje tenga un diccionario de clases que lo implementan. Este algoritmo se basa en dicha idea.

Conceptualmente, cada selector posee un diccionario con las clases que implementan dicho mensaje. Cuando se envía un mensajes, este algoritmo busca en dicho diccionario la clase del objeto receptor del mensaje. En caso de no encontrarla, realiza la misma secuencia que **DTS**, buscando el mensaje nuevamente en el mismo diccionario pero para la superclase. Si la superclases es *nil*, envía el mensaje *#doesNotUnderstand* a *self*.

6.3.2 Implementación

Este algoritmo está implementado en la clase **DTSWithClassesML**.

Durante la implementación de este algoritmo surguió un problema que impidió implementarlo tal cual cómo se esperaba. El problema está relacionado con el formato de la clase Symbol.

La clase Symbol es la que modela los selectores. O sea, el selector *at:* utilizado para enviar el mensaje *#at:* a una clase, es instancia de la clase Symbol. Este algoritmo se basa en que cada selector (o sea, cada instancia de la clase Symbol) tenga un diccionario que le permita determinar que clases implementan mensajes con este selector como nombre. Por lo tanto, para poder implementar este algoritmo era necesario modificar la clase Symbol y agregar una variable de instancia que referencie a dicho diccionario.

Debido a que la clase Symbol es de formato variable y de bytes[26], agregar una variable de instancia para referenciar a dicho diccionario de clases es imposible al menos que se modifique su formato y estructura. Symbol es una de las clases que la Virtual Machine debe conocer su estructura para poder funcionar correctamente, por lo que modificar su formato implicaba modificar la Virtual Machine en todo lugar donde se

[26] Las clases en Smalltalk tiene diferentes formatos de representación en memoria. Formato variable y de bytes indica que las instancias de esa clase tienen dimensión variable por instancia y que es indexable de a bytes. Las clases comúnmente utilizadas son de formato fijo y de word, esto implica que todas las instancias tienen la misma dimensión y que es indexable de a word, puesto que cada word es una variable de instancia que referencia a otros objetos.

referenciaba a Symbol y crear una imagen nueva con el formato nuevo de Symbol. Por más que la Virtual Machine de Squeak está implementada en Smalltalk, su diseño no es bueno y el código está ampliamente acoplado puesto que está hecho de tal manera que permita su traducción al lenguaje C facilmente.

Realizar este trabajo de modificación estaba fuera del alcance de la investigación, por lo que se creó un diccionario global cuya clave son los selectores y valor los diccionarios de clases que lo implementan. De esta manera cada vez que se envía un mensaje, el algoritmo busca el diccionario de clases dentro del diccionario global de selectores y luego realiza la búsqueda del mensaje en sí.

El tiempo de búsqueda del diccionario de clases dentro del diccionario global de selectores es computado como tiempo extra para que no afecte el tiempo final de ejecución del algoritmo.

6.3.3 Estadísticas

A continuación, en la Tabla 10, se puede observar los resultados estadísticos de este algoritmo. Esta tabla (al igual que las tablas de los siguientes algoritmos) posee las siguientes columnas:

- Tiempo Total de ejecución: Muestra el tiempo total de ejecución del test case.
- Porcentaje de Diferencia con DTS: Muestra la diferencia de tiempo con respecto a DTS en porcentaje. Cuanto más grande el valor, mejor la performance. (O sea, menos tiempo se tardó en ejecutar el test)
- Tiempo de ML: Indica el tiempo que insumió el algoritmo de lookup.
- Porcentaje de Diferencia con DTS: Muestra la diferencia de tiempo con respecto a tiempo que insumió el algoritmo de DTS. Cuanto más grande el valor, mejor la performance. (O sea, menos tiempo tardó el algoritmo en resolver las búsquedas).
- Porcentaje: Porcentaje de tiempo utilizado por el algoritmo de búsqueda con respecto al tiempo total de ejecución del test case.

Para este algoritmo, la ejecución de los test cases resultaron ser en promedio un 5,15% más rápida que **DTS**, con un máximo de 9,07%.

El tiempo de lookup es en promedio un 14,04% más rápido que **DTS**, con un máximo del 22,68%.

	Tiempo Total		Tiempo ML		Procentaje
Test	Actual	% Dif. DTS	Actual	% Dif. DTS	
testCompiler	5020.84	6.62	969.12	20.81	19.30
testDecompiler	1176.37	9.07	216.24	22.24	18.39
testFileOut	8249.36	6.49	1103.14	22.68	13.37
testPrintHierarchy	1722.27	5.60	275.91	12.90	16.02
testRestoreDisplay	5979.91	1.40	839.98	4.08	14.05
testTranscriptShow	839.13	1.74	180.44	1.55	21.50
Mínimo		1.40		1.55	13.37
Máximo		9.07		22.68	21.50
Promedio		5.15		14.04	17.11

Tabla 10: Estadísticas de tiempo del Algoritmo DTS with Classes

6.3.4 Conclusión

La implementación de este algoritmo implica la modificación de las siguientes clases:

- La clase Symbol: para que colabore con un diccionario de métodos por clase
- La clase Behavior: no debe tener un diccionario de métodos sino una colección de mensajes.

El proceso de agregar un mensaje a una clase implicaría insertar dicho mensaje dentro de la colección de mensajes de la clase y agregar en el diccionario de dicho mensaje la clase para la cual se creo el mensaje, con el método correspondiente. Borrar un mensaje de una clase sería simplemente hacer el proceso inverso.

Como se pudo observar en la tabla de resultados, hay un promedio de 5% de ganancia en velocidad de ejecución. Esta ganancia se debe a que los diccionarios de búsqueda de un mensaje son en promedio más chicos que los de las clases. Esto significa que el número de mensajes implementados por una clase es mayor respecto al número de clases que implementan un mensaje. En Tabla 11 se puede ver los valores correspondientes a esta aseveración.

	Promedio	Máximo	Mínimo	Mediana
DTS	9.81	202	0	3
DTS With Classes	1.58	88	1	1

Tabla 11: Cantidad de entradas en los diccionarios de DTS y DTS With Classes

6.4 Hierarchy Branch

6.4.1 Descripción

Este es quizá el algoritmo más complicado de todos los implementados y cubre también un espacio no investigado en la literatura.

Este algoritmo trata de evitar tener que realizar la búsqueda de la implementación de un mensaje en la jerarquía de herencia en caso de no encontrarse la misma en la clase del objeto receptor, a diferencia del algoritmo DTS que busca en toda la rama de la jerarquía de herencia

Por lo tanto, asociando la implementación de un mensaje con la "rama" de la jerarquía de herencia para la cual es válido, se evita realizar la búsqueda utilizada por el algoritmo DTS.

La idea de implementar este algoritmo surgió a partir de ciertos datos estadísticos sobre la relación entre mensajes y la cantidad de implementaciones de cada mensaje. A continuación se presentan estos datos.

Datos estadísticos de la Relación entre Mensajes, Cantidad de Implementaciones y Referencias

Estas estadísticas relacionan los mensajes con la cantidad de implementaciones que poseen (métodos) y la cantidad de veces que son enviados.

Los resultados de este estudio reveló un hecho poco mencionado en la literatura estudiada, el cual muestra que la gran mayoría de mensajes dentro de Smalltalk son implementados un única vez (un 79% como veremos más adelante) y que es muy baja la cantidad de mensajes que tienen más de 6 implementaciones (mensajes que denominamos '*megamórficos*').

La implementación utilizada para realizar esta estadística se puede ver en el Apéndice B. Debido a su característica estática y sencillez, no merece mayores explicaciones.

El resultado de analizar todos los mensajes con respecto a la cantidad de veces que son enviados se muestra en la Tabla 12. La misma muestra que hay una gran cantidad de mensajes que nunca son enviados, o que solo se envían en un solo método.

	Env. 0	Env. 1	Env. 2	Env. 3	Env. 4	Env. 5	Env. >6
Cantidad de Mensajes	1743	4959	2287	1218	804	486	2493
Porcentaje	12,46%	35,46%	16,35%	8,71%	5,75%	3,47%	17,83%

Tabla 12: Relación mensajes definidos y enviados.

La otra medida que se tomó fue la cantidad de implementaciones que posee cada mensaje. La Tabla 13 muestra dicho resultado.

	1 impl.	2 impl.	3 impl.	4 impl.	5 impl.	6 impl.	>6 impl.
Cantidad de Mensajes	10989	1715	503	261	198	69	246
Porcentaje relacionado	79,13%	12,34%	3,63%	1,87%	1,42%	0,49%	1,77%

Tabla 13: Relación entre mensaje y cantidad de implementaciones (métodos) asociados

Como se puede ver, la mayor cantidad de mensajes tienen una sola implementación (79%) siendo los que tiene dos implementaciones también un porcentaje alto (12%). Ejemplos de estos mensajes son los implementados en Object únicamente *(#error:*, *#subclassResponsibility*, etc).

Hay que tener en cuenta que la implementación de un mensaje puede estar definida tanto en la raíz de la jerarquía de herencia como en una hoja. Esto implica que por más que un mensaje tenga una sola implementación, no solo instancias de la clase donde se implementa el mensaje responderán al mismo. Este es uno de los puntos flojos de IC y PIC debido a que estas técnicas guardan en la cache la clase del objeto receptor del mensaje y no todas aquellas clases para las cuales también es válido el método, como por ejemplo las subclases (siempre y cuando estas no lo redefinan).

Esa desventaja es la que ataca activamente este algoritmo. La idea es que cuando se encuentre la implementación de un mensaje, se sepa para que conjunto de clases dicha implementación es válida, sin tener que hacer búsquedas adicionales.

Por otro lado, hay que tener en cuenta que la existencia de una gran cantidad de mensajes con una sola implementación no implica que el porcentaje de envíos de dicho mensaje sea tan alta. Por lo tanto, también se midió la relación entre la cantidad de implementaciones que tiene un mensaje y la cantidad de veces que es enviado.

Los resultados de la Tabla 14 muestran dicha relación. Como se puede ver, entre un 21% y 47% de mensajes que se envían tienen una sola implementación.

Estos valores fueron obtenidos luego de correr todos los test cases y almacenar en un diccionario los mensajes enviados junto a la clase del objeto receptor, y la cantidad de veces que objetos instancias de dicha clase recibieron dicho mensaje. Luego se analizó para todos los mensajes enviados la cantidad de implementaciones que tenía cada uno.

Como conclusión se puede decir que hay una gran proporción de mensajes enviados que poseen una sola implementación (promedio del 33%), la cual se incrementa considerablemente en los test que involucran el uso de la pantalla. El máximo llega a un 47%.

Tests	Cantidad de Implementaciones						
	1	2	3	4	5	6	>6
TestCompiler	21%	10%	7%	0%	4%	2%	53%
TestDecompiler	27%	14%	9%	3%	4%	0%	38%
TestFileOut	24%	3%	1%	1%	1%	2%	64%
TestPrintHierarchy	47%	13%	4%	3%	2%	5%	23%
TestRestoreDisplay	44%	23%	4%	4%	4%	4%	14%
TestTranscriptShow	37%	27%	5%	2%	2%	3%	21%
Máximo	47%	27%	9%	4%	4%	5%	64%
Mínimo	21%	3%	1%	0%	1%	0%	14%
Promedio	33%	15%	5%	2%	3%	3%	36%

Tabla 14: Relación cantidad de mensajes enviados y cantidad de implementaciones

Los datos mostrados en esta sección corresponden a la imagen de Squeak versión 2.5. Se puede argumentar que estos datos pueden variar fácilmente, por ejemplo creando una jerarquía de clases que haga crecer la cantidad de mensajes implementados. Sin embargo, el trabajo se basa sobre imágenes reales de trabajo y no suposiciones, de allí la importancia de esta medida.

Numeración de las clases

A partir de los datos estadísticos que se muestran en la sección anterior, se dedujo la posibilidad de relacionar la implementación de un mensaje con un conjunto de clases y no con una clase únicamente.

El conjunto de clases con las que se puede relacionar la implementación de un mensaje se encuentra íntimamente ligado a la jerarquía de herencia. Si observamos la Figura 43, se puede ver la que implementación de *m2* en la clase **A** es la misma que para las clases **B, C, D y E**. Esto se debe a la definición de herencia como mecanismo de *sharing* de código. (Ver [Stein et. al]).

Debido a esta característica, se concluyó que era importante poder determinar rápidamente si una clase era subclase de otra. Para ello se otorgó a cada clase dos números de la siguiente manera:

- El primero, denominado **from,** es el resultado de numerar la jerarquía de herencia de manera “depth first”
- El segundo, denominado **to**, asocia a cada clase el mayor número de todas sus subclases.

Por lo tanto, sea ***from*** el número que se asigna a cada clase y ***to*** el mayor número de sus subclases, se puede concluir que:

B es subclase(A) sii A.from <= B.from <= A.to

Figura 42: Regla de Subclase para algoritmo de Búsqueda HierarchyBranch

Por ejemplo, utilizando la jerarquía de Figura 43, se puede concluir fácilmente que **D** es subclase de **A** comparando 1 <= 4 <= 6. Con las herramientas comunes que ofrece

Smalltalk, determinar si una clase es subclase de otra consume demasiado tiempo, más del admisible para un algoritmo de method lookup.

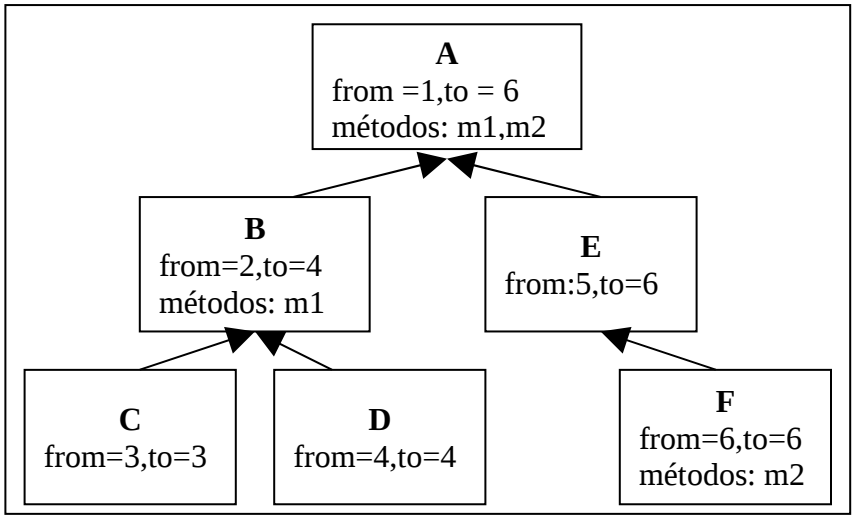


Figura 43: Jerarquía de Ejemplo para Algoritmo Hierarchy Branch

Colección Ordenada de Búsqueda

Además de numerar las clases como se menciona en la sección anterior, se creó para cada selector una lista ordenada de objetos instancia de **MessageImplementationBranch**, cada uno de ellos conteniendo dos números (los llamaremos ***from*** y ***to*** también) y una referencia a un método compilado (llamaremos a esta variable de instancia ***compiledMethod***). El orden de esta colección está dado por la variable de instancia ***from***.

El pseudo-código de las funciones de creación de esta lista se puede ver a continuación.

```
CrearListaDe ( S: Selector ) → Lista Ordenada
  Sea S un Selector
  Sea O1 objeto instancia de MessageImplementationBranch
  Sea ListaDeS una Lista Ordenada de objetos instancia de
MessageImplementationBranch

  ListaDeS ← Nueva Lista Ordenada.
  Para toda Clase C que implemente el mensaje S
    O1 ← Nuevo MessageImplementationBranch
    O1.from ←  C.from
    O1.to ← C.to
    O1.CompiledMethod ← MetodoCompiladoPara(C,S)
    Si existe O en ListaDeS talque O.from <= O1.from <= O.to entonces
      Particionar(O,O1,ListaDeS)
    Agregar O1 en listaDeS

  Devolver ListaDeS
```

Figura 44: Función CrearListaDe del Algoritmo Hierarchy Branch

```
MetodoCompiladoPara ( C: Class,S: Selector ) → ComplileMethod
  Devolver la implementación de S para la clase C
```

Figura 45: Función MetodoCompiladoPara

```
Particionar (O: MessageImplementationBranch, O1: MessageImplementationBranch,
ListaDeS: ListaOrdenada )
'Ver que O.from no puede ser igual a O1.from porque estaría agregando el mismo
'CompiledMethod para la misma clase más de una vez

viejoTo ← O.to
O.to ← O1.from – 1
Si O1.from < O.to entonces
  O2 ← Nuevo MessageImplementationBranch
  O2.from ← O1.to + 1
  O2.to ← viejoTo
  O2.CompiledMethod ← O.CompiledMethod
  Agregar O2 en ListaDeS
```

Figura 46: Función Particionar del Algoritmo Hierarchy Branch

Como resultado de la función *crearListaDe*, se obtiene una lista ordenada de objetos instancia de **MessageImplementationBranch**. Cada uno de estos objetos indican el rango de los números de las clases que implementan un mensaje. Por lo tanto, buscar la implementación de un mensaje para una clase dada será buscar el objeto instancia de **MessageImplementationBranch** cuyo rango contenga el número from de la clase.

```
BuscarImplementacionDe ( S: Selector, C: Clase ) → CompiledMethod

ListaDeS ← ListaDe ( S )
Si existe O talque O.from <= C.from <= O.to
        devolver O.compiledMethod
sino
        'No existe la implementación para la clase C, envío el mensaje
#doesNotUnderstand
        devolver BuscarImplementacionDe (#doesNotUnderstand:,C )
```

Figura 47: Implementación del Algoritmo de búsqueda para Hierarchy Branch

Como se puede observar, este algoritmo cumple con el objetivo de tener que realizar la búsqueda del método en la jerarquía de herencia y reemplaza la metodología utilizada en **DTS** por una búsqueda binaria dentro de la lista de implementaciones de un selector.

Para aquellos mensajes que tiene una sola implementación, el algoritmo es extremadamente veloz porque encontrar la implementación para dichos mensajes implica

únicamente verificar si la clase del objeto receptor se encuentra dentro del rango especificado por el único **MessageImplementationBranch** de la lista.

Otro punto interesante de este algoritmo es el poder determinar el tiempo de búsqueda máximo en el cual se puede incurrir enviando mensajes en un conjunto de colaboraciones. Esto se debe a que el comportamiento del algoritmo está basado en realizar una búsqueda binaria dentro de una lista ordenada de elementos, por lo que el orden de la búsqueda es de **O (n/2)**, siendo **n** la cantidad de objetos instancia de **MessageImplementationBranch** que posee la lista, lo cual es completamente determinable para un mensaje dado.

Por lo tanto, el tiempo máximo que demandará el algoritmo de "method lookup" para un conjunto de colaboraciones se puede concluir con el siguiente algoritmo:

```
TiempoDeBusquedaPara( CC: ConjuntoDeColaboraciones ) → Tiempo

Tiempo ← 0
Para cada M, mensaje enviado en CC
        N ← Dimensión de la lista de implementaciones de M dividido 2
        Tiempo ← Tiempo + N * TiempoIncurridoEnUnaComparacion

devolver Tiempo
```

Figura 48: Determinación del Tiempo De Búsqueda para el algoritmo Hierarchy Branch

Dentro del algoritmo se utiliza la constante **TiempoIncurridoEnUnaComparacion** que indica el Tiempo que insume para verificar si el número de una clase está contenido en el rango indicado por un objeto instancia de **MessageImplementationBranch.**

6.4.2 Implementación

Este algoritmo está implementado en la clase **HierarchyBranchML**, y utiliza dos clases adicionales para crear las listas de implementaciones y la numeración de las clases. Estas clases son:

- **ClassInterval**: Representa el intervalo de numeración de una clase. Tiene las variables de instancia, **from** y **to** que indican el número de una clase y el mayor número de sus subclases respectivamente.

- **MessageImplementationBranch**: Como se mencionó anteriormente, representa el intervalo de clases (o sea, la rama de la jerarquía de herencia) para los cuales es válido la implementación de un mensaje. Por cuestiones de reusabilidad de código, subclasea **ClassInterval**. Posee una variable de instancia adicional, denominada **method** que referencia al **CompiledMethod** en cuestión.

Debido a los mismos problemas mencionados en la implementación del algoritmo "**DTS With Clases**", no se modificó la estructura de las clases **Class** para que contenga su numeración ni la clase **Symbol** para que referencie a la lista de implementaciones. Esto se

solucionó con dos diccionarios, uno que relaciona a cada clase con su intervalo de números y otro que relaciona cada selector con su lista de implementaciones.

Las búsquedas que se realizan en estos diccionarios son computados como tiempo extra.

6.4.3 Estadísticas

En la Tabla 15 se puede observar los datos estadísticos de este algoritmo.

El tiempo de ejecución de los test fue en promedio un 1,8% más rápido que usando DTS, y el tiempo consumido de lookup fue un 9,65% en promedio mayor al de DTS

	Tiempo Total		Tiempo ML		Procentaje
Test	Actual	% Dif. DTS	Actual	% Dif. DTS	
TestCompiler	5482.56	-1.97	1402.77	-14.63	25.59
TestDecompiler	1153.56	10.84	269.27	3.18	23.34
TestFileOut	9106.08	-3.23	1915.50	-34.26	21.04
TestPrintHierarchy	1721.78	5.62	310.75	1.90	18.05
TestRestoreDisplay	6122.12	-0.95	946.57	-8.09	15.46
TestTranscriptShow	849.84	0.48	194.28	-6.00	22.85
Mínimo		-3.23		-34.26	15.46
Máximo		10.84		3.18	25.59
Promedio		1.80		-9.65	21.05

Tabla 15: Estadísticas de tiempo del Algoritmo HierarchyBranch

Un dato estadístico interesante sobre este algoritmo es la distribución de los selectores con respecto a la cantidad de ramas de implementación. En la Tabla 16 se puede ver dicha relación

Cant. De Ramas	Cant. De Selectores	Cant. De Ramas	Cant. De Selectores
1	2724	14	3
2	298	15	5
3	235	21	1
4	64	22	2
5	51	23	2
6	24	27	1
7	19	28	2
8	20	33	1
9	10	47	1
10	14	48	1
11	9	74	1
12	7	85	1
13	6	131	1

Tabla 16: Relación Cant. de Ramas y Cant. de Selectores

Como se puede observar, hay 2724 selectores que tienen una sola rama de implementación. Esto implica, selectores que son definidos en una sola clase y nunca son redefinidos en sus subclases o implementados en otras clases. Para estos selectores, el algoritmo va a ser extremadamente rápido.

Sin embargo, hay casos contrarios a este: hay un selector que tiene 131 ramas de implementación. Este selector es *#initialize*. Para este selector, el algoritmo va a ser tremendamente lento debido a la gran cantidad de ramas sobre las que el algoritmo debe buscar la implementación. Peor aún, el mensaje *#initialize* es enviado constantemente en la mayoría de las clases cada vez que se crea una nueva instancia.

En Tabla 17 se detalla los mensajes que denominamos *megamórficos*. En dicha tabla se muestran aquellos mensajes que tiene 21 o más ramas de implementación.

Cant. De Ramas	Mensajes
21	*#name*
22	*#deepCopy, #at:*
23	*#release, #at:put:*
27	*#reset*
28	*#hash, #size*
33	*#copy*
47	*#=*
48	*#storeOn:*
74	*#new*
85	*#printOn:*
131	*#initialize*

Tabla 17: Selectores megamórficos

Para estos mensajes, el algoritmo va a ser más lento que **DTS** debido a la potencial cantidad de comparaciones que tendrá que hacer para encontrar una implementación.

6.4.4 Conclusión

Como todos los algoritmos de method lookup, su utilidad debe ser analizada desde tres puntos de vista: la performance, la implementación de algoritmo de lookup y el mantenimiento de las relaciones que necesita el algoritmo para funcionar. La implementación del algoritmo en lo que respecta al lookup, se puede ver en el punto 6.4.2.

La performance del algoritmo no es superior a la de **DTS**. En algunos casos es superior (como en el test case testDecompiler) y para otros es inferior (como en el test case testFileOut). Como vimos, en promedio es solo un 1,8% mejor.

Mantener la estructura que necesita este algoritmo para funcionar implica:

- Modificar las clase **Symbol** y **Behavior** de manera similar a **DTS With Clases**
- Agregar las clases **MessageImplementationBranch** y **ClassInterval**
- Mantener la numeración de las clases cada vez que se agrega o borra una clase

- Mantener las ramas de implementación por mensaje cada vez que se implementa o borra un método.

Todo esto implica un cambio bastante grande dentro de la implementación de metaprogramación de los Smalltalks actuales y según demuestran las estadísticas de tiempo, no se justifica debido a la poca diferencia de tiempo final en la ejecución.

Como única ventaja de este algoritmo, se puede decir que el tiempo de lookup de un conjunto de colaboraciones puede ser determinado fácilmente, como se mencionó en la sección 6.4.2.

6.5 Hierarchy Branch With Dictionary

6.5.1 Descripción

Este algoritmo realiza una pequeña modificación al algoritmo de **Hierarchy Branch**.

La idea se basa en acelerar la búsqueda para aquellos mensajes que poseen gran cantidad de implementaciones. Ejemplos de estos mensajes son *#new* o *#initialize*. Por ejemplo, el mensaje *#initialize* tiene 130 “ramas” distintas de implementación en la imagen que se utilizó de prueba.

Para estos mensajes, tener que realizar una búsqueda binaria y poder evitar la búsqueda en las superclases como **DTS**, termina siendo una desventaja por la gran cantidad de comparaciones que debe hacer para encontrar la implementación correcta.

Tomemos el ejemplo del mensaje *#initialize*, el cual como vimos tiene 130 “ramas” de implementación posibles. La búsqueda de la implementación de ese mensaje puede llegar a necesitar 65 comparaciones, (Recordar lo mencionado en la sección anterior sobre el orden del algoritmo) lo cual es demasiado alto, más teniendo en cuenta que casi todas las clases implementan el mensaje *#initialize* puesto que este mensaje tiene como responsabilidad “inicializar nuevas instancias”. Por ende, utilizando el algoritmo **DTS**, el resultado de buscar la implementación de *#initialize* para un objeto dado va a ser generalmente más rápido que usando **HierarchyBranch**.

Teniendo en cuenta está desventaja del algoritmo **HierarchyBranch** es que se decidió modificarlo y para aquellos mensajes que tengan un número considerable de implementaciones, modificar la lista ordenada de instancias de **MessageImplementationBranch** por un diccionario que relaciona las clases con la implementación del mensaje.

Este diccionario se parece más a una **VirtualTable** que al diccionario de mensajes utilizado por **DTS With Classes**, puesto que no contiene únicamente las clases en las cuales se implementó efectivamente el mensaje sino todas aquellas clases cuyas instancias pueden responder el mensaje. Por lo tanto, tomando como ejemplo la jerarquía de la Figura 43, para el método *m1*, este diccionario tendría como claves a las clases **A,B,C,D,E,F** puesto que el mensaje puede ser respondido por instancias de todas estas clases.

Esta decisión de diseño hace que se evite tener que realizar la búsqueda en las superclases tal cual como sucede con **DTS**.

Una desventaja de esta decisión es la cantidad de memoria que se necesita para soportar dichos diccionarios. En la sección 6.5.3 se muestra este valor.

6.5.2 Implementación

Este algoritmo está implementado en la clase **HierarchyBranchWithDicML**.

La misma es subclase de **HierarchyBranchML** y sobreescribe el mensaje *#methodLookupForSelector: aSelector inClass: aClass* de tal manera que si el selector utiliza un diccionario para almacenar sus implementaciones, realiza una búsqueda en el diccionario sino envía el mismo mensaje *#methodLookupForSelector: aSelector inClass: aClass* a super que realiza la búsqueda según el algoritmo **HierarchyBranch**.

6.5.3 Estadísticas

Se tomaron tres estadísticas de tiempo para tres configuraciones distintas. La primera (Tabla 18) utilizando el diccionario para aquellos selectores que tuvieran más de 15 ramas de implementación, la segunda (Tabla 19) para aquellos selectores que tuvieran más de 7 ramas de implementación y por último (Tabla 20) una para aquellos selectores que tuvieran más de 1 rama de implementación.

	Tiempo Total		Tiempo ML		Porcentaje
Test	Actual	% Dif. DTS	Actual	% Dif. DTS	
TestCompiler	5075.51	5.60	1130.85	7.59	22.28
TestDecompiler	1212.35	6.29	203.55	26.81	16.79
TestFileOut	8781.00	0.46	1640.36	-14.97	18.68
TestPrintHierarchy	1781.49	2.35	301.50	4.82	16.92
TestRestoreDisplay	6043.05	0.35	917.12	-4.73	15.18
TestTranscriptShow	888.49	-4.05	184.07	-0.43	20.71
Mínimo		-4.05		-14.97	15.18
Máximo		6.29		26.81	22.28
Promedio		1.84		3.18	18.43

Tabla 18: Estadísticas del Algoritmo HierarchyBranchWithDictionary para 15 ramas

Como se puede observar, a medida que se utilizan diccionarios para más selectores, se obtienen mejores tiempos. Esto se debe a que el tiempo de búsqueda en una colección ordenada para aquellos selectores que poseen muchas ramas de implementación es reemplazado por una búsqueda en un diccionario.

Además, estos diccionarios están “explotados” con todas las clases que saben responder al mensaje no solo con aquellas que lo implementan, por lo que se evita tener que hacer la búsquedas en las superclases como lo hace **DTS**. Todo esto hace que el algoritmo mejore su performance a medida que se utilicen diccionarios para más selectores.

Se puede observar, comparando los resultados con los de **HierarchyBranchML**, que utilizando diccionarios para aquellos mensajes que tengan más de 15 ramas de

implementación el algoritmo es más rápido. Lo mismo sucede comparándolo con **DTS**, en promedio el algoritmo es 3,18% más rápido.

Estos números van mejorando a medida que se van utilizando diccionarios para más selectores. Para aquellos selectores con más de 7 ramas de implementación se observa una mejora de performance del 5,47% en total y un 12,44% en el algoritmo, y para la configuración que utiliza diccionarios para aquellos selectores que tengan más de una implementación se obtienen resultados más satisfactorios (6,39% más rápida la ejecución de los test y 22,04% más rápida la ejecución del algoritmo de lookup)

	Tiempo Total		Tiempo ML		Porcentaje
Test	Actual	% Dif. DTS	Actual	% Dif. DTS	
testCompiler	5015.29	6.72	1003.21	18.02	20.01
testDecompiler	1150.58	11.07	194.34	30.12	16.91
testFileOut	8502.15	3.62	1294.39	9.28	15.22
testPrintHierarchy	1693.92	7.15	284.31	10.25	16.78
testRestoreDisplay	6088.55	-0.40	878.34	-0.30	14.43
testTranscriptShow	814.19	4.66	169.99	7.25	20.87
Mínimo		-0.40		-0.30	14.43
Máximo		11.07		30.12	20.87
Promedio		5.47		12.44	17.37

Tabla 19: Estadísticas del Algoritmo HierarchyBranchWithDictionary para 7 ramas

	Tiempo Total		Tiempo ML		Porcentaje
Test	Actual	% Dif. DTS	Actual	% Dif. DTS	
TestCompiler	4866.99	9.48	898.56	26.57	18.46
TestDecompiler	1170.96	9.49	170.99	38.51	14.60
TestFileOut	8366.83	5.15	1220.03	14.49	14.58
TestPrintHierarchy	1719.39	5.76	246.27	22.26	14.32
TestRestoreDisplay	5903.36	2.66	770.26	12.05	13.05
TestTranscriptShow	804.28	5.82	149.59	18.39	18.61
Mínimo		2.66		12.05	13.05
Máximo		9.49		38.51	18.61
Promedio		6.39		22.04	15.60

Tabla 20: Estadísticas del Algoritmo HierarchyBranchWithDictionary para 1 rama

Sin embargo, utilizar los diccionarios de esta manera hace que el consumo de memoria sea muy grande. Recordar que estos diccionarios son similares a la tablas utilizadas en la técnica de **Virtual Tables**, por ende, como se comentó en 4.4.1, el espacio de memoria que utilizan es muy grande.

En la Tabla 21 se puede observar como va creciendo la cantidad de diccionarios y entradas de diccionarios a medida que se incrementa la cantidad de selectores que los utilizan.

Cant. De Ramas	Cant. De Diccionarios	Entradas en Dicc.	Cant. De SortedCollections
15	14	6951	3489
7	88	16299	3415
1	779	55870	2724

Tabla 21: Cantidad de Diccionarios vs. Cant. de SortedCollections

6.5.4 Conclusión

Este algoritmo mejora los tiempos de **HierarchyBranchML** a costa de utilizar más memoria. Se llegó al extremo de utilizar solo el algoritmo de **HierarchyBranchML** para aquellos mensajes que tienen 1 sola implementación y utilizar diccionarios completos de búsqueda para el resto.

El mantener la estructura de objetos necesaria para que este algoritmo funcione tiene la misma complejidad que **HierarchyBranchML**, más la complejidad asociada a mantener los diccionarios de búsqueda para aquellos selectores que así lo requieran.

Los resultados obtenidos con este algoritmo debido a utilizar diccionarios de búsqueda '*a la'* **Virtual Tables** fue lo que llevó a crear el algoritmo **Full Dictionary**, cuyos resultados se muestran a continuación.

6.6 Full Dictionary

6.6.1 Descripción

Como vimos en el capitulo 4, el algoritmo de búsqueda más rápido es el de **Virtual Tables**. También se comento sobre la imposibilidad de utilizar este algoritmo en Smalltalk por la gran cantidad de memoria que consumiría. Este algoritmo busca un solución de compromiso entre **DTS** y **Virtual Tables**. A diferencia de **DTS** que hace que cada clase tenga un diccionario de los mensaje definidos únicamente en esa clase**, Full Dictioary** completa ese diccionario con todos los mensajes implementados por las superclases de la clase en cuestión.

Entonces, utilizando un decisión de diseño similar a **HierarchyBranch With Dictionaries, Full Dictionary** evita tener que realizar la búsqueda de la implementación de un mensaje en toda la jerarquía de herencia puesto que si el mismo no se encuentra en el diccionario de mensajes de la clase del objeto receptor, simplemente se asume que ese mensaje no está implementado en dicha clase.

Como es de suponer, utilizar este algoritmo para todas las clases de Smalltalk traería el mismo problema de consumo de memoria que tiene el algoritmo **Virtual Tables**, puesto que en definitiva los dos algoritmos utilizan el hecho que la clase conozca no solo las implementaciones de ella sino también la de sus superclases. Para evitar este problema, este algoritmo es utilizado solo para un conjunto de clases. La idea es utilizarlo solo para aquellas clases más criticas dentro del funcionamiento del sistema.

Otra diferencia con el algoritmo de **Virtual Tables** es que **Full Dictionary** no numera los mensajes para utilizarlos como índice dentro de una tabla sino que simplemente

utiliza un Diccionario que relaciona selectores con su implementación (objetos instancia de **CompiledMethod**), de la misma manera que lo hace **DTS**.

6.6.2 Implementación

Este algoritmo está implementado en la clase **FullDictionaryML**, quien subclasea **DispatchTableSearchML**.

Su implementación es más que sencilla. Utiliza el mismo algoritmo que **DTS**.

Se podría haber especializado el algoritmo de tal manera para que si un mensaje no es encontrado en una clase que utiliza Full Dictionary, envíe el mensaje *#doesNotUnderstand:* directamente y evitar la búsqueda en la jerarquía de herencia, sin embargo se decidió no hacerlo porque entonces por cada mensaje enviado se debería estar verificando si la clase utiliza **Full Dictionary** o **DTS** y actuar en consecuencia. Hacer esta comparación consumiría en definitiva más tiempo (se debería realizar por cada mensaje enviado) que realizar la búsqueda para aquellos mensajes no implementados, puesto que esto sucede en casos excepcionales.

6.6.3 Estadísticas

Una gran ventaja de este algoritmo es que puede ser utilizado solo en aquellas clases que se crea conveniente. Por lo tanto, se utilizaron dos configuraciones distintas para estas estadísticas. La primera utiliza este algoritmo en toda la jerarquía de clases de *Collection*, debido a que son clases muy utilizadas. La segunda agrega la jerarquía de *View* y de *Magnitud*.

	Tiempo Total		Tiempo ML		Porcentaje
Test	Actual	% Dif. DTS	Actual	% Dif. DTS	
testCompiler	4925.67	8.39	969.55	20.77	19.68
testDecompiler	1160.41	10.31	206.18	25.86	17.79
testFileOut	8330.46	5.57	1231.64	13.68	14.79
testPrintHierarchy	1717.53	5.86	283.06	10.64	16.48
testRestoreDisplay	5963.14	1.67	822.64	6.06	13.80
testTranscriptShow	832.81	2.48	177.72	3.04	21.34
Mínimo		1.67		3.04	13.80
Máximo		10.31		25.86	21.34
Promedio		5.71		13.34	17.31

Tabla 22: Estadísticas del Algoritmo FullDictionary para la jerarquía de Collection

Como se puede observar en la Tabla 22, utilizando este algoritmo solo para la jerarquía de *Collection* se obtiene una mejora del 5,71% promedio en la performance final de los test y una mejora del 13,34 % del algoritmo con respecto a **DTS**.

La diferencia de consumo de memoria se puede observar en la Tabla 23. Utilizando este algoritmo solamente para la jerarquía de *Collection*, se incrementa la cantidad de entradas del diccionario de 622 a 5621. Esto implica un aumento del uso de memoria en

97,63K (97,63K = (5621-622)*20[27]/1024). Este valor es despreciable para imagenes que ocupan cerca de 10Mg, como ser la imagen de la última versión de Squeak.

	Cantidad de Entradas
DTS	622
FullDictionary	5621

Tabla 23: Cantidad de entradas en los diccionarios de búsqueda para FullDictionary en Collection

El resultado de la segunda medida se puede observar en la Tabla 24. La performance final es en promedio un 7,43% mejor y el algoritmo mejora en un 17,95%. En la Tabla 25 se puede observar la cantidad de entradas utilizadas con esta nueva configuración, lo cual implica un total de 188,55K adicionales (188,55=(10917-1263)*20/1024).

	Tiempo Total		Tiempo ML		Porcentaje
Test	Actual	% Dif. DTS	Actual	% Dif. DTS	
TestCompiler	4967.25	7.62	920.91	24.75	18.54
TestDecompiler	1084.41	16.18	195.65	29.65	18.04
TestFileOut	8367.51	5.15	1212.39	15.02	14.49
TestPrintHierarchy	1654.91	9.29	261.61	17.42	15.81
TestRestoreDisplay	5914.48	2.47	776.99	11.28	13.14
TestTranscriptShow	820.81	3.88	165.69	9.60	20.18
Mínimo		2.47		9.60	13.14
Máximo		16.18		29.65	20.18
Promedio		7.43		17.95	16.70

Tabla 24: Estadísticas para el Algoritmo FullDictionary usado en Collection/View/Magnitud

	Cantidad de Entradas
DTS	1263
FullDictionary	10917

Tabla 25: Cantidad de entradas para los diccionarios de FullDictionary usado con Collection/View/Magnitud

6.6.4 Conclusión

Como se pudo observar de los valores obtenidos para la segunda configuración, se obtiene un promedio del 7,43% de mejora de performance final (con un máximo del 16,18%) tan solo consumiendo 188K adicionales de memoria, esto es un 1,88% de una imagen de 10 Mg.

[27] Este calculo se basa en el espacio utilizado por Squeak para cada referencia y el espacio utilizado por cada nueva entrada de un diccionario. Squeak utiliza palabras de 32 bits (4 bytes) para las referencias, y por cada entrada de un diccionario se utiliza una entrada de un array que referencia a una instancia de la clase Association, que esta compuesta por dos referencias adicionales. Por cada instancia de Association se utilizan 4 palabras (2 para el header del objeto y 2 para las variables de instancia). Como resultado, para cada nueva entrada de un diccionario se utiliza 5 palabras, o sea 20 bytes.

Este es el algoritmo más promisorio de todos los estudiados, puesto que:

- Es sencillo de implementar (usa el mismo algoritmo que DTS)
- No requiere grandes cambios en la metaprogramación de Smalltalk
- Su estructura de información es fácil de mantener
- Usándolo en algunas jerarquías importantes de Smalltalk o clases "hot spots"[28] se obtienen resultados de performance importantes
- Se puede configurar sobre qué clases se desea aplicar, por lo que...
- ... su consumo de memoria puede ser limitado

El hecho de que se puedan configurar las clases sobre las que este algoritmo se aplica, limita la desventaja del consumo de memoria de la técnica de **Virtual Tables** y permite que las aplicaciones estén configuradas para que el algoritmo de Method lookup sea más rápido para aquellas clases reconocidas como "hot spots".

6.7 Conclusión Final

En la Tabla 26 se muestra la diferencia de performance de cada algoritmo con respecto a **DTS**. Como se puede ver, el algoritmo que mejores resultados tuvo fue **FullDictionary**.

Test	DTS With Classes	Hierarchy Branch	Hierarchy Branch With Dictionay[29]	Full Dictionary
TestCompiler	6.62	-1.97	6.72	7.62
TestDecompiler	9.07	10.84	11.07	16.18
TestFileOut	6.49	-3.23	3.62	5.15
TestPrintHierarchy	5.60	5.62	7.15	9.29
TestRestoreDisplay	1.40	-0.95	-0.40	2.47
TestTranscriptShow	1.74	0.48	4.66	3.88
Mínimo	1.40	-3.23	-0.40	2.47
Máximo	9.07	10.84	11.07	16.18
Promedio	5.15	1.80	5.47	7.43

Tabla 26: Relación de performance de los algortimos de búsqueda con respecto a DTS

[28] Se denominan "Hot Spots" o punto calientes, a aquellas porciones de código o clases (en este caso) para las cuales mejorar la performance de ellos implica un gran cambio de performance en el total del sistema.
[29] Esa columna muestra los valores correspondientes a usar Hierarchy Branch With Dictionary para mensajes con más de 7 ramas de implementación.

7. Implementación de Distintas Estrategias de Caching

Este capítulo muestra la implementación de distintas estrategias de caching. Los objetivos buscados con estas distintas implementaciones fueron:

- Probar el framework comentado en el capítulo 5
- Probar distintas estrategias de caching

Para realizar la prueba de cada algoritmo, se utilizó el mismo conjunto de test cases detallado en 6.1. Cada estrategia tiene una descripción de funcionamiento, una descripción sobre la implementación, una cuadro de estadísticas y una conclusión final. Hacia el final se muestra una tabla comparativa de todas ellas.

7.1 Global Lookup Cache

7.1.1 Descripción

Esta es la implementación del algoritmo de **GLC** comentado en el capítulo 4.3.1.

Se utilizó la clase **GlobalCachingStrategy** como estrategia de caching. Para probar el comportamiento del algoritmo, se utilizaron distintas estrategias de backup, como **DispatchTableSearchML**, **DTSWithClassesML**, etc.

7.1.2 Implementación

Este algoritmo utiliza la clase **GlobalCachingStrategy** como estrategia de method lookup y de caching. La implementación del comportamiento de la cache es el mismo que utiliza Squeak.

La cache fue implementada utilizando un Array bi-dimensional. Cada fila contiene el resultado de buscar la implementación de un mensaje con el algoritmo de backup y está compuesta por las siguientes columnas:

1. **Selector**: Contiene el selector sobre el cual se realizó la búsqueda
2. **Clase**: Contiene la clase en la cual se buscó el mensaje (no dónde fue encontrado efectivamente)
3. **Método**: Contiene la instancia de CompiledMethod correspondiente a la implementación del mensaje buscado.
4. **Indice de Primitiva:** En caso de ser la implementación de este mensaje una primitiva, esta columna contiene el índice correspondiente, 0 sino.

En Figura 49 se muestra el pseudo código de búsqueda dentro de la cache. Se puede observar que primero se obtiene una fila posible donde puede estar ubicada la implementación del mensaje en cuestión. Este valor se obtiene a partir de realizar un hashing en el que participan el selector del mensaje y la clase del objeto que recibe el mensaje. Luego se verifica que la columna 1 y 2 sean iguales al selector y clase en cuestión, respectivamente. Si esta comparación resulta exitosa, se devuelve la implementación del mensaje, el cual está referenciado por la columna 3. Si no es exitosa, se intenta nuevamente con las dos filas contiguas. Si ninguna de ellas contiene el selector y clase buscados, se

devuelve *nil* como resultado de la búsqueda, lo cual indica que se produjo un "Cache Miss".

```
BuscarMetodoPara( S: Selector, C: Clase ) → Metodo

FilaEnCache ← HashEntre ( S,C )
Si la celda de la cache dada por FilaEnCache, columna 1, contiene S y
   la celda de la cache dada por FilaEnCache, columna 2, contiene C entonces
        Devolver (celda de la cache dada por FilaEnCache, columna 3.)

FilaEnCache ← FilaEnCache + 1
Si la celda de la cache dada por FilaEnCache, columna 1, contiene S y
   la celda de la cache dada por FilaEnCache, columna 2, contiene C entonces
        Devolver (celda de la cache dada por FilaEnCache, columna 3)

FilaEnCache ← FilaEnCache + 1
Si la celda de la cache dada por FilaEnCache, columna 1, contiene S y
   la celda de la cache dada por FilaEnCache, columna 2, contiene C entonces
        Devolver (celda de la cache dada por FilaEnCache, columna 3)

Devolver nil
```

Figura 49: Pseudo código de búsqueda en la cache global (GLC)

Debido a que se prueba hasta tres veces para ver si un mensaje está en la cache, el algoritmo de mantenimiento debe tener en cuenta esta característica cuando agrega elementos a la misma. A continuación se puede ver el pseudo código del mismo.

```
AgregarACache( S: Selector, C: Clase, M: Metodo, I: IndiceDePrimitiva )

'Busco una fila libre de la cache
FilaEnCache ← HashEntre ( S,C )
De 1 a 3 hacer
        Si la fila de la cache dada por FilaEnCache esta vacía entonces
                GuardarEnLaFilaDeLaCache ( S,C,M,I,FilaEnCache )
                Salir
        Sino
                FilaEnCache ←FilaEnCache + 1

'No encontré espacio libre, lo pongo en la primer fila y limpio las otras dos
FilaEnCache ← HashEntre ( S,C )
GuardarEnLaFilaDeLaCache ( S,C,M,I,FilaEnCache )

De 1 a 2 hacer
        FilaEnCache ← FilaEnCache + 1
        VaciarFilaDeLaCache ( FilaEnCache )
```

Figura 50: Pseudo Código de mantenimiento de la cache global

Como se puede observar, lo primero que hace este algoritmo es buscar una fila vacía para guardar el selector, la clase, la referencia al método y el número de primitiva encontrados en el algoritmos de backup. Para ello obtiene la fila donde debería guardar dichos valores utilizando la misma técnica que cuando se busca en la cache, realizando un hashing entre el selector y la clase. A partir de ese número de fila, busca la primer fila vacía entre las tres siguientes y guarda los valores en la primera que encuentre vacía. Si ninguna de esas tres filas está vacía, guarda los valores en la primer fila (cuyo número fue obtenido a partir del hashing realizado entre el selector y la clase) y borra las entradas de las siguientes dos filas.

Esta manera de inserción de elementos en la cache evita que la misma se llene, teniendo luego que, por medio de otro algoritmo, liberar espacio de la misma.

Como se comentó al principio de esta sección, la metodología de uso de cache fue "copiada" de la implementación de Squeak. Hay muchos factores a estudiar de esta implementación y buscar alternativas distintas de mantenimiento, los cuales están propuestos en el capítulo de "Trabajo a Futuro".

7.1.3 Estadísticas

Este algoritmo fue probado con distintos tipos de algoritmos de method lookup de backup, sin embargo como era de esperarse el comportamiento de la cache global no depende del algoritmo de backup utilizado. La diferencia de tiempos de un cache miss entre '**GLC** con **DTS'** o '**GLC** con **DTWWithClases'** es proporcional a la diferencia que existe entre **DTS** y **DTSWithClases**.

Consumo de memoria de la cache

La cantidad de filas utilizadas en la cache fue de 512. Esto implica que la misma ocupa un total de 8Kb de memoria (512 Filas * 4 Entradas por fila * 4 bytes por entrada = 8192 bytes = 8Kb).

Este valor es un 0,078% de la dimensión de las imágenes que se están utilizando actualmente en Squeak, las cuales rondan los 10Mg.

Estadísticas de GLC con DTS

Como se puede observar en la Tabla 27, la diferencia total de ejecución de los test cases con respecto al algoritmo de **DTS** es de un 16,06% mejor en promedio con un máximo del 19,33%. Esto muestra la eficacia de utilizar una cache global como estrategia de caching del algoritmo de method lookup.

También se puede observar en la misma tabla, que la mejora en performance del algoritmo de method lookup total (esto incluye caching más algoritmo de backup) es del 58,04% en promedio, con un pico del 64,94%.

	Tiempo Total		Tiempo ML		Porcentaje
Test	Actual	% Dif. DTS	Actual	% Dif. DTS	
TestCompiler	4337.22	19.33	435.50	64.41	10.04
TestDecompiler	1048.24	18.98	97.51	64.94	9.32
TestFileOut	7547.99	14.44	544.91	61.81	7.22
TestPrintHierarchy	1534.54	15.89	147.04	53.58	9.58
TestRestoreDisplay	5419.47	10.64	345.81	60.51	6.38
TestTranscriptShow	707.91	17.10	104.52	42.98	14.70
Mínimo		10.64		42.98	6.38
Máximo		19.33		64.94	14.70
Promedio		16.06		58.04	9.54

Tabla 27: Estadísticas de GLC usando DTS

En la Tabla 28 se puede observar el comportamiento de la cache. Los datos muestran resultados similares a los encontrados en la literatura que trata el tema. El cache hit es bastante alto, un 91% promedio, con un máximo del 97% y un mínimo del 78%.

También en la misma tabla se puede observar que del tiempo total utilizado por este algoritmo un 19,88% se utiliza en el algoritmo de backup para resolver los cache miss, un 41% se utiliza buscando en la cache misma y un 5,21% se pierde en el mantenimiento de la cache. El resto es utilizado por el algoritmo mismo.

Como última columna se muestra el porcentaje de cache utilizada luego de finalizar la ejecución del test case. En promedio, la cache está ocupada en un 56,27%, con un mínimo muy bajo, 26,63%.

	% Cache Hit	% Cache Miss	% Alg. Backup	% Bús. Cache	% Mant. Cache	% Cache Usada
testCompiler	96.72	3.28	12.36	47.57	2.30	62.17
testDecompiler	92.05	7.95	21.06	41.14	5.18	26.63
testFileOut	97.25	2.75	8.80	49.93	2.04	64.06
testPrintHierarchy	87.73	12.27	26.13	34.68	10.41	60.94
testRestoreDisplay	95.28	4.72	12.50	46.40	3.25	58.79
testTranscriptShow	78.38	21.62	38.41	26.31	8.10	65.04
Mínimo	78.38	2.75	8.80	26.31	2.04	26.63
Máximo	97.25	21.62	38.41	49.93	10.41	65.04
Promedio	91.24	8.77	19.88	41.00	5.21	56.27

Tabla 28: Comportamiento de la cache en GLC con DTS

Estadísticas de GLC con FullDictionary

Como se mencionó con anterioridad, el algoritmo de backup utilizado por **GLC** no influye en su comportamiento, por lo tanto se muestra a continuación únicamente cómo se comporta este algoritmo utilizando **FullDictionary** de algoritmo de backup.

El motivo por el cual se seleccionó **FullDictionary** para esta muestra es porque es el que mejor resultado dio en el estudio realizado sobre las distintas estrategias de method lookup en lo que respecta a performance.

En la Tabla 29 se puede observar los tiempos finales de usar **GLC** con **FullDictionary**. Como se puede ver, la mejora respecto a **DTS** es del 16,56%, tan solo un 0,50% mejor que utilizando **GLC** con **DTS**. Sin embargo la diferencia entre los máximo (23,80% y 19,33%) es del 4,47%.

Esta diferencia se produce para el test case denominado 'testDecompiler', que si se observa la Tabla 24 es el test case sobre el cual **FullDictionary** tuvo una mejor performance con respecto a DTS.

	Tiempo Total		Tiempo ML		Porcentaje
Test	Actual	% Dif. DTS	Actual	% Dif. DTS	
testCompiler	4531.38	15.72	444.47	63.68	9.81
testDecompiler	985.84	23.80	91.72	67.02	9.31
testFileOut	7720.35	12.48	552.89	61.25	7.16
testPrintHierarchy	1507.46	17.37	143.47	54.71	9.52
testRestoreDisplay	5446.39	10.19	354.85	59.48	6.51
testTranscriptShow	684.70	19.82	92.18	49.71	13.46
Mínimo		10.19		49.71	6.51
Máximo		23.80		67.02	13.46
Promedio		16.56		59.31	9.29

Tabla 29: Estadísticas de GLC usando FullDictionary con Collection/Magnitud/View

En la Tabla 30 se detalla el comportamiento de la cache utilizando **FullDictionary**. El resultado es similar a cuando se utiliza **DTS**, sin embargo se puede observar una diferencia del 2,7% en el tiempo consumido por el algoritmo de Backup a favor de **FullDictionary**, como era de esperarse.

	% Cache Hit	% Cache Miss	% Alg. Backup	% Bús. Cache	% Mant. Cache	% Cache Usada
TestCompiler	96.13	3.87	10.12	48.23	2.80	60.68
TestDecompiler	93.16	6.84	15.55	44.11	4.44	37.30
TestFileOut	97.01	2.99	7.78	49.81	2.20	64.30
TestPrintHierarchy	88.17	11.83	26.20	36.84	6.00	62.50
testRestoreDisplay	94.89	5.11	11.76	44.48	6.10	54.43
testTranscriptShow	78.01	21.99	31.48	30.61	9.39	58.79
Mínimo	78.01	2.99	7.78	30.61	2.20	29.30
Máximo	97.01	21.99	31.48	49.81	9.39	62.50
Promedio	91.23	8.77	17.15	42.35	5.15	50.50

Tabla 30: Comportamiento de la cache utilizando GLC con FullDictionary

7.1.4 Conclusión

Este algoritmo de caching es el más utilizado por todas las implementaciones de Smalltalk. Es sencillo, no ocupa demasiada memoria y es rápido. Como vimos anteriormente, se pensaron en otras estrategias de caching como **IC** o **PIC**, sin embargo la complejidad de implementación de las misma hace que no todos los Smalltalks las utilicen.

7.2 Selector Cache

7.2.1 Descripción

Este algoritmo utiliza una cache dedicada por selector y no una cache global. El objetivo que se buscó con este algoritmo fue comparar los resultados contra **GLC** que es la única técnica de cache de algoritmos "estáticos" que se encuentra en la literatura.

7.2.2 Implementación

La estrategia de caching está implementada en la clase **SelectorCachingStrategy**. Al igual que **GLC**, este algoritmo de caching está preparado para utilizar cualquier otro algoritmo de method lookup como backup.

Debido a las mismas limitaciones que se comentaron en **DTSWithClasses** sobre la posibilidad de modificar Symbol, se utilizó un diccionario para relacionar cada selector con su cache correspondiente.

La cache de cada selector esta compuesta por un array bi-dimensional, donde cada fila contiene el resultado de hacer la búsqueda con el algoritmo de backup y esta compuesta por las siguientes columnas:

1. **Clase**: Contiene la clase en la cual se buscó el mensaje (no donde fue encontrado efectivamente)
2. **Método**: Contiene la instancia de CompiledMethod correspondiente a la implementación del mensaje buscado.
3. **Indice de Primitiva:** En caso de ser la implementación de este mensaje una primitiva, esta columna contiene el índice correspondiente, 0 sino.

Como se puede ver, a diferencia de **GLC** que posee cuatro columnas por entrada, este algoritmo reduce a tres las columnas por entrada debido a que el selector ya está discriminado.

El algoritmo de búsqueda dentro de la cache es similar al de **GLC** con la diferencia que solo se realiza una sola prueba. Si no se encuentra la clase para la cual se está realizando la búsqueda en la primera entrada, directamente se devuelve un cache miss. Esta decisión se tomó debido a la escasa dimensión de la cache.

7.2.3 Estadísticas

Las estadísticas que se presentan a continuación utilizan como algoritmo de backup a **DTS**. Como dijimos en el punto anterior, la utilización de distintos algoritmos de backup no modifican el comportamiento de la cache.

	Tiempo Total		Tiempo ML		Porcentaje
Test	Actual	% Dif. DTS	Actual	% Dif. DTS	
testCompiler	4894.01	8.98	554.76	54.67	11.33
testDecompiler	1069.04	17.37	140.01	49.66	13.08
testFileOut	8211.97	6.91	675.40	52.66	8.22
testPrintHierarchy	1600.73	12.26	156.93	50.46	9.80
testRestoreDisplay	5947.19	1.94	461.30	47.32	7.76
testTranscriptShow	750.59	12.10	101.95	44.38	13.59
Mínimo		1.94		44.38	7.76
Máximo		17.37		54.67	13.59
Promedio		9.93		49.86	10.63

Tabla 31: Estadísticas de Selector Cache con DTS

Como se puede observar, los resultados no superan a **GLC**. En promedio se obtuvo una mejora de performance del 9,93% sobre **DTS**.

También se puede observar que el comportamiento de la cache es muy dispar con respecto a **GLC**. Por empezar hay menos cache hits, un 88% en promedio. Esto se debe a que **SelectorCache** solo realiza un intento cuando está buscando en la cache.

Otro punto interesante es que los algoritmos probados solo llegan a utilizar un 32% de la cache. Este es un dato interesante a tener en cuenta para aumentar el cache hit y disminuir el espacio de memoria utilizado.

	% Cache Hit	% Cache Miss	% Alg. Backup	% Bús. Cache	% Mant. Cache	% Cache Usada
testCompiler	91.58	8.42	20.56	42.26	3.30	34.34
testDecompiler	86.15	13.85	28.46	32.72	4.40	30.29
testFileOut	96.04	3.96	12.40	47.81	2.37	32.67
testPrintHierarchy	84.22	15.78	31.12	33.84	4.85	31.91
testRestoreDisplay	92.85	7.15	21.82	41.49	2.70	32.61
testTranscriptShow	78.28	21.72	40.54	27.91	5.29	31.67
Mínimo	78.28	3.96	12.40	27.91	2.37	30.29
Máximo	96.04	21.72	40.54	47.81	5.29	32.67
Promedio	88.19	11.81	25.82	37.67	3.82	32.25

Tabla 32: Comportamiento de la cache de SelectorCache

En la Tabla 33 se puede ver para cada test case la distribución de uso de las caches de los selectores. En promedio, hay 161 selectores cuya cache solo esta ocupada por una entrada, y solo 3 selectores (en promedio) que utilizan la cache en su totalidad (o sea, las cuatro entradas).

Esto tiene mucho que ver con la cantidad de implementaciones que tiene un mensaje. Aquellos mensajes que tengan una sola implementación, son candidatos a tener menos entradas en la cache (no se puede asegurar que sea 1 (una) debido a que el mensaje

puede ser enviado a subclasses de la clase donde está definido) y aquellos mensajes 'megamórficos' son candidatos a que se queden cortos con una cache de 4 entradas. Esta información podría ser utilizada para mejorar el espacio de memoria ocupado por este algoritmo y mejorar el cache hit.

El espacio de memoria consumido por este algoritmo en una imagen actual de Squeak sería de 709 Kb (Cantidad de mensajes*Dimensión de Cache en Bytes = 15128 * (3 * 4 * 4) Bytes = 15128*48 bytes = 726144 bytes = 709.125 Kb). Esto equivaldría a un 7,9% de la imagen total de versiones actuales de Squeak.

	1 Entrada	2 Entradas	3 Entradas	4 Entradas
testCompiler	111.67	22.67	10.33	2.67
testDecompiler	135.00	22.00	12.00	1.00
testFileOut	72.33	23.00	2.00	1.67
testPrintHierarchy	230.00	40.00	15.00	10.00
testRestoreDisplay	179.00	26.00	12.00	0.00
testTranscriptShow	238.00	29.00	16.00	5.00
Mínimo	72.33	22.00	2.00	0.00
Máximo	238.00	40.00	16.00	10.00
Promedio	161.00	27.11	11.22	3.39

Tabla 33: Distribución de uso de la cache por test case

Como se puede deducir, el consumo de memoria es inaceptable, por lo que habría que utilizar técnicas para disminuir el espacio utilizado. Una posibilidad sería no crear la cache de un mensaje hasta que el mismo sea enviado. De esta manera solo tendrían cache aquellos mensajes que son realmente utilizados, disminuyendo drásticamente el espacio de cache necesario.

En la Tabla 34 se puede ver cual sería el espacio de memoria utilizado por este algoritmo, de utilizar esta técnica, para cada test case.

	Num. Caches	Kb totales
testCompiler	147	6.91
testDecompiler	170	7.97
testFileOut	99	4.64
testPrintHierarchy	295	13.83
testRestoreDisplay	217	10.17
testTranscriptShow	288	13.50

Tabla 34: Espacio de Memoria utilizado por la cache para c/test case

7.2.4 Conclusión

Este algoritmo de caching no logra superar la performance obtenida por **GLC** y además necesita más espacio de memoria para trabajar.

Utilizando la técnica mencionada en el punto anterior para disminuir el consumo de memoria, se estaría dentro de valores de memoria aceptables. Al mismo tiempo, si para

cada mensaje se estudia cual sería la dimensión de cache apropiada, se podría mejorar el cache hit y disminuir aún más el uso total de memoria.

7.3 Hierarchy Branch Cache

7.3.1 Descripción

Este algoritmo funciona únicamente con el algoritmo **HierarchyBranchML** como backup y utiliza una cache dedicada por selector al igual que **SelectorCache**.

Aprovecha la característica de búsqueda que ofrece **HierarchyBranchML**, de tal manera que una entrada en la cache contiene el intervalo de clases para las cuales la implementación del mensaje es válido.

7.3.2 Implementación

Este algoritmo tiene varias implementaciones dependiendo de la técnica de mantenimiento de cache utilizado. Sin embargo, todas estas implementaciones subclasifican la clase abstracta **HBCachingStrategy.**

Debido a las mismas limitaciones que se comentaron en **DTSWithClasses** sobre la posibilidad de modificar *Symbol*, se utilizó un diccionario para relacionar cada selector con su cache correspondiente.

La cache de cada selector esta compuesta por un array bi-dimensional, donde cada fila contiene el resultado de hacer la búsqueda con el algoritmo de backup y está compuesta por las siguientes columnas:

1. **From**: Contiene el número **from** del objeto **MessageImplementationBranch** encontrado por el algoritmo **HierarchyBranchML.**
2. **To:** Contiene el número **to** del objeto **MessageImplementationBranch** encontrado por el algoritmo **HierarchyBranchML.**
3. **Método**: Contiene la instancia de **CompiledMethod** correspondiente a la implementación del mensaje buscado.
4. **Indice de Primitiva:** En caso de ser la implementación de este mensaje una primitiva, esta columna contiene el índice correspondiente, 0 sino.

Cuando se desea verificar si la implementación de un mensaje se encuentra en una entrada de la cache, se verifica que el número de la clase del objeto receptor se encuentre entre el **from** y el **to** de la entrada de la cache correspondiente al mensaje.

Para verificar si un mensaje se encuentra en la cache, se realiza la comparación indicada en el párrafo anterior para cada entrada de la cache de manera secuencial hasta llegar al final de la cache o se encuentre una entrada vacía.

Como se mencionó al principio, con este algoritmo se probaron distintas técnicas de mantenimiento de cache.

La primera borra toda la cache cuando la misma está llena y se desea agregar una nueva entrada. Luego de borrar toda la cache, pone la nueva entrada en la primer posición.

El algoritmo de caching que utiliza esta técnica está implementado en **HBEraseAllCachingStrategy.**

La segunda reutiliza entradas ya ocupadas, y las selecciona de manera circular, es decir, la primera vez utiliza la posición 1, luego la 2, luego la 3 hasta llegar a la mayor posición de la cache, en cuyo caso empieza nuevamente desde la primer posición. Esta técnica está implementada en la clase **HBCacheCircularAddCachingStrategy.**

La tercer técnica utiliza una cache de una única entrada evitando así tener que realizar un mantenimiento especial. Esta técnica está implementada en **HBOneEntryCachingStrategy.**

7.3.3 Estadísticas

De todas la técnicas de mantenimiento y dimensiones de cache probadas, la que mejor resultado dio fue **HBCacheCircularAddCachingStrategy** con una dimensión de tres entradas por mensaje.

Como se puede ver en Tabla 35 el algoritmo llega a mejorar en un 10% con respecto a **DTS** en promedio, con un máximo del 20%. Aunque es un porcentaje de mejora interesante, no alcanza para mejorar la performance de **GLC**, que tiene una promedio del 16% mejor.

	Tiempo Total		Tiempo ML		Porcentaje
Test	Actual	% Dif. DTS	Actual	% Dif. DTS	
TestCompiler	4828.79	10.19	515.56	57.87	10.67
TestDecompiler	1035.15	19.99	105.15	62.19	10.16
TestFileOut	8244.70	6.54	694.71	51.31	8.43
TestPrintHierarchy	1604.32	12.06	162.42	48.73	10.13
TestRestoreDisplay	5961.23	1.70	440.85	49.66	7.40
TestTranscriptShow	751.97	11.94	103.63	43.46	13.78
Mínimo		1.70		43.46	7.40
Máximo		19.99		62.19	13.78
Promedio		10.40		52.20	10.09

Tabla 35: Comportamiento del algoritmo HierarchyBranchCache utilizando técnica Circular Add

Lo que es interesante destacar es que este algoritmo tiene mejor performance que **Hierarchy Branch With Dictionary** y además necesita menos espacio de memoria para funcionar, por lo que es la mejor alternativa a ser utilizado con el algoritmo de **Hierarchy Branch**.

En la Tabla 36 se puede observar el comportamiento de la cache. Hay un promedio de cache hit del 93%, con un máximo del 97%. Estos números son un poco mejores que **GLC**, que en promedio tiene un 91% de cache hit. Sin embargo, aunque el porcentaje de cache hit sea mejor, el tiempo de búsqueda en la misma es muy alto, un 47% contra un 41% del **GLC**, lo cual demuestra que el algoritmo de búsqueda en cache de **GLC** es mejor que la utilizada por este algoritmo.

	% Cache Hit	% Cache Miss	% Alg. Backup	% Bús. Cache	% Mant. Cache
TestCompiler	97.16	2.84	7.36	53.63	3.04
TestDecompiler	94.72	5.28	10.69	49.62	4.98
TestFileOut	97.51	2.49	7.31	53.54	2.65
TestPrintHierarchy	91.01	8.99	15.58	40.82	7.36
TestRestoreDisplay	95.74	4.26	10.67	50.43	4.41
TestTranscriptShow	85.21	14.79	20.32	36.70	10.28
Mínimo	85.21	2.49	7.31	36.70	2.65
Máximo	97.51	14.79	20.32	53.54	10.28
Promedio	93.56	6.44	11.99	47.46	5.45

Tabla 36: Comportamiento de la cache para alg. HierarchyBranch con técnica Circular Add

En la Tabla 37 se puede observar cuantas entradas están ocupadas en la cache para cada test case y en Tabla 38 se puede observar el espacio de memoria utilizado por la cache.

El espacio de memoria utilizado por este algoritmo no es muy superior al de **GLC**. **GLC** necesita 8K para la cache, mientras que este utilizó un máximo de 10K.

	1 Entrada	2 Entradas	3 Entradas
TestCompiler	57.00	11.00	6.33
TestDecompiler	144.00	16.00	10.00
testFileOut	78.00	13.33	3.67
testPrintHierarchy	252.00	17.00	26.00
testRestoreDisplay	187.00	20.00	6.67
testTranscriptShow	249.00	21.00	18.00
Mínimo	57.00	11.00	3.67
Máximo	252.00	21.00	26.00
Promedio	161.17	16.39	11.78

Tabla 37: Distribución de uso de la cache por test case

	Num. Caches	Kb totales
TestCompiler	74	2.61
TestDecompiler	170	5.98
TestFileOut	95	3.34
testPrintHierarchy	295	10.37
testRestoreDisplay	214	7.51
testTranscriptShow	288	10.13

Tabla 38: Memoria utilizada por la cache para c/ test case

7.3.4 Conclusión

Una desventaja de esta algoritmo de caching es que no se puede realizar hashing a partir de la clase y el mensaje de tal manera que permita buscar rápidamente en la cache. Este problema se debe a que cada entrada está compuesta por un rango de clases para las

cuales el mensaje es válido. Debido a este problema, la búsqueda dentro de la cache se realiza de manera secuencial, lo cual toma mucho tiempo.

Aunque los resultados son mejores que **DTS**, no logra mejorar la performance obtenida por **GLC**.

7.4 Conclusión Final

Finalmente, el algoritmo de **GLC** probó ser el más adecuado como mecanismo de caching para lenguajes de objetos dinámicos como Smalltalk.

Test	GLC	Selector Cache	Hierarchy Branch Cache
testCompiler	19.33	8.98	10.19
testDecompiler	18.98	17.37	19.99
testFileOut	14.44	6.91	6.54
testPrintHierarchy	15.89	12.26	12.06
testRestoreDisplay	10.64	1.94	1.70
testTranscriptShow	17.10	12.10	11.94
Mínimo	10.64	1.94	1.70
Máximo	19.33	17.37	19.99
Promedio	16.06	9.93	10.40

Tabla 39: Comparación de las estrategias de Caching con respecto a DTS

Sin embargo, más allá de lo que probamos como performance y comportamiento de cache para cada algoritmo, cabe destacar que el framework utilizado soportó distintos tipos de algoritmos y está preparado para soportar otras técnicas de caching como de profiling.

En Tabla 39 se puede observar la comparación de todos los algoritmos de caching utilizados.

8. Conclusiones Finales

La programación con Objetos tiene grandes ventajas sobre otros paradigmas de programación. Uno de ellos, sino el principal, es el uso de polimorfismo para disminuir la complejidad de los modelos implementados. Sin embargo, como vimos a lo largo de este trabajo, el polimorfismo no es gratis y su implementación, a través del algoritmo de *method lookup* puede hacer usable o no a un sistema.

Este trabajo realizó primero una investigación sobre las distintas implementaciones de method lookup y categorizó a los mismos en estáticos y dinámicos. Para los ambientes de programación como Smalltalk, se mostró que los estáticos son impractibles debido justamente a su característica rígida y al consumo de tiempo impuesto para su mantenimiento.

Dentro de los algoritmos dinámicos, los más rápidos son aquellos que utilizan cache locales y modificables según el comportamiento del programa como **PIC**. Esta técnica es, en la actualidad, la más utilizada en las implementaciones comerciales de Smalltalk.

El trabajo continuó con la implementación de un framework de *method lookup* desarrollado con Squeak. Este framework de despacho de mensajes tenía por objetivos brindar un ambiente de prueba de distintos algoritmos de lookup y caching, como así también la posibilidad de obtener estadísticas sobre los mismos. Estos objetivos se cumplieron y fueron mostrados más adelante por las distintas implementaciones de los algoritmos realizados.

El framework demostró ser fácilmente instanciable para distintos tipos de algoritmos de búsqueda y de caching. Se logró un buen grado de polimorfismo, de tal manera que un algoritmo de caching puede ser visto como una implementación especial de un algoritmo de *method lookup*. También los distintos profilers pueden ser intercambiados sin ningún tipo de problema.

El framework permite tomar distintos tipos de medidas indicativas del comportamiento de los algoritmos. Inicialmente, las medidas que tomamos en este trabajo de investigación se concentraron en la problemática de la performance y el uso de memoria.

El framework permite modificar el algoritmo de lookup utilizado dinámicamente, o sea, mientras se está ejecutando el programa. También permite especificar algoritmos distintos determinados por clase.

La implementación de este framework permitió obtener conclusiones interesantes con respecto a Squeak. Se pudo comprobar la veracidad de la afirmación realizada por los integrantes de grupo creador de Squeak sobre la posibilidad de programar en Smalltalk la misma Virtual Machine y de correr un Squeak dentro de otro. Esta característica disminuyó bastante los tiempos necesarios para realizar los algoritmos y las pruebas presentadas en este trabajo.

Sin embargo, la implementación de la VM escrita en Smalltalk deja mucho que desear a nivel diseño y programación debido a las limitaciones que ofrece el conversor de código Smalltalk a código C. Dentro de este campo hay mucho por mejorar, y dichas mejoras serían vistas con buenos ojos dentro de la comunidad Squeak. Una de las mejoras que se podría realizar es independizar el manejo de memoria de la ejecución del byte code propiamente dicho, que actualmente se encuentran muy acoplados. Esto permitiría investigar sobre distintos algoritmos de *garbage collection* o sobre distintos tipos de byte code sin interferir el uno con el otro.

Luego de implementado el framework, se pasó a implementar distintos algoritmos de *method lookup* que permitieron en algunos casos constatar la veracidad de la bibliografía estudiada y obtener nuevos datos interesantes.

Se demostró en primera instancia que el algoritmo **DTS With Classes,** el cual trabaja de manera inversa a **DTS**, tiene un 5% de mejora en performance que este último, dato que sorporendió bastante debido a que las implementaciones de todos los Smalltalk conocidos utilizan **DTS** como algoritmo básico de búsqueda.

Se desarrollo un algoritmo muy fácil de implementar (**FullDictionary**) y que no consume demasiados recursos, que supera en performance tanto a **DTS** como **DTS With Classes**. Este algoritmo se basa en polular los diccionarios de mensajes de ciertas clases comúnmente utilizadas con todos los mensajes implementados en las clases de la jerarquía de herencia correspondiente (característica similar al algoritmo **Virtual Table**). Las mejoras de performance con respecto a **DTS** alcanzó en las configuración probada a un 17,95%.

Dentro de los algoritmos probados, la gran decepción fue el algoritmo denominado **HierarchyBranch** y sus derivados. Pesnabamos que iba a superar en performance a **FullDictionary** y por consiguiente a **DTS**, aunque en la práctica demostró tener serios problemas de performance y recursos que lo hacen impracticable.

También se probaron varios algoritmos de caching, algunos direccionados completamente a mejorar la performance de algunos algoritmos de búsqueda específicos como **Hierarchy Branch Cache**. Sin embargo, de las pruebas realizadas, **GLC** demostró una vez más ser el algoritmo más rápido y conveniente por el poco uso de memoria que realiza. Los algoritmos relacionados a cachear únicamente por mensaje, como **Selector Cache**, no lograron superarlo.

En conclusión, este trabajo provee la implementación de un framework de *method lookup* que puede ser utilizado para realizar más investigación o profundizar la aquí presentada y la implementación de varios algoritmos que pueden ser utilizado como base de futuros trabajos. En lo que respecta a algoritmos de method lookup, este trabajo encontró una implementación, **FullDictionary**, que supera en performance a la implementación más utilizada, **DTS**.

9. Trabajo a Futuro

Es mucho lo que se puede hacer relacionado al tema de *method lookup*. Es nuestra creencia que el programador debería tener más control sobre el algoritmo de *method lookup*, de tal manera que puede elegir con cual desea trabajar, cómo desea configurarlo, etc. Algo similar propone Johnson en [Foote et al.]

Por lo tanto, un punto a ampliar a partir de esta tesis, es "abrir" el *method lookup* al programador, permitir que lo pueda configurar para situaciones específicas, como por ejemplo tener un tiempo de lookup constante (útil para sistemas real-time), mejorar la performance para un conjunto de clases importantes del sistema (como se probó con **FullDictionary**) o utilizar distintos algoritmos no provistos por el ambiente.

Para ello es necesario modificar la VM de tal manera que no asuma la existencia de un único algoritmo de lookup (tal cual como se realizó en este trabajo de investigación). Esta modificación debería ser complementada con una ampliación al metamodelo de tal manera que permita mantener fácilmente las estructuras de los algoritmos de lookup utilizados cada vez que se modifican métodos, clases, jerarquías, etc.

"Abrir" el algoritmo de *method lookup* ayudaría a mejorar los resultados que se pueden obtener a través de metaprogramación. Por ejemplo, se ha mostrado en [Bour98] que tener un despacho de mensajes especializado por clase es una gran ventaja y permite que características de los objetos puedan ser controladas por medio del algoritmo de despacho de mensajes. Ejemplos concretos de utilización serían:

- Se podría implementar un algoritmo que para cada mensaje enviado, automáticamente se envíe un mensaje indicando que se esta por ejecutar el mensaje original y luego enviar otro indicando que se ejecutó el mensaje original. Esto permitiría:
  - Realizar un *tracing* de los mensajes recibidos por un objeto (Sería tan sencillo como grabar en un stream el objeto y el mensaje enviado)
  - Implementar varios patterns como el Singleton (el mensaje new debería devolver siempre la misma instancia), el Proxy (se deberían delegar todos los mensajes el objeto real), etc.
- Se podrían implementar algoritmos de despacho múltiple, lo cual facilitaría la programación de objetos "subjetivos"

Otro tema de investigación que se puede ver beneficiado con mayores estudios sobre *method lookup* es Subjetividad. Subjetividad permite que los objetos tengan comportamiento diferente dependiendo del contexto con el que se relacionan. Tener un algoritmo de despacho múltiple de mensajes es uno de los requisitos esenciales de los ambientes de Subjetividad como se muestra en [Alencar] y [Smith et. al].

Por lo tanto, el despacho múltiple de mensajes es un tema de ampliación de esta tesis. El despacho múltiple tiene en cuenta, para determinar qué método debe ejecutarse, no solo al objeto que recibe el mensaje sino también al emisor y a los colaboradores que el

mensaje utiliza. El framework creado permitiría fácilmente implementar un algoritmo de tales características para ser utilizado en un ambiente de "*objetos subjetivos*", por lo tanto este es otro punto donde este trabajo de investigación puede ser ampliado.

Este trabajo no profundizó la investigación con respecto a la técnica de caching utilizado por el algortimo de **GLC** implementado. Este aspecto merece mayor investigación para tratar de lograr un comportamiento de cache más rápido y un mejor mantenimiento de la misma.

La implementación de los algoritmos realizados en esta investigación encontraron varios problemas relacionados a la representación de ciertas clases, como la clase Symbol y Behavior. Estos problemas fueron resueltos momentaneamente para continuar con la investigación, sin embargo sería conveniente dedicar tiempo a la modificación de la VM y de ciertas clases como Symbol, de tal manera que permitan la implementación directa de estos algoritmos. Hacer esto implica más tiempo y dedicación que investigación puesto que existen actualmente, los mecanismo para poder realizarlo.

El framework implementado tiene un fuerte acoplamiento entre los objetos que representan los algoritmos de lookup y los profilers utilizados para medir su comportamiento. Este acoplamiento se debe a que era objetivo de esta investigación obtener mediciones sobre performance y uso de memoria de los distintos algoritmos implementados. Sin embargo dicho acoplamiento podría disminuirse por medio de la utilización del pattern denominado Proxy o por medio de metaprogramación.

Mejorar la implementación de la VM escrita en Smalltalk es otros de los posibles trabajos a realizar que además sería visto con buenos ojos dentro de la comunidad Squeak. La implementación actual deja mucho que desear a nivel diseño y programación debido a las limitaciones que impone el conversor de código Smalltalk a código C.

Una de las mejoras que se podría realizar es independizar el manejo de memoria de la ejecución del byte code propiamente dicho, que actualmente se encuentran muy acoplados. Esto permitiría investigar sobre distintos algoritmos de *garbage collection* o sobre distintos tipos de *byte code* sin interferir el uno con el otro.

10. Referencias

[Agesen et al.] O. Agesen, J. Palsberg, M. Schwartzbach, *Type Inference of Self: Analysis of Objects with Dynamic and Multiple Inheritance*, ECOOP'93

[Alencar] P.S.C. Alencar, D.D. Cowan, C.J.P. Lucena, *"A Formal Model to Support Subject-Oriented Programming"*, OOPSLA'95

[Andre et al.] P. André y J.-C. Royer. *Optimizing Method Search with Lookup Caches and Incremental Coloring*. 1992, ACM OOPSLA '92 Conference Proceedings.

[Auslander et al] J Auslander, M. Philipose, C. Chambers, S. Eggers y B. Bershad, *Fast, Effective Dynamic Compilation*, 1996 ACM PLDI'96.

[Bracha et al.] G. Bracha, D. Griswold, *Strongtalk: Typechecking Smalltalk in a Production Evironment*, 1993 ACM OOSPLA'93

[Booch] Booch, *Object-Oriented Analisys and Desing with Applications*, Second Edition, Addison-Wesley.

[Bour98] N. Bouraqadi-Saadani, T. Ledoux, F. Rivard, *"Safe Metaclass Programming"*, OOSPLA '98

[Brooks] F. Brooks, *The Mythical man-Month*, 20th Aniversary Edition, Addison-Wesley, 1996.

[Callegari et al] C. Callegari, Eliaschev L, Tanner C. *Subjetividad en un Ambiente de Objetos*. Tesis de Licenciatura, Universidad de Buenos Aires.

[Chambers] C. Chambers, *Object-Oriented Multi-Methods in Cecil*. ECCOP'92, Conference Preceedings.

[Dahl et al.] O. Dahl, B. Myrhaug, *Simula Implementation Guide*, 1973, Publication S 477, NCC

[Dean et al.] J. Dean y C. Chambers, *Towards Better Inlining Decisions using Inlining Trials*, 1994 ACM LISP'94.

[Dean et al2] J. Dean, G. DeFouw, D. Grove, V. Litvinov y C. Chamber, *Vortex: An Optimizing Compiler for Object-Oriented Languages*, 1996 ACM OOPSLA'96

[Denckery et al.] P. Denckery, K. Durre, J Heuft, *Optimization of Parser Tables for Portable Compilers*, 1984, TOPLAS 6(4): Páginas 546-572

[Deutsch et al.] L. Deutsch, A. Schiffman. *Efficient Implementation of the Smalltalk-80.* Proceedings del 11er. Symposium on the Principles of Programming Languages,

[Dixon et al.] R. Dixon, T. McKee, P. Schweitzer, y M. Vaughan. *A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance*. 1989, ACM OOPSLA '89 Conference Proceedings, pp. 211-214.

[Dolby et al.] J. Dolby y A. Chien, *An Evaluation of Automatic Object Inline Allocation Techniques*, 1998 ACM OOPSLA'98

[Driesen] K. Driesen, *Multiple Dispatch Techniques: a survey*, University of Santa Barbara, California

[Driesen2] Karel Driesen, *Software and Hardware Thechniques for Efficient Polymorphic Calls*, PhD Thesis, University Of California, Santa Barbara.

[Driesen3] K. Driesen, *Method Lookup Strategies in Dynamically Typed Object-Oriented Programming Languages*. Master's Thesis, Vrije Universiteit Brussel, 1993.

[Driesen et al.] K. Driesen, U. Holzle, J. Vitek, *Message Dispatch on Pipelined Processors*, ECOOP'95

[Driesen et al2] K. Driesen, U. Holzle, *The Direct Cost of Virtual Function Calls in C++*, 1996 ACM OOSPLA Conference.

[Driesen et al3] K. Driesen, U. Holzle, J. Vitek, *Thecnical Report TRCS 94-620*, 1994, University of California.

[Ernst et al.] M. Ernst, C. Kaplan, C. Chambers. *Predicate Dispatching: A Unified Theory of Dispatch*. ECOOP'98, Conference Proceedings.

[Falcone] J. Falcone, *The Analysis of the Smalltalk-80 System at Hewleet-Packard*. En [Krasner], páginas 207-237.

[Foote et al.] B. Foote, R. Johnson, *Reflectives Facilities in Smalltalk-80*

[Fowler] Fowler, *Analysis Patterns, Reusable Object Models*, Addison-Wesley

[GOF] E. Gamma, J. Vlissides, Jonhson, Helm, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[GOF2] S. Alpert, K. Brown, B Woolf, *The Design Patterns Smalltalk Companion*, Addison-Wesley, 1998

[Goldberg1] Adele Goldberg y David Robson. *Smalltalk-80: The Language*. Second Edition, Addison-Wesley Publishing Company

[Goldberg2] Adele Goldberg, David Robson, *Smalltalk-80: The Language and its Implementation*, Capítulos: 28 al 31, First Edition, Addison-Wesley Publising Company.

[Gosling] A. Gosling, *The Java Programming Language*, Java Series, Addison-Wesley.

[Gosling et al.] A. Gosling, J. Steele, *The Java Language Specification*, Java Series, Addison-Wesley

[Holzle] U Holzle, D. Ungar, *Do Object-Orinted Languages Need Special hardware Support*, ECOOP'95

[Holzle et al.] U. Holzle, C. Chambers, D. Ungar, *Optimazing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches*, ECCOP'91

[Holzle et al2] U. Holzle, D. Ungar, *Optimizing Dynamically-Dispatched Call with Run-Time Type Feedback*, SIGPLAN 1994.

[Ingalls] D. Ingalls, *Design Principles Behind Smalltalk*, Agosto 1981, BYTE Magazine, The McGraw-Hill Companies, Inc.

[Ingalls2] D. Ingalls, *A Simple Technique for Handling Multiple Polymorphism*, OOSPLA'86, Conference Preceedings.

[Ingalls et al] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, A. Kay, *Back to the Future: The Story of Squeak*, A practical Smalltalk Written in Itself, Apple Computer, Walt Disney Imagineering.

[Kiczalez] G. Kiczales, *CSC-517 Object Oriented Languages and Systems*, North Carolina State University, Fall 1996, Notas del curso.

[Kiczales97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes,Jean-Marc Loingtier, John Irwin, *"Aspect-Oriented Programming"*, ECOOP 97

[Kiczalez et. al] G. Kiczales y L. Rodriguez, *Eficient Method Dispatch in PCL*, 1990 ACM Conference on Lisp and Functional Programming. Pages 99-105.

[Kiczalez et al2] G. Kiczales, J. des Rivieres, D. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991

[Krasner] Gleen Krasner, *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, 1983.

[Leavens et al.] G. Leavens, T. Millstein, *Multiple Dispatch as Dispatch on Tuples*, 1998 ACM OOSPLA Conference.

[Lindholm et al.] Lindholm, Yelling, *The Java Virtual Machine Specification*, Java Series, Addison-Wesley.

[Martin et al.] R. Martin, D. Riehle, F. Buschmann, *Pattern Languages of Program Design 3*, Addison Wesley, 1998

[Mendhekar] A. Mendhekar, G. Kiczales, J. Lamping, *Compilation Strategies as Object*, 1994 ACM OOPLSA'94.

[Meyer] B. Meyer, *Object-Oriented Software Construction*, Second Edition, Prentince Hall PTR

[Meyers1] Meyers, *Effective C++*, Addison-Wesley.

[Meyers2] Meyers, *More Effective C++*, Addison-Wesley.

[Paepcke] A. Paepcke, *Object-Oriented Programming: The CLOS Perspective*, MIT Press, 1993.

[Parmas] D. Parmas, *On the Cirteria to Be used in Decomposing Systems into Modules*, en Classics in Software Engineering, Yourdon Press, 1979.

[Piumarta] I. Piumarta, *Delayed Code Generation in a Smalltalk-80 Compiler*, Thenical Report Series UMCS-93-7-1, University of Manchester

[Sessions] R. Sessions, *Object Persistence*, Prentince Hall PTR

[Smith et. al] R. Smith, D. Ungar, "*A simple and Unifying Approach to Subjective Objects*", Sun Microsystems Laboratories, Inc.

[Smith et al.2] R. Smith, D. Ungar, *Programming as an Experience: The Inspiration for Self*, Sun Microsystems Laboratories.

[Stein et. al] L. Stein, H. Liberman, D. Ungar, *A Shared View of Sharing: The Treaty of Orlando*

[Stroustrup1] B. Stroustrup, *The C++ Programming Language*, Second Edition, Addison-Wesley

[Stroustrup2] B. Stroustrup, *The Design and Evolution of C++*, Addison-Wesley.

[Taylor] Taylor, *Object-Oriented Technology: A manager´s Guide*, Addison-Wesley.

[Ungar] D. Ungar. *The Desing and Evaluation of a High-Performance Smalltalk System*. PhD Thesis, MIT Press, 1987.

[Ungar et al.] D. Ungar, R. Smith, C. Chambers, U. Holzle, *Object, Message, and Performance: How they coexist in Self*, Octubre 1992, IEEE Computer

[Ungar et al2] D. Ungar, C. Chambers, B. Chang, U. Holzle, *Organizing Programs Without Classes*, Lisp And Symbolic Computations: An Internations Journal, 4, 3, 1991

[Ungar et al3] D. Ungar, C. Chambers, B. Chang, U. Holzle, *Parents are Shared Parts of Objects: Inheritance and Encapsulation in SELF*, Lisp And Symbolic Computations: An Internations Journal, 4, 3, 1991

[Ungar et al4] D. Ungar, D. Patterson. *What Price Smalltalk?*, en IEEE Computer 20(1), Enero de 1987

[Vitek1] J. Vitek, *Compact Dispatch Tables for Dynamically Typed Programming Languages*, Technical Report, University of Geneva, Centre Univesitaire d'Informatique.

[Vitek2] J. Vitek, *Compact Dispatch Tables for Dianymically Typed Programming Languages*, Master Thesis, University of Geneva, Centre Univesitaire d'Informatique.

[Vitek et al.1] J. Vitek y R. Horspool, *Compact Dispatch Tables for Dianymically Typed Programming Languages*, University of Geneva, Centre Univesitaire d'Informatique. Dept. of Cumputer Science, Univ. of Victoria.

[Vitek et al.2] J. Vitek y R. Horspool, *Taming Message Passing: Efficient Method Look-Up for Dynamically Typed Languages*, ECOOP'94.

[Vlissides] J. Vlissides, *Pattern Hatching: Design Patterns applied*, Addison-Wesley, 1998

2 Apéndice A – Notación Gráfica Utilizada

La notación de diseño utilizada es la definida por el **OMG** (Object Management Group) que se denomina **UML** (Unified Modeling Language). La espcificación de la misma puede ser obtenida en www.omg.org

En este capítulo daremos una pequeña reseña de los diagramas utilizados en este trabajo, para que pueda ser entendido sin necesidad de leer la especificación. Se asume el conocimiento del lector sobre los elementos que definen el paradigma de objetos como ser objeto, clase, herencia, colaboración, polimorfismo, etc. Los mismo se encuentran definidos en los capítulos 2 y 3 de este trabajo.

10.1 Diagrama de Paquetes

Un paquete denota un conjunto de clases. Generalmente las clases de un mismo paquete modelan el mismo problema conceptual. Los paquetes pueden tener dependencias entre sí, esto significa que clases de un paquete colaboran directa o indirectamente con clases del otro paquete.

A continuación se muestra la representación gráfica utilizada para los paquetes y la relación entre los mismos.

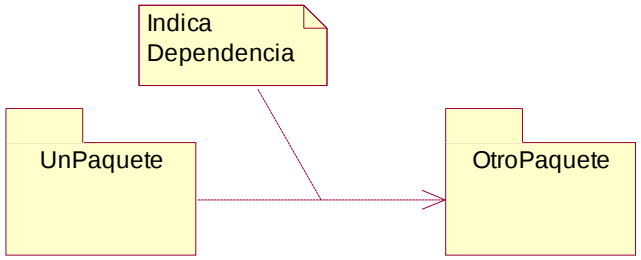


Figura 51: Ejemplo de Diagrama de Paquetes

10.2 Diagrama de Clases

El Diagrama de Clases muestra las clases que componen el modelo que se está presentando y la relación de herencia entre las misma. También muestra las relaciones de conocimiento o contención que se existen entre objetos instancias de dichas clases. Hay que tener en cuenta que estas relaciones se dibujan entre clases pero realmente suceden a otro nivel, el de instancias.

Las clases se representan con rectángulos que contienen el nombre de las mismas en la parte superior y el protocolo que definen en la parte inferior.

A continuación se puede observar un diagrama de clases de ejemplo.

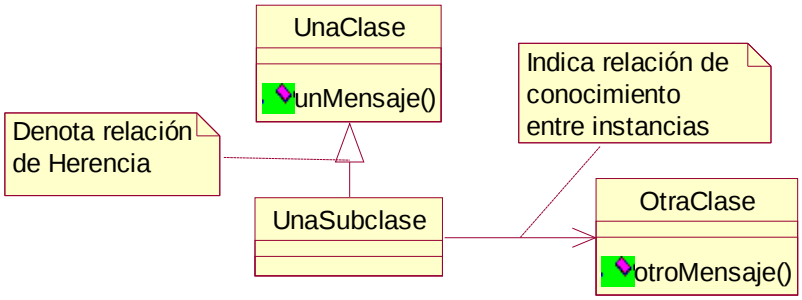


Figura 52: Ejemplo de Diagrama de Clases

10.3 Diagrama de Secuencia

El Diagrama de Secuencia muestra la colaboración entre varios objetos que surje a partir del envío de un mensaje. Generalmente tienden a dar una idea de la colaboración entre los objetos a alto nivel, sin mucho detalle de implementación.

Los objetos son representados con rectángulos que poseen el nombre con que es conocido el objeto y la clase de la cual son instancia. La colaboración entre objetos es representada con flechas que parten de un objeto a otro.

A continuación se muestra un ejemplo de un diagrama de secuencia.

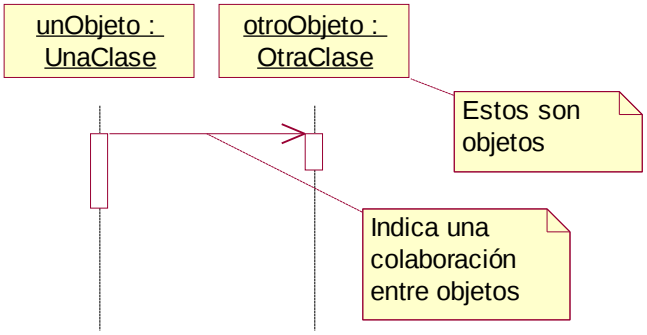


Figura 53: Ejemplo de Diagrama de Objetos

11. Apéndice B – Código de la Implementación

11.1 Thesis-Interpreter

11.1.1 RunnableInterpreter

```
InterpreterSimulatorLSB subclass: #RunnableInterpreter
	instanceVariableNames: 'endRunning '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Thesis-Interpreter'!

RunnableInterpreter>>primitiveQuit
	endRunning := true.

RunnableInterpreter>>basicRun: anImageFileName runBlock: aBlock

	| display |
	Smalltalk snapshot: true andQuit: false.
	display := DisplayScreen allInstances at: 1.
	display restoreAfter: [
		self openOn: anImageFileName. "extraMemory: 4000000"
		self class initialize. "I have to do this to initialize primitiveTable...."
		aBlock value.
		self close. ].
	Cursor normal show.

	^self.

RunnableInterpreter>>profile: anImageFileName

	self basicRun: anImageFileName runBlock: self profileBlock.

RunnableInterpreter>>profileBlock
	^ [ MessageTally
		spyOn: self runBlock
		every: 1.].

RunnableInterpreter>>profileSends: anImageFileName

	self basicRun: anImageFileName runBlock: self profileSendsBlock.

RunnableInterpreter>>profileSendsBlock
	^ [ MessageTally
		tallySends: self runBlock. ].

RunnableInterpreter>>run: anImageFileName

	self basicRun: anImageFileName runBlock: self runBlock.

RunnableInterpreter>>runBlock

	^ [byteCount := 0.
	self internalizeIPandSP.
	self fetchNextBytecode.
	endRunning := false.
	[endRunning] whileFalse:
		[self dispatchOn: currentBytecode in: BytecodeTable.
		byteCount := byteCount + 1].
	self externalizeIPandSP.].


"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "

RunnableInterpreter class
```

```
	instanceVariableNames: "!

RunnableInterpreter class>>run
	^self run: self defaultImageFileName.

RunnableInterpreter class>>run: anImageFileName
	^super new
		run: anImageFileName.

RunnableInterpreter class>>defaultImageFileName

	^'\development\thesis\squeak\ce\squeak2.1.image'.
```

11.1.2 ConfigMLInterpreter

```
RunnableInterpreter subclass: #ConfigMLInterpreter
	instanceVariableNames: 'methodLookupStrategy test testMethodLookupStrategy originalMethodLookupStrategy
methodLookupStrategies '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Thesis-Interpreter'!

ConfigMLInterpreter>>initialize

	self initializeMethodLookupStrategies.
	^super initialize

ConfigMLInterpreter>>initializeMethodLookupStrategies

	methodLookupStrategies := Dictionary new.
	methodLookupStrategies
		at: 1 put: [ DispatchTableSearchML on: self. ];
		at: 2 put: [ GlobalCachingStrategy on: self withMethodLookupStrategy: (DispatchTableSearchML on:
self). ];
		at: 3 put: [ DTSWithClassesML on: self. ];
		at: 4 put: [ HierarchyBranchML on: self. ];
		at: 5 put: [ HierarchyBranchWithDicML on: self. ];
		at: 6 put: [ GlobalCachingStrategy on: self withMethodLookupStrategy: (DTSWithClassesML on: self). ];
		at: 7 put: [ GlobalCachingStrategy on: self withMethodLookupStrategy: (HierarchyBranchML on: self). ];
		at: 8 put: [ GlobalCachingStrategy on: self withMethodLookupStrategy: (HierarchyBranchWithDicML on:
self). ];
		at: 9 put: [ SelectorCachingStrategy on: self withMethodLookupStrategy: (DispatchTableSearchML on:
self). ];
		at: 10 put: [ DispatchTableSearchML on: self. ]; "FullDictionaryML on: self. ];"
		at: 11 put: [ GlobalCachingStrategy on: self withMethodLookupStrategy: (FullDictionaryML on: self). ];
		at: 12 put: [ HBEraseAllCachingStrategy on: self withMethodLookupStrategy: (HierarchyBranchML on:
self).];
		at: 13 put: [ HBGlobalCircularAddCachingStrategy on: self withMethodLookupStrategy:
(HierarchyBranchML on: self).];
		at: 14 put: [ HBCacheCircularAddCachingStrategy on: self withMethodLookupStrategy:
(HierarchyBranchML on: self).];
		at: 15 put: [ HBOneEntryCachingStrategy on: self withMethodLookupStrategy: (HierarchyBranchML on:
self).].

ConfigMLInterpreter>>currentMethodLookupStrategy

	^methodLookupStrategy

ConfigMLInterpreter>>currentMethodLookupStrategy: aMethodLookupStrategy

	methodLookupStrategy := aMethodLookupStrategy.

ConfigMLInterpreter>>originalMethodLookupStrategy

	^originalMethodLookupStrategy

ConfigMLInterpreter>>originalMethodLookupStrategy: aMethodLookupStrategy

	originalMethodLookupStrategy := aMethodLookupStrategy.
```

```
ConfigMLInterpreter>>restoreOriginalMethodLookupStrategy

	self currentMethodLookupStrategy: self originalMethodLookupStrategy.

ConfigMLInterpreter>>testMethodLookupStrategy

	^testMethodLookupStrategy.

ConfigMLInterpreter>>testMethodLookupStrategy: aMethodLookupStrategy

	testMethodLookupStrategy := aMethodLookupStrategy.

ConfigMLInterpreter>>useTestMethodLookupStrategy

	self originalMethodLookupStrategy: self currentMethodLookupStrategy.
	self currentMethodLookupStrategy: self testMethodLookupStrategy.

ConfigMLInterpreter>>addToMethodCacheSel: selector class: class method: meth primIndex: primIndex

	^self error: 'Should not be called'

ConfigMLInterpreter>>flushAtCache

	1 to: AtCacheTotalSize do: [ :i | atCache at: i put: 0 ].

ConfigMLInterpreter>>flushMethodCache

	methodLookupStrategy flushCache.

ConfigMLInterpreter>>lookupInMethodCacheSel: selector class: class

	^self error: 'Should not be called'

ConfigMLInterpreter>>primitiveFlushCacheByMethod

	methodLookupStrategy primitiveFlushCacheSelective.

ConfigMLInterpreter>>primitiveFlushCacheSelective

	methodLookupStrategy primitiveFlushCacheSelective.

ConfigMLInterpreter>>findNewMethodInClass: aClass

	^self currentMethodLookupStrategy findMethodForSelector: messageSelector inClass: aClass.

ConfigMLInterpreter>>internalFindNewMethod

	^self currentMethodLookupStrategy findMethodForSelector: messageSelector inClass: lkupClass.

ConfigMLInterpreter>>superFindNewMethodInClass: aClass

	^super findNewMethodInClass: aClass

ConfigMLInterpreter>>maxPrimitiveIndex

	^MaxPrimitiveIndex

ConfigMLInterpreter>>messageDictionaryIndex

	^MessageDictionaryIndex.

ConfigMLInterpreter>>messageSelector

	^messageSelector.

ConfigMLInterpreter>>messageSelector: aSelector

	messageSelector := aSelector.
```

```
ConfigMLInterpreter>>methodArrayIndex

	^MethodArrayIndex.

ConfigMLInterpreter>>newMethod
	^newMethod

ConfigMLInterpreter>>newMethod: aNewMethod
	newMethod := aNewMethod.
	primitiveIndex := self primitiveIndexOf: aNewMethod.
	primitiveIndex > MaxPrimitiveIndex ifTrue:
		["If primitiveIndex is out of range, set to zero before putting in cache.
		This is equiv to primFail, and avoids the need to check on every send."
		primitiveIndex := 0].

ConfigMLInterpreter>>newMethod: aNewMethod newPrimitiveIndex: anIndex
	newMethod := aNewMethod.
	primitiveIndex := anIndex.

ConfigMLInterpreter>>primitiveIndex
	^primitiveIndex

ConfigMLInterpreter>>selectorCannotInterpret

	^self splObj: SelectorCannotInterpret.

ConfigMLInterpreter>>selectorDoesNotUnderstand

	^self splObj: SelectorDoesNotUnderstand.

ConfigMLInterpreter>>selectorStart

	^SelectorStart

ConfigMLInterpreter>>initializeCache: aMethodLookupStrategy with: anInitializationMessagesArrayOrNil
	| association selector parameter |

	anInitializationMessagesArrayOrNil = self nilObject ifFalse: [
		1 to: (self lengthOf: anInitializationMessagesArrayOrNil) do: [ :anIndex |
			association := self fetchPointer: (anIndex - 1) ofObject: anInitializationMessagesArrayOrNil.
			selector := (self stringOf: (self fetchPointer: 0 ofObject: association)) asSymbol.
			parameter := self fetchPointer: 1 ofObject: association.
			(self isIntegerObject: parameter) ifTrue: [ parameter := self integerValueOf: parameter ].
			aMethodLookupStrategy performCacheInitalization: selector with: parameter. ].
		].

ConfigMLInterpreter>>initializeMethodLookup: aMethodLookupStrategy with: anInitializationMessagesArrayOrNil

	| association selector parameter |

	anInitializationMessagesArrayOrNil = self nilObject ifFalse: [
		1 to: (self lengthOf: anInitializationMessagesArrayOrNil) do: [ :anIndex |
			association := self fetchPointer: (anIndex - 1) ofObject: anInitializationMessagesArrayOrNil.
			selector := (self stringOf: (self fetchPointer: 0 ofObject: association)) asSymbol.
			parameter := self fetchPointer: 1 ofObject: association.
			(self isIntegerObject: parameter) ifTrue: [ parameter := self integerValueOf: parameter ].
			aMethodLookupStrategy performInitalization: selector with: parameter. ].
		].

ConfigMLInterpreter>>methodLookupForKey: aMethodLookupKey

	| methodLookupStrategyCreationBlock |

	"Si el numero es 0, significa que hay que seguir usando el mismo MethodLookup"
	aMethodLookupKey=0 ifTrue: [
		^self methodLookupStrategy.]
	ifFalse: [
		methodLookupStrategyCreationBlock := methodLookupStrategies
			at: aMethodLookupKey ifAbsent: [self error: 'Invalid Method Lookup Key' ].
		^methodLookupStrategyCreationBlock value. ].
```

```
ConfigMLInterpreter>>primitiveUseCacheMethodLookup

	| methodLookupKey methodLookup initializationMessages cacheInitializationMessages |

	methodLookupKey := self integerValueOf: (self stackValue: 2).
	methodLookup := self methodLookupForKey: methodLookupKey.
	initializationMessages := self stackValue: 1.
	cacheInitializationMessages := self stackValue: 0.
	self initializeMethodLookup: methodLookup with: initializationMessages.
	self initializeCache: methodLookup with: cacheInitializationMessages.
	self testMethodLookupStrategy: methodLookup.

ConfigMLInterpreter>>primitiveUseMethodLookup

	| methodLookupKey methodLookup initializationMessages |

	methodLookupKey := self integerValueOf: (self stackValue: 1).
	methodLookup := self methodLookupForKey: methodLookupKey.
	initializationMessages := self stackValue: 0.
	self initializeMethodLookup: methodLookup with: initializationMessages.
	self testMethodLookupStrategy: methodLookup.

ConfigMLInterpreter>>primitiveStartTest
	| name |

	name := self stringOf: (self stackValue: argumentCount - 1).
	test := Test named: name for: self testMethodLookupStrategy.
	self useTestMethodLookupStrategy.

	Smalltalk garbageCollect.

	test startProfiling.

ConfigMLInterpreter>>primitiveStopTest

	| printingType |

	test stopProfiling.
	self restoreOriginalMethodLookupStrategy.

	printingType := self integerValueOf: (self stackValue: argumentCount - 1).

	printingType=0 ifTrue: [
		self printTestStats. ]
	ifFalse: [
		self printTestStatsAsCsv.].

ConfigMLInterpreter>>printTestStats
	| stream |

	stream := WriteStream on: String new.
	test printOn: stream.
	Transcript
		show: stream contents;
		show: '----------------------------------------------------';
		cr.

ConfigMLInterpreter>>printTestStatsAsCsv
	| stream |

	stream := WriteStream on: String new.
	test printAsCsvOn: stream.
	Transcript
		show: stream contents;
		cr.

"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

ConfigMLInterpreter class
```

```
	instanceVariableNames: '!

ConfigMLInterpreter class>>PrimStartTestIndex
	^ 571.

ConfigMLInterpreter class>>PrimStopTestIndex
	^ 572.

ConfigMLInterpreter class>>PrimUseCacheMethodLookup
	^ 574.

ConfigMLInterpreter class>>PrimUseMethodLookup
	^ 573.

ConfigMLInterpreter class>>run: anImageFileName using: aMethodLookupStrategyClass

	| interpreter |

	interpreter := self new.
	interpreter
		currentMethodLookupStrategy: (aMethodLookupStrategyClass on: interpreter);
		run: anImageFileName
		yourself.

ConfigMLInterpreter class>>runUsing: aMethodLookupStrategyClass

	^self run: (self defaultImageFileName) using: aMethodLookupStrategyClass

ConfigMLInterpreter class>>runUsingDTS

	^self runUsing: DispatchTableSearchML.

ConfigMLInterpreter class>>runUsingGLC

	^self runUsing: GlobalCachingStrategy.

ConfigMLInterpreter class>>profileSends: anImageFileName using: aMethodLookupStrategyClass

	| interpreter |

	interpreter := self new.
	interpreter
		currentMethodLookupStrategy: (aMethodLookupStrategyClass on: interpreter);
		profileSends: anImageFileName
		yourself.

ConfigMLInterpreter class>>profileSendsUsing: aMethodLookupStrategyClass

	^self profileSends: (self defaultImageFileName) using: aMethodLookupStrategyClass

ConfigMLInterpreter class>>profileSendsUsingDTS

	^self profileSendsUsing: DispatchTableSearchML.

ConfigMLInterpreter class>>profileSendsUsingGLC

	^self profileSendsUsing: CachedMethodLookup.

ConfigMLInterpreter class>>profile: anImageFileName using: aMethodLookupStrategyClass

	| interpreter |

	interpreter := self new.
	interpreter
		currentMethodLookupStrategy: (aMethodLookupStrategyClass on: interpreter);
		profile: anImageFileName
		yourself.

ConfigMLInterpreter class>>profileUsing: aMethodLookupStrategyClass
```

```
	^self profile: (self defaultImageFileName) using: aMethodLookupStrategyClass


ConfigMLInterpreter class>>profileUsingDTS

	^self profileUsing: DispatchTableSearchML.

ConfigMLInterpreter class>>profileUsingGLC

	^self profileUsing: CachedMethodLookup.

ConfigMLInterpreter class>>run: anImageFileName using: aMethodLookupStrategyClass profile: aNumberOfByteCodes

	| interpreter |

	interpreter := self new.
	interpreter
		currentMethodLookupStrategy: (aMethodLookupStrategyClass on: interpreter);
		openOn: anImageFileName;
		profile: aNumberOfByteCodes;
		yourself.

ConfigMLInterpreter class>>run: anImageFileName using: aMethodLookupStrategyClass profileSends: aNumberOfByteCodes

	| interpreter |

	interpreter := self new.
	interpreter
		currentMethodLookupStrategy: (aMethodLookupStrategyClass on: interpreter);
		openOn: anImageFileName;
		profileSends: aNumberOfByteCodes;
		yourself.

ConfigMLInterpreter class>>initializePrimitiveTable
	super initializePrimitiveTable.
	PrimitiveTable at: (self PrimStartTestIndex) put: #primitiveStartTest.
	PrimitiveTable at: (self PrimStopTestIndex) put: #primitiveStopTest.
	PrimitiveTable at: (self PrimUseMethodLookup) put: #primitiveUseMethodLookup.
	PrimitiveTable at: (self PrimUseCacheMethodLookup) put: #primitiveUseCacheMethodLookup.
```

11.1.3 Test

```
Object subclass: #Test
	instanceVariableNames: 'name executionStopWatch methodLookupStrategy '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Thesis-Interpreter'!


Test>>printAsCsvOn: aStream

	aStream
		nextPutAll: self name;
		comma.

	self printExecutionTimeAsCsvOn: aStream.
	aStream comma.

	self methodLookupStrategy printAsCsvOn: aStream enclosedTime: (self normalizedExecutionTime).

Test>>printExecutionTimeAsCsvOn: aStream

	aStream
		nextPutAll: self normalizedExecutionTime asMilliseconds valueAsString.

Test>>printExecutionTimeOn: aStream

	aStream
		nextPutAll: 'Tiempo Total de Ejecuci—n: ';
		nextPutAll: self normalizedExecutionTime asMilliseconds printString.
```

```
Test>>printOn: aStream

	aStream
		nextPutAll: 'Nombre del Test: ';
		nextPutAll: self name;
		cr.

	self printExecutionTimeOn: aStream.
	aStream cr.

	self methodLookupStrategy printOn: aStream enclosedTime: (self normalizedExecutionTime).

Test>>normalizedExecutionTime

	^self realExecutionTime - self methodLookupStrategy deviationTime.

Test>>realExecutionTime

	^self executionStopWatch expendedTime.

Test>>startProfiling

	StopWatch initializeTicksStack.
	self executionStopWatch: StopWatch new.
	self methodLookupStrategy startProfiling.
	self executionStopWatch start.

Test>>stopProfiling

	self executionStopWatch stop.
	self methodLookupStrategy stopProfiling.

Test>>executionStopWatch

	^executionStopWatch

Test>>executionStopWatch: aStopWatch

	executionStopWatch := aStopWatch.

Test>>methodLookupStrategy

	^methodLookupStrategy

Test>>methodLookupStrategy: aMethodLookupStrategy

	methodLookupStrategy := aMethodLookupStrategy

Test>>name
	^name.

Test>>name: aName
	name := aName.

"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

Test class
	instanceVariableNames: ''!

Test class>>named: aName for: aMethodLookupStrategy

	^(super new)
		name: aName;
		methodLookupStrategy: aMethodLookupStrategy;
		yourself.

Test class>>new
```

```
	^self error: 'Usar named:for:'
```

11.2 Thesis-Method Lookup Strategies

11.2.1 MethodLookupStrategy

```
Object subclass: #MethodLookupStrategy
	instanceVariableNames: ''
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Thesis-Method Lookup strategies'!

MethodLookupStrategy>>createProfiler

	^self subclassResponsibility

MethodLookupStrategy>>deviationTime

	^self subclassResponsibility

MethodLookupStrategy>>extraTime

	^self subclassResponsibility

MethodLookupStrategy>>lookupTime

	^self subclassResponsibility

MethodLookupStrategy>>profiler

	^self subclassResponsibility

MethodLookupStrategy>>profiler: aProfiler

	self subclassResponsibility

MethodLookupStrategy>>profilingResult

	^self subclassResponsibility

MethodLookupStrategy>>profilingResult: aProfiler

	self subclassResponsibility

MethodLookupStrategy>>startProfiling

	self profiler: self createProfiler.
	self profiler startProfiling.

MethodLookupStrategy>>stopProfiling

	self profiler stopProfiling.
	self profilingResult: self profiler.
	self profiler: NullMethodLookupProfiler new.

	^self profilingResult.

MethodLookupStrategy>>flushCache

	^self subclassResponsibility.

MethodLookupStrategy>>primitiveFlushCacheByMethod

	^self subclassResponsibility.

MethodLookupStrategy>>primitiveFlushCacheSelective

	^self subclassResponsibility.
```

```
MethodLookupStrategy>>interpreter

	^self subclassResponsibility

MethodLookupStrategy>>interpreter: anInterpreter

	self subclassResponsibility

MethodLookupStrategy>>findMethodForSelector: aSelector inClass: aClass

	^self subclassResponsibility

MethodLookupStrategy>>printAsCsvOn: aStream enclosedTime: aStopWatchTime

	self printNameAsCsvOn: aStream.
	self printProfilingResultAsCsvOn: aStream enclosedTime: aStopWatchTime.

MethodLookupStrategy>>printNameAsCsvOn: aStream

	aStream
		nextPutAll: self class name;
		comma.

MethodLookupStrategy>>printProfilingResultAsCsvOn: aStream enclosedTime: aStopWatchTime

	self profilingResult printAsCsvOn: aStream enclosedTime: aStopWatchTime.

MethodLookupStrategy>>printNameOn: aStream

	aStream
		nextPutAll: 'Lookup Strategy: ';
		nextPutAll: self class name;
		cr.

MethodLookupStrategy>>printOn: aStream enclosedTime: aStopWatchTime

	self printNameOn: aStream.
	self printProfilingResultOn: aStream enclosedTime: aStopWatchTime.

MethodLookupStrategy>>printProfilingResultOn: aStream enclosedTime: aStopWatchTime

	self profilingResult printOn: aStream enclosedTime: aStopWatchTime.

MethodLookupStrategy>>performInitalization: selector with: parameter

	self subclassResponsibility.
```

11.2.2 BasicMethodLookupStrategy

```
MethodLookupStrategy subclass: #BasicMethodLookupStrategy
	instanceVariableNames: 'interpreter profiler profilingResult nilObj maxPrimitiveIndex '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Thesis-Method Lookup strategies'!

BasicMethodLookupStrategy>>deviationTime

	^self profilingResult totalDeviationTime.

BasicMethodLookupStrategy>>extraTime

	^self profilingResult totalExtraTime.

BasicMethodLookupStrategy>>lookupTime

	^self profilingResult methodLookupTime.
```

```
BasicMethodLookupStrategy>>profiler

	^profiler

BasicMethodLookupStrategy>>profiler: aMethodLookupProfiler

	profiler := aMethodLookupProfiler

BasicMethodLookupStrategy>>profilingResult

	^profilingResult

BasicMethodLookupStrategy>>profilingResult: aProfiler

	profilingResult := aProfiler

BasicMethodLookupStrategy>>lookupMethodForSelector: aSelector inClass: aClass

	self subclassResponsibility.

BasicMethodLookupStrategy>>findMethodForSelector: aSelector inClass: aClass

	"Uses the template pattern.
	Subclasses must implement lookupMethodForSelector: inClass:"

	| result |

	profiler selector: aSelector sendTo: aClass.
	profiler profileMethodLookup: [ result := self lookupMethodForSelector: aSelector inClass: aClass ].
	^result.

BasicMethodLookupStrategy>>interpreter

	^interpreter

BasicMethodLookupStrategy>>interpreter: anInterpreter

	interpreter := anInterpreter.
	self initializeInterpreterData.

BasicMethodLookupStrategy>>initialize

	profiler := NullMethodLookupProfiler new.

BasicMethodLookupStrategy>>initializeInterpreterData

	nilObj := interpreter nilObject.
	maxPrimitiveIndex := interpreter maxPrimitiveIndex.

BasicMethodLookupStrategy>>performInitalization: aSelector with: aCollaborator

	self perform: aSelector with: aCollaborator.

BasicMethodLookupStrategy>>lookupInteger: anInteger inDictionary: aDictionary

	| array arraySize association key hash start |

	"Fetchs the inst. vars array"
	array := interpreter fetchPointer: 1 ofObject: aDictionary.

	hash := interpreter integerValueOf: anInteger.
	arraySize := interpreter fetchWordLengthOf: array.
	arraySize > 4096 ifTrue:
		[ hash := hash * (arraySize//4096)].

	start := hash \\ arraySize.
	start to: arraySize-1 do:
		[ :index |
		association := interpreter fetchPointer: index ofObject: array.
		association=nilObj ifTrue:
```

```
					[^ nilObj ]
				ifFalse:
					[key := interpreter fetchPointer: 0 ofObject: association.
					key=anInteger ifTrue:
						[^ interpreter fetchPointer: 1 ofObject: association. ].
					].
				].

	0 to: start-1 do:
				[ :index |
				association := interpreter fetchPointer: index ofObject: array.
				association=nilObj ifTrue:
					[^ nilObj ]
				ifFalse:
					[key := interpreter fetchPointer: 0 ofObject: association.
					key=anInteger ifTrue:
						[^ interpreter fetchPointer: 1 ofObject: association. ].
					].
				].

	^ nilObj.

BasicMethodLookupStrategy>>lookupObject: anObject inDictionary: aDictionary

	| array arraySize association key hash start |

	"Fetchs the inst. vars array"
	array := interpreter fetchPointer: 1 ofObject: aDictionary.

	hash := interpreter hashBitsOf: anObject.
	arraySize := interpreter fetchWordLengthOf: array.
	arraySize > 4096 ifTrue:
				[ hash := hash * (arraySize//4096)].

	start := hash \\ arraySize.
	start to: arraySize-1 do:
				[ :index |
				association := interpreter fetchPointer: index ofObject: array.
				association=nilObj ifTrue:
					[^ nilObj ]
				ifFalse:
					[key := interpreter fetchPointer: 0 ofObject: association.
					key=anObject ifTrue:
						[^ interpreter fetchPointer: 1 ofObject: association. ].
					].
				].

	0 to: start-1 do:
				[ :index |
				association := interpreter fetchPointer: index ofObject: array.
				association=nilObj ifTrue:
					[^ nilObj ]
				ifFalse:
					[key := interpreter fetchPointer: 0 ofObject: association.
					key=anObject ifTrue:
						[^ interpreter fetchPointer: 1 ofObject: association. ].
					].
				].

	^ nilObj.

"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

BasicMethodLookupStrategy class
	instanceVariableNames: ''!

BasicMethodLookupStrategy class>>on: anInterpreter

	^super new
		interpreter: anInterpreter;
```

```
		initialize;
		yourself.
```

11.2.3 CachedMethodLookup

```
BasicMethodLookupStrategy subclass: #CachedMethodLookup
	instanceVariableNames: 'cachingStrategy methodLookupStrategy '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Thesis-Method Lookup strategies'!

CachedMethodLookup>>printCachingStrategyAsCsvOn: aStream

	self cachingStrategy printAsCsvOn: aStream enclosedTime: self lookupTime.
CachedMethodLookup>>printOn: aStream enclosedTime: aStopWatchTime

	super printNameOn: aStream.
	self printResultOn: aStream enclosedTime: aStopWatchTime.
	self printMethodLookupStrategyOn: aStream.
	self printCachingStrategyOn: aStream.

CachedMethodLookup>>primitiveFlushCacheSelective

	cachingStrategy primitiveFlushCacheSelective.

CachedMethodLookup>>printMethodLookupStrategyAsCsvOn: aStream

	self methodLookupStrategy printAsCsvOn: aStream enclosedTime: (self lookupTime + self methodLookupStrategy
deviationTime)

CachedMethodLookup>>testInit: aStringPointer

	Transcript show: (self interpreter stringOf: aStringPointer).

CachedMethodLookup>>flushCache

	cachingStrategy flushCache.

CachedMethodLookup>>methodLookupStrategy: aMethodLookupStrategy

	methodLookupStrategy _ aMethodLookupStrategy.

CachedMethodLookup>>deviationTime

	^self profilingResult totalDeviationTime + self methodLookupStrategy deviationTime + self cachingStrategy
deviationTime.

CachedMethodLookup>>lookupMethodForSelector: aSelector inClass: aClass

	| foundOnCache |

	foundOnCache _ cachingStrategy findMethodForSelector: aSelector inClass: aClass.
	foundOnCache ifFalse: [
		methodLookupStrategy findMethodForSelector: aSelector inClass: aClass.
		cachingStrategy addSelector: aSelector forClass: aClass
			method: (interpreter newMethod) primitiveIndex: (interpreter primitiveIndex).
	].

CachedMethodLookup>>performInitalization: aSelector with: aCollaborator

	self methodLookupStrategy perform: aSelector with: aCollaborator.

CachedMethodLookup>>printMethodLookupStrategyOn: aStream

	self methodLookupStrategy printOn: aStream enclosedTime: (self lookupTime + self methodLookupStrategy
deviationTime)

CachedMethodLookup>>primitiveFlushCacheByMethod
```

```
	cachingStrategy primitiveFlushCacheByMethod.

CachedMethodLookup>>cachingStrategy

	^cachingStrategy

CachedMethodLookup>>printResultAsCsvOn: aStream enclosedTime: aStopWatchTime

	aStream
		nextPutAll: self lookupTime asMilliseconds valueAsString;
		comma;
		nextPutAll: ((self lookupTime percentOf: aStopWatchTime) roundTo: 0.01) printString;
		comma.

CachedMethodLookup>>cachingStrategy: aCachingStrategy
	cachingStrategy _ aCachingStrategy.

CachedMethodLookup>>stopProfiling

	super stopProfiling.
	self methodLookupStrategy stopProfiling.
	self cachingStrategy stopProfiling.

CachedMethodLookup>>printResultOn: aStream enclosedTime: aStopWatchTime

	aStream
		nextPutAll: 'Tiempo total de Lookup: ';
		nextPutAll: self lookupTime asMilliseconds printString;
		nextPutAll: ' (';
		nextPutAll: ((self lookupTime percentOf: aStopWatchTime) roundTo: 0.01) printString;
		nextPutAll: '%)';
		cr.

CachedMethodLookup>>createProfiler

	^BasicMethodLookupProfiler new.

CachedMethodLookup>>cacheLookupTime

	^self cachingStrategy lookupCacheTime.

CachedMethodLookup>>cacheMaintenanceTime

	^self cachingStrategy maintenanceTime.

CachedMethodLookup>>methodLookupStrategy

	^methodLookupStrategy

CachedMethodLookup>>printCachingStrategyOn: aStream

	self cachingStrategy printOn: aStream enclosedTime: self lookupTime.

CachedMethodLookup>>performCacheInitalization: aSelector with: aCollaborator

	self cachingStrategy perform: aSelector with: aCollaborator.

CachedMethodLookup>>startProfiling

	super startProfiling.
	self methodLookupStrategy startProfiling.
	self cachingStrategy startProfiling.

CachedMethodLookup>>lookupTime

	^self profilingResult methodLookupTime - self methodLookupStrategy deviationTime - self cachingStrategy
deviationTime.

CachedMethodLookup>>printAsCsvOn: aStream enclosedTime: aStopWatchTime
```

```
	super printNameAsCsvOn: aStream.
	self printResultAsCsvOn: aStream enclosedTime: aStopWatchTime.
	self printMethodLookupStrategyAsCsvOn: aStream.
	self printCachingStrategyAsCsvOn: aStream.

CachedMethodLookup class>>on: anInterpreter

	^self on: anInterpreter methodLookupCreationBlock: self defaultMLCreationBlock.

CachedMethodLookup class>>on: anInterpreter methodLookupCreationBlock: aMLCreationBlock

	^self
		on: anInterpreter
		methodLookupCreationBlock: aMLCreationBlock
		cachingCreationBlock: self defaultCachingCreationBlock.


CachedMethodLookup class>>defaultCachingCreationBlock

	^[ :anInterpreter | GlobalCachingStrategy on: anInterpreter ].

CachedMethodLookup class>>defaultMLCreationBlock

	^[ :anInterpreter | DispatchTableSearchML on: anInterpreter ].

CachedMethodLookup class>>on: anInterpreter methodLookupCreationBlock: aMLCreationBlock cachingCreationBlock:
aCachingCreationBlock

	^(super on: anInterpreter)
		methodLookupStrategy: (aMLCreationBlock value: anInterpreter);
		cachingStrategy: (aCachingCreationBlock value: anInterpreter );
		yourself.
```

11.2.4 DTSWithClassesML

```
BasicMethodLookupStrategy subclass: #DTSWithClassesML
	instanceVariableNames: 'selectorsDictionary '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Thesis-Method Lookup strategies'!

DTSWithClassesML>>lookupMethodForSelector: aSelector inClass: aClass

	| selectorDictionary result |

	profiler profileExtraTime: [ selectorDictionary _ self selectorDictionaryFor: aSelector ].
	result _ self lookupMethodInDictionary: selectorDictionary forClass: aClass.

	^result

DTSWithClassesML>>primitiveFlushCacheSelective

	"Do nothing. I don't have cache..."
DTSWithClassesML>>selectorsDictionary: anDictionaryPointer
	"This message is sent to set the reference to the dictionary that holds the DTS information"

	selectorsDictionary _ anDictionaryPointer

DTSWithClassesML>>primitiveFlushCacheByMethod

	"Do nothing. I don't have cache..."
DTSWithClassesML>>lookupMethodInDictionary: selectorDictionary forClass: class

	| currentClass newMethod rclass |

	currentClass _ class.
	[currentClass ~= nilObj] whileTrue:
		[newMethod _ self lookupObject: currentClass inDictionary: selectorDictionary.
		newMethod=nilObj ifFalse:
			[interpreter newMethod: newMethod.
```

```
                    ^newMethod].
                currentClass _ interpreter superclassOf: currentClass].

        "Could not find #doesNotUnderstand: -- unrecoverable error."
        interpreter messageSelector = interpreter selectorDoesNotUnderstand ifTrue:
        [self error: 'Recursive not understood error encountered'].

        "Cound not find a normal message -- raise exception #doesNotUnderstand:"
        interpreter pushRemappableOop: class.  "may cause GC!"
        interpreter createActualMessageTo: class.
        rclass _ interpreter popRemappableOop.
        interpreter messageSelector: self selectorDoesNotUnderstand.
        ^ self lookupMethodInDictionary: selectorDictionary forClass: rclass.

DTSWithClassesML>>flushCache

        "Do nothing. I don't have cache..."

DTSWithClassesML>>selectorDictionaryFor: aSelector

        ^self lookupObject: aSelector inDictionary: selectorsDictionary.

DTSWithClassesML>>createProfiler

        ^BasicMethodLookupProfiler new.
```

11.2.5 DispatchTableSearchML

```
BasicMethodLookupStrategy subclass: #DispatchTableSearchML
        instanceVariableNames: 'messageDictionaryIndex selectorStart methodArrayIndex '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Thesis-Method Lookup strategies'!

DispatchTableSearchML>>lookupMethod: aSelector inDictionary: dictionary
        "This method lookup tolerates integers as Dictionary keys to support
        execution of images in which Symbols have been compacted out"
        | length index mask wrapAround nextSelector methodArray |

        length _ interpreter fetchWordLengthOf: dictionary.
        mask _ length - selectorStart - 1.
        (interpreter isIntegerObject: aSelector)
                ifTrue:
                [index _ (mask bitAnd: (interpreter integerValueOf: aSelector)) + selectorStart]
                ifFalse:
                [index _ (mask bitAnd: (interpreter hashBitsOf: aSelector)) + selectorStart].
        "It is assumed that there are some nils in this dictionary, and search will
        stop when one is encountered.  However, if there are no nils, then wrapAround
        will be detected the second time the loop gets to the end of the table."
        wrapAround _ false.
        [true] whileTrue:
                [nextSelector _ interpreter fetchPointer: index
                                        ofObject: dictionary.
                nextSelector=nilObj ifTrue: [^false].
                nextSelector=aSelector
                        ifTrue: [methodArray _ interpreter fetchPointer: methodArrayIndex
                                                        ofObject: dictionary.
                                interpreter newMethod: ( interpreter fetchPointer:  index - selectorStart
                                                        ofObject: methodArray).
                                ^ true].
                index _ index + 1.
                index = length
                        ifTrue: [wrapAround ifTrue: [^false].
                                wrapAround _ true.
                                index _ selectorStart]]

DispatchTableSearchML>>lookupMethodForSelector: aSelector inClass: aClass

        | currentClass dictionary found rclass |
```

```
		currentClass _ aClass.
		[currentClass ~= nilObj]
			whileTrue:
			[dictionary _ interpreter fetchPointer: messageDictionaryIndex ofObject: currentClass.
			found _ self lookupMethod: aSelector inDictionary: dictionary.
			found ifTrue: [^ currentClass].
			currentClass _ interpreter superclassOf: currentClass].

		"Could not find #doesNotUnderstand: -- unrecoverable error."
		aSelector= interpreter selectorDoesNotUnderstand ifTrue:
			[self error: 'Recursive not understood error encountered'].

		"Cound not find a normal message -- raise exception #doesNotUnderstand:"
		interpreter pushRemappableOop: aClass.  "may cause GC!"
		interpreter createActualMessageTo: aClass.
		rclass _ interpreter popRemappableOop.
		^ self lookupMethodForSelector: (interpreter selectorDoesNotUnderstand) inClass: rclass.

DispatchTableSearchML>>flushCache

		"Do nothing. I don't have cache..."

DispatchTableSearchML>>primitiveFlushCacheByMethod

		"Do nothing. I don't have cache..."

DispatchTableSearchML>>createProfiler

		^BasicMethodLookupProfiler new.

DispatchTableSearchML>>primitiveFlushCacheSelective

		"Do nothing. I don't have cache..."

DispatchTableSearchML>>initializeInterpreterData

		super initializeInterpreterData.
		messageDictionaryIndex _ interpreter messageDictionaryIndex.
		selectorStart _ interpreter selectorStart.
		methodArrayIndex _ interpreter methodArrayIndex.
```

11.2.6 FullDictionary

```
'From Squeak2.6 of 11 October 1999 [latest update: #1559] on 21 August 2001 at 3:34:43 pm'!
DispatchTableSearchML subclass: #FullDictionaryML
		instanceVariableNames: ''
		classVariableNames: ''
		poolDictionaries: ''
		category: 'Thesis-Method Lookup strategies'!

FullDictionaryML>>lookupMethodForSelector: aSelector inClass: aClass

		"I have no diference with DTS..."
		^super lookupMethodForSelector: aSelector inClass: aClass.
```

11.2.7 HierarchyBranchML

```
BasicMethodLookupStrategy subclass: #HierarchyBranchML
		instanceVariableNames: 'classIntervals selectorsImplementations lastMessageImplementationBranch '
		classVariableNames: ''
		poolDictionaries: ''
		category: 'Thesis-Method Lookup strategies'!

HierarchyBranchML>>classNumberFor: aBehavior

		| classInterval |
		classInterval _ self classIntervalFor: aBehavior.
		^interpreter fetchInteger: 0 ofObject: classInterval.
```

```
HierarchyBranchML>>lastFrom

	^interpreter fetchInteger: 0 ofObject: lastMessageImplementationBranch.

HierarchyBranchML>>primitiveFlushCacheSelective

	"Do nothing. I don't have cache..."

HierarchyBranchML>>createProfiler

	^BasicMethodLookupProfiler new.

HierarchyBranchML>>lastTo

	^interpreter fetchInteger: 1 ofObject: lastMessageImplementationBranch.

HierarchyBranchML>>classIntervalFor: aBehavior

	^self lookupObject: aBehavior inDictionary: classIntervals.

HierarchyBranchML>>flushCache

	"Do nothing. I don't have cache..."

HierarchyBranchML>>classIntervals: aDictionaryPointer

	classIntervals _ aDictionaryPointer

HierarchyBranchML>>selectorsImplementations: aDictionaryPointer

	selectorsImplementations _ aDictionaryPointer.

HierarchyBranchML>>selectorImplementationsFor: aSelector

	^self lookupObject: aSelector inDictionary: selectorsImplementations.

HierarchyBranchML>>lookupMethodForSelector: aSelector inClass: aClass

	| selectorImplementations classNumber result |

	profiler profileExtraTime: [
		selectorImplementations _ self selectorImplementationsFor: aSelector.
		classNumber _ self classNumberFor: aClass. ].

	result _ self compiledMethodFor: classNumber selectorImplementations: selectorImplementations.

	result=nilObj ifTrue: [
		profiler profileExtraTime: [
			selectorImplementations _ self selectorImplementationsFor: interpreter
selectorDoesNotUnderstand.].
		result _ self compiledMethodFor: classNumber selectorImplementations: selectorImplementations.
		(result=interpreter nilObject) ifTrue: [ self error: 'MNU no implementado'.].].

	interpreter newMethod: result.

	^result.

HierarchyBranchML>>compiledMethodFor: aClassNumber selectorImplementations: aSortedCollectionPointer

	"Implements the quick search algorithm"

	| array low mid high messageImplementationBranch |

	array _ interpreter fetchPointer: 0 ofObject: aSortedCollectionPointer. "Fetchs the array instVar"
	low _ (interpreter fetchInteger: 1 ofObject: aSortedCollectionPointer)-1. "Fetchs the firstIndex instVar"
	high _ (interpreter fetchInteger: 2 ofObject: aSortedCollectionPointer)-1. "Fetch the lastIndex instVar"
	[low <= high] whileTrue:
		[mid _ low + high // 2.
		messageImplementationBranch _ interpreter fetchPointer: mid ofObject: array.
		(interpreter fetchInteger:0 ofObject: messageImplementationBranch) <= aClassNumber ifTrue: [
```

```
                        (interpreter fetchInteger:1 ofObject: messageImplementationBranch) > aClassNumber ifTrue: [
                                "I save the branch to allow use it with out looking for it again"
                                lastMessageImplementationBranch _ messageImplementationBranch.
                                ^(interpreter fetchPointer:2 ofObject: messageImplementationBranch) ]
                        ifFalse: [
                                low _ mid + 1.]. ]
                ifFalse: [
                        high _ mid - 1.].
                ].

        "No definition found, return nil"
        ^nilObj.

HierarchyBranchML>>primitiveFlushCacheByMethod

        "Do nothing. I don't have cache..."
```

11.2.8 HierarchyBranchWithDicML

```
HierarchyBranchML subclass: #HierarchyBranchWithDicML
        instanceVariableNames: 'sortedCollection '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Thesis-Method Lookup strategies'!

HierarchyBranchWithDicML>>for1compiledMethodFor: aClassNumber selectorImplementations: aSortedCollectionPointer

        "Just for collections of 1 element"

        | array low messageImplementationBranch |

        array _ interpreter fetchPointer: 0 ofObject: aSortedCollectionPointer. "Fetchs the array instVar"
        low _ (interpreter fetchInteger: 1 ofObject: aSortedCollectionPointer)-1.
        messageImplementationBranch _ interpreter fetchPointer: low ofObject: array.
        ^(interpreter fetchPointer:2 ofObject: messageImplementationBranch).


HierarchyBranchWithDicML>>lookupMethodForSelector: aSelector inClass: aClass

        "I reimplemented the method to have a better performance"

        | selectorImplementations classNumber result doesNotUnderstand |

        profiler profileExtraTime: [
                selectorImplementations _ self selectorImplementationsFor: aSelector. ].

        "I check if the implementation is a sorted collection or a dictionary"
        (interpreter fetchClassOf: selectorImplementations)=sortedCollection ifTrue: [
                profiler profileExtraTime: [ classNumber _ self classNumberFor: aClass. ].
                result _ self compiledMethodFor: classNumber selectorImplementations: selectorImplementations. ]
        ifFalse: [
                result _self lookupObject: aClass inDictionary: selectorImplementations. ].

        result=nilObj ifTrue: [
"               I have commented this because it assumes that are all messages are stored in SortedCollections
                profiler profileExtraTime: [
                        selectorImplementations _ self selectorImplementationsFor: interpreter
selectorDoesNotUnderstand.
                        classNumber isNil ifTrue: [ classNumber _ self classNumberFor: aClass. ]. ].
                result _ self compiledMethodFor: classNumber selectorImplementations: selectorImplementations.
                (result=interpreter nilObject) ifTrue: [ self error: 'MNU no implementado'.]."
                doesNotUnderstand _ interpreter selectorDoesNotUnderstand.
                aSelector=doesNotUnderstand ifTrue: [ self error: 'MNU no implementado'.].
                result _ self lookupMethodForSelector: doesNotUnderstand inClass: aClass.
        ].

        interpreter newMethod: result.

        ^result
```

```
HierarchyBranchWithDicML>>sortedCollection: aClassPointer

	sortedCollection _ aClassPointer
```

11.3 Thesis- Caching Strategies

11.3.1 CachingStrategy

```
BasicMethodLookupStrategy subclass: #CachingStrategy
	instanceVariableNames: ''
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Thesis-Caching strategies'!

CachingStrategy>>printResultAsCsvOn: aStream enclosedTime: aStopWatchTime

	aStream
		nextPutAll: self lookupTime asMilliseconds valueAsString;
		comma;
		nextPutAll: ((self lookupTime percentOf: aStopWatchTime) roundTo: 0.01) printString;
		comma.

CachingStrategy>>printProfilingResultAsCsvOn: aStream enclosedTime: aStopWatchTime

	self profilingResult printAsCsvOn: aStream enclosedTime: aStopWatchTime.

CachingStrategy>>printUsedEntriesAsCsvOn: aStream

	aStream
		nextPutAll: self usedEntries printString;
		comma;
		nextPutAll: ((self usedEntries*100/self cacheEntries) roundTo: 0.01) printString;
		comma.

CachingStrategy>>printMethodLookupStrategyOn: aStream

	self methodLookupStrategy printOn: aStream enclosedTime: (self lookupTime + self methodLookupStrategy
extraTime)

CachingStrategy>>printAsCsvOn: aStream enclosedTime: aStopWatchTime

	super printNameAsCsvOn: aStream.
	self printResultAsCsvOn: aStream enclosedTime: aStopWatchTime.
	self printMethodLookupStrategyAsCsvOn: aStream.
	self printNameAsCsvOn: aStream.
	self printProfilingResultAsCsvOn: aStream enclosedTime: self lookupTime.
	self printUsedEntriesAsCsvOn: aStream.

CachingStrategy>>primitiveFlushCacheSelective

	self subclassResponsibility

CachingStrategy>>lookupMethodForSelector: aSelector inClass: aClass

	self subclassResponsibility

CachingStrategy>>addSelector: aSelector forClass: aClass method: aMethod primitiveIndex: aPrimitiveIndex

	self subclassResponsibility

CachingStrategy>>initialize

	self profiler: NullCachingProfiler new.

CachingStrategy>>printOn: aStream enclosedTime: aStopWatchTime

	super printNameOn: aStream.
	self printResultOn: aStream enclosedTime: aStopWatchTime.
```

```
	self printMethodLookupStrategyOn: aStream.
	self printNameOn: aStream.
	self printProfilingResultOn: aStream enclosedTime: self lookupTime.
	self printUsedEntriesOn: aStream.

CachingStrategy>>stopProfiling

	self methodLookupStrategy stopProfiling.
	^super stopProfiling.

CachingStrategy>>printResultOn: aStream enclosedTime: aStopWatchTime

	aStream
		nextPutAll: 'Tiempo total de Lookup: ';
		nextPutAll: self lookupTime asMilliseconds printString;
		nextPutAll: ' (';
		nextPutAll: ((self lookupTime percentOf: aStopWatchTime) roundTo: 0.01) printString;
		nextPutAll: '%)';
		cr.

CachingStrategy>>printProfilingResultOn: aStream enclosedTime: aStopWatchTime

	self profilingResult printOn: aStream enclosedTime: aStopWatchTime.

CachingStrategy>>flushCache

	self subclassResponsibility

CachingStrategy>>printMethodLookupStrategyAsCsvOn: aStream

	self methodLookupStrategy printAsCsvOn: aStream enclosedTime: (self lookupTime + self methodLookupStrategy
extraTime)

CachingStrategy>>printNameAsCsvOn: aStream

	aStream
		nextPutAll: self class name;
		comma.

CachingStrategy>>lookupTime

	^self profilingResult methodLookupTime - self methodLookupStrategy deviationTime.

CachingStrategy>>methodLookupStrategy

	^self subclassResponsibility

CachingStrategy>>printUsedEntriesOn: aStream

	aStream
		nextPutAll: 'Used Entries: ';
		nextPutAll: self usedEntries printString;
		nextPutAll: ' (';
		nextPutAll: ((self usedEntries*100/self cacheEntries) roundTo: 0.01) printString;
		nextPutAll: '%)';
		cr.

CachingStrategy>>deviationTime

	^self profilingResult totalDeviationTime + self methodLookupStrategy deviationTime

CachingStrategy>>primitiveFlushCacheByMethod

	self subclassResponsibility
CachingStrategy>>printNameOn: aStream

	aStream nextPutAll: 'Caching Strategy: ';
		nextPutAll: self class name;
		cr.
```

```
CachingStrategy>>performInitalization: aSelector with: aCollaborator

	self methodLookupStrategy perform: aSelector with: aCollaborator.

CachingStrategy>>startProfiling

	self methodLookupStrategy startProfiling.
	^super startProfiling.

CachingStrategy>>cacheEntries

	self subclassResponsibility.
CachingStrategy>>performCacheInitalization: aSelector with: aCollaborator

	self perform: aSelector with: aCollaborator.

CachingStrategy>>usedEntries

	self subclassResponsibility.

CachingStrategy class>>new

	^super new
		initialize;
		yourself.
```

11.3.2 BasicCachingStrategy

```
CachingStrategy subclass: #BasicCachingStrategy
	instanceVariableNames: 'methodLookupStrategy '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Thesis-Caching strategies'!

BasicCachingStrategy>>methodLookupStrategy

	^methodLookupStrategy.

BasicCachingStrategy>>lookupMethodForSelector: aSelector inClass: aClass

	| foundOnCache |

	profiler profileCacheLookup: [ foundOnCache _ self lookupInCache: aSelector forClass: aClass.].
	foundOnCache ifFalse: [
		profiler incrementCacheMiss.
		methodLookupStrategy findMethodForSelector: aSelector inClass: aClass.
		profiler profileCacheMaintenance: [ self addSelector: aSelector forClass: aClass
			method: (interpreter newMethod) primitiveIndex: (interpreter primitiveIndex). ].
	].

BasicCachingStrategy>>lookupInCache: aSelector forClass: aClass

	^self subclassResponsibility.

BasicCachingStrategy>>methodLookupStrategy: aStrategy

	methodLookupStrategy _ aStrategy.

BasicCachingStrategy class>>on: anInterpreter withMethodLookupStrategy: aStrategy

	^super new
		interpreter: anInterpreter;
		methodLookupStrategy: aStrategy;
		yourself.

BasicCachingStrategy class>>on: anInterpreter

	^self on: anInterpreter withMethodLookupStrategy: (DispatchTableSearchML on: anInterpreter).
```

11.3.3 AbstractSelectorCachingStrategy

```
BasicCachingStrategy subclass: #AbstractSelectorCachingStrategy
	instanceVariableNames: ''
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Thesis-Caching strategies'!

AbstractSelectorCachingStrategy>>usedEntries

	| usedEntries |

	usedEntries _ 0.
	self selectorsCache do: [ :aCache |
		usedEntries _ usedEntries + (self countUsedEntries: aCache).
	].

	^usedEntries.

AbstractSelectorCachingStrategy>>printAsCsvOn: aStream enclosedTime: aStopWatchTime

	super printAsCsvOn: aStream enclosedTime: aStopWatchTime.
	self printDictionaryAsCsv: self cacheStats on: aStream.

AbstractSelectorCachingStrategy>>countUsedEntries: aCache

	^self subclassResponsibility

AbstractSelectorCachingStrategy>>printDictionaryAsCsv: aDictionary on: aStream

	aDictionary associationsDo: [ :assoc |
		aStream
			nextPutAll: assoc key printString;
			comma;
			nextPutAll: assoc value printString;
			comma.
	].

AbstractSelectorCachingStrategy>>createProfiler

	^BasicCachingProfiler new.

AbstractSelectorCachingStrategy>>printOn: aStream enclosedTime: aStopWatchTime

	super printOn: aStream enclosedTime: aStopWatchTime.
	aStream
		nextPutAll: 'Uso de la cache: ';
		nextPutAll: self cacheStats printString.

AbstractSelectorCachingStrategy>>cacheEntries

	^self selectorsCache size*self cacheEntries.

AbstractSelectorCachingStrategy>>cacheStats

	| result usedEntries count |
	result _ Dictionary new.

	self selectorsCache do: [ :anArray |
		usedEntries _ self countUsedEntries: anArray.
		count _ result at: usedEntries ifAbsent: [0].
		result at: usedEntries put: count+1.].

	^result.

AbstractSelectorCachingStrategy>>selectorsCache

	^self subclassResponsibility
```

11.3.4 GlobalCachingStrategy

```
BasicCachingStrategy subclass: #GlobalCachingStrategy
	instanceVariableNames: 'methodCache methodCacheEntries methodCacheClass methodCacheMask
methodCacheMethod methodCachePrim methodCacheSelector cacheProbeMax '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Thesis-Caching strategies'!

GlobalCachingStrategy>>createProfiler

	^BasicCachingProfiler new.

GlobalCachingStrategy>>primitiveFlushCacheByMethod

	| probe oldMethod |

	profiler profileCacheMaintenance: [
		oldMethod _ interpreter stackTop.
		probe _ 0.
		1 to: self methodCacheEntries do: [:i |
			(methodCache at: probe + methodCacheMethod) = oldMethod ifTrue:
				[methodCache at: probe + methodCacheSelector put: 0].
			probe _ probe + self methodCacheEntrySize].
		interpreter compilerFlushCacheHook: oldMethod.		"Flush the dynamic compiler's inline caches."
	].

GlobalCachingStrategy>>methodCacheSize

	^self methodCacheEntries*self methodCacheEntrySize.

GlobalCachingStrategy>>methodCacheEntries

	^methodCacheEntries.

GlobalCachingStrategy>>defaultMethodCacheEntries

	^512.

GlobalCachingStrategy>>cacheEntries

	^self methodCacheEntries.

GlobalCachingStrategy>>addSelector: selector forClass: class method: meth primitiveIndex: primIndex
	"Add the given entry to the method cache.
	The policy is as follows:
		Look for an empty entry anywhere in the reprobe chain.
		If found, install the new entry there.
		If not found, then install the new entry at the first probe position
			and delete the entries in the rest of the reprobe chain.
		This has two useful purposes:
			If there is active contention over the first slot, the second
				or third will likely be free for reentry after ejection.
			Also, flushing is good when reprobe chains are getting full."
	| probe hash |
	hash _ selector bitXor: class.  "drop low-order zeros from addresses"

	0 to: cacheProbeMax-1 do:
		[:p | probe _ (hash >> p) bitAnd: methodCacheMask.
		(methodCache at: probe + methodCacheSelector) = 0 ifTrue:
				["Found an empty entry -- use it"
				methodCache at: probe + methodCacheSelector put: selector.
				methodCache at: probe + methodCacheClass put: class.
				methodCache at: probe + methodCacheMethod put: meth.
				methodCache at: probe + methodCachePrim put: primIndex.
				^ nil]].

	"OK, we failed to find an entry -- install at the first slot..."
	probe _ hash bitAnd: methodCacheMask.  "first probe"
	methodCache at: probe + methodCacheSelector put: selector.
```

```
		methodCache at: probe + methodCacheClass put: class.
		methodCache at: probe + methodCacheMethod put: meth.
		methodCache at: probe + methodCachePrim put: primIndex.

		"...and zap the following entries"
		1 to: cacheProbeMax-1 do:
			[:p | probe _ (hash >> p) bitAnd: methodCacheMask.
			methodCache at: probe + methodCacheSelector put: 0].


GlobalCachingStrategy>>initialize

		super initialize.
		self methodCacheEntries: self defaultMethodCacheEntries.

GlobalCachingStrategy>>lookupInCache: selector forClass: class
		"This method implements a simple method lookup cache. If an entry for the given selector and class is found in the cache, set the values of 'newMethod' and 'primitiveIndex' and return true. Otherwise, return false."
		"About the re-probe scheme: The hash is the low bits of the XOR of two large addresses, minus their useless lowest two bits. If a probe doesn't get a hit, the hash is shifted right one bit to compute the next probe, introducing a new randomish bit. The cache is probed CacheProbeMax times before giving up."
		"WARNING: Since the hash computation is based on the object addresses of the class and selector, we must rehash or flush when compacting storage. We've chosen to flush, since that also saves the trouble of updating the addresses of the objects in the cache."

		| hash probe |
		hash _ selector bitXor: class.  "shift drops two low-order zeros from addresses"

		probe _ hash bitAnd: methodCacheMask.  "first probe"
		(((methodCache at: probe + methodCacheSelector) = selector) and:
			[(methodCache at: probe + methodCacheClass) = class]) ifTrue:
				[interpreter newMethod: (methodCache at: probe + methodCacheMethod)
					newPrimitiveIndex: (methodCache at: probe + methodCachePrim).
				^ true	"found entry in cache; done"].

		probe _ (hash >> 1) bitAnd: methodCacheMask.  "second probe"
		(((methodCache at: probe + methodCacheSelector) = selector) and:
			[(methodCache at: probe + methodCacheClass) = class]) ifTrue:
				[interpreter newMethod: (methodCache at: probe + methodCacheMethod)
					newPrimitiveIndex: (methodCache at: probe + methodCachePrim).
				^ true	"found entry in cache; done"].

		probe _ (hash >> 2) bitAnd: methodCacheMask.
		(((methodCache at: probe + methodCacheSelector) = selector) and:
			[(methodCache at: probe + methodCacheClass) = class]) ifTrue:
				[interpreter newMethod: (methodCache at: probe + methodCacheMethod)
					newPrimitiveIndex: (methodCache at: probe + methodCachePrim).
				^ true	"found entry in cache; done"].

		^ false

GlobalCachingStrategy>>initializeMethodCache

		methodCacheSelector _ 1.
		methodCacheClass _ 2.
		methodCacheMethod _ 3.
		methodCachePrim _ 4.
		methodCacheMask _ (self methodCacheEntries - 1) * self methodCacheEntrySize.
		cacheProbeMax _ 3.

		methodCache _ Array new: self methodCacheSize.
		1 to: self methodCacheSize do: [ :i | methodCache at: i put: 0 ].


GlobalCachingStrategy>>usedEntries

		| entry usedEntries |

		usedEntries _ 0.
		0 to: (self cacheEntries)-1 do: [ :anIndex |
```

```
                entry _ methodCache at: (anIndex*4) + 1.
                (entry isNil or: [ entry=0 ]) not ifTrue: [ usedEntries _ usedEntries + 1 ].
        ].

        ^usedEntries.

GlobalCachingStrategy>>flushCache

        profiler profileCacheMaintenance: [
                1 to: self methodCacheSize do: [ :i | methodCache at: i put: 0 ].
                interpreter flushAtCache.].




GlobalCachingStrategy>>methodCacheEntrySize

        ^4.

GlobalCachingStrategy>>primitiveFlushCacheSelective

        | selector probe |

        profiler profileCacheMaintenance: [
                selector _ interpreter stackTop.
                probe _ 0.
                1 to: self methodCacheEntries do: [ :i |
                        (methodCache at: probe + methodCacheSelector) = selector ifTrue:
                                [methodCache at: probe + methodCacheSelector put: 0].
                        probe _ probe + self methodCacheEntrySize].
        ].

GlobalCachingStrategy>>methodCacheEntries: aNumber

        methodCacheEntries _ aNumber.
        self initializeMethodCache.
```

11.3.5 HBCachingStrategy

```
AbstractSelectorCachingStrategy subclass: #HBCachingStrategy
        instanceVariableNames: 'selectorsCache cacheEntries cacheEntriesByEntrySize entrySize '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Thesis-Caching strategies'!

HBCachingStrategy>>cacheEntries

        ^cacheEntries.

HBCachingStrategy>>initialize

        super initialize.
        selectorsCache _ Dictionary new.
        cacheEntries _ 3.
        entrySize _ 4.
        cacheEntriesByEntrySize _ cacheEntries * entrySize.

HBCachingStrategy>>primitiveFlushCacheByMethod

        ^self error: 'Not implemented yet'.

HBCachingStrategy>>flushCache

        profiler profileCacheMaintenance: [
                selectorsCache _ Dictionary new.
                interpreter flushAtCache.].

HBCachingStrategy>>createSelectorCache
```

```
	^Array new: cacheEntriesByEntrySize.

HBCachingStrategy>>countUsedEntries: aCache

	| entry usedEntries |

	usedEntries _ 0.
	1 to: cacheEntriesByEntrySize by: entrySize do: [ :anIndex |
		entry _ aCache at: anIndex.
		"Stop counting on the first nil"
		entry notNil ifTrue: [ usedEntries _ usedEntries + 1 ] ifFalse: [^usedEntries].
	].

	^usedEntries

HBCachingStrategy>>selectorsCache

	^selectorsCache.

HBCachingStrategy>>primitiveFlushCacheSelective

	^self error: 'Not implemented yet'.

HBCachingStrategy>>addSelector: aSelector forClass: aClass method: aMethod primitiveIndex: aPrimitiveIndex

	^self subclassResponsibility.

HBCachingStrategy>>lookupInCache: aSelector forClass: aClass

	| cache classNumber from |

	profiler profileExtraCacheLookupTime: [
		cache _ selectorsCache at: aSelector ifAbsentPut: [self createSelectorCache.].
		classNumber _ methodLookupStrategy classNumberFor: aClass. ].

	1 to: cacheEntriesByEntrySize by: entrySize do: [ :index |
		from _ cache at: index.
		from isNil ifTrue: [ ^false. ].
		from <= classNumber ifTrue: [
			classNumber < (cache at: (index+1)) ifTrue: [
				interpreter newMethod: (cache at: (index+2) ) newPrimitiveIndex: (cache at:
(index+3) ).
				^true.
			].
		].
	].

	^false.
```

11.3.6 HBCacheCircularAddCachingStrategy

```
HBCachingStrategy subclass: #HBCacheCircularAddCachingStrategy
	instanceVariableNames: ''
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Thesis-Caching strategies'!

HBCacheCircularAddCachingStrategy>>addSelector: aSelector forClass: aClass method: aMethod primitiveIndex:
aPrimitiveIndex

	| cache indexToAdd |

	profiler profileExtraMaintenanceTime: [ cache _ selectorsCache at: aSelector. ].

	indexToAdd _ cache last.

	cache at: indexToAdd put: methodLookupStrategy lastFrom.
	cache at: (indexToAdd+1) put: methodLookupStrategy lastTo.
	cache at: (indexToAdd+2) put: aMethod.
	cache at: (indexToAdd+3) put: aPrimitiveIndex.
```

```
	cache at: (cache size) put: (indexToAdd + entrySize)\\cacheEntriesByEntrySize.

HBCacheCircularAddCachingStrategy>>createSelectorCache

	| cache size |
	size _ cacheEntriesByEntrySize + 1.
	cache _ Array new: size.
	cache at: size put: 1.
	^cache.
```

11.3.7 HBEraseAllCachingStrategy

```
HBCachingStrategy subclass: #HBEraseAllCachingStrategy
	instanceVariableNames: ''
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Thesis-Caching strategies'!

HBEraseAllCachingStrategy>>addSelector: aSelector forClass: aClass method: aMethod primitiveIndex: aPrimitiveIndex

	| cache |

	profiler profileExtraMaintenanceTime: [ cache _ selectorsCache at: aSelector. ].

	1 to: cacheEntriesByEntrySize by: entrySize do: [ :index |
		(cache at: index) isNil ifTrue: [
			cache at: index put: methodLookupStrategy lastFrom.
			cache at: (index+1) put: methodLookupStrategy lastTo.
			cache at: (index+2) put: aMethod.
			cache at: (index+3) put: aPrimitiveIndex.
			^self.
		].
	].

	"No space in the cache. Put the new entry at first position"
	cache at: 1 put: methodLookupStrategy lastFrom.
	cache at: 2 put: methodLookupStrategy lastTo.
	cache at: 3 put: aMethod.
	cache at: 4 put: aPrimitiveIndex.

	"Putting nil in the second position means that the rest of the cache is empy"
	cache at: 5 put: nil.
```

11.3.8 HBGlobalCircularAddCachingStrategy

```
HBCachingStrategy subclass: #HBGlobalCircularAddCachingStrategy
	instanceVariableNames: 'indexToAdd '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Thesis-Caching strategies'!

HBGlobalCircularAddCachingStrategy>>addSelector: aSelector forClass: aClass method: aMethod primitiveIndex:
aPrimitiveIndex

	| cache |

	profiler profileExtraMaintenanceTime: [ cache _ selectorsCache at: aSelector. ].

	1 to: cacheEntriesByEntrySize by: entrySize do: [ :index |
		(cache at: index) isNil ifTrue: [
			cache at: index put: methodLookupStrategy lastFrom.
			cache at: (index+1) put: methodLookupStrategy lastTo.
			cache at: (index+2) put: aMethod.
			cache at: (index+3) put: aPrimitiveIndex.
			^self.
		].
	].

	cache at: indexToAdd put: methodLookupStrategy lastFrom.
```

```
		cache at: (indexToAdd+1) put: methodLookupStrategy lastTo.
		cache at: (indexToAdd+2) put: aMethod.
		cache at: (indexToAdd+3) put: aPrimitiveIndex.

		indexToAdd _ (indexToAdd + entrySize)\\cacheEntriesByEntrySize.

HBGlobalCircularAddCachingStrategy>>initialize

		super initialize.
		indexToAdd _ 1.
```

11.3.9 HBOneEntryCachingStrategy

```
HBCachingStrategy subclass: #HBOneEntryCachingStrategy
		instanceVariableNames: ''
		classVariableNames: ''
		poolDictionaries: ''
		category: 'Thesis-Caching strategies'!

HBOneEntryCachingStrategy>>lookupInCache: aSelector forClass: aClass

		| cache classNumber from |

		profiler profileExtraCacheLookupTime: [
			cache _ selectorsCache at: aSelector ifAbsentPut: [self createSelectorCache.].
			classNumber _ methodLookupStrategy classNumberFor: aClass. ].

		from _ cache at: 1.
		from isNil ifTrue: [ ^false. ].
		from <= classNumber ifTrue: [
			classNumber < (cache at: 2) ifTrue: [
				interpreter newMethod: (cache at: 3 ) newPrimitiveIndex: (cache at: 4 ).
				^true.
			].
		].

		^false.

HBOneEntryCachingStrategy>>initialize

		super initialize.
		selectorsCache _ Dictionary new.
		cacheEntries _ 1.
		entrySize _ 4.
		cacheEntriesByEntrySize _ cacheEntries * entrySize.

HBOneEntryCachingStrategy>>addSelector: aSelector forClass: aClass method: aMethod primitiveIndex: aPrimitiveIndex

		| cache |

		profiler profileExtraMaintenanceTime: [ cache _ selectorsCache at: aSelector. ].

		cache at: 1 put: methodLookupStrategy lastFrom.
		cache at: 2 put: methodLookupStrategy lastTo.
		cache at: 3 put: aMethod.
		cache at: 4 put: aPrimitiveIndex.
```

11.3.10 SelectorCachingStrategy

```
AbstractSelectorCachingStrategy subclass: #SelectorCachingStrategy
		instanceVariableNames: 'selectorsCache cacheEntries '
		classVariableNames: ''
		poolDictionaries: ''
		category: 'Thesis-Caching strategies'!

SelectorCachingStrategy>>lookupInCache2: aSelector class: aClass

		| cache probe |

		profiler profileExtraLookupTime: [ cache _ selectorsCache at: aSelector ifAbsentPut: [self createSelectorCache.]. ].
```

```
		probe _ ((aClass>>2) bitAnd: 3)+1.
		(cache at: probe)=aClass ifTrue: [
			interpreter newMethod: (cache at: probe + 4 )
						newPrimitiveIndex: (cache at: probe + 8 ).
			^true.
		].

		probe=4 ifTrue: [
			probe _ 1. ]
		ifFalse: [
			probe _ probe + 1. ].

		(cache at: probe)=aClass ifTrue: [
			interpreter newMethod: (cache at: probe + 4 )
						newPrimitiveIndex: (cache at: probe + 8 ).
			^true.
		].

		^false.
SelectorCachingStrategy>>flushCache

		profiler profileCacheMaintenance: [
			selectorsCache _ Dictionary new.
			interpreter flushAtCache.].

SelectorCachingStrategy>>addSelector: aSelector forClass: aClass method: aMethod primitiveIndex: aPrimitiveIndex

		| cache probe |

		profiler profileExtraMaintenanceTime: [ cache _ selectorsCache at: aSelector. ].
		probe _ ((aClass>>2) bitAnd: 3)+1.
		cache at: probe  put: aClass.
		cache at: probe + 4 put: aMethod.
		cache at: probe + 8 put: aPrimitiveIndex.

SelectorCachingStrategy>>lookupInCache: aSelector forClass: aClass

		| cache probe |

		profiler profileExtraCacheLookupTime: [ cache _ selectorsCache at: aSelector ifAbsentPut: [self
createSelectorCache.]. ].
		probe _ ((aClass>>2) bitAnd: 3)+1.
		(cache at: probe)=aClass ifTrue: [
			interpreter newMethod: (cache at: probe + 4 )
						newPrimitiveIndex: (cache at: probe + 8 ).
			^true.
		].

		^false.

SelectorCachingStrategy>>countUsedEntries: aCache

		| entry usedEntries |

		usedEntries _ 0.
		1 to: 4 do: [ :anIndex |
			entry _ aCache at: anIndex.
			(entry isNil or: [entry=0]) not ifTrue: [ usedEntries _ usedEntries + 1 ].
		].

		^usedEntries

SelectorCachingStrategy>>primitiveFlushCacheSelective

		^self error: 'Not implemented yet'.

SelectorCachingStrategy>>createSelectorCache

		^Array new: (3 * cacheEntries).
```

```
SelectorCachingStrategy>>primitiveFlushCacheByMethod

	^self error: 'Not implemented yet'.

SelectorCachingStrategy>>addSelector2: aSelector forClass: aClass

	| cache probe |

	profiler profileExtraMaintenanceTime: [ cache _ selectorsCache at: aSelector. ].
	probe _ ((aClass>>2) bitAnd: 3)+1.
	(cache at: probe) isNil ifTrue: [
		cache at: probe  put: aClass.
		cache at: probe + 4 put: (interpreter newMethod).
		cache at: probe + 8 put: (interpreter primitiveIndex).
		^self. ].

	probe=4 ifTrue: [
		probe _ 1. ]
	ifFalse: [
		probe _ probe + 1. ].

	(cache at: probe) isNil ifTrue: [
		cache at: probe  put: aClass.
		cache at: probe + 4 put: (interpreter newMethod).
		cache at: probe + 8 put: (interpreter primitiveIndex).
		^self ].

	"There is no empty space to add the entry, so I erase this one and..."
	cache at: probe  put: nil.
	cache at: probe + 4 put: nil.
	cache at: probe + 8 put: nil.

	"Add the new entry in the first place..."
	probe _ ((aClass>>2) bitAnd: 3)+1.
	cache at: probe  put: aClass.
	cache at: probe + 4 put: (interpreter newMethod).
	cache at: probe + 8 put: (interpreter primitiveIndex).

SelectorCachingStrategy>>initialize

	super initialize.
	selectorsCache _ Dictionary new.
	cacheEntries _ 4.

SelectorCachingStrategy>>cacheEntries

	^cacheEntries.

SelectorCachingStrategy>>selectorsCache

	^selectorsCache.
```

11.4 Thesis-Profiling Support

11.4.1 StopWatch

```
Object subclass: #StopWatch
	instanceVariableNames: 'totalTicks numberOfMeasurements '
	classVariableNames: 'MeasurementDeviation TicksPerSecond '
	poolDictionaries: ''
	category: 'Thesis-Profiling support'!

StopWatch>>totalExpendedTime

	^self expendedTime + self deviationTime.

StopWatch>>stop

	| elapsedTicks |
```

```
		elapsedTicks _ self elapsedTicks.
		totalTicks _ totalTicks + elapsedTicks.
		numberOfMeasurements _ numberOfMeasurements + 1.
		^elapsedTicks.

StopWatch>>printOn: aStream

		aStream
			nextPutAll: self expendedTime asMilliseconds printString;
			cr;
			nextPutAll: self totalExpendedTime asMilliseconds printString.

StopWatch>>clear

		self initialize.

StopWatch>>expendedTime

		^StopWatchTicks new: totalTicks.

StopWatch>>expendedMilliseconds

		^self expendedTime asMilliseconds

StopWatch>>initialize

		totalTicks _ 0.
		numberOfMeasurements _ 0.

StopWatch>>elapsedTicks

		<primitive: 603>

StopWatch>>expendedSeconds

		^self expendedTime asSeconds.

StopWatch>>measure: aBlock

		self start.
		aBlock value.
		^self stop.

StopWatch>>start

		self saveCurrentTicks.

StopWatch>>saveCurrentTicks

		<primitive: 602>

StopWatch>>expendedTicks

		^self expendedTime asTicks.

StopWatch>>deviationTime

		^StopWatchTicks new:  (self class measurementDeviation * numberOfMeasurements).

StopWatch>>currentTicks

		<primitive: 600>

StopWatch class>>testComposite

		| sum sw1 sw2 sw3 |

		sw1 _ StopWatch new.
		sw2 _ StopWatch new.
```

```
	sw3 _ StopWatch new.

	sw1 measure: [
		sum _ 0.
		1 to: 1000 do: [ :index |
			sum _ sum + 1. ].
	].
	sw1 clear.

	sw3 measure: [
	sw1 measure: [
		sum _ 0.
		1 to: 1000 do: [ :index |
			sum _ sum + 1. ].
	].

	sw2 measure: [
		sum _ 0.
		1 to: 1000 do: [ :index |
			sum _ sum + 1. ].
	].
	].
	self halt.
	^Array
		with: sw3 expendedTime value
		with: sw1 totalExpendedTime value asFloat
		with: sw2 totalExpendedTime value asFloat
		with: (sw3 expendedTime - sw1 totalExpendedTime - sw2 totalExpendedTime) value asFloat.

StopWatch class>>initialize

	self initializeTicksPerSecond.
	self initializeMeasurementDeviation.

StopWatch class>>initializeTicksStack

	<primitive: 604>

StopWatch class>>ticksMeasuring

	| coll stopWatch |
	coll _ OrderedCollection new.
	stopWatch _ StopWatch new.
	1 to: self cyclesForDeviation do: [ :j |
		coll add: ( StopWatch ticksToRun: [ 1 to: 100 do: [ :i |
			stopWatch measure: [ 2 + 2 ]. ]. ]).
	].
	^coll average.

StopWatch class>>ticksWithOutMeasuring

	| coll |
	coll _ OrderedCollection new.
	1 to: self cyclesForDeviation do: [ :j |
		coll add: ( StopWatch ticksToRun: [ 1 to: 100 do: [ : i | 2 + 2 ].]).
	].
	^coll average.

StopWatch class>>calculateMeasurementDeviation

	| ticksMeasuring tickWithOutMeasuring |

	ticksMeasuring _ self ticksMeasuring.
	tickWithOutMeasuring _ self ticksWithOutMeasuring.

	^(ticksMeasuring - tickWithOutMeasuring)/self cyclesForDeviation.

StopWatch class>>ticksToRun: aBlock

	StopWatch saveCurrentTicks.
```

```
		aBlock value.
		^StopWatch elapsedTicks.

StopWatch class>>cyclesForDeviation

		^ 200.

StopWatch class>>elapsedTicks

		<primitive: 603>

StopWatch class>>basicTicksPerSecond

		<primitive: 601>

StopWatch class>>ticksPerMillisecond

		^self ticksPerSecond / 1000

StopWatch class>>initializeMeasurementDeviation

		MeasurementDeviation _ self calculateMeasurementDeviation.

StopWatch class>>timeToRun: aBlock

		^StopWatchTicks new: (StopWatch ticksToRun: aBlock).

StopWatch class>>ticksPerSecond

		^TicksPerSecond

StopWatch class>>new

		^super new initialize.

StopWatch class>>saveCurrentTicks

		<primitive: 602>

StopWatch class>>measurementDeviation

		^MeasurementDeviation.

StopWatch class>>initializeTicksPerSecond

		TicksPerSecond := self basicTicksPerSecond.
```

11.4.2 StopWatchTime

```
Object subclass: #StopWatchTime
		instanceVariableNames: ''
		classVariableNames: 'Cero '
		poolDictionaries: ''
		category: 'Thesis-Profiling support'!

StopWatchTime>>multiplyByANumber: aNumber

		self subclassResponsibility

StopWatchTime>>+ aStopWatchTime

		self subclassResponsibility.

StopWatchTime>>- aStopWatchTime

		self subclassResponsibility.

StopWatchTime>>printOn: aStream

		self subclassResponsibility
```

```
StopWatchTime>>asTicks
	self subclassResponsibility

StopWatchTime>>* aStopWatchTimeOrANumber

	^aStopWatchTimeOrANumber multiplyStopWatchTime: self.

StopWatchTime>>/ aStopWatchTimeOrANumber

	^aStopWatchTimeOrANumber divideStopWatchTime: self.

StopWatchTime>>divideByANumber: aNumber

	self subclassResponsibility

StopWatchTime>>percentOf: aStopWatchTime

	^self asTicks value*100/(aStopWatchTime asTicks value)

StopWatchTime>>value

	self subclassResponsibility.

StopWatchTime>>asSeconds

	self subclassResponsibility

StopWatchTime>>asMilliseconds

	self subclassResponsibility

StopWatchTime>>divideStopWatchTime: aStopWatchTime

	self subclassResponsibility

StopWatchTime>>multiplyStopWatchTime: aStopWatchTime

	self subclassResponsibility

StopWatchTime>>value: aValue

	self subclassResponsibility.

StopWatchTime class>>initializeCero

	Cero _ StopWatchTicks new: 0.

StopWatchTime class>>cero

	Cero isNil ifTrue: [ self initializeCero. ].
	^Cero.

StopWatchTime class>>new: aValue

	^super new
		value: aValue;
		yourself.
```

11.4.3 Milliseconds

```
StopWatchTime subclass: #Milliseconds
	instanceVariableNames: 'stopWatchTicks '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Thesis-Profiling support'!

Milliseconds>>asSeconds

	^Seconds new: (self value/1000).
```

```
Milliseconds>>divideByANumber: aNumber

	^Milliseconds new: (self value / aNumber ).

Milliseconds>>printOn: aStream

	aStream
		nextPutAll: (self value roundTo: 0.01) printString;
		space;
		nextPutAll: 'Millis.'.

Milliseconds>>- aStopWatchTime

	^(self stopWatchTicks - (aStopWatchTime asTicks)) asMilliseconds.

Milliseconds>>multiplyByANumber: aNumber

	^Milliseconds new: (self value * aNumber ).

Milliseconds>>value: aValue

	^self stopWatchTicks value: (aValue*StopWatch ticksPerMillisecond)

Milliseconds>>multiplyStopWatchTime: aStopWatchTime

	^Milliseconds new: (self value * (aStopWatchTime asMilliseconds value))

Milliseconds>>stopWatchTicks

	stopWatchTicks isNil ifTrue: [ stopWatchTicks _ StopWatchTicks new: 0 ].
	^stopWatchTicks.

Milliseconds>>asTicks

	^self stopWatchTicks.

Milliseconds>>valueAsString

	^(self value roundTo: 0.01) printString.

Milliseconds>>+ aStopWatchTime

	^(self stopWatchTicks + (aStopWatchTime asTicks)) asMilliseconds.

Milliseconds>>value

	^(self stopWatchTicks value)/(StopWatch ticksPerMillisecond)

Milliseconds>>asMilliseconds

	^self

Milliseconds>>divideStopWatchTime: aStopWatchTime

	^Milliseconds new: (aStopWatchTime asMilliseconds value / (self value)).
```

11.4.4 Seconds

```
StopWatchTime subclass: #Seconds
	instanceVariableNames: 'stopWatchTicks '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Thesis-Profiling support'!

Seconds>>value
	^(self stopWatchTicks value)/(StopWatch ticksPerSecond)

Seconds>>+ aStopWatchTime
```

```
	^(self stopWatchTicks + (aStopWatchTime asTicks)) asSeconds.

Seconds>>multiplyByANumber: aNumber

	^Seconds new: (self value * aNumber ).

Seconds>>printOn: aStream

	aStream
		nextPutAll: (self value asFloat printString);
		space;
		nextPutAll: 'Seconds'.

Seconds>>asMilliseconds

	^Milliseconds new: (self value*1000).

Seconds>>value: aValue

	^self stopWatchTicks value: (aValue*StopWatch ticksPerSecond)

Seconds>>asTicks

	^self stopWatchTicks.

Seconds>>divideStopWatchTime: aStopWatchTime

	^Seconds new: (aStopWatchTime asSeconds value / (self value)).

Seconds>>stopWatchTicks

	stopWatchTicks isNil ifTrue: [ stopWatchTicks _ StopWatchTicks new: 0 ].
	^stopWatchTicks.

Seconds>>asSeconds

	^self

Seconds>>divideByANumber: aNumber

	^Seconds new: (self value / aNumber ).

Seconds>>multiplyStopWatchTime: aStopWatchTime

	^Seconds new: (self value * (aStopWatchTime asSeconds value)).

Seconds>>- aStopWatchTime

	^(self stopWatchTicks - (aStopWatchTime asTicks)) asSeconds.
```

11.4.5 StopWatchTicks

```
StopWatchTime subclass: #StopWatchTicks
	instanceVariableNames: 'ticks '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Thesis-Profiling support'!

StopWatchTicks>>divideByANumber: aNumber

	^StopWatchTicks new: (self value / aNumber ).

StopWatchTicks>>multiplyStopWatchTime: aStopWatchTime

	^StopWatchTicks new: (self value * aStopWatchTime asTicks value).

StopWatchTicks>>divideStopWatchTime: aStopWatchTime

	^StopWatchTicks new: ( aStopWatchTime asTicks value / self value).
```

```
StopWatchTicks>>+ aStopWatchTime

	^StopWatchTicks new: (self value + aStopWatchTime asTicks value).

StopWatchTicks>>value

	^ticks.

StopWatchTicks>>- aStopWatchTime

	^StopWatchTicks new: (self value - aStopWatchTime asTicks value).

StopWatchTicks>>printOn: aStream

	aStream
		nextPutAll: (self value printString);
		space;
		nextPutAll: 'Ticks'.

StopWatchTicks>>asSeconds

	^Seconds new: (self value/StopWatch ticksPerSecond)

StopWatchTicks>>asMilliseconds

	^Milliseconds new: (ticks/StopWatch ticksPerMillisecond)

StopWatchTicks>>multiplyByANumber: aNumber

	^StopWatchTicks new: (self value * aNumber ).

StopWatchTicks>>value: aValue

	ticks _ aValue.

StopWatchTicks>>asTicks

	^self.
```

11.5 Thesis-Method Lookup Profiling

11.5.1 MethodLookupProfiler

```
Object subclass: #MethodLookupProfiler
	instanceVariableNames: ''
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Thesis-Method Lookup Profiling'!

MethodLookupProfiler>>extraDeviationTime

	self subclassResponsibility

MethodLookupProfiler>>selector: messageSelector sendTo: lookupClass

	self subclassResponsibility.

MethodLookupProfiler>>totalExtraTime

	^self extraTime + self extraDeviationTime

MethodLookupProfiler>>sendMessages

	self subclassResponsibility

MethodLookupProfiler>>startProfiling

	self subclassResponsibility.
```

```
MethodLookupProfiler>>methodLookupTime

	self subclassResponsibility

MethodLookupProfiler>>initialize

MethodLookupProfiler>>methodLookupDeviationTime

	self subclassResponsibility

MethodLookupProfiler>>extraTime

	self subclassResponsibility

MethodLookupProfiler>>stopProfiling

	self subclassResponsibility.

MethodLookupProfiler>>printAsCsvOn: aStream enclosedTime: aStopWatchTime

	self subclassResponsibility.

MethodLookupProfiler>>totalDeviationTime

	^self extraTime + self extraDeviationTime + self methodLookupDeviationTime.

MethodLookupProfiler>>profileMethodLookup: aBlock

	self subclassResponsibility.

MethodLookupProfiler>>printOn: aStream enclosedTime: aStopWatchTime

	self subclassResponsibility.

MethodLookupProfiler>>profileExtraTime: aBlock

	self subclassResponsibility.

MethodLookupProfiler class>>new

	^super new
		initialize;
		yourself.
```

11.5.2 BasicMethodLookupProfiler

```
MethodLookupProfiler subclass: #BasicMethodLookupProfiler
	instanceVariableNames: 'totalSendMessages methodLookupStopWatch extraTimeStopWatch '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Thesis-Method Lookup Profiling'!

BasicMethodLookupProfiler>>printMethodLookupTimeAsCsvOn: aStream enclosedTime: aStopWatchTimeOrNil

	aStream
		nextPutAll: self methodLookupTime asMilliseconds valueAsString;
		comma.
	aStopWatchTimeOrNil isNil ifFalse: [
		aStream
			nextPutAll: ((self methodLookupTime percentOf: aStopWatchTimeOrNil) roundTo: 0.01)
printString;
			comma. ].

BasicMethodLookupProfiler>>printMethodLookupTimeOn: aStream enclosedTime: aStopWatchTimeOrNil

	aStream nextPutAll: 'Tiempo de Method Lookup: '.
	self methodLookupTime asMilliseconds  printOn: aStream.
	aStopWatchTimeOrNil isNil ifFalse: [
		aStream
```

```
                                nextPutAll: '( ';
                                nextPutAll: ((self methodLookupTime percentOf: aStopWatchTimeOrNil) roundTo: 0.01)
printString;
                                nextPutAll: '%)'. ].

BasicMethodLookupProfiler>>initialize

        totalSendMessages _ 0.
        methodLookupStopWatch _ StopWatch new.
        extraTimeStopWatch _ StopWatch new.

BasicMethodLookupProfiler>>printOn: aStream

        ^self printOn: aStream enclosedTime: nil.

BasicMethodLookupProfiler>>methodLookupDeviationTime

        ^methodLookupStopWatch deviationTime.

BasicMethodLookupProfiler>>methodLookupTime

        ^methodLookupStopWatch expendedTime - (self extraTime + self extraDeviationTime ).

BasicMethodLookupProfiler>>sendMessages

        ^totalSendMessages

BasicMethodLookupProfiler>>selector: aSelector sendTo: aClass

        totalSendMessages _ totalSendMessages + 1.

BasicMethodLookupProfiler>>extraDeviationTime

        ^extraTimeStopWatch deviationTime.

BasicMethodLookupProfiler>>printSendMessagesAsCsvOn: aStream

        aStream
                nextPutAll: self sendMessages printString;
                comma.

BasicMethodLookupProfiler>>printAsCsvOn: aStream enclosedTime: aStopWatchTime

        self printMethodLookupTimeAsCsvOn: aStream enclosedTime: aStopWatchTime.
        self printSendMessagesAsCsvOn: aStream.

BasicMethodLookupProfiler>>extraTime

        ^extraTimeStopWatch expendedTime.

BasicMethodLookupProfiler>>startProfiling

        methodLookupStopWatch clear.
        extraTimeStopWatch clear.


BasicMethodLookupProfiler>>profileExtraTime: aBlock

        extraTimeStopWatch measure: aBlock.

BasicMethodLookupProfiler>>printSendMessagesOn: aStream

        aStream
                nextPutAll: 'Total de mensajes buscados: ';
                nextPutAll: self sendMessages printString;
                cr.

BasicMethodLookupProfiler>>stopProfiling
```

```
BasicMethodLookupProfiler>>printOn: aStream enclosedTime: aStopWatchTime

	self printMethodLookupTimeOn: aStream enclosedTime: aStopWatchTime.
	aStream cr.
	self printSendMessagesOn: aStream.

BasicMethodLookupProfiler>>profileMethodLookup: aBlock

	methodLookupStopWatch measure: aBlock.
```

11.5.3 HierarchyBranchMLProfiler

```
BasicMethodLookupProfiler subclass: #HierarchyBranchMLProfiler
	instanceVariableNames: 'tries '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Thesis-Method Lookup Profiling'!

HierarchyBranchMLProfiler>>initialize

	super initialize.
	tries _ Dictionary new.

HierarchyBranchMLProfiler>>prolifeNumberOfTries: aNumberOfTries

	| currentTryCount |
	currentTryCount _ tries at: aNumberOfTries ifAbsent: [ tries at: aNumberOfTries put: 0 ].
	tries at: aNumberOfTries put: currentTryCount + 1.

HierarchyBranchMLProfiler>>printOn: aStream enclosedTime: aTime

	super printOn: aStream enclosedTime: aTime.
```

11.5.4 NullMethodLookupProfiler

```
MethodLookupProfiler subclass: #NullMethodLookupProfiler
	instanceVariableNames: ''
	classVariableNames: 'Singleton '
	poolDictionaries: ''
	category: 'Thesis-Method Lookup Profiling'!

NullMethodLookupProfiler>>printOn: aStream enclosedTime: aStopWatchTime

	^self printOn: aStream.

NullMethodLookupProfiler>>profileExtraTime: aBlock

	aBlock value.

NullMethodLookupProfiler>>stopProfiling

NullMethodLookupProfiler>>profileMethodLookup: aBlock

	aBlock value.

NullMethodLookupProfiler>>sendMessages

	^0.

NullMethodLookupProfiler>>methodLookupDeviationTime

	^StopWatchTime cero.

NullMethodLookupProfiler>>extraDeviationTime

	^StopWatchTime cero.


NullMethodLookupProfiler>>selector: messageSelector sendTo: lookupClass
```

```
NullMethodLookupProfiler>>extraTime

	^StopWatchTime cero.

NullMethodLookupProfiler>>methodLookupTime

	^StopWatchTime cero.

NullMethodLookupProfiler>>startProfiling

NullMethodLookupProfiler>>printAsCsvOn: aStream

NullMethodLookupProfiler class>>initializeSingleton

	Singleton _ super new.

NullMethodLookupProfiler class>>new

	Singleton ifNil: [ self initializeSingleton.].
	^Singleton.
```

11.6 Thesis-Caching Profiling

11.6.1 CachingProfiler

```
BasicMethodLookupProfiler subclass: #CachingProfiler
	instanceVariableNames: ''
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Thesis-Caching Profiling'!

CachingProfiler>>maintenanceTime

	^self subclassResponsibility

CachingProfiler>>printMaintenanceTimeAsCsvOn: aStream enclosedTime: aStopWatchTime

	aStream
		nextPutAll: self maintenanceTime asMilliseconds valueAsString;
		comma;
		nextPutAll: ((self maintenanceTime percentOf: aStopWatchTime) roundTo: 0.01) printString.

CachingProfiler>>printCacheHitsOn: aStream

	aStream
		nextPutAll: 'Cache Hits: ';
		nextPutAll: self cacheHits printString;
		nextPutAll: ' (';
		nextPutAll: (self cacheHitsPerc roundTo: 0.01) printString;
		nextPutAll: '%)'.

CachingProfiler>>printLookupTimeAsCsvOn: aStream

	aStream
		nextPutAll: self lookupCacheTime asMilliseconds valueAsString.

CachingProfiler>>profileCacheMaintenance: aBlock

	^self subclassResponsibility

CachingProfiler>>printLookupTimeAsCsvOn: aStream enclosedTime: aStopWatchTime

	aStream
		nextPutAll: self lookupCacheTime asMilliseconds valueAsString;
		comma;
		nextPutAll: ((self lookupCacheTime percentOf: aStopWatchTime) roundTo: 0.01) printString.

CachingProfiler>>printAsCsvOn: aStream enclosedTime: aStopWatchTime
```

```
	self printCacheHitsAsCsvOn: aStream.
	aStream comma.
	self printCacheMissAsCsvOn: aStream.
	aStream comma.
	self printLookupTimeAsCsvOn: aStream enclosedTime: aStopWatchTime.
	aStream comma.
	self printMaintenanceTimeAsCsvOn: aStream enclosedTime: aStopWatchTime.
	aStream comma.

CachingProfiler>>lookupCacheTime

	^self subclassResponsibility

CachingProfiler>>printLookedUpMessageOn: aStream

	aStream nextPutAll: 'Total de mensajes buscados: '.
	self lookedUpMessagesInCache printOn: aStream.

CachingProfiler>>printLookupTimeOn: aStream

	aStream
		nextPutAll: 'Tiempo total de Busqueda ';
		nextPutAll: self lookupCacheTime asMilliseconds printString.

CachingProfiler>>printMaintenanceTimeOn: aStream enclosedTime: aStopWatchTime

	aStream
		nextPutAll: 'Total de tiempo de Mantenimiento de la Cache: ';
		nextPutAll: self maintenanceTime asMilliseconds printString;
		nextPutAll: ' (';
		nextPutAll: ((self maintenanceTime percentOf: aStopWatchTime) roundTo: 0.01) printString;
		nextPutAll: '%)'.

CachingProfiler>>totalDeviationTime

	^super totalDeviationTime + self lookupCacheDeviationTime + self maintenanceDeviationTime + self
totalExtraLookupCacheTime + self totalExtraMaintenanceTime.

CachingProfiler>>totalExtraLookupCacheTime

	^self subclassResponsibility

CachingProfiler>>printMaintenanceTimeOn: aStream

	aStream
		nextPutAll: 'Total de tiempo de Mantenimiento ';
		nextPutAll: self maintenanceTime asMilliseconds printString.

CachingProfiler>>printLookedUpMessageAsCsvOn: aStream

	self lookedUpMessagesInCache printOn: aStream.

CachingProfiler>>totalExtraMaintenanceTime

	^self subclassResponsibility

CachingProfiler>>lookupCacheDeviationTime

	^self subclassResponsibility

CachingProfiler>>printOn: aStream

	self printLookedUpMessageOn: aStream.
	aStream cr.
	self printCacheHitsOn: aStream.
	aStream cr.
	self printCacheMissOn: aStream.
	aStream cr.
	self printLookupTimeOn: aStream.
```

```
	aStream cr.
	self printMaintenanceTimeOn: aStream.

CachingProfiler>>maintenanceDeviationTime

	^self subclassResponsibility

CachingProfiler>>printMaintenanceTimeAsCsvOn: aStream

	aStream
		nextPutAll: self maintenanceTime asMilliseconds valueAsString.

CachingProfiler>>profileCacheLookup: aBlock

	^self subclassResponsibility

CachingProfiler>>printOn: aStream enclosedTime: aStopWatchTime

	self printCacheHitsOn: aStream.
	aStream cr.
	self printCacheMissOn: aStream.
	aStream cr.
	self printLookupTimeOn: aStream enclosedTime: aStopWatchTime.
	aStream cr.
	self printMaintenanceTimeOn: aStream enclosedTime: aStopWatchTime.
	aStream cr.

CachingProfiler>>profileExtraCacheLookupTime: aBlock.

	self subclassResponsibility.

CachingProfiler>>cacheHits

	^self subclassResponsibility

CachingProfiler>>printCacheHitsAsCsvOn: aStream

	aStream
		nextPutAll: self cacheHits printString;
		comma;
		nextPutAll: (self cacheHitsPerc roundTo: 0.01) printString.

CachingProfiler>>printCacheMissOn: aStream

	aStream
		nextPutAll: 'Cache Miss: ';
		nextPutAll: self cacheMiss printString;
		nextPutAll: ' (';
		nextPutAll: (self cacheMissPerc roundTo: 0.01) printString;
		nextPutAll: '%)'.

CachingProfiler>>profileExtraMaintenanceTime: aBlock.

	self subclassResponsibility.

CachingProfiler>>printLookupTimeOn: aStream enclosedTime: aStopWatchTime

	aStream
		nextPutAll: 'Tiempo total de Bœsqueda en Cache: ';
		nextPutAll: self lookupCacheTime asMilliseconds printString;
		nextPutAll: ' (';
		nextPutAll: ((self lookupCacheTime percentOf: aStopWatchTime) roundTo: 0.01) printString;
		nextPutAll: '%)'.

CachingProfiler>>printCacheMissAsCsvOn: aStream

	aStream
		nextPutAll: self cacheMiss printString;
		comma;
		nextPutAll: (self cacheMissPerc roundTo: 0.01) printString.
```

```
CachingProfiler>>cacheMiss

	^self subclassResponsibility

CachingProfiler>>lookedUpMessagesInCache

	^self subclassResponsibility

CachingProfiler>>methodLookupTime

	^super methodLookupTime - ( self lookupCacheDeviationTime + self maintenanceDeviationTime + self
totalExtraLookupCacheTime + self totalExtraMaintenanceTime).

CachingProfiler>>cacheMissPerc

	^self cacheMiss*100/self lookedUpMessagesInCache.

CachingProfiler>>cacheHitsPerc

	^self cacheHits*100/self lookedUpMessagesInCache.

CachingProfiler>>incrementCacheMiss

	^self subclassResponsibility

CachingProfiler class>>new

	^super new
		initialize;
		yourself.
```

11.6.2 BasicCachingProfiler

```
CachingProfiler subclass: #BasicCachingProfiler
	instanceVariableNames: 'totalLookedUpMessages totalCacheMiss lookupStopWatch maintenanceStopWatch
extraLookupTimeStopWatch extraMaintenanceTimeStopWatch '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Thesis-Caching Profiling'!

BasicCachingProfiler>>totalExtraLookupCacheTime

	^extraLookupTimeStopWatch expendedTime + extraLookupTimeStopWatch deviationTime.

BasicCachingProfiler>>initialize

	super initialize.
	self totalLookedUpMessages: 0.
	self totalCacheMiss: 0.
	lookupStopWatch _ StopWatch new.
	maintenanceStopWatch _ StopWatch new.
	extraLookupTimeStopWatch _ StopWatch new.
	extraMaintenanceTimeStopWatch _ StopWatch new.

BasicCachingProfiler>>lookupCacheDeviationTime

	^lookupStopWatch deviationTime.

BasicCachingProfiler>>totalExtraMaintenanceTime

	^extraMaintenanceTimeStopWatch expendedTime + extraMaintenanceTimeStopWatch deviationTime.

BasicCachingProfiler>>maintenanceTime

	^maintenanceStopWatch expendedTime - self totalExtraMaintenanceTime.

BasicCachingProfiler>>lookupCacheTime

	^lookupStopWatch expendedTime - self totalExtraLookupCacheTime.
```

```
BasicCachingProfiler>>profileCacheLookup: aBlock

	totalLookedUpMessages _ totalLookedUpMessages + 1.
	lookupStopWatch measure: aBlock.

BasicCachingProfiler>>incrementCacheMiss

	totalCacheMiss _ totalCacheMiss + 1

BasicCachingProfiler>>profileExtraMaintenanceTime: aBlock

	extraMaintenanceTimeStopWatch measure: aBlock.

BasicCachingProfiler>>startProfiling

	super startProfiling.
	lookupStopWatch clear.
	maintenanceStopWatch clear.
	extraLookupTimeStopWatch clear.
	extraMaintenanceTimeStopWatch clear.

BasicCachingProfiler>>profileCacheMaintenance: aBlock

	maintenanceStopWatch measure: aBlock.

BasicCachingProfiler>>cacheMiss

	^totalCacheMiss

BasicCachingProfiler>>totalLookedUpMessages: aNumber

	totalLookedUpMessages _ aNumber.

BasicCachingProfiler>>totalCacheMiss: aNumber

	totalCacheMiss _ aNumber

BasicCachingProfiler>>profileExtraCacheLookupTime: aBlock

	extraLookupTimeStopWatch measure: aBlock.

BasicCachingProfiler>>maintenanceDeviationTime

	^maintenanceStopWatch deviationTime.

BasicCachingProfiler>>lookedUpMessagesInCache

	^totalLookedUpMessages

BasicCachingProfiler>>cacheHits

	^self lookedUpMessagesInCache-self cacheMiss.
```

11.6.3 NullCachingProfiler

```
CachingProfiler subclass: #NullCachingProfiler
	instanceVariableNames: ''
	classVariableNames: 'Singleton '
	poolDictionaries: ''
	category: 'Thesis-Caching Profiling'!

NullCachingProfiler>>cacheHits

	^0

NullCachingProfiler>>startProfiling

NullCachingProfiler>>lookupCacheTime
```

```
		^StopWatchTime cero.
NullCachingProfiler>>profileCacheLookup: aBlock
		aBlock value.
NullCachingProfiler>>profileMethodLookup: aBlock
		aBlock value.
NullCachingProfiler>>lookupCacheDeviationTime
		^StopWatchTime cero.
NullCachingProfiler>>methodLookupDeviationTime
		^StopWatchTime cero.
NullCachingProfiler>>selector: messageSelector sendTo: lookupClass
NullCachingProfiler>>maintenanceTime
		^StopWatchTime cero.
NullCachingProfiler>>profileCacheMaintenance: aBlock
		aBlock value.
NullCachingProfiler>>methodLookupTime
		^StopWatchTime cero.
NullCachingProfiler>>stopProfiling
NullCachingProfiler>>sendMessages
		^0.
NullCachingProfiler>>maintenanceDeviationTime
		^StopWatchTime cero.
NullCachingProfiler>>extraTime
		^StopWatchTime cero.
NullCachingProfiler>>cacheMiss
		^0
NullCachingProfiler>>incrementCacheMiss
NullCachingProfiler>>extraDeviationTime
		^StopWatchTime cero.

NullCachingProfiler>>lookedUpMessagesInCache
		^0
NullCachingProfiler>>profileExtraTime: aBlock
		aBlock value.
NullCachingProfiler class>>initializeSingleton
		Singleton _ super new.
```

```
NullCachingProfiler class>>new

	Singleton isNil ifTrue: [ self initializeSingleton ].
	^Singleton.
```

11.7 Thesis-DTS With Classes Support

11.7.1 DTSWithClassesMLProxy

```
ConfigMLInterpreterProxy subclass: #DTSWithClassesMLProxy
	instanceVariableNames: ''
	classVariableNames: 'SelectorsDictionary '
	poolDictionaries: ''
	category: 'Thesis-DTS with Classes'!

DTSWithClassesMLProxy class>>selectorsDictionary

	^SelectorsDictionary

DTSWithClassesMLProxy class>>initializeInterpreter

	"Assumes that SelectorsDictionary has the right values"
	ConfigMLInterpreterProxy runPrimitiveMessageToMethodLookupStrategy: #setSelectorsDictionary: with: self
selectorsDictionary.

DTSWithClassesMLProxy class>>initializeSelectorsDictionary

	| selector method selectorDictionary |
	SelectorsDictionary _ IdentityDictionary new.
	Smalltalk allBehaviorsDo:
		[:aBehavior |
		aBehavior methodDictionary associationsDo:
			[:anAssoc |
			selector _ anAssoc key.
			method _ anAssoc value.
			selectorDictionary _ SelectorsDictionary at: selector ifAbsent:
				[ SelectorsDictionary at: selector put: IdentityDictionary new.].
			selectorDictionary at: aBehavior put: method.
			].
		].

	SelectorsDictionary rehash.
	SelectorsDictionary do: [ :aDictionary | aDictionary rehash ].
	Smalltalk garbageCollect.

DTSWithClassesMLProxy class>>printStatistics

	Transcript
		show: DTSWithClassesMLProxy dtsDictionariesSize average asFloat printString; cr;
		show: DTSWithClassesMLProxy dtsWithClassesDictionariesSizes average asFloat printString; cr;
		show: DTSWithClassesMLProxy dtsDictionariesSize max asFloat  printString; cr;
		show: DTSWithClassesMLProxy dtsWithClassesDictionariesSizes max asFloat  printString; cr;
		show: DTSWithClassesMLProxy dtsDictionariesSize min asFloat printString; cr;
		show: DTSWithClassesMLProxy dtsWithClassesDictionariesSizes min asFloat printString; cr;
		show: DTSWithClassesMLProxy dtsDictionariesSize median asFloat  printString; cr;
		show: DTSWithClassesMLProxy dtsWithClassesDictionariesSizes median asFloat   printString; cr.

DTSWithClassesMLProxy class>>dtsDictionariesSize

	|  allBehaviors |

	allBehaviors _ Object allSubclasses
		add: Object;
		yourself.
	^allBehaviors asOrderedCollection collect: [ :aBehavior | aBehavior methodDictionary size. ].

DTSWithClassesMLProxy class>>dtsWithClassesDictionariesSizes

	^self selectorsDictionary collect: [ :aSelectorDictionary | aSelectorDictionary size. ].
```

11.8 Thesis-FullDictionary Support

11.8.1 FullDictionaryML

```
Object subclass: #FullDictionaryML
	instanceVariableNames: ''
	classVariableNames: 'FullDictionaries OriginalMethodDictionaries '
	poolDictionaries: ''
	category: 'Thesis-Full Dictionary'!

FullDictionaryML class>>selectDTS

	OriginalMethodDictionaries keysDo: [ :aBehavior |
		aBehavior recreateMethodDictionary. ].

FullDictionaryML class>>createFullDictionaryFor: aBehavior

	| fullDictionary |
	fullDictionary _ MethodDictionary new.
	aBehavior allSelectors do: [ :aSelector |
		fullDictionary at: aSelector put: (aBehavior lookupSelector: aSelector ) ].

	fullDictionary rehash.
	^fullDictionary.

FullDictionaryML class>>numberOfFullDictionaryEntries

	| number |
	self selectFullDictionary.
	number _ OriginalMethodDictionaries keys inject: 0 into: [ :aNumber :aBehavior |
		aNumber _ aNumber + aBehavior methodDictionary size. ].
	self selectDTS.

	^number.

FullDictionaryML class>>initializeFor: aCollectionOfSuperclasses

	self initializeOriginalMethodDictionaries: aCollectionOfSuperclasses.
	self initializeFullDictionaries: aCollectionOfSuperclasses.

FullDictionaryML class>>selectFullDictionary

	FullDictionaries keysDo: [ :aBehavior |
		aBehavior methodDictionary: (FullDictionaries at: aBehavior). ].

FullDictionaryML class>>initializeOriginalMethodDictionaries: aCollectionOfSuperclasses

	OriginalMethodDictionaries _ Dictionary new.
	aCollectionOfSuperclasses do: [ :aSuperclass |
		OriginalMethodDictionaries at: aSuperclass put: aSuperclass methodDictionary.
		aSuperclass allSubclasses do: [ :aBehavior |
			OriginalMethodDictionaries at: aBehavior put: aBehavior methodDictionary. ]. ].

FullDictionaryML class>>test

	|vtbl |

	FullDictionaries keysDo: [ :aBehavior |
		vtbl _ FullDictionaries at: aBehavior.
		aBehavior allSelectors do: [ :aSelector |
			(aBehavior lookupSelector: aSelector)==(vtbl at: aSelector) ifFalse: [
				self error: 'Test failed'. ].
		].
	].

	^true.

FullDictionaryML class>>numberOfDTSEntries
```

```
	self selectDTS.
	^OriginalMethodDictionaries keys inject: 0 into: [ :aNumber :aBehavior |
		aNumber _ aNumber + aBehavior methodDictionary size. ].

FullDictionaryML class>>initializeFullDictionaries: aCollectionOfSuperclasses

	FullDictionaries _ Dictionary new.
	aCollectionOfSuperclasses do: [ :aSuperclass |
		FullDictionaries at: aSuperclass put: (self createFullDictionaryFor: aSuperclass ).
		aSuperclass allSubclasses do: [ :aBehavior |
			FullDictionaries at: aBehavior put: (self createFullDictionaryFor: aBehavior ). ]. ].
```

11.9 Thesis-Hierarchy Branch Support

11.9.1 ClassInterval

```
Object subclass: #ClassInterval
	instanceVariableNames: 'from to '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Thesis-Hierarchy Branch'!

ClassInterval>>printOn: aStream

	aStream nextPutAll: 'From: '.
	self from printOn: aStream.
	aStream nextPutAll: ' To: '.
	self to printOn: aStream.

ClassInterval>>partitionateUsing: aClassInterval

	| currentTo newInterval |
	currentTo _ self to.
	self to: (aClassInterval from).
	(currentTo=aClassInterval to) ifTrue: [
		newInterval _ nil. ]
	ifFalse: [
		newInterval _ self shallowCopy.
		newInterval from: (aClassInterval to);
			to: currentTo.
	].
	^newInterval.

ClassInterval>>includes: aNumberOrClassNumbers

	"Verifies that the range of numbers in aClassNumbers are between the range of self.
	 It is inclusive"

	(aNumberOrClassNumbers isKindOf: Number ) ifTrue: [
		^(self from<=aNumberOrClassNumbers) and: [ self to>aNumberOrClassNumbers ].]
	ifFalse: [
		^(self from<=aNumberOrClassNumbers from) and: [ self to>=aNumberOrClassNumbers to ].].

ClassInterval>>to

	^to

ClassInterval>>to: aNumber

	to _ aNumber

ClassInterval>>from: aNumber

	from _ aNumber

ClassInterval>>from

	^from
```

```
ClassInterval class>>from: from to: to

	^super new
		from: from;
		to: to.
```

11.9.2 HierarchyBranchML

```
Object subclass: #HierarchyBranchML
	instanceVariableNames: ''
	classVariableNames: 'ClassIntervals SelectorImplementations '
	poolDictionaries: ''
	category: 'Thesis-Hierarchy Branch'!

HierarchyBranchML class>>lookupMethod: aSelector class: aClass

	| messageImplementationCollection classNumber |

	messageImplementationCollection _ self selectorImplementationsFor: aSelector.
	classNumber _ (self classIntervalFor: aClass) from.
	messageImplementationCollection do: [ :aMessageImplementation |
		(aMessageImplementation includes: classNumber) ifTrue: [ ^aMessageImplementation method. ].
	].

	^nil.

HierarchyBranchML class>>reorderImplementationsPostCondition: aSortedCollection original: anOriginalCollection
addedCollection: addedCollection
	"This method verifies if the reordering was done ok. Just for test."

	| index messageImplementationA messageImplementationB |

	index _ 1.
	[index < aSortedCollection size] whileTrue: [
		messageImplementationA _ aSortedCollection at: index.
		messageImplementationB _ aSortedCollection at: (index + 1).
		messageImplementationA to=messageImplementationB from ifTrue: [
			messageImplementationA method==messageImplementationB method ifTrue: [
				self error: 'Error en el algoritmo de reordenamiento'. ].
		]
		ifFalse: [
			messageImplementationA to>messageImplementationB from ifTrue: [
				self error: 'Error en el algoritmo de reordenamiento'. ].
		].

		index _ index + 1.
	].


HierarchyBranchML class>>testBothDirections

	self testFromMethodDictionaryToSelectorImplementations.
	self testFromSelectorImplementationsToMethodDictionary.

HierarchyBranchML class>>classFor: aClassNumber ifAbsent: aBlock

	ClassIntervals associationsDo: [ :assoc |
		(aClassNumber=assoc value from) ifTrue: [
			^assoc key. ].
	].

	^aBlock value.

HierarchyBranchML class>>selectorImplementationsFor: aSelector

	^self selectorImplementations at: aSelector

HierarchyBranchML class>>classIntervals: aDictionary

	ClassIntervals _ aDictionary
```

```
HierarchyBranchML class>>testFromMethodDictionaryToSelectorImplementations

	self testMethodsForBehavior: Object.

HierarchyBranchML class>>testFromSelectorImplementationsToMethodDictionary

	self selectorImplementations associationsDo: [ :anAssociation |
		self testMethodFor: anAssociation key inMessageImplementationCollection: anAssociation value.
	].

HierarchyBranchML class>>reorderEqualFrom: messageImplementationA  b: messageImplementationB collection:
aSortedCollection addedCollection: addedCollection

	"The only reason why there are to intervals with the same from is because one
	is the product of a reordering"
	(addedCollection includes: messageImplementationA) ifTrue: [
		"If the smallest o equal interval was the added, just remove it becuase its method it is not
		the right one. The right one is the original."
		aSortedCollection remove: messageImplementationA.]
	ifFalse: [
		(addedCollection includes: messageImplementationB) ifTrue: [
			aSortedCollection remove: messageImplementationB.
			"Now I have to verify is there is some classes that still have the method referenced by
			messageImplementationB"
			(messageImplementationA to<messageImplementationB to) ifTrue: [
				messageImplementationB from: (messageImplementationA to).
				aSortedCollection add: messageImplementationB.]
			ifFalse: [
				(messageImplementationA to>messageImplementationB to) ifTrue: [
					self error: 'Error en algoritmo de reordenamiento'. ].
			].
		]
		ifFalse: [
				self error: 'Error en algoritmo de reordenamiento'
		].
	].

HierarchyBranchML class>>numerateClassesFrom: aClass startingWith: aNumber on: aClassIntervals

	| aClassFrom |
	aClassFrom _ aNumber + 1.
	aClass subclassesDo:
		[ :aSubClass |
		aClassFrom _ self numerateClassesFrom: aSubClass startingWith: aClassFrom on: aClassIntervals.].

	aClassIntervals at: aClass put: (ClassInterval from: aNumber to: aClassFrom).
	^aClassFrom.

HierarchyBranchML class>>classIntervals

	^ClassIntervals

HierarchyBranchML class>>createSelectorImplementations

	| allSelectorsImplementations selector method aBehaviorInterval implementationIntervals |

	allSelectorsImplementations _ Dictionary new.
	Smalltalk allBehaviorsDo:
		[ :aBehavior |
		aBehaviorInterval := self classIntervalFor: aBehavior.
		aBehavior methodDictionary associationsDo:
			[ :association |
			selector _ association key.
			method _ association value.
			implementationIntervals _ allSelectorsImplementations at: selector ifAbsent:
				[implementationIntervals _ SortedCollection sortBlock: [ :a :b |
					(a from < b from) or: [ a from=b from and: [ a to<=b to ] ].].
				allSelectorsImplementations at: selector put: implementationIntervals. ].
			implementationIntervals add:
```

```
                                        (MessageImplementationBranch classInterval: aBehaviorInterval
                                        method: method). ].
                ].

        self reorderAllImplementations: allSelectorsImplementations.
        allSelectorsImplementations rehash.
        ^allSelectorsImplementations.

HierarchyBranchML class>>selectorImplementations

        ^SelectorImplementations

HierarchyBranchML class>>testMethodsForBehavior: aBehavior

        | selector |

        aBehavior methodDictionary associationsDo: [ :anAssociation |
                selector _ anAssociation key.
                (self lookupMethod: selector class: aBehavior)==(anAssociation value) ifFalse: [
                        (selector==#DoIt and: [aBehavior==UndefinedObject]) ifFalse: [
                        self error: 'Error de ', selector printString, ' en clase ', aBehavior printString.]. ].
        ].

        aBehavior subclassesDo: [ :aSubclass |
                self testMethodsForBehavior: aSubclass ].

HierarchyBranchML class>>initializeClassIntervals

        ClassIntervals _ self createClassIntervals.

HierarchyBranchML class>>unifyContinuousIntervals: aSortedCollection

        | index messageImplementationsA messageImplementationsB |

        index _ 1.
        [index < aSortedCollection size] whileTrue: [
                messageImplementationsA _ aSortedCollection at: index.
                messageImplementationsB _ aSortedCollection at: index+1.
                ((messageImplementationsA to=messageImplementationsB from) and: [
                messageImplementationsA method==messageImplementationsB method]) ifTrue: [
                        messageImplementationsA to: messageImplementationsB to.
                        aSortedCollection remove: messageImplementationsB. ]
                ifFalse: [
                        index _ index + 1. ].
        ].

HierarchyBranchML class>>printBranchesOn: aStream

        | branches |

        branches _ self branches.
        branches keys asSortedCollection do: [ :aNumberOfImpl |
                aStream nextPutAll: aNumberOfImpl printString;
                        nextPutAll: ', ';
                        nextPutAll: (branches at: aNumberOfImpl) key printString;
                        cr. ].
HierarchyBranchML class>>classIntervalFor: aBehavior

        ^ClassIntervals at: aBehavior

HierarchyBranchML class>>reorderImplementations: aSortedCollection

        | index messageImplementationA messageImplementationB messageImplementationC originalCollection
addedCollection |

        originalCollection _ aSortedCollection species sortBlock: (aSortedCollection sortBlock).
        aSortedCollection do: [ :anObject | originalCollection add: (anObject shallowCopy). ].
        addedCollection _ OrderedCollection new.

        index _ 1.
```

```
	[index < aSortedCollection size] whileTrue: [
		messageImplementationA _ aSortedCollection at: index.
		messageImplementationB _ aSortedCollection at: (index + 1).
		"two from can only be equal after breaking an interval"
		(messageImplementationA from=messageImplementationB from) ifTrue: [
			"The to can be equal or less than because of the order"
			self reorderEqualFrom: messageImplementationA  b: messageImplementationB collection:
aSortedCollection addedCollection: addedCollection.
		]
		ifFalse: [
			(messageImplementationA includes: messageImplementationB) ifTrue: [
				messageImplementationC _ messageImplementationA
					partitionateUsing: messageImplementationB.
				messageImplementationC isNil ifFalse: [
					aSortedCollection add: messageImplementationC.
					addedCollection add: messageImplementationC.].
			].
		index _ index + 1.
		].
	].

	self unifyContinuousIntervals: aSortedCollection.
	self reorderImplementationsPostCondition: aSortedCollection original: originalCollection addedCollection:
addedCollection.

HierarchyBranchML class>>testMethodFor: aSelector inMessageImplementation: aMessageImplementation

	| aBehavior |
	aMessageImplementation from to: (aMessageImplementation to-1) do: [ :aNumber |
		aBehavior _ self behaviorFor: aNumber.
		(aBehavior lookupMethod: aSelector)==(aMessageImplementation method) ifFalse: [
			(aSelector==#DoIt and: [aBehavior==UndefinedObject]) ifFalse: [
				self error: 'Error en ', aSelector printString, ' en clase ', aBehavior printString.]. ].
	].
HierarchyBranchML class>>classFor: aClassNumber

	^self classFor: aClassNumber ifAbsent: [self error: 'Clase no encontrada'].

HierarchyBranchML class>>selectorImplementations: aDictionary

	SelectorImplementations _ aDictionary.

HierarchyBranchML class>>behaviorFor: aNumber

	self classIntervals associationsDo: [ :anAssociation |
		anAssociation value from=aNumber ifTrue: [ ^anAssociation key ].
	].

	^nil.

HierarchyBranchML class>>initialize

	self initializeClassIntervals.
	self initializeSelectorImplementations.
	Smalltalk garbageCollect.

HierarchyBranchML class>>createClassIntervals

	| classIntervals |

	classIntervals _ IdentityDictionary new.
	self numerateClassesFrom: Object startingWith: 1 on: classIntervals.
	classIntervals rehash.
	^classIntervals.

HierarchyBranchML class>>compiledMethodFor: aSelector of: aClass

	| classInterval methodFinder |

	classInterval _ self classIntervalFor: aClass.
```

```
	methodFinder _ self methodFinderFor: aSelector.

	^methodFinder compiledMethodFor: classInterval.

HierarchyBranchML class>>initializeSelectorImplementations

	SelectorImplementations _ self createSelectorImplementations.

HierarchyBranchML class>>testMethodFor: aSelector inMessageImplementationCollection:
aMessageImplementationCollection

	aMessageImplementationCollection do: [ :aMessageImplementation |
		self testMethodFor: aSelector inMessageImplementation: aMessageImplementation ].

HierarchyBranchML class>>branches

	| branches size assoc |

	branches _ Dictionary new.
	self selectorImplementations associationsDo: [ :anAssociation |
		size _ anAssociation value size.
		assoc _ branches at: size ifAbsent: [
			branches at: size put: (Association key:0 value: OrderedCollection new).].
		assoc key: assoc key + 1.
		assoc value add: (anAssociation key).
	].

	^branches.

HierarchyBranchML class>>reorderAllImplementations: aDictionary

"	self halt.
	self reorderImplementations: (aDictionary at: #newFrom:)."

	aDictionary associationsDo: [ :anAssociation |
		"Transcript show: anAssociation key; cr."
		self reorderImplementations: anAssociation value. ].
```

11.9.3 HierarchyBranchWithDicML

```
HierarchyBranchML subclass: #HierarchyBranchWithDicML
	instanceVariableNames: ''
	classVariableNames: 'ImplementationThreadshold SelectorImplementationsWithDic '
	poolDictionaries: ''
	category: 'Thesis-Hierarchy Branch'!

HierarchyBranchWithDicML class>>initialize

	super initialize.
	self initializeSelectorsImplementationsWithDic.
	Smalltalk garbageCollect.

HierarchyBranchWithDicML class>>numberOfDictionaries

	^self selectorImplementationsWithDic count: [ :aCollection | aCollection class=Dictionary ].

HierarchyBranchWithDicML class>>addAsSortedCollection: aSelectorCollection

	aSelectorCollection do: [ :selector |
		SelectorImplementationsWithDic at: selector put: (super selectorImplementationsFor: selector).].

HierarchyBranchWithDicML class>>selectorImplementationsWithDic

	^SelectorImplementationsWithDic

HierarchyBranchWithDicML class>>implementationThreadshold

	^ImplementationThreadshold.

HierarchyBranchWithDicML class>>numberOfSortedCollections
```

```
	^self selectorImplementationsWithDic count: [ :aCollection | aCollection class=SortedCollection ].

HierarchyBranchWithDicML class>>addAsDictionary: aSelectorCollection

	aSelectorCollection do: [ :selector |
		SelectorImplementationsWithDic at: selector put: (self createMethodDictionaryFor: selector).].

HierarchyBranchWithDicML class>>implementationThreadshold: aNumber

	ImplementationThreadshold _ aNumber.

HierarchyBranchWithDicML class>>createMethodDictionaryFor: aSelector

	| implementations methodDictionary |

	methodDictionary _ Dictionary new.
	implementations _ self selectorImplementationsFor: aSelector.
	implementations do: [ :messageImplementationBranch |
		messageImplementationBranch from to: (messageImplementationBranch to-1) do: [:classNumber |
			methodDictionary at: (self classFor: classNumber)
				put: (messageImplementationBranch method).].
	].

	methodDictionary rehash.
	^methodDictionary
HierarchyBranchWithDicML class>>numberOfDictionaryEntries

	^self selectorImplementationsWithDic inject: 0 into: [ :valor :dic |
		dic class=Dictionary ifTrue: [
			valor _ valor + dic size. ].
		valor. ]

HierarchyBranchWithDicML class>>initializeSelectorsImplementationsWithDic

	| branches selectors |

	SelectorImplementationsWithDic _ Dictionary new.
	branches _ self branches.
	branches associationsDo: [ :association |
		selectors _ association value value.
		association key <= self implementationThreadshold ifTrue: [
			self addAsSortedCollection: selectors.
		]
		ifFalse: [
			self addAsDictionary: selectors.
		].
	].
```

11.9.4 MessageImplementationBranch

```
ClassInterval subclass: #MessageImplementationBranch
	instanceVariableNames: 'method '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Thesis-Hierarchy Branch'!

MessageImplementationBranch>>method

	^method

MessageImplementationBranch>>method: aMethod

	method _ aMethod.

MessageImplementationBranch class>>classInterval: aClassInterval method: aMethod

	^self new
		from: aClassInterval from;
		to: aClassInterval to;
```

```
		method: aMethod;
		yourself.
```

11.10 Thesis-Statistics Support

11.10.1 ConfigMLInterpreterProxy

```
Object subclass: #ConfigMLInterpreterProxy
	instanceVariableNames: ''
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Thesis-Statistics Support'!

ConfigMLInterpreterProxy class>>runPrimitiveUseMethodLookup: aMethodLookupKey initializationArray: anArrayOrNil
	<primitive: 572>
	self error: 'Error running primitive #runPrimitiveUseMethodLookup:'.

ConfigMLInterpreterProxy class>>runPrimitiveMessageToMethodLookupStrategy: aMessage with: aCollaborator
	<primitive: 573>
	self error: 'Error running primitive #runPrimitiveMessageToMethodLookupStrategy:with:'.

ConfigMLInterpreterProxy class>>runPrimitiveUseCacheMethodLookup: aMethodLookupKey initializationArray: anArrayOrNil
cacheInitializationArray: anotherInitializationArrayOrNil

	<primitive: 573>
	self error: 'Error running primitive #runPrimitiveUseMethodLookup:'.

ConfigMLInterpreterProxy class>>runPrimitiveMessageToMethodLookupStrategy: aMessage
	<primitive: 573>
	self error: 'Error running primitive #runPrimitiveMessageToMethodLookupStrategy:'.

ConfigMLInterpreterProxy class>>runPrimitiveStartTest: aName
	<primitive: 570>
	self error: 'Error running primitive #runPrimitiveStartTest:'.

ConfigMLInterpreterProxy class>>runPrimitiveStopTest: aStatsPrintingType

	<primitive:571>
	self error: 'Error running primitive #runPrimitiveStopTest'.
```

11.10.2 MessageSentAnalizer

```
Object subclass: #MessageSentAnalizer
	instanceVariableNames: 'messageDictionary analizedSet totalMessagesSend '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Thesis-Statistics Support'!

MessageSentAnalizer>>printResultAsCsv: aFileName on: aStream
	| messageSends |
	aStream nextPutAll: aFileName.
	1 to: 6 do: [ :numberOfImplementators |
		messageSends _ self numberOfSendsFor: numberOfImplementators with: [ :a :b | a=b ].
		aStream
			nextPutAll: ',';
			nextPutAll: (numberOfImplementators asString);
			nextPutAll: ',';
			nextPutAll: ((messageSends*100/totalMessagesSend) asInteger asString).].

	messageSends _ self numberOfSendsFor: 6 with: [ :a :b | a<b ].
	aStream
		nextPutAll: ',7,';
		nextPutAll: ((messageSends*100/totalMessagesSend) asInteger asString).

MessageSentAnalizer>>printOn: aStream

	| messageSends |
	1 to: 6 do: [ :numberOfImplementators |
```

```
                messageSends _ self numberOfSendsFor: numberOfImplementators with: [ :a :b | a=b ].
                self print: numberOfImplementators with: messageSends on: aStream.].

        messageSends _ self numberOfSendsFor: 6 with: [ :a :b | a<b ].
        self print: 6 with: messageSends on: aStream.
MessageSentAnalizer>>print: numberOfImplementators with: messageSends on: aStream
        aStream nextPutAll: 'Messages Send with ';
                nextPutAll: (numberOfImplementators asString);
                nextPutAll: ' implementation(s): ';
                nextPutAll: (messageSends asString);
                nextPutAll: ', ';
                nextPutAll: ((messageSends*100/totalMessagesSend) asInteger asString);
                nextPutAll: '%';
                cr.

MessageSentAnalizer>>numberOfSendsFor: numberOfImplementators with: aBooleanBlock
        ^ analizedSet inject: 0 into: [ :preValue :messageStatistics |
                (aBooleanBlock value: numberOfImplementators value: messageStatistics numberOfImplementators)
                ifTrue: [preValue _ preValue + messageStatistics numberOfSends.].
                preValue.].

MessageSentAnalizer>>readFile: aFileName
        | stream |
        stream _ ReferenceStream fileNamed: aFileName.
        messageDictionary _ Dictionary new.
        messageDictionary _ stream next.
        stream close.

MessageSentAnalizer>>printResult
        | string |
        string _ String streamContents: [ :aStream |
                self printOn: aStream. ].
        Transcript show: string.

MessageSentAnalizer>>basicAnalize
        | selector selectorStatistics |
        totalMessagesSend _ 0.
        analizedSet _ Set new.
        messageDictionary associationsDo: [ :association |
                "Gets the selector"
                selector _ association key allButFirst asSymbol.
                selectorStatistics _ SelectorStatistics on: selector.
                "Sets the number of implementators"
                selectorStatistics numberOfImplementators: (Smalltalk allImplementorsOf: selector) size.
                "Saves the number of sents..."
                selectorStatistics numberOfSends: ((association value) inject:0 into: [ :preValue :newValue | preValue _
preValue + newValue]).
                totalMessagesSend _ totalMessagesSend + selectorStatistics numberOfSends.
                analizedSet add: selectorStatistics.].

MessageSentAnalizer>>analize: aFileName
        self readFile: aFileName.
        self basicAnalize.

MessageSentAnalizer>>printResultAsCsv: aFileName
        | string |
        string _ String streamContents: [ :aStream | self printResultAsCsv: aFileName on: aStream.
                aStream cr. ].

        Transcript show: string.

MessageSentAnalizer class>>fileNamed: aString
        ^ super fileNamed: aString.
```

11.10.3 SelectorStatistics

```
Object subclass: #SelectorStatistics
        instanceVariableNames: 'selector numberOfImplementators numberOfSends '
        classVariableNames: ''
        poolDictionaries: ''
```

```
		category: 'Thesis-Statistics Support'!

SelectorStatistics>>numberOfImplementators: aNumber
		numberOfImplementators _ aNumber.

SelectorStatistics>>selector
		^selector.

SelectorStatistics>>numberOfSends: aNumber
		numberOfSends _ aNumber.

SelectorStatistics>>numberOfSends
		^numberOfSends.

SelectorStatistics>>numberOfImplementators
		^numberOfImplementators.

SelectorStatistics>>selector: aSelector
		selector _ aSelector.

SelectorStatistics class>>on: aSelector
		^super new selector: aSelector.
```

11.10.4 ThesisBenchmarks

```
Object subclass: #ThesisBenchmarks
	instanceVariableNames: ''
	classVariableNames: 'StatsPrintingType '
	poolDictionaries: ''
	category: 'Thesis-Statistics Support'!

ThesisBenchmarks>>runAllTests
	self runAllTestsWithMethodLookupKey: 0.

ThesisBenchmarks>>testFileOut
	| fileStream |
	FileDirectory deleteFilePath: ThesisBenchmarks FileInOutName.
	fileStream _ FileStream fileNamed: ThesisBenchmarks FileInOutName.
	AbstractSound fileOutOn: fileStream.
	fileStream close.

ThesisBenchmarks>>perform: aSymbol times: aNumber

	aNumber timesRepeat: [ self perform: aSymbol ].
	Smalltalk beep.
	Smalltalk snapshot: false andQuit: true.

ThesisBenchmarks>>runTest: testNameSymbol times: aCounter withMethodLookupKey: aMethodLookupKey

	self runTest: testNameSymbol times: aCounter withMethodLookupKey: aMethodLookupKey initializationArray: nil.

ThesisBenchmarks>>runTest: testNameSymbol times: aCounter
	self runTest: testNameSymbol times: aCounter withMethodLookupKey: 0.

ThesisBenchmarks>>runTest: testNameSymbol
	self runTest: testNameSymbol times: 1.

ThesisBenchmarks>>runAllTestsUsingGLCWithFullDictionaries

	FullDictionaryML selectFullDictionary.
	self runAllTestsWithMethodLookupKey: 11
		initializationArray: nil
		cacheInitializationArray: self globalLookupCacheInitializationBlock.
	FullDictionaryML selectDTS.

ThesisBenchmarks>>testPrintHierarchy
	| stringHierarchy |
	stringHierarchy _ Number printHierarchy.
	Transcript show: stringHierarchy.

ThesisBenchmarks>>runAllTestsUsingDTSWithClassesML

	self runAllTestsWithMethodLookupKey: 3
		initializationArray: self dtsWithClassesInitializationArray.

ThesisBenchmarks>>runAllTestsUsingHBCachingWithHBML

	self error: 'Invalid'.

ThesisBenchmarks>>runAllTestsUsingSLCWithDTS

	self runAllTestsWithMethodLookupKey: 9
		initializationArray: nil
		cacheInitializationArray: nil.

ThesisBenchmarks>>runAllTestsWithMethodLookupKey: aMethodLookupKey
	^self runAllTests: 1 withMethodLookupKey: aMethodLookupKey

ThesisBenchmarks>>runAllTestsUsingHBEraseAllCachingStrategy

	self runAllTestsWithMethodLookupKey: 12
		initializationArray: self hierarchyBranchInitializationArray
```

```
		cacheInitializationArray: nil.

ThesisBenchmarks>>testRestoreDisplay
	ScheduledControllers restore.
	ScheduledControllers activeController view emphasize

ThesisBenchmarks>>hierarchyBranchWithDicInitializationArray

	^ Array
			with: (Association key: 'classIntervals:' value: HierarchyBranchWithDicML classIntervals)
			with: (Association key: 'selectorsImplementations:' value: HierarchyBranchWithDicML
selectorImplementationsWithDic)
			with: (Association key: 'sortedCollection:' value: SortedCollection).

ThesisBenchmarks>>testDecompiler
	| decompiler method |
	method _ ThesisBenchmarks class compiledMethodAt: #LongMethod.
	decompiler _ ThesisBenchmarks class decompilerClass new.
	^decompiler decompile: #LongMethod
		in: ThesisBenchmarks class
		method: method.

ThesisBenchmarks>>globalLookupCacheInitializationBlock

	^ Array
		with: (Association key: 'methodCacheEntries:' value: 512).

ThesisBenchmarks>>statsPrintingType

	^self class statsPrintingType.

ThesisBenchmarks>>runAllTestsUsingHierarchyBranchWithDicML

	self runAllTestsWithMethodLookupKey: 5
		initializationArray: self hierarchyBranchWithDicInitializationArray.

ThesisBenchmarks>>runAllTestsUsingHBCacheCircularAddCachingStrategy

	self runAllTestsWithMethodLookupKey: 14
		initializationArray: self hierarchyBranchInitializationArray
		cacheInitializationArray: nil.

ThesisBenchmarks>>testFileIn
	|fileStream|
	fileStream _ FileStream readOnlyFileNamed: ThesisBenchmarks FileInOutName.
	fileStream fileIn.

ThesisBenchmarks>>runAllTestsUsingGLCWithHBWithDic

	self runAllTestsWithMethodLookupKey: 8
		initializationArray: self hierarchyBranchWithDicInitializationArray
		cacheInitializationArray: self globalLookupCacheInitializationBlock.

ThesisBenchmarks>>runAllTests: aTimesNumber withMethodLookupKey: aMethodLookupKey
	self runAllTests: aTimesNumber withMethodLookupKey: aMethodLookupKey initializationArray: nil.

ThesisBenchmarks>>runAllTestsUsingGLCWithDTS

"	FullDictionaryML selectDTS."
	self runAllTestsWithMethodLookupKey: 2
		initializationArray: nil
		cacheInitializationArray: self globalLookupCacheInitializationBlock.

ThesisBenchmarks>>runAllTestsUsingDispatchTableSearchML

"	FullDictionaryML selectDTS."
	self runAllTestsWithMethodLookupKey: 1.

ThesisBenchmarks>>runTest: testNameSymbol times: aCounter withMethodLookupKey: aMethodLookupKey
initializationArray: anArrayOrNil
```

```
	ConfigMLInterpreterProxy runPrimitiveUseMethodLookup: aMethodLookupKey initializationArray: anArrayOrNil.
	self startTest: testNameSymbol asString.
	1 to: aCounter do: [ :index | self perform: testNameSymbol.].
	self stopTest.

ThesisBenchmarks>>statsPrintingType: aValue

	self class statsPrintingType: aValue.

ThesisBenchmarks>>testCompiler
	| sourceCode compiler methodNode |
	sourceCode _ ThesisBenchmarks class sourceCodeAt: #LongMethod.
	compiler _ ThesisBenchmarks class compilerClass new.
	methodNode _ compiler compile: sourceCode
		in: ThesisBenchmarks class
		notifying: nil
		ifFail: [self error: 'Error Compiling #LongMethod'].
	^methodNode sourceMap.


ThesisBenchmarks>>runAllTestsUsingHBCachingWithHBWithDicML

	self error: 'Can''t use HierarchyBranchCaching with HierarchyBranchWitchDic'.

ThesisBenchmarks>>hierarchyBranchInitializationArray

	^ Array
			with: (Association key: 'classIntervals:' value: HierarchyBranchML classIntervals)
			with: (Association key: 'selectorsImplementations:' value: HierarchyBranchML
selectorImplementations).

ThesisBenchmarks>>runAllTestsUsingHierarchyBranchML

	self runAllTestsWithMethodLookupKey: 4
		initializationArray: self hierarchyBranchInitializationArray.

ThesisBenchmarks>>runInteractiveTest
	| name |
	name _ FillInTheBlank request: 'Test name?'.
	self startTest: name.
	self error: 'Start the test. Select Procced to ended'.
	self stopTest.

ThesisBenchmarks>>runTest: testNameSymbol withMethodLookupKey: aMethodLookupKey
	self runTest: testNameSymbol times: 1 withMethodLookupKey: aMethodLookupKey.

ThesisBenchmarks>>runTest: testNameSymbol times: aCounter withMethodLookupKey: aMethodLookupKey
initializationArray: anArrayOrNil cacheInitializationArray: anotherArrayOrNil

	ConfigMLInterpreterProxy runPrimitiveUseCacheMethodLookup: aMethodLookupKey initializationArray:
anArrayOrNil cacheInitializationArray: anotherArrayOrNil.
	self startTest: testNameSymbol asString.
	1 to: aCounter do: [ :index | self perform: testNameSymbol.].
	self stopTest.

ThesisBenchmarks>>runAllTestsUsingHBOneEntryCachingStrategy

	self runAllTestsWithMethodLookupKey: 15
		initializationArray: self hierarchyBranchInitializationArray
		cacheInitializationArray: nil.

ThesisBenchmarks>>runAllTestsWithMethodLookupKey: aMethodLookupKey initializationArray: anArrayOrNil
cacheInitializationArray: anotherArrayOrNil

	^self runAllTests: 1
		withMethodLookupKey: aMethodLookupKey
		initializationArray: anArrayOrNil
		cacheInitializationArray: anotherArrayOrNil.
```

```
ThesisBenchmarks>>runAllTestsUsingAllML

	| categorySelectors |
	categorySelectors _ ThesisBenchmarks organization listAtCategoryNamed: 'test support using CacheML' asSymbol.
	categorySelectors do: [ :selector | self perform: selector. ].

	categorySelectors _ ThesisBenchmarks organization listAtCategoryNamed: 'test support using ML' asSymbol.
	categorySelectors do: [ :selector | self perform: selector. ].

ThesisBenchmarks>>startTest: testName
"	Smalltalk garbageCollect."
	ConfigMLInterpreterProxy runPrimitiveStartTest: testName

ThesisBenchmarks>>stopTest
	ConfigMLInterpreterProxy runPrimitiveStopTest: self statsPrintingType.

ThesisBenchmarks>>runAllTestsUsingFullDictionariesML

	FullDictionaryML selectFullDictionary.
	self runAllTestsWithMethodLookupKey: 10.
	FullDictionaryML selectDTS.

ThesisBenchmarks>>runAllTestsUsingGLCWithHB

	self runAllTestsWithMethodLookupKey: 7
		initializationArray: self hierarchyBranchInitializationArray
		cacheInitializationArray: self globalLookupCacheInitializationBlock.

ThesisBenchmarks>>testTranscriptShow
	Transcript clear.
	Transcript show: 'Hello world!'; cr.

ThesisBenchmarks>>runAllTestsUsingGLCWithDTSWithClasses

	self runAllTestsWithMethodLookupKey: 6
		initializationArray: self dtsWithClassesInitializationArray
		cacheInitializationArray: self globalLookupCacheInitializationBlock.
ThesisBenchmarks>>runAllTestsWithMethodLookupKey: aMethodLookupKey initializationArray: anArrayOrNil
	^self runAllTests: 1 withMethodLookupKey: aMethodLookupKey initializationArray: anArrayOrNil

ThesisBenchmarks>>runAllTests: aTimesNumber
	self runAllTests: aTimesNumber withMethodLookupKey: 0.

ThesisBenchmarks>>runAllTests: aTimesNumber withMethodLookupKey: aMethodLookupKey initializationArray: anArrayOrNil
cacheInitializationArray: anotherArrayOrNil

	| categorySelectors |
	categorySelectors _ ThesisBenchmarks organization listAtCategoryNamed: #tests.
	categorySelectors do: [ :selector |
		self runTest: selector times: aTimesNumber withMethodLookupKey: aMethodLookupKey
			initializationArray: anArrayOrNil cacheInitializationArray: anotherArrayOrNil. ].

ThesisBenchmarks>>dtsWithClassesInitializationArray

	^ Array
		with: (Association key: 'selectorsDictionary:' value: DTSWithClassesMLProxy selectorsDictionary).

ThesisBenchmarks>>initialize

	self statsPrintingType: 1.

ThesisBenchmarks>>runAllTestsUsingHBGlobalCircularAddCachingStrategy

	self runAllTestsWithMethodLookupKey: 13
		initializationArray: self hierarchyBranchInitializationArray
		cacheInitializationArray: nil.

ThesisBenchmarks>>runAllTests: aTimesNumber withMethodLookupKey: aMethodLookupKey initializationArray: anArrayOrNil
	| categorySelectors |
```

```
	categorySelectors _ ThesisBenchmarks organization listAtCategoryNamed: #tests.
	categorySelectors do: [ :selector |
		self runTest: selector times: aTimesNumber withMethodLookupKey: aMethodLookupKey
			initializationArray: anArrayOrNil. ].

ThesisBenchmarks class>>statsPrintingType: aType

	StatsPrintingType _ aType.

ThesisBenchmarks class>>FileInOutName
	^ '\Development\thesis\AbstractSound.st'

ThesisBenchmarks class>>LongMethod
	"This method is for test porpouses only.
	 Used by testCompiler and testDecompiler."

	| var1 var2 var3 var4 var5 var6 |
	var1 _ 'Hello World!'.
	Transcript show: var1; cr.
	var2 _ 1.
	var3 _ 100.
	var2 to: var3 do: [ :index | var1 _ var1 + index asString ].
	var4 _ var1 size.
	var4 _ var4 factorial.

	var5 _ var2=10.
	var5 ifTrue: [ var2 _ 'True'. ]
		ifFalse: [ var2 _ 'False' ].

	[var2<20] whileTrue: [
		var2 _ var2 + 1.
		var6 _ 'Bye now'.
		Transcript show: var6. ].

	var6 _ 10 factorial.
	var1 _ 0.
	[var6>11 factorial] whileFalse: [
		var6 _ var6 - 1.
		var1 _ var1 + 1.
		var1%10=0 ifTrue: [ Transcript show: var6 asString].].

	^false.

ThesisBenchmarks class>>statsPrintingType

	^StatsPrintingType
```

11.11 Código Para Calculos

11.11.1 Cantidad de mensajes que cada clase puede responder

```
"Cuenta la cantidad total de mensajes a los que cada clase puede responder"
|counter|
counter := 0.
Smalltalk allBehaviorsDo: [ :aBehavior |
	counter := counter + aBehavior allSelectors size.].
^counter.

812229
```

11.11.2 Cantidad de mensajes implementados en una imagen

```
| allMessages |
allMessages := Set new.
Object allSubclasses do: [ :aBehavior |
	allMessages addAll: (aBehavior selectors) ].
```

```
allMessages addAll: (Object selectors).
^allMessages size.
15128
```

11.12 Código agregado a la VM

11.12.1 Definiciones

```
#include "windows.h"
#define TICKS_STACK_SIZE 10
int ticksStackPosition;
LONGLONG ticksStack[TICKS_STACK_SIZE];

int primitiveCurrentTicks ( void );
int primitiveTicksPerSecond ( void );
int primitiveSaveCurrentTicks ( void );
int primitiveElapsedTicks ( void );
int primitiveInitializeTicksStack ( void );
int positive64BitIntegerFor ( LONGLONG aLargeInteger );

primitiveResponde (Modificacion)
int primitiveResponse(void)  {
.
.
.

		case 599:
			primitiveFail ();
			break;
		case 600:
			successFlag = primitiveCurrentTicks ();
			break;
		case 601:
			successFlag = primitiveTicksPerSecond ();
			break;
		case 602:
			successFlag = primitiveSaveCurrentTicks ();
			break;
		case 603:
			successFlag = primitiveElapsedTicks ();
			break;
		case 604:
			successFlag = primitiveInitializeTicksStack ();
			break;
		case 605:
.
.
}
```

11.12.2 primitiveCurrentTicks

```
int primitiveCurrentTicks (void)
{
    int object;
    int sp;

		BOOL result;
		LARGE_INTEGER performanceCount;

		result = QueryPerformanceCounter ( &performanceCount );
		/* begin pop: */
		stackPointer -= 1 * 4;
		/* begin push: */
		object = positive64BitIntegerFor (performanceCount.QuadPart);
		longAtput(sp = stackPointer + 4, object);
		stackPointer = sp;
```

```
	return result;
}
```

11.12.3 primitiveTicksPerSecond

```
int primitiveTicksPerSecond (void)
{
   int object;
   int sp;

	BOOL result;
	LARGE_INTEGER frequency;

	result = QueryPerformanceFrequency( &frequency );

	/* begin pop: */
	stackPointer -= 1 * 4;
	/* begin push: */
	object = positive32BitIntegerFor (frequency.QuadPart); //positive64BitIntegerFor (frequency.QuadPart);
	longAtput(sp = stackPointer + 4, object);
	stackPointer = sp;

	return result;
}
```

11.12.4 primitiveSaveCurrentTicks

```
int primitiveSaveCurrentTicks ( void )
{
	BOOL result;
	LARGE_INTEGER currentTicks;

	if ( ticksStackPosition>=TICKS_STACK_SIZE )
		result = false;
	else
	{
		result = QueryPerformanceCounter ( &currentTicks );
		ticksStack[ticksStackPosition] = currentTicks.QuadPart;
		ticksStackPosition++;
	}

	return result;
}
```

11.12.5 primitiveElapsedTicks

```
int primitiveElapsedTicks ( void )
{
   int object;
   int sp;
	BOOL result;
	LONGLONG elapsedTicks;
	LARGE_INTEGER currentTicks;

	ticksStackPosition--;
	if ( ticksStackPosition<0 )
	{
		ticksStackPosition = 0;
		result = false;
	}
	else
	{
		result = QueryPerformanceCounter ( &currentTicks );
		elapsedTicks = currentTicks.QuadPart-ticksStack[ticksStackPosition];
		/* begin pop: */
		stackPointer -= 1 * 4;
		/* begin push: */
		object = positive32BitIntegerFor (elapsedTicks);
		longAtput(sp = stackPointer + 4, object);
```

```
                stackPointer = sp;
        }

        return result;
}
```

11.12.6 primitiveInitializeTicksStack

```
int primitiveInitializeTicksStack ( void )
{
        ticksStackPosition=0;
        return true;
}

int positive64BitIntegerFor ( LONGLONG aLargeInteger )
{
        int newLargeInteger;
        int i;

        newLargeInteger = instantiateClassindexableSize (classLargePositiveInteger(),8,0 );
        for ( i=0;aLargeInteger!=0 && i<8;i++ )
        {
                byteAtput(((((char *) newLargeInteger)) + 4) + i, aLargeInteger & 255);
                aLargeInteger = aLargeInteger >> 8;
        }

        return newLargeInteger;
}
```