Universidad de Buenos Aires

Facultad de Ciencias Exactas y Naturales

Departamento de Computación

Tesis de Licenciatura

Integración de los patrones de Diseño en los ambientes de desarrollo Orientados a Objetos

Autor

Ignacio Sagulo
LU 186/90
isagulo@gmail.com

Directores:

Lic. Hernán Wilkinson

Junio 2012

Resumen

Los patrones de diseño marcaron un hito en la Programación Orientada a Objetos. Un *patrón* define una solución genérica para un tipo de problema de diseño que se presenta de manera recurrente. Los patrones de diseño se popularizaron con la aparición del libro conocido como "Gang of Four " (GoF), en el cual se realiza una descripción estructurada y sistemática de los patrones. Desde entonces, la aplicación de los patrones produjo importantes beneficios como elevar el nivel de abstracción al diseñar, reutilizar diseños que han probado ser exitosos en el pasado, facilitar la comunicación del equipo y obtener diseños más flexibles.

A pesar de que los patrones de diseño han tenido un gran impacto y difusión en la industria, su aplicación continúa siendo un proceso manual. En el trabajo se describirán diversos enfoques existentes en la literatura que han buscado automatizar el uso de patrones. Se observará como estas herramientas normalmente sirven para generar código pero no tienen una buena integración con los ambientes de desarrollo. Por ejemplo, una limitación típica es que no reaccionan a modificaciones sobre el código que violan las restricciones que impone el uso de un determinado patrón.

En la presente tesis se investiga la integración de los patrones de diseño en los ambientes de desarrollo. Se buscará que los patrones de diseño se puedan usar como bloques de construcción de más alto nivel dentro del entorno. Por ejemplo, el programador podría "instanciar” patrones de diseño, y validar que se están usando en forma correcta. Integrar adecuadamente los patrones dentro del entorno, permitiría potenciar sus beneficios a través de su aplicación consistente a lo largo de todo el ciclo de desarrollo.

Para lograr la integración de los patrones de diseño con el entorno de desarrollo, la estrategia que se ha seguido en este trabajo es la creación de un Framework[1]. Dicho Framework se ha implementado en Visual Works, y se ha puesto a prueba inicialmente para un patrón que logró integrarse exitosamente dentro del ambiente. En las conclusiones del trabajo, se analiza la factibilidad de extender y generalizar el enfoque seguido en la presente tesis para otros patrones de diseño.

[1] En la presente tesis, no usaremos la palabra framework con la connotación de framework de objetos sino como modelo que define un marco.

Abstract

The design patterns marked a milestone in the Object Oriented Programming. A *pattern* defines a generic solution for a kind of design problem that appears in a recurring way. The design patterns became popular with the appearance of the book known as "Gang of Four" (GoF). This book makes a systematic and structured description of the patterns. Since then, the patterns application has generated important benefits like raise the abstraction level when designing, reusing designs that have proved to be successful in the past, facilitate the communication in the team and achieve more flexible designs.

In spite that the design patterns have had a great impact and spreading in the industry, its application is still a manual process. In this work will describe several approaches existing in the literature that have searched to automate the use of patterns. It will observe how these tools can generate some code but they don´t have a good integration with the development environments. For example, a typical shortcoming is that they don’t react to code modifications that infringe the constraints that impose the use of certain pattern.

The present thesis researches the integration of the design patterns in the development environments. It will search that design patterns can be used like construction blocks of higher level inside the development environment. For example, the programmer might be able to instantiate design patterns and validate that they are being used in a proper way. Integrating the design patterns into the environment in a suitable way, would allow increase their benefits throughout a sound use along all the development cycle

To achieve integrating design patterns in the development environment, the strategy that has been followed in this work is creating a Framework. This Framework has been implemented in Visual Works, and has been initially tested with one pattern that has been successfully integrated inside the environment. The feasibility of extend the approach followed in the present thesis to other design patterns is going to be analyzed in the conclusions of this work.

Dedicado a:

A mi mujer, Romina, que siempre me apoyó y ayudó en todo lo que necesité para poder realizar este trabajo.

A mi hijo Juanchi, un gran curioso, con el que siempre tenemos charlas muy interesantes sobre el mundo y sus objetos

A mi madre y a mi padre que siempre estimularon en mí las ganas conocer y estudiar.

Índice

1 Introducción....8
1.1 Objetivos....8
1.2 Relevancia de la propuesta 9
1.3 Problemas observados en las herramientas de patrones....9
1.4 Organización de la tesis....10
2 Background....11
2.1 Conceptos esenciales del Paradigma Orientado a Objetos....11
2.1.1 Objetos y mensajes....11
2.1.2 Clases e Instancias....12
2.1.3 Herencia y Polimorfismo....12
2.1.4 Tipo....13
2.2 Patrones de diseño 14
2.2.1 ¿Qué es un patrón de diseño? 15
2.2.2 La aparición del concepto de patrón....15
2.2.3 ¿Cómo se describe un patrón de Diseño? 15
2.2.4 Ventajas de la utilización de los patrones de Diseño....17
3 Revisión del Estado del Arte 19
3.1 Caso de Estudio....19
3.1.1 Patrón Decorator....19
3.1.2 Ejemplo de Patrón Decorator....21
3.2 Los patrones como construcciones del lenguaje de programación....23
3.2.1 Discusión....27
3.3 Reificación de los patrones como Componentes de Diseño....28
3.3.1 Discusión....29
3.4 LePUS: un lenguaje para especificar patrones de Diseño....30
3.4.1 Modelo del Diseño....30
3.4.2 Elementos de LePUS....31
3.4.3 Especificación de patrones en LePUS....33
3.4.3.1 Patrón Decorator en LePUS....33
3.4.4 Discusión....33
3.4.4.1 Falta de una representación adecuada de los mensajes....33
3.4.4.2 Ausencia del concepto de receptor del mensaje. 35
3.4.4.3 Restricciones no realistas a las implementaciones de un patrón....35
3.4.4.4 Utilidad del formalismo....36
3.5 Los patrones como un Framework....38
3.5.1 Discusión....38
4 Dificultades de modelar los patrones de Diseño....39
4.1 Los patrones: a la vez definición de problema y solución....39
4.2 ¿Los patrones son formalizables? 40
4.3 ¿Los patrones se describen con el mismo nivel de precisión?....41
4.4 Las múltiples variantes de implementación de un patrón 43
4.5 "Desviaciones" al aplicar patrones de diseño....43
4.6 Conclusiones....44
5 El Framework de Patrones....45
5.1 Enfoque y Metodología del trabajo....46
5.1.1 Discusión de la estrategia de implementación....46
5.1.1.1 Necesidad de fuerte interacción con el entorno....46
5.1.1.2 Elección de Smalltalk como lenguaje de implementación....47
5.1.1.3 Accesibilidad para el programador....48

5.1.1.4 Feed-back de la aplicación práctica y construcción iterativa....................48
5.1.1.5 Extensibilidad 49
5.1.2 Observaciones sobre el alcance del modelo de patrones................................49
5.2 Visión general...50
5.3 Modelo de roles...52
5.3.1 Actores y Roles...53
5.3.2 Estrategias de Definición de Roles...55
5.3.3 Roles en el patrón Decorator..56
5.3.3.1 Component..56
5.3.3.2 Mensajes a decorar del Componente...56
5.3.3.3 Decorator...56
5.3.3.4 Variable de instancia para referenciar al Component..............................57
5.3.3.5 Decorators...57
5.4 Predicados y Expresiones 58
5.4.1 Predicado del Patrón Decorator..58
5.4.1.1 Fórmula lógica del patrón de Estudio..58
5.4.1.2 Representación de fórmulas en el Framework..60
5.4.2 Objetos Evaluables y Contextos de Evaluación..62
5.4.3 Los patrones como Contextos de Evaluación..63
5.4.3.1 Ejemplos de Resolución de Búsqueda de Actores.....................................64
5.4.4 Evaluación de Expresiones...65
5.4.5 Evaluación de Predicados...67
5.5 Validación de patrones..69
5.6 Construcción de Patrones 71
5.6.1 Construcción de Definiciones de Patrones...72
5.6.2 Construcción de Instancias de Patrones...74
5.7 Interacción entre el ambiente de Desarrollo y el framework de Patrones............75
5.7.1 Descripción funcional para el Caso de Estudio 75
5.7.2 Diseño de la interacción ambiente-framework de patrones............................78
5.7.2.1 Representación de los cambios de código en Visual Works.....................78
5.7.2.2 Colaboraciones entre el ambiente y el framework de patrones.................80
5.8 Referencia de las Clases del Framework...83
5.8.1 Predicates..83
5.8.1.1 SendMessageRelation..83
5.8.1.2 ForAllPredicate...86
5.8.1.3 MessageImplementedRelation. 86
5.8.1.4 Predicados binarios..87
5.8.2 Method...88
5.8.3 ClassSet...89
5.8.4 Protocol..90
5.8.5 MessageDeclaration...90
6 Conclusiones..91
6.1 Resultados obtenidos con el Framework. 91
6.2 Factibilidad de extender el Framework propuesto...92
7 Trabajos Futuros...93
8 Apéndice...96
8.1 Patrón Visitor...97
8.2 Setup del entorno..98
9 Referencias...101

Tabla de Figuras

Figura 1 - Estructura Patrón Decorator....20
Figura 2 - Ejemplo Patrón Decorator en Visual Works....21
Figura 3 - Parsing de Patrones....26
Figura 4 - Definición e Instancia de patrón....51
Figura 5 - Modelo de Roles....52
Figura 6 - Estrategia de Definición de Roles....55
Figura 7 - Ejemplos de Búsquedas de Actores....64
Figura 8 - MessageImplementationsFunction....66
Figura 9 - Jerarquía de EvaluationResult....67
Figura 10 - Validación de Patrones....69
Figura 11 - Pattern Builders....71
Figura 12 - Interacción con el Entorno. Agregar método a Component(1/2)....77
Figura 13 - Interacción con el Entorno. Agregar método a Component(2/2)....78
Figura 14 - Representación de los cambios de código en Visual Works....79
Figura 15 - Colaboraciones entre el ambiente y el framework de patrones....80
Figura 16 - Evaluación de SendMessageRelation....85
Figura 17 - Predicados binarios....87
Figura 18 - Resultado de evaluar predicados binarios....88
Figura 19 - Jerarquía de ClassSets....89
Figura 20 - Jerarquía de Protocols....90
Figura 21 - Estructura patrón Visitor....97

1 Introducción

En la presente tesis se investigará la integración de los patrones de diseño en los ambientes de desarrollo orientados a objetos. Se buscará que los patrones de diseño se puedan usar como bloques de construcción de más alto nivel dentro del entorno de desarrollo. Por ejemplo, el programador podría "instanciar patrones de diseño", validar que los patrones existentes se usen correctamente y consultar los patrones utilizados en el sistema.

Para dar soporte a estas funcionalidades, será necesario construir un *modelo* de los patrones de diseño. La construcción de este modelo presenta numerosas dificultades de carácter teórico-práctico: ¿los patrones de diseño se pueden formalizar para soportar su automatización? las múltiples variantes y consideraciones de implementación que tiene un patrón de diseño ¿se pueden representar de forma tal de obtener un soporte de patrones de diseño útil para el desarrollador? En la sección 4, luego de describir en la sección *"Revisión del Estado del Arte"* los distintos enfoques existentes para modelar patrones de diseño, se realizará un análisis detallado de las dificultades que conlleva la construcción de modelos para los patrones. Hasta el momento, pareciera que los patrones de diseño están a un nivel de abstracción que los ha hecho resistentes al soporte de herramientas y a una integración eficaz en los ambientes de desarrollo.

1.1 Objetivos

El problema que el presente trabajo de investigación enfrentará entonces, es integrar eficazmente los patrones de diseño en los ambientes de desarrollo de forma tal de lograr los siguientes objetivos específicos:

- Permitir al programador usar algunos de los patrones existentes en [Gamma95] como bloques de construcción de más alto nivel:
  - Documentar el diseño del sistema por medio de los patrones utilizados
  - Facilitar la búsqueda y el mantenimiento de los patrones empleados en el sistema
- Soportar el concepto de "instanciación de patrones". Como se mencionó anteriormente, cada patrón es una solución genérica que define una estructura de roles y responsabilidades. Cuando se usa un patrón en un diseño en particular, los diferentes roles son cumplidos por diferentes objetos. El modelo deberá conocer y trabajar con esta información.
- Validar el uso correcto de los patrones de diseño. Cada patrón impone un conjunto de restricciones sobre el código. El modelo deberá detectar problemas en el uso de los patrones y reportarlos al programador
- Generación de código para simplificar tareas repetitivas de instanciación de patrones
- Manipulación de las instancias de patrones a través de operaciones específicas dependiendo del tipo de patrón. Como ejemplos de estas operaciones se pueden nombrar:

- Agregar un estado en una instancia del patrón "State". En este caso, se realizarán las recomendaciones necesarias para que el programador modifique el código de tal forma que el nuevo estado sea usado correctamente.
- Agregar una ConcreteFactory a una instancia del patrón AbstractFactory.

- Generación de consejos al programador sobre posibles modificaciones a realizar en el código.

1.2 Relevancia de la propuesta

Los patrones de diseño han tenido, y tienen, una gran importancia en la construcción de software OO. Sin embargo, la aplicación de los mismos continúa siendo un *proceso manual*. Si se logran cumplir los objetivos de integración de los patrones en los ambientes de desarrollo, se obtendrían los siguientes beneficios:

- Se enriquece el entorno de desarrollo gracias a la incorporación de los patrones de diseño como objetos de primera clase. Existiría una comunicación bidireccional entre los patrones de diseño y los elementos de la implementación (jerarquías, clases y métodos):
  - En la dirección del diseño al código, cuando se instancian, manipulan y consultan los patrones de diseño, el código relacionado se modifica y localiza consecuentemente.
  - En la dirección del código al diseño, cuando se realizan modificaciones en el código que no respetan las restricciones que impone un patrón, estas modificaciones se detectan y reportan al programador.
- Las tareas de desarrollo se simplifican ya que programador realizaría su trabajo en un nivel más alto de abstracción interactuando con los patrones de diseño que estarían "vivos" dentro del ambiente.
- una parte importante de la documentación del diseño como el uso de patrones estaría integrada y permanentemente sincronizada con el código. Cuando el diseño se representa con herramientas no integradas con el código, se crean artefactos que es necesario actualizar independientemente. Este hecho junto con las presiones en los tiempos de desarrollo, provoca que generalmente, la documentación de diseño quede desactualizada.

1.3 Problemas observados en las herramientas de patrones

Si bien existen algunas herramientas que trabajan con patrones, éstas normalmente sólo asisten en la generación de código pero tienen importantes limitaciones:

- no permiten visualizar los patrones utilizados. Se presenta entonces el problema de la *trazabilidad* que se describe en [Ager98]. Cuando se utilizan muchos patrones de diseño en una aplicación, generalmente existirán clases que juegan distintos roles en los diferentes patrones. La trazabilidad de estos roles se pierde cuando se pasa del diseño al código.

- no permiten modificar los patrones previamente definidos con la herramienta
- no reaccionan a modificaciones sobre el código que violan las restricciones que impone el uso de un patrón determinado.
- si se producen errores en la validación de un patrón, la herramienta no puede sugerir acciones correctivas y llevarlas a cabo.

Todas estas limitaciones se deben generalmente a la falta de un *modelo* que represente los patrones de diseño, y que a su vez tenga una fuerte interacción con el ambiente de desarrollo. Como observamos anteriormente, en la presente tesis se investigará cómo realizar esta integración, permitiendo aplicar los patrones de diseño de una manera más consistente y eficaz durante todo el proceso de desarrollo.

1.4 Organización de la tesis

A continuación se describe la organización general del presente trabajo, resumiendo el contenido de las secciones que lo componen:

- En la sección *"Background"* (2) se repasan algunos conceptos de POO y de patrones de diseño, que, posteriormente, son utilizados en el transcurso de la tesis.
- En la sección *"Revisión del Estado del Arte"* (3) se expondrán varios trabajos sobre patrones de diseño, que se plantean objetivos similares a los de esta tesis. Los trabajos elegidos ejemplifican los distintos enfoques existentes en la literatura. Las presentaciones de los distintos trabajos y del Framework de patrones, se realizarán, en general, sobre el patrón que se muestra en la sección *"Caso de Estudio"* (3.1). Esto facilitará al lector la comparación del enfoque seguido en la presente tesis con los trabajos expuestos en esta sección.
- Para que una herramienta pueda dar soporte a los patrones de diseño, es necesario definir algún tipo de modelo de los mismos. En la sección 4, se discutirán las dificultades que se presentan cuando se quiere armar un modelo sobre los patrones de diseño. Es importante entender estas dificultades, porque son la causa de la "resistencia" de los patrones al soporte de herramientas
- En la sección 5 se presenta el *"Framework de Patrones"*[2] que se ha desarrollado durante esta tesis, para buscar integrar los patrones de diseño en el ambiente de desarrollo. Inicialmente, se discuten las razones por las cuales se decidió construir un framework como estrategia de implementación, y luego, se describe el diseño de dicho Framework siguiendo el caso de estudio.
- En la sección 6 se presentarán las conclusiones del trabajo, haciendo un balance de los logros obtenidos con respecto a los objetivos planteados. Finalmente en la sección 7, se discuten posibles tareas para seguir avanzando con el enfoque planteado en esta tesis

[2] Como se ha observado anteriormente en el Resumen, en la presente tesis, no usaremos la palabra framework con la connotación de framework de objetos sino como modelo que define un marco

2 Background

2.1 Conceptos esenciales del Paradigma Orientado a Objetos

Los conceptos aquí volcados forman parte del capítulo uno del "Blue Book" [Gold86], 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*"[3].

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.

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.

---

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

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 definida 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 está 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 sí 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 sus propias subclases.

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 está 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 múltiple-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 compuesta por el nombre, el tipo y nombre de los colaboradores externos que recibe (parámetros) y el tipo de resultado que otorga[4].

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[5]) sin

[4] Java agrega a la definición de cada mensaje las excepciones que puede producir.
[5] 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.

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 qué 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 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 circunstancia 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. Lenguajes débilmente 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).

2.2 Patrones de diseño

Los patrones de diseño marcaron un hito en la Programación Orientada a Objetos. Un *patrón* propone una solución genérica para un tipo de problema de diseño que se presenta frecuentemente en la práctica. El concepto de patrón se popularizó con la publicación de [Gamma95], conocido como el libro de "Gang of Four " (GoF). Este texto es un *catálogo de patrones*, en el cual cada patrón se describe siguiendo un template fijo formado por un conjunto de secciones. Esta sistematicidad en la descripción, logró que los patrones fueran más fáciles de aprender, comparar y aplicar. El mérito del libro de GoF es haber logrado capturar la experiencia existente en el campo del Diseño OO de forma tal que pueda ser reusada por todos.

2.2.1 ¿Qué es un patrón de diseño?

Para contestar a esta pregunta podemos decir que:

- Es una solución genérica para un tipo de problema de diseño recurrente. A esta solución, se llega generalizando y abstrayendo soluciones utilizadas en el pasado en forma exitosa.
- Un patrón no define una implementación o diseño en concreto. Es como un template que describe en forma abstracta un problema y su solución. La especificación de la solución abarca la definición de:
    - el conjunto de participantes (clases y/o instancias)
    - los roles y la distribución de responsabilidades que cada participante tiene dentro de la solución
    - las colaboraciones entre los participantes.

- Además un patrón describe los compromisos (*trade-off*) de su uso en términos de flexibilidad, extensibilidad y uso de recursos.
- Por último, pero no menos importante cada patrón tiene un nombre de una o dos palabras que intenta capturar de forma intuitiva el problema que apunta a resolver así como la solución propuesta.

2.2.2 La aparición del concepto de patrón

Es interesante observar que la idea de patrón surgió en el campo de la Arquitectura y fue concebida inicialmente por Alexander en [Alex79] quien dio la siguiente definición:

Un patrón es una regla de tres partes, que expresa una relación entre cierto contexto, un problema y una solución. Como un elemento en el mundo, un patrón es una relación entre un contexto, un sistema de fuerzas que ocurre repetidamente en ese contexto y una cierta configuración espacial que permite a estas fuerzas resolverse a sí mismas. Como un elemento de un lenguaje, un patrón es una instrucción que muestra cómo tiene que usarse esta configuración espacial, una y otra vez, para resolver un determinado sistema de fuerzas, donde quiera que el contexto lo haga relevante.

En resumen, un patrón es al mismo tiempo, una cosa que sucede en el mundo y una regla que nos dice cómo y cuándo crear dicha cosa. Un patrón es un proceso y una cosa, es una descripción de un elemento que está vivo y una descripción del proceso por el cual creamos dicho elemento (p. 247)

2.2.3 ¿Cómo se describe un patrón de Diseño?

Para describir *cada* patrón de diseño en GoF se sigue un template fijo que se detalla a continuación:

Nombre de patrón y clasificación

El nombre del patrón expresa la esencia del patrón en forma resumida. Un buen nombre es vital porque el nombre pasa a formar parte del *vocabulario* de diseño. Los patrones se clasifican en las siguientes categorías:

- *Creacionales*: tienen como problema el proceso de creación de objetos
- *Estructurales*: tratan con la composición de objetos y clases
- *De comportamiento*: caracterizan las maneras en las que las clases y los objetos interactúan y distribuyen las responsabilidades.

Intención

La intención es un breve enunciado que responde las siguientes preguntas: ¿qué es lo que hace el patrón de diseño? ¿Cuál es el problema o aspecto del diseño que el patrón apunta a resolver?

También conocido como

Si existen otras formas conocidas de nombrar al mismo patrón, se enumeran en esta sección.

Motivación

Es un escenario que muestra un problema de diseño y cómo la estructura de clases y objetos asociada al patrón soluciona el problema. Este escenario de aplicación concreta ayuda a entender la descripción más abstracta que se define en las subsiguientes secciones.

Aplicabilidad

¿Cuáles son las situaciones en las que se puede aplicar el patrón de diseño? ¿Cuáles son ejemplos de pobres diseños que este patrón puede resolver? ¿Cómo reconocemos estas situaciones?

Estructura

Contiene una representación gráfica de las clases que participan del patrón. También se usan diagramas de interacción para ilustrar las secuencias de requerimientos y colaboraciones entre objetos.

Participantes

Las clases y objetos que participan en el patrón de diseño y la descripción de sus responsabilidades

Colaboraciones

Cómo los participantes colaboran para llevar a cabo sus responsabilidades

Consecuencias

¿Cómo el patrón soporta sus objetivos? ¿Cuáles son los compromisos y resultados de usar el patrón? ¿Qué aspecto de la estructura del sistema el patrón permite variar independientemente?

Implementación

¿Qué riesgos, consejos o técnicas debemos conocer cuando implementamos el patrón? ¿Existen aspectos específicos del lenguaje a tener en cuenta?

Código ejemplo

Fragmentos de código que ilustran como se podría implementar el patrón.

Usos conocidos

Ejemplos de patrones encontrados en sistemas reales

Patrones relacionados

¿Qué patrones de diseño están cercanamente relacionados a este patrón? ¿Cuáles son las diferencias importantes? ¿Junto con qué otros patrones se puede usar este patrón?

2.2.4 Ventajas de la utilización de los patrones de Diseño

La utilización de patrones en el diseño de aplicaciones OO produce los siguientes beneficios:

- *Permite diseñar en un nivel de abstracción más alto*. Esto se consigue porque cada patrón define una microarquitectura de objetos que permite resolver un determinado tipo de problema
- *Flexibilidad en los Diseños*. La utilización de patrones prepara el diseño para ser flexibles en determinados aspectos y ante cierto tipo de cambios. Cuando utilizamos correctamente los patrones, sin sobre-diseñar, el sistema se puede adaptar mucho más fácilmente a los cambios que se presenten.
- *Reutilización de diseños exitosos*. Los patrones existentes en el libro de GoF son soluciones que ya probaron ser exitosas para determinado tipo de problemas que ocurren de manera recurrente. Entonces, al estar frente a un problema en el cual podemos aplicar un patrón, podemos usar una solución ya probada y conocemos las ventajas que nos traerá, pero también, los compromisos de su aplicación
- *Facilita la comunicación de los miembros del equipo*. Los patrones incrementan el vocabulario de diseño. Proponer utilizar un patrón para resolver un problema específico, determina varias decisiones de diseño que

no es necesario detallar porque se desprenden de la definición o restricciones que el patrón propuesto implica.

- *Simplifican la documentación*. Al aclarar que estamos usando un patrón de diseño determinado, indicamos de un manera mucho concisa el problema que estamos resolviendo y las características de la solución

3 Revisión del Estado del Arte

En esta sección, se expondrán algunos trabajos que modelan a los patrones de diseño. Se analizará un caso significativo de cada uno de los distintos enfoques existentes en la literatura. Por ejemplo, uno de estos enfoques modela a los patrones de diseño como nuevas construcciones dentro del lenguaje de programación, otro define un lenguaje formal para representar el diseño de un programa orientado a objetos dentro del cual se pueden especificar los patrones de diseño, etc. Para cada enfoque analizado, se discutirá en qué medida dicho enfoque contribuye a cumplir con los objetivos descriptos en la sección 1.1. Como se observó en la introducción, en general, para ilustrar cada uno de los enfoques de la literatura, y posteriormente en El Framework de Patronesla sección 5 en la que se describe el Framework de Patrones, se usará como ejemplo el Caso de Estudio que se describe a continuación

3.1 Caso de Estudio

El patrón que se usará como caso de estudio es patrón Decorator que se describirá a continuación. Adicionalmente, como ejemplo de aplicación o instanciación de dicho patrón, en varios casos se usará la jerarquía de componentes Visuales en VisualWorks que también se mostrará en esta sección

3.1.1 Patrón Decorator

Intención

Poder agregar dinámicamente a un objeto responsabilidades adicionales. El uso del patrón Decorator provee una alternativa flexible a la subclasificación para extender la funcionalidad

Aplicabilidad

- agregar responsabilidades a objetos individuales en forma dinámica y transparente, esto es, sin afectar a otros objetos
- para responsabilidades que pueden eliminarse
- cuando la extensión por subclasificación es impráctica. Algunas veces, es posible realizar un gran número de extensiones independientes y esto produciría una explosión de subclases para soportar cada combinación posible.

Estructura

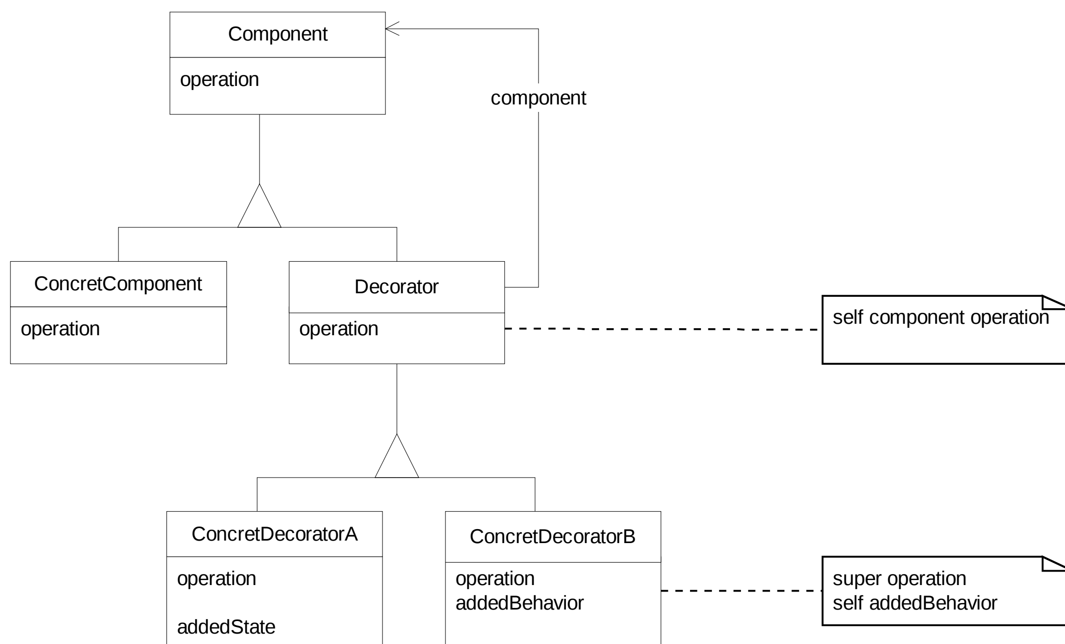


Figura 1 - Estructura Patrón Decorator

Participantes

- **Component**: define la interface de los objetos a los cuales se les puede agregar responsabilidades dinámicamente.
- **ConcretComponent**: define un objeto al cual se le pueden agregar responsabilidades adicionales. Los componentes concretos son decorados, o wrapeados, por subclases de Decorators.
- **Decorator**: mantiene una referencia a un objeto Component e implementa la misma interface que Component.
- **ConcretDecorator**: son las subclases de Decorator que agregan funcionalidad al Component.

Colaboraciones

El *Decorator* forwardea los requerimientos a su objeto C*omponent*. Puede opcionalmente, ejecutar operaciones adicionales antes y después de forwaredear el requerimiento.

Observaciones

Esta variante se caracteriza porque el participante Decorator es una subclase del Component y tiene una variable de instancia a través de la cual referencia al Component. Hay que observar que la relación de herencia entre Decorator y Component no es la única alternativa de implementación del patrón Decorator. Lo que sí es necesario es que el Decorator y el Component sean polimórficos respecto del protocolo de Component, de tal manera que la existencia del Decorator sea transparente para el cliente. Además, no necesariamente el Decorator debe referenciar al Component con una variable de instancia, basta con que pueda obtener al Component de alguna forma.

3.1.2 Ejemplo de Patrón Decorator

Como ejemplo de aplicación del patrón Decorator se va a tomar la jerarquía de componentes Visuales en VisualWorks. Esta aplicación se detalla en [Alper98]. A continuación, se muestran algunas de las clases de dicha jerarquía para comprender la aplicación del patrón:

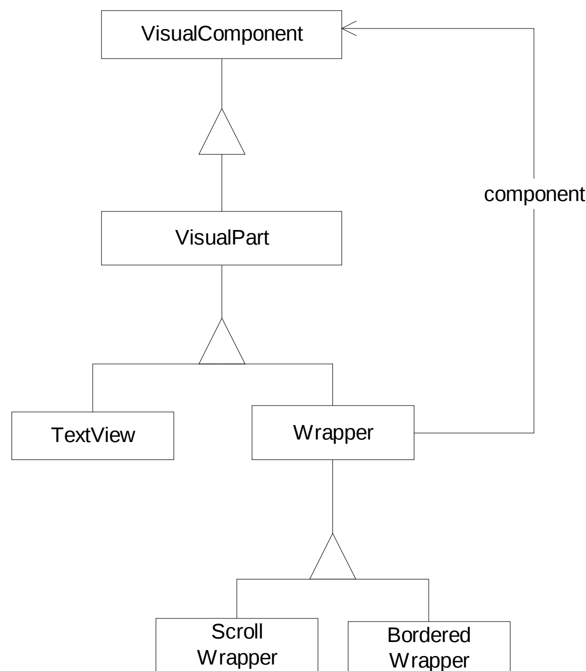
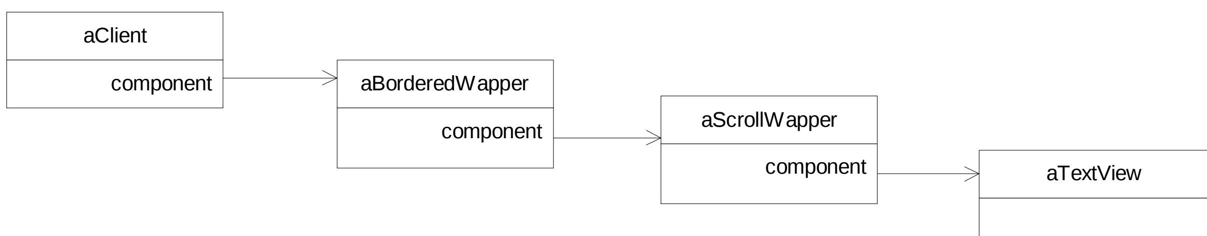


Figura 2 - Ejemplo Patrón Decorator en Visual Works

- **VisualComponent (Component):** es una clase abstracta que representa un objeto que puede crear una representación visual de sí mismo. Las subclases implementan algoritmos particulares para producir presentaciones visuales de distintos tipos de información.
- **Wrapper (Decorator)**: esta clase no hace más que definir una variable de instancia *'component'* e implementar los mensajes de Component forwardéandolos a dicha variable.

- **ScrollWrapper y BorderedWrapper (ConcretDecorator)**: son subclases de Wrapper que agregan la funcionalidad de poder scrolear y graficar bordes en un componente visual
- **TextView (ConcretComponent)**: es un componente visual que se especializa en mostrar texto

Un cliente de un `VisualComponent` podría tener asociado un `Wrapper` ya que éste implementa la misma interface que `VisualComponent`. Los decorators se pueden anidar en forma recursiva. Como se muestra en la figura, `'aClient'` colabora con el wraper `'aBorderedWapper'` que se encarga de proveer el borde a su componente. El `BorderedWrapper` tiene a su vez como componente `'aScrollWrapper'` que se encarga de agregarle la capacidad de scrolear a su componente que es un `TextView`. Para el ejemplo de los componentes visuales se podría tener:



3.2 Los patrones como construcciones del lenguaje de programación

En esta sección se analizarán la línea de trabajos [Hedin97], [Hedin00] y [Hedin00b] en la cual se propone pensar la aplicación de un patrón de diseño como un cierto *tipo de construcción del lenguaje* en el cual se especifican los objetos que juegan roles particulares dentro del patrón y las reglas que estos objetos deben seguir. La técnica con la que se realiza esta especificación es *"Extensión de Atributos"* que se presenta en [Hedin96].

La técnica de extensión de atributos requiere un *compilador abierto* que exponga la semántica estática del lenguaje de programación a través del Abstract Sintax Tree (AST). Además de exponer el AST, el compilador debe permitir modificar los nodos que conforman el árbol agregando nuevos atributos o ecuaciones para calcular otros atributos. De esta forma, la semántica del lenguaje base se puede extender y agregar nuevas verificaciones sobre el código fuente. Estas verificaciones podrían incluir la forma en la que se usa una librería o como se muestra en [Hedin97] la utilización de patrones de diseño.

Para especificar el rol que un elemento del programa (clase, método o variable) juega en un determinado patrón, el programador debe "anotar" el programa usando un tipo particular de comentario. Por ejemplo, en el siguiente código:

```
/*= DecoratorPattern_Decorator =*/
VisualPart subclass: #Wrapper
        instanceVariableNames: 'component '
        classVariableNames: ''
```

se especifica que la clase Wrapper juega el rol Decorator en una instancia del patrón Decorator. No es necesario marcar en el código fuente todos los roles existentes en un patrón de diseño. Para definir una aplicación del patrón Decorator bastaría con marcar los siguientes roles:

Rol	**Descripción**
Component	La clase abstracta para los componentes que son decorados
Decorator	Una subclase especial de Component que sirve como Decorator abstracto. Tiene un *DecoratedComponent* y posiblemente algunas *DecoratingImplementations*
DecoratedComponent	Una variable de instancia declarada en *Decorator* la cual referencia al *Component* decorado

Estos roles se llaman ***roles definidos*** ya que es necesario que el programador los defina explícitamente usando comentarios en el código fuente.

Cada nodo del AST que representa a una *clase* del sistema será una instancia de **`ClassDeclaration`**. Usaremos Smalltalk para representar los objetos que conforman el

AST pero el lenguaje que se parsea puede ser cualquier lenguaje OO. La clase `ClassDeclaration` tendrá un atributo `boolean` por cada uno de los posibles roles que eventualmente puede jugar la clase. En el código que se muestra a continuación los atributos de la clase ClassDeclaration se tratarán como nuevos mensajes definidos en dicha clase. Por ejemplo, para dar soporte al patrón Decorator, `ClassDeclaration` tiene los siguientes mensajes:

```
ClassDeclaration>>isComponent
      isComponent isNil ifTrue: [ ^false] ifFalse: [^ isComponent]

ClassDeclaration>>isDecorator
      isDecorator isNil ifTrue: [ ^false] ifFalse: [^ isDecorator]
```

Por ejemplo, si el mensaje `isComponent` es `true` para una clase C, entonces la clase C juega el rol `Component` en una instancia del patrón `Decorator`. En caso contrario no juega este rol. Por default, si el programador no especifica ningún valor, `isComponent` es false (la clase *no* es Component)

A partir de los roles definidos en el programa, la herramienta puede *inferir* que determinados elementos del sistema juegan otros roles en los patrones de diseño. Estos roles se llaman ***roles derivados***. Los roles derivados para el patrón Decorator se muestran a continuación:

Rol	**Descripción**
Operation	Cualquier método declarado en *Component*
DecoratingImplementation	Una implementación de *Operation* en *Decorator* o *ConcreteDecorator* que delega la llamada en *DecoratedComponent* y opcionalmente ejecuta acciones adicionales antes y/o después de la delegación
ConcretDecorator	Una subclase de Decorator la cual puede contener *DecoratingImplementations*
ConcreteComponent	Una subclase de *Component* que no es *Decorator* ni una subclase de Decorator

Un rol derivado se modela con un atributo *sintetizado* en un nodo del AST [Aho86]. Un atributo sintetizado es un atributo cuyo valor se calcula en función de los valores de los atributos de otros nodos del AST. Los atributos de un AST son calculados por el compilador a medida que éste va reconociendo las estructuras de lenguaje de programación.

Como dijimos anteriormente cada declaración de una clase del lenguaje OO que estamos parseando se representa con una instancia de `ClassDeclaration`. Si B es una subclase de A, entonces existirán 2 instancias de `ClassDeclaration`: `classDeclA` y `classDeclB`. Una instancia de `ClassDeclaration` puede acceder a la declaracion de la superclase de su clase asociada, usando el método `superClassDeclaration`. Por ejemplo, si a `classDeclB` se le envía el mensaje `superClassDeclaration` se obtiene como respuesta `classDeclA`.

Entonces, para definir si una clase juega el rol de `ConcreteDecorator` se calcula el atributo `'isConcreteDecorator'` como se muestra a continuación

```
ClassDeclaration>>isConcreteDecorator
      self superClassDeclaration isNil
            ifTrue: [^false]
```

```
                    ifFalse: [ ^self superClassDeclaration isDecorator |
                           self superClassDeclaration isConcreteDecorator]
```

Esta definición entonces establece que una clase C juega el rol `ConcreteDecorator`, si y sólo si tiene una superclase y esa superclase juega a su vez el rol `Decorator` o `ConcreteDecorator`.

En [Hedin97] se define también que los roles de un patrón deben cumplir ciertas relaciones entre sí. Para verificar cada una de estas relaciones se requiere un atributo sintetizado. Por ejemplo, en la implementación estándard del patrón Decorator una regla sería: *"El Decorator debe ser una subclase del Component".* El mensaje asociado a esta regla es:

```
ClassDeclaration>>decoratorIsSubclassOfComponent
       "Si la regla no se cumple tira una excepcion"
       ( self isDecorator and: [self superClassDeclaration notNil]
                          and: [self superClassDeclaration isComponent not] )
             ifTrue: [ Exception raiseSignal:
                           "Decorator Pattern: decorator has not a Component as
                            superClass"]
```

Otro ejemplo de regla para el patrón Decorator sería que una *DecoratingImplementation* debe forwardear la llamada de la correspondiente *Operation* sobre su *DecoratedComponent*

En [Hedin00] se presenta la implementación de un prototipo de herramienta llamado *DPDOC*, el cual soporta la documentación del uso de patrones de diseño basado en las ideas discutidas en [Hedin97]. A diferencia de [Hedin97] en donde los roles de los patrones de diseño se representan usando atributos en los nodos del AST, en [Hedin00] los patrones de diseño tienen una representación explícita y una gramática independiente. Cuando el programador quiere instanciar un patrón de diseño debe *codificar* un patrón siguiendo la sintaxis para definir patrones. Por ejemplo, para definir una instancia de patrón Decorator se debería escribir algo como:

```
Decorator:
       name: 'decorator1'
       Component: VisualComponent;
       Decorator: Wrapper;
       DecoratedComponent: component;
```

La herramienta parsea el código fuente del programa junto con esta definición y genera el *Arbol sintáctico del programa* ***(ASTprog)*** y el *Arbol sintáctico de las instancias de patrones* ***(ASTpat)***. Ambos están relacionados como se muestra en la siguiente figura:

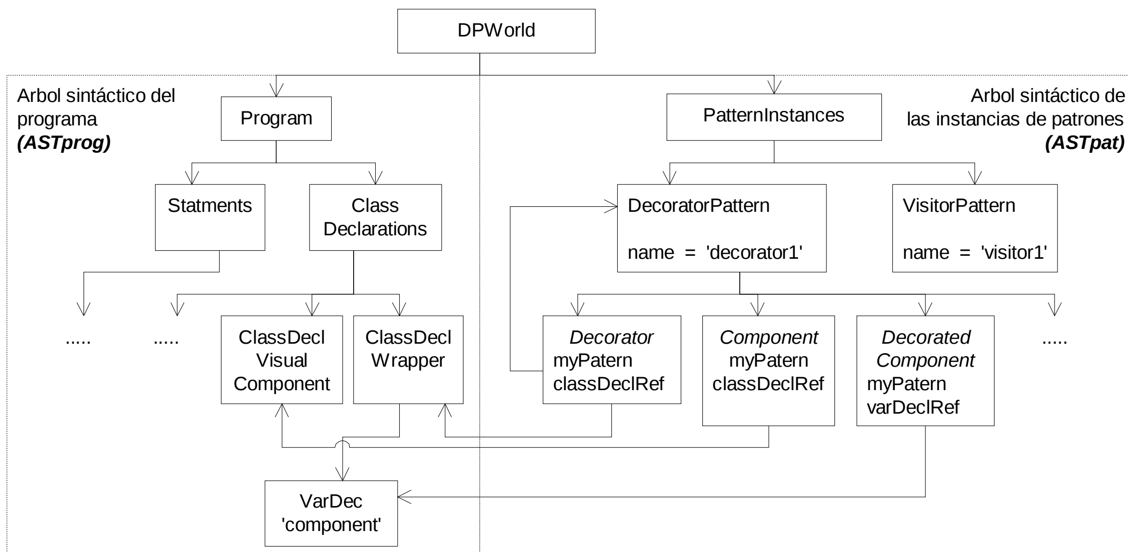


Figura 3 - Parsing de Patrones

Podemos observar lo siguiente:

- Los nodos que forman parte del ASTprog, son nodos que crea el compilador del lenguaje y representan las *declaraciones* realizadas en el código fuente.
- Cada instancia de patrón (por ej 'decorator1') se representa explícitamente como un nodo dentro del ASTpat. Un nodo que representa a una instancia de patrón tiene a su vez referencias a los nodos de los distintos roles ('decorator1' referencia a los roles Decorator, Component y DecoratorComponent)
- Cada *nodo rol **nr*** referencia al nodo del ASTprog tal que su elemento de programa asociado juega el rol de nr dentro de la instancia de patrón a la cual **nr** pertenece. Por ejemplo, el nodo rol Decorator referencia a la *declaracion* de la clase Wrapper, porque la clase Wrapper juega el rol Decorator dentro del patrón 'decorator1'.

DPDOC está construída dentro de una herramienta de desarrollo de lenguajes interactiva llamada APPLAB (APPlication language LABoratory). APPLAB se usa principalmente para testear gramáticas para nuevos lenguajes de programación mientras que éstos se encuentran en desarrollo. Los usuarios de APPLAB pueden editar los programas y las gramáticas al mismo tiempo.

Dentro de la herramienta *DPDOC* agregar un patrón de diseño implica definir la gramática del patrón y las reglas semánticas asociadas a cada producción de dicha gramática. Las reglas semánticas especifican acciones a tomar (por ejemplo, código a ejecutar) cuando el compilador reconoce una determinada estructura del lenguaje (ver [Aho86]). El ambiente de APPLAB proveé mecanismos que permiten a las reglas semánticas acceder a las declaraciones ya parseadas por el compilador. Esto permite que se establezcan los links entre el ASTpat y el ASTprog. Dentro de las reglas semánticas de un patrón de diseño se pueden codificar las validaciones del estilo *"El Decorator debe ser una subclase del Component"*. Cuando el usuario termina de editar el programa y los patrones utilizados, el programa se compila y las validaciones de los patrones que fallan se informan como si fueran errores o warnings de compilación.

3.2.1 Discusión

Resultan interesantes los conceptos de *rol definido* y *rol derivado*. Es bueno que una herramienta pueda, a partir de unos pocos roles definidos explícitamente por el programador, inferir los roles que los distintos componentes del sistema juegan en los patrones de diseño. Esta característica hace que los patrones sean más sencillos de instanciar y manipular ya que el programador debe suministrar menos información. Sin embargo, si un rol en un patrón *sólo* puede ser derivado y no puede ser definido, la herramienta sería muy limitada. Como bien se observa en [Hedin97]:

"En general, el uso de roles derivados tiene la desventaja de atar más la definición del patrón a una implementación específica. Por ejemplo, el hecho de que todos los métodos en Component juegen el rol de Operation, excluye la posibilidad de que el programador defina métodos en Component que están fuera del patrón Decorator. Por otra parte, los roles derivados tienen como ventaja que evitan que el programador tenga que declarar explícitamente todos los roles de un patrón"

La limitación de quedar atado a una implementación específica se puede observar en los trabajos analizados ya que el rol *Operation* es un rol derivado y lo juegan ***todos*** los métodos declarados en *Component*. En consecuencia, el programador no puede decorar sólo un subconjunto de los métodos existentes en Component. Lo ideal sería que por default la herramienta trate la mayor cantidad de roles posibles como derivados, pero si el programador lo necesita, pueda explícitamente definir qué elemento del sistema jugará un determinado rol.

Como se puede observar en el Apendice A de [Hedin00], los patrones de diseño se definen en el código asociado a las acciones semánticas. La forma en la que quedan expresados los patrones es poco declarativa y su comprensión se dificulta ya que es necesario entender la interacción con el Compilador y las estructuras de bajo nivel que éste maneja.

Los trabajos asumen la existencia de un Compilador abierto que tenga siempre disponible la representación de todas las declaraciones existentes en el código fuente. La compilación de la gramática de patrones se realiza en forma simultánea a la compilación del código fuente para generar los árboles sintácticos que se muestran en la Figura 3 - Parsing de Patrones. No se aclara cómo este enfoque se podría implementar en ambientes reales. En particular, en ambientes con compilación incremental, donde se compila y se agrega un método a una clase, como no se compilan todos las estructuras completamente, no se podrían ejecutar las acciones semánticas asociadas a los patrones de diseño y, en consecuencia, las estructuras de la Figura 3 - Parsing de Patrones quedarían des-sincronizadas con respecto a los cambios que se producen en el ambiente.

Por otra parte, como se observa en las conclusiones de la evaluación existente en [Tab00], la utilidad de una herramienta de documentación de código como DPDOC radica en su integración con el entorno en el cual el código se produce. Como APPLAB no es un verdadero entorno de desarrollo, se presentan muchas complicaciones al evaluar la usabilidad y utilidad de DPDOC.

3.3 Reificación de los patrones como Componentes de Diseño

En [Same02] se plantea que los patrones de diseño se representen de una forma concreta como *"Componentes de Diseño"*. Un c*omponente de diseño* es un artefacto que existe en un **repositorio** de componentes. Entonces, el programador para instanciar un patrón de diseño se conectaría a este al repositorio de componentes (patrones). Los problemas que se buscan encarar con este enfoque son:

- *falta de información de Diseño durante el mantenimiento de sistemas*. Los componentes que constituyen un sistema están pobremente documentados y la intención original con la que fueron concebidos no queda clara y a menudo desaparece durante el mantenimiento.
- *Intangibilidad de los patrones de diseño*. A pesar de que los patrones tienen una gran aceptación y uso, el hecho de que éstos no se traten como artefactos de software impide que se pueda desarrollar una ingeniería de software basada en patrones de diseño.

Un Componente de Diseño está descripto en el repositorio en 3 niveles de abstracción distintos:

- **Descripción del componente**. La descripción de un componente contendría básicamente todas las secciones que se usan para describir un patrón en [Gamma95] salvo la Estructura, los Participantes y las Colaboraciones. De esta forma, la descripción de un componente provee información general como la motivación, la aplicabilidad y las consecuencias pero no define cómo debe ser concretamente el diseño que surge de querer utilizar un determinado componente.
- **Modelo de roles del componente**. Un componente determinado podría tener uno o más *modelos de roles*. El autor señala que para cada variante de implementación de un patrón se debería definir un modelo de Roles diferente. Por ejemplo, el modelo de roles del *"Patrón Visitor"* (8.1) sería el siguiente:

```
roleModel {
      component:   Visitor Pattern
      name: default

      roleType Visitor: declara un mensaje  visitXXX  por cada clase
      existente en la jerarquía de Elements
      roleType ConcretVisitor: cada ConcretVisitor implementa los
      mensajes visitXXX  ...
      roleType Element: define un mensaje acceptVisitor ...
      roleType ConcretElement: implementa el mensaje accepVisitor ...
}
```

Del ejemplo anterior se puede observar que un modelo de roles se corresponde aproximadamente con las secciones *participantes* y *colaboraciones* que existen en la descripción de un patrón. Durante la instanciación de un componente, el programador debe asignar una clase del sistema a cada uno de los roles.

- **Implementación del componente**. Nuevamente, para un determinado modelo de roles podrían existir distintas implementaciones. En este caso, cada implementación se correspondería con la elección de un determinado lenguaje

de programación. Por ejemplo, la implementación del componente asociado al Patrón Visitor en Smalltalk sería:

```
roleModel {
	component:   Visitor Pattern
	rolemodel: default
	plataform: Smalltalk

	roleType Visitor
		visit$ConcretElement$: a$ConcretElement$
			self subclassResponsibility
	roleType ConcretVisitor extends Visitor
	roleType Element
		accept: a$Visitor$
	roleType ConcretElement
		accept: a$Visitor$
			a$Visitor$ visit$ConcretElement$ self
}
```

Instanciación de un Componente

Durante la instanciación de un componente los nombres de roles encerrados por el signo **'$'** se reemplazan por los nombres de las clases que juegan dichos roles. Como pueden existir varias clases que jueguen un mismo rol, un determinado template puede reemplazarse varias veces. Por ejemplo, para la clase que juegue el rol de *Visitor*, a partir del template

```
visit$ConcretElement$: a$ConcretElement$
```

se generarán los métodos

```
visitConcretElementA: aConcretElementA
		self subclassResponsibility
visitConcretElementB: aConcretElementB
		self subclassResponsibility
```

Suponiendo que ConcretElementA y ConcretElementB son las clases que juegan el rol de ConcretElement.

3.3.1 Discusión

- En el modelo de implementación se define un mecanismo que puede resultar útil para generar código.
- En el modelo de roles se explicitan los roles que el programador debe asignar para instanciar un componente. Sin embargo, las colaboraciones y las relaciones que deben existir entre los distintos roles sólo se describen de manera informal usando texto. Como consecuencia, luego de que instancia un componente de diseño, y se realizan modificaciones sobre el código, no hay forma de validar que el patrón continúe siendo bien utilizado.
- Como se observó anteriormente uno de los problemas que se pretende atacar es que la intención de un determinado diseño *no se desvirtue* durante la fase mantenimiento. Este problema queda sin resolver porque no se encara un problema previo que es validar si un patrón se está usando correctamente o no.
- En la herramienta se busca la independencia del repositorio de Componentes de Diseño con respecto al lenguaje de programación pero no se encara problema de

cómo la herramienta debería interactuar con el ambiente de desarrollo a los efectos de mantener la consistencia código-diseño.

- En definitiva, tanto este enfoque como el que se presenta en [Bud96], sólo sirven para generar código cuando se instancia un patrón.

3.4 LePUS: un lenguaje para especificar patrones de Diseño

En [Eden00] y [Eden02] se define el lenguaje LePUS (*LanguagE for Patterns Uniform Specification*)[6]. LePUS es un lenguaje formal que utiliza notaciones lógico-matemáticas y que tiene además una representación gráfica. Los objetivos principales que planteó el autor del lenguaje fueron:

- poder dar una especificación precisa de las micro-arquitecturas que definen los patrones de diseño.
- encontrar o definir los "bloques de construcción" y abstracciones que permitan describir y entender los patrones de diseño.

3.4.1 Modelo del Diseño

La estrategia seguida para definir el lenguaje es representar las clases, métodos, atributos y las relaciones entre ellos en una estructura matemática. El primer concepto que se define es el *Modelo del Diseño* de un programa orientado a objetos. Formalmente, si **P** es un programa válido, entonces el Modelo de Diseño de P se denota:

```
M(P) = < U, R>
```

donde:

- **U**: es el conjunto de *ground entities* o átomos que existen en **P** . A su vez este conjunto está particionado en:
- **C**: el conjunto de todas las clases existentes en P
- **F**: el conjunto de todos los métodos existentes en P
- **S**: el conjunto de todas las signaturas de métodos existentes en P
- **R**: es el conjunto de todas las relaciones existentes entre las entidades de U (*ground relations*)

Por ejemplo, si P es el siguiente programa:

```
abstract class Decorator {
   abstract void Draw();
}
class BorderDecorator extends Decorator {
   void Draw() {
      Decorator.Draw(); //...
   }
   void Rotate(int degree) { //...
   }
   int BorderWidth;
}
```

[6] Una descripción del lenguaje se puede consultar en la página http://www.eden-study.org/lepus/.

Entonces, cada uno de los componentes de *Modelo de Diseño* de P será:

Clases (**C**)	Métodos (**F**)	Signaturas (**S**)	Ground Relations (**R**)

`Decorator`
`BorderDeco`
`rator`
`int`
	`Decorator.Draw`
	`BorderDecorator.Draw`
		`<"Draw">`
		`<"Rotate",int>`
			`Abstract(Decorator)`
			`Abstract(Decorator.Draw)`
			`Member(int,BorderDecorator)`
			`Member(Decorator.Draw,Decorator)`
			`Member(BorderDecorator.Draw,BorderDecorator)`
			`SignatureOf(Decorator.Draw,<"Draw">)`
			`SignatureOf(BorderDecorator.Draw,<"Draw">)`
			`SignatureOf(BorderDecorator.Rotate,<"Rotate"`
			`,int>)`
			`Inherit(BorderDecorator,Decorator)`
			`Invoke(BorderDecorator.Draw,Decorator.Draw)`

Algunas de las otras relaciones existentes son:

- **Abstract: C∪F** . *Abstract(c)* es verdera si la clase **c** es abstracta. *Abstract(f)* es verdaera si el método f es abstracto.
- **Inherit: CxC**. *Inherit(c1, c2)* es verdadera si la clase c1 hereda directa o indirectamente de c2.
- **Invoke: FxF** . *Invoke(f, g)* es verdadera si dentro del método **f** existe una expresión cuya evaluación invoca a **g**. Se ignora el flujo de control.
- **Forward: FxF** *Forward(f, g)* es verdadera si se cumple que *Invoke(f, g)* y además los parámetros con los que se invoca a **g** son los parámetros formales de **f**
- **Create: FxC**. *Create(f,c)* es verdadera si el cuerpo del método **f** contiene una expresión cuya evaluación crea una instancia de **c.**
- **ReturnType: FxC**. *ReturnType(f,c)* es verdadera si el tipo de retorno del método f es C

3.4.2 Elementos de LePUS

Los elementos del lenguaje son los siguientes:

- **Variables**: una variable es un símbolo cuyos posibles valores están dentro de un dominio específico. El dominio de una variable siempre debe ser un conjunto uniforme. Un *conjunto uniforme* es un conjunto en el cual todos los elementos son del mismo tipo. Por ejemplo, C, F y S son conjuntos uniformes. Pero también los conjuntos de orden superior (o sea conjuntos cuyos elementos son conjuntos) son posibles dominios. Los siguientes son ejemplos de declaraciones de variables:

Declaración	**Descripción**
decorator : C	Declara que la variable 'decorator' puede tomar como valor alguna clase c tal que $c \in C$
ConcretDecorators: P(C)	Declara que 'ConcretDecorators ' puede tomar como valor algún conjunto C' tal que $C' \in P(C)$
Visit : $P^2(F)$	Se declara la variable 'Visit' cuyos valores posibles son conjuntos de conjuntos de métodos
Visitors : H	el dominio H es el conjunto de todas las jerarquías existentes en un

	programa dado. Un conjunto de clases C se dice que es una jerarquía sii existe una clase root $\in$ C tal que: root es abstracta y las demás clase de C heredan de root directa o indirectamente. Visitors tiene como valores posibles alguna jerarquía h tal que h $\in$ H

- **Relaciones**: además de las *"Ground Relations"* que deberían poder establecerse a partir del análisis estático del código fuente. LePUS define los siguientes *Predicados*:

  - `Clan(F,C)` donde F es un conjunto de métodos y C un conjunto de clases. Este predicado es verdadero si todos los métodos de F tienen la misma signatura y cada método de F está definido en exactamente una clase de C. Este predicado se lee como *"el conjunto de funciones en F forma un clan en C"*. Por ejemplo, el conjunto de métodos:

```
{Visitor>>visitConcretElementA ,
ConcretVisitor1>>visitConcretElementA,
ConcretVisitor2>>visitConcretElementA }
```

    en el Patrón Visitor forman un Clan en la jerarquía de Visitors donde esta jerarquía es el conjunto de clases

```
{Visitor, ConcretVisitor1,ConcretVisitor2 }
```

  - `Tribe(F,C)` es verdadero si `F` es un conjunto de conjuntos de métodos tal que `∀f∈F: Clan(f,C)`. Por ejemplo si tomamos `F` como:

```
{ {Visitor>>visitConcretElementA,
ConcretVisitor1>>visitConcretElementA,
ConcretVisitor2>>visitConcretElementA } ,
{Visitor>>visitConcretElementB,
ConcretVisitor1>>visitConcretElementB,
ConcretVisitor2>>visitConcretElementB } }
```

    entonces se cumple que `Tribe(F,Visitors)`

Por otra parte, dada una relación `R` definida en `DomxRan` el predicado `Total(R, Dom,Ran)` es veradero sii `∀x∈Dom, ∃y∈Ran: xRy` . Una relación total `R` se denota como $R^{\rightarrow}$`(Dom,Ran)`. Si la Relacion es una funcion biyectiva se denota $R^{\leftrightarrow}$`(Dom,Ran)`

Operadores: El único operador definido en el lenguaje es el operador selección

$\otimes : P^{m}(S) \times P^{n}(C) \rightarrow P^{m+n}(F)$

Este operador dada una signatura (mensaje) o un conjunto de signaturas permite recuperar todos las implementaciones de ese mensaje en una clase o un conjunto de clases. Por ejemplo, en el Patrón Visitor:

```
acceptVisitor : S
```

La expresion `acceptVisitor ⊗ Elements` devuelve el conjunto de métodos siguiente:

```
{ Element>>acceptVisitor: , ConcretElementA>>acceptVisitor: ,
ConcretElementB>>acceptVisitor: }
```

3.4.3 Especificación de patrones en LePUS

Un patrón de diseño en LePUS *se define* a través de una fórmula ϕ(x1, ... xn) en la cual las variables *x1* ... *xn* son las variables libres en ϕ. Se dice que las variables *x1* ... *xn* son los *participantes* del patrón y que las relaciones existentes en ϕ son las *colaboraciones* que el patrón determina.

Dado un programa **P** y una asignación de entidades **A** = < a1, ... an>, en la cual se mapean las variables *x1, ... xn* con elementos de C, F o S, se dice que <a1, ... an> es una *instancia del patrón ϕ* en el contexto de A si ϕ(a1, ... an) es verdadera en M(P). Por ejemplo, si se toma el patrón trivial `Invoke(f1,f2)` en el Modelo del Diseño dado como ejemplo, y se considera la asignación

```
A = <BorderDecorator.Draw→f1, Decorator.Draw → f2 >,
```

entonces es fácil ver que `<BorderDecorator.Draw, Decorator.Draw>` es una instancia del patrón `Invoke(f1,f2)` en el contexto de A ya que la fórmula

`Invoke(BorderDecorator.Draw,Decorator.Draw)` es verdadera en M(P)

A continuación se mostrará como se especifica en LePUS el *"Patrón Decorator"* (3.1.1)

3.4.3.1 Patrón Decorator en LePUS

Los participantes del patrón (variables) se declaran en la sección superior y las relaciones que deben existir entre ellos en la parte inferior.

```
component, decorator : C
ConcreteComponents, ConcreteDecorators : P(C)
Operations : P(S)
-----------------------------------------------------------------
Abstract(component)
Abstract(decorator)
Inherit(decorator, component)
Member(component, decorator)
Inherit→(ConcreteDecorators, decorator)
Inherit→(ConcreteComponents, component)
Forward↔(Operations ⊗ ConcreteDecorators, Operations ⊗decorator)
Forward↔(Operations ⊗ decorator, Operations ⊗component)
Operations ⊗ ConcreteComponents
```

3.4.4 Discusión

3.4.4.1 Falta de una representación adecuada de los mensajes

Un problema que observamos en LePUS es que el concepto de mensaje, central al paradigma de objetos, se modela sólo tangencialmente en la idea de Signatura. El papel central lo ocupan los métodos y las relaciones entre métodos (por ejemplo `Forward(metodo1, metodo2)` e `Invoque(metodo1, metodo2)`)[7]. En general, a causa del binding dinámico, la existencia de relaciones entre métodos no se puede inferir a partir del análisis estático del código fuente. En [Eden00] no se describe en suficiente detalle el problema de cómo generar el Modelo del Diseño de un programa. (ver sección 3.4.1). En particular, para el caso de las invocaciones entre métodos sólo se muestran ejemplos sencillos como por ejemplo:

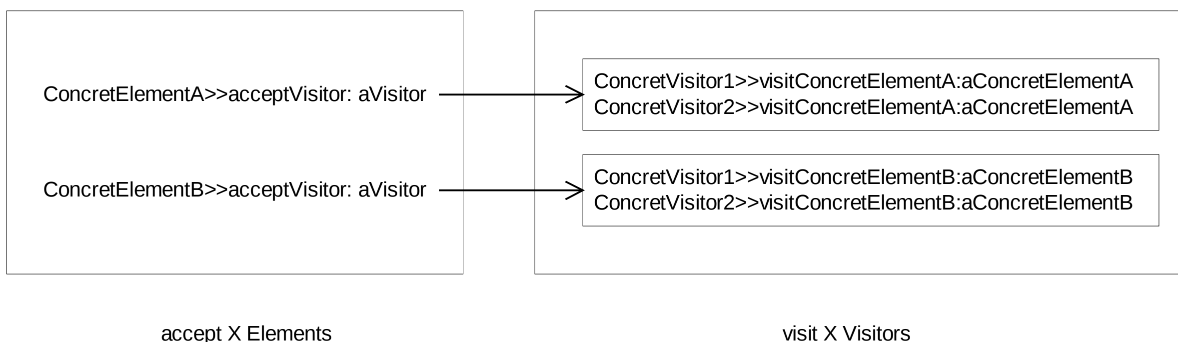
```
class BorderDecorator extends Decorator {
   void Draw() {
      Decorator.Draw(); //...
}
```

En este caso, excepcionalmente, sí se puede concluir que se cumple relación `Invoke(BorderDecorator.Draw, Decorator.Draw)` pero este tipo de construcción es particular de lenguajes como C++. En general, si analizamos el código fuente de un método en particular, podremos determinar qué mensajes se envían dentro de dicho método pero no podremos saber qué métodos se invocan. En consecuencia, la relaciones `Forward` e `Invoque` definidas entre métodos no son adecuadas para chequear estáticamente las propiedades que debe cumplir un patrón de diseño. Por ejemplo, para el Patrón Visitor en LePUS se define que se debe cumplir el siguiente isomorfismo:

```
Invoque↔(accept ⊗ Elements, visit ⊗ Visitors )
donde

accept: S
Elements, Visitors: H
```

`accept ⊗ Elements` es un Clan de métodos en la jerarquía de Elements y `visit ⊗ Visitors` es un Tribe de métodos en la jerarquía de Visitors. Gráficamente, este isomorfismo se puede representar de la siguiente forma:



Al carecer LePUS del concepto de mensaje, no define cómo realizar el chequeo estático de la existencia de este isomorfismo. Lo único que se podría chequear en forma estática es que dentro de cada método `ConcretElementX>>acceptVisitor: aVisitor` se envíe a `'aVisitor'` el mensaje correcto que sería el siguiente:

```
ConcretElementX>>acceptVisitor: aVisitor
      aVisitor visitConcretElementX:  self
```

[7] El concepto de método que se usa en LePUS es el que se usa normalmente en el Paradigma de POO: un método es una de las posibles implementaciones de un *mensaje*. Siempre un método está definido en una clase determinada.

Por otra parte, en ninguno de los 2 isomorfismos planteados la fórmula del patrón Decorator (3.4.3.1):

```
Forward↔(Operations ⊗ ConcreteDecorators, Operations ⊗decorator)
Forward↔(Operations ⊗ decorator, Operations ⊗component)
```

queda claro que si vale `Forward(met1, met2)` entonces `met1` y `met2` *deberían* tener la misma signatura (como lo establece el patrón Decorator, ver sección 3.1.1). O sea, la segunda expresión anterior, solamente nos indica que para que un patrón Decorator sea válido, debería existir una correspondencia biunívoca entre las implementaciones de `Operations` en el `decorator` (`Operations ⊗ decorator`) y las implementaciones de `Operations` en el `component` (`Operations ⊗ component`). Pero podría valer que un `decorator` delege un mensaje enviándole otro mensaje a su `component`:

```
Decorator>>doSomeThing: anObject
       component doAnotherThing: anObject
```

en este caso se cumple la relacion `Forward(Decorator>>doSomeThing, Component>>doAnotherThing)` pero los métodos no tienen la misma signatura.

3.4.4.2 Ausencia del concepto de receptor del mensaje.

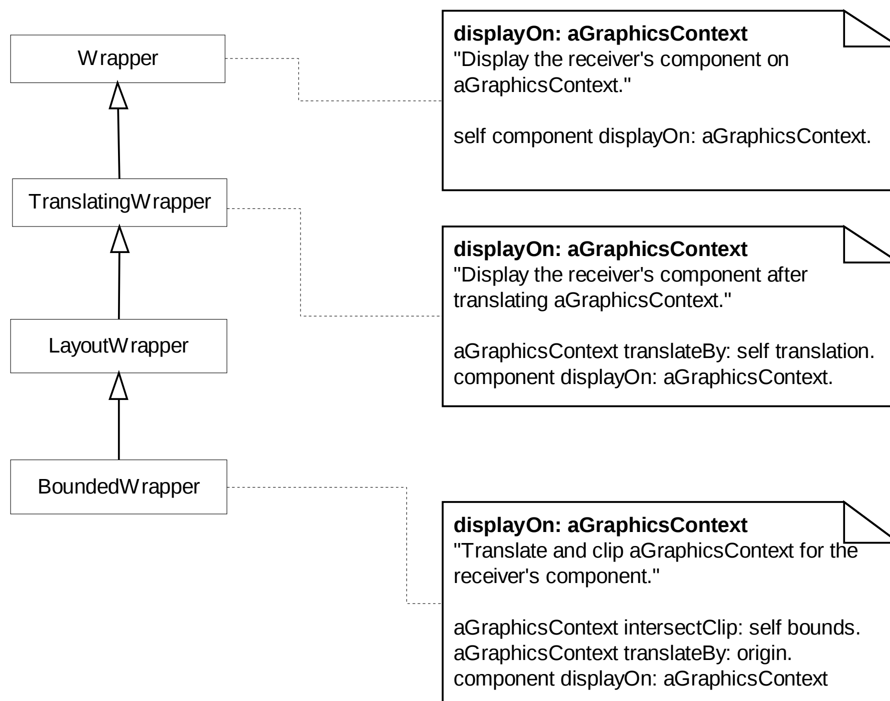
En LePUS no existe ningún mecanismo para explicitar quién debe ser el *receptor* de un determinado mensaje. Por ejemplo, en el caso del Patrón Decorator se especifica que un Decorator referencia a un Component en la sentencia `Member(component, decorator)`, pero no se define a través de qué variable de instancia lo referencia. En consecuencia, no se especifica claramente que cada ConcretDecorator debe forwardear directa o indirectamente los mensajes recibidos a una variable de instancia específica

3.4.4.3 Restricciones no realistas a las implementaciones de un patrón

En la fórmula del patrón Decorator (3.4.3.1) la expresión

```
Forward↔(Operations ⊗ ConcreteDecorators, Operations ⊗decorator)
```

Para ser verdadera requiere que cada ConcreteDecorator forwardeé el mensaje al componente usando la implementación que tiene el Decorator. Si bien ésta es una alternativa posible para forwardear el mensaje al componente, es muy frecuente observar que el mensaje se forwardea directamente a colaborador `component`. En el ejemplo del patrón Decorator (3.1.2) se observan las siguientes implementaciones del mensaje `displayOn:`



Se puede ver que tanto BoundedWrapper como TranslatingWrapper forwardean el mensaje `displayOn:` directamente al colaborador `component`. TranslatingWapper podría forwardear el mensaje usando la implementación existente en Wrapper usando

```
super displayOn: aGraphicsContext
```

en lugar de

```
component displayOn: aGraphicsContext.
```

Pero BoundedWrappper no puede usar `super displayOn:` porque cambiaría la semántica del método. Por otra parte, en Smalltalk no existe la posibilidad de especificar desde una subclase el uso de método existente en una superclase. En C++ BoundedWrapper podría usar

```
Wrapper.displayOn (aGraphicsContext)
```

En conclusión, la expresión `Forward⁻(Operations ⊗ ConcreteDecorators,Operations ⊗decorator)` no es adecuada para describir como el patrón Decorator se aplica en la práctica.

3.4.4.4 Utilidad del formalismo

En [Eden00], se muestra a grandes rasgos como una especificación LePUS se traduciría a Prolog. Basado en esta traducción, el autor describe un prototipo de herramienta realizada en Prolog con la cual se podrían reconocer, validar y aplicar patrones. Por ejemplo, la validación de un patrón se realizaría de la siguiente forma:

- un programa **P** se traduce a una base de datos Prolog en la cual los `atomos y facts` de Prolog representan las *ground entities* y *ground relations* de LePUS

(ver sección 3.4.1). Esta traducción se haría automáticamente o con asistencia humana.

- Para ver si un conjunto de atomos `a1,... aN` consituyen una instancia de un patrón, se ejecuta una *query* Prolog usando `a1,... aN` como argumentos de la query. Observar que los átomos se mapean con las ground entities de LePUS, que a su vez abarcan las clases, métodos y signaturas existentes en **P**. De esta forma, podríamos saber si las entidades asociadas a los atomos `a1,... aN` en **P** cumplen un determinado patrón.

Este enfoque, debido a la falta de integración de la herramienta con el entorno de desarrollo, presenta los problemas que se describieron en la sección 1.3. Por otra parte, el prototipo de la herramienta no se encuentra disponible y entonces no se pueden conocer sus detalles. Tampoco resultaría factible usar LePUS como base para una herramienta integrada a un IDE por los problemas que se expusieron anteriormente: falta de una representación adecuada de los mensajes, ausencia del concepto de receptor del mensaje. y restricciones no realistas a las implementaciones de un patrón (3.4.4.1, 3.4.4.2 y 3.4.4.3). En síntesis, si bien resulta interesante la búsqueda de la formalización del *diseño de un programa*, la utilidad del formalismo obtenido está lejos de ser probada.

3.5 Los patrones como un Framework

En [Meij96] y [Grui97] los patrones de diseño se representan a través de *“el modelo de fragmentos”*. En este modelo, cada componente relevante dentro de un patrón (métodos, atributos, clases, relaciones de herencia, relaciones entre clases y los mismos patrones) se representa como un fragmento.

Los fragmentos están interconectados por roles. Cada fragmento puede jugar diferentes roles para distintos fragmentos. Internamente cada fragmento tiene un conjunto de slots a través de los cuales se implementan los roles y se almacena el comportamiento. Cada patrón tiene asociado un prototipo que contiene la información de los roles definidos en el patrón y el comportamiento para chequear que sus instancias respeten ciertas propiedades. Cuando el programador quiere usar un determinado patrón en su aplicación, se produce la clonación del prototipo correspondiente. El nuevo objeto resultado de la clonación, que es una instancia de patrón, tiene una referencia a su prototipo del cual obtiene su comportamiento usando delegación.

El modelo de fragmentos usa fuertemente las ideas de prototipación, clonación y delegación como se definen en el lenguaje Self [Ungar87]. La implementación de la herramienta fue hecha en Smalltalk [Gold86].

3.5.1 Discusión

- Este modelo al centrarse en dar una representación uniforme para todos los elementos de un programa, termina reduciendo los diferentes niveles de abstracción a sólo fragmentos que se interconectan por roles. Esto hace que el diseño de la aplicación quede representado como un grafo de fragmentos que es de difícil comprensión y manipulación.
- Uno de los objetivos de introducir los patrones en un ambiente de desarrollo es simplificar la representación del diseño. Sin embargo, este modelo hace necesario aprender un concepto nuevo (fragmento) que no posee ningún valor semántico particular.

4 Dificultades de modelar los patrones de Diseño

En la sección *"Revisión del Estado del Arte"* se analizaron los distintos enfoques existentes para modelar a los patrones de diseño. Además, para cada enfoque, se discutieron las ventajas y problemas que presenta. En esta sección, se profundizará el análisis de los problemas que se presentan al modelar a los patrones de diseño. Buscaremos dar respuesta a las siguientes preguntas:

- ¿Qué características de los patrones de diseño los hace resistentes al soporte de herramientas?
- ¿Por qué estas características hacen complejo crear un modelo de patrones que sirva de base para la creación de herramientas?
- ¿Cuáles son las razones por las que los trabajos realizados hasta el momento en herramientas de patrones presentan los problemas descriptos en 1.3 ?

Esta discusión nos permitirá tener una mayor comprensión de las dificultades del problema al que nos enfrentamos y nos servirá como contexto para la presentación del enfoque seguido en este trabajo.

4.1 Los patrones: a la vez definición de problema y solución

Como vimos en la sección 2.2.3, un patrón abarca tanto la descripción de un problema de diseño como la descripción de una de las posibles soluciones a dicho problema. El *problema* que ataca el patrón se encuentra en las secciones *Intención, Motivación y Aplicabilidad*. En la sección de Motivación se muestra un ejemplo de cómo puede utilizarse el patrón para resolver un problema concreto de diseño. La sección de Aplicabilidad está expresada en un nivel mayor de abstracción y describe las situaciones en las que el patrón puede utilizarse, ayudando a reconocer dichas situaciones. Por otra parte, la *solución* se expone en las secciones Estructura, Participantes y Colaboraciones.

En el trabajo [Vlis98], Vlissides destaca la importancia de la *intención o problema* que se apunta a resolver con determinado patrón de diseño. Este aspecto de los patrones, que constituye una parte esencial, es a menudo eclipsado por el foco que se pone en la Estructura. Vlissides observa que:

- un error muy común es intentar diferenciar los patrones realizando *sólo* una comparación de sus diagramas de Estructura. Los diagramas de estructura son una parte de la descripción de un patrón. Existen patrones como el State y el Strategy que tienen estructuras casi idénticas y, sin embargo, su intención y el comportamiento en run-time de las implementaciones de estos patrones son muy distintas. Si queremos entender las diferencias entre un patrón y otro, tenemos que analizar la intención que es el diferenciador más conciso.
- el hecho de seguir en forma estricta un patrón de diseño, en sí mismo no tiene ningún valor. Un patrón es un medio para conseguir un *fin* y no un fin en sí

mismo. El fin es atacar el problema de diseño definido por la intención del patrón.

- se debe ser cuidadoso en la fase de documentación cuando se describe una solución en término de patrones. Si se identifica que un conjunto de clases siguen un determinado patrón, es necesario estar seguro de que se sigue la intención de dicho patrón. Si la conexión con la intención es tenue, no hay que mencionar el patrón porque sólo se genera confusión.

Por otra parte, Eden en [Eden00b] observa que "la investigación en herramientas de soporte y lenguajes de especificación se centra invariablemente en el *segmento de solución* de los patrones de diseño. O sea, apuntan a capturar las abstracciones detrás de todas las posibles implementaciones válidas de un patrón. Los lenguajes de especificación que evolucionaron de la investigación del catálogo de GoF reformulan la información incluida en las secciones *estructura, participantes y colaboraciones*". Esto, como se observa en [Eden98] también es válido para LePUS.

Coincidimos con Vlissides en la afirmación de que los patrones *no* son un fin en sí mismo, sino que son un *medio* para resolver un problema. Cuando frente a un problema concreto, se quiere aplicar en forma estricta la estructura de solución presentada en GoF, se corre el riesgo de descuidar la identificación y resolución efectiva de los matices que presenta dicho problema. Esta actitud es la que Vlissides denomina *legalismo de los patrones*.

4.2 ¿Los patrones son formalizables?

Para crear un modelo sobre los patrones de diseño, es preciso volver a la pregunta inicial: ¿qué es un patrón de diseño? Nuestra aproximación para responder a esta pregunta fue: un patrón es una *solución genérica* para un tipo de problema de diseño que ocurre frecuentemente. Ahora bien, esta solución genérica, ¿*especifica* de forma no ambigua un conjunto de restricciones que la implementación debe cumplir para seguir al patrón de diseño? Para analizar esta pregunta es interesante contrastar las posturas antagónicas existentes en la literatura.

Por una parte, John Vlissides, uno de los autores de [Gamma95], observa en [Vlis98] que:

- el objetivo del diagrama que está en la sección de Estructura de un patrón, es mostrar un *ejemplo* de una implementación que se ve frecuentemente en la práctica. Este diagrama *no* pretende ser una especificación de cómo debe implementarse un patrón.
- el uso de una notación semi-formal en la Estructura, sugiere a los lectores un nivel de precisión que *no* era buscado por los autores del catálogo. Este malentendido da origen al "legalismo de los patrones" que pretende que cada vez que se use un patrón se adhiera en forma rigurosa a la estructura descripta en GoF.

Por otra parte, Ammon Eden, cuyos trabajos hemos analizado en la sección 3.4, observa en [Eden00b] que:

- los medios de especificación utilizados en GoF (descripciones verbales mezcladas con jerga técnica, fragmentos de código fuente, diagramas de clase y de objetos), a pesar de su uso sistemático, no son lo suficientemente rigurosos para permitir una interpretación no ambigua de los patrones de diseño.
- las especificaciones coherentes de los patrones son esenciales para mejorar su comprensión (por ejemplo poder establecer si un patrón usa a otro, o si un patrón es un caso particular de otro patrón), para permitir un razonamiento formal sobre sus propiedades y relaciones y para soportar y automatizar su aplicación.
- existe una tendencia dentro de la comunidad de patrones que no ve favorablemente los intentos de analizar científicamente a los patrones de diseño. Citando a Eden, según esta visión irracional *"un patrón es una idea, un elemento de un lenguaje y un concepto cuasi-corpóreo cuya esencia es intangible, elusiva y por lo tanto más allá del alcance de una expresión literal. Un 'buen' patrón se aleja de una mera prescripción micro-arquitectural por una cualidad inmaterial que no puede ser explícitamente expresada ... "*

Observamos entonces, una clara diferencia de criterio con respecto a lo que deberíamos esperar de un patrón de diseño. Vlissides no pretende que un patrón sea una especificación mientras que Eden recalca que es imprescindible que un patrón se defina usando un lenguaje formal y para ello propone LePUS (ver 3.4). Eden considera que la formalización es un *pre-requisito* para la creación de herramientas que permitan automatizar la utilización de patrones de diseño. Sin embargo, como hemos observado en la sección 3.4.4, la existencia de un formalismo no asegura la construcción de una herramienta de utilidad práctica, ya que el propio formalismo puede no ser adecuado.

4.3 ¿Los patrones se describen con el mismo nivel de precisión?

Este problema está relacionado con la discusión expuesta en el punto 4.1 pero se plantea al observar que las soluciones que describen los distintos patrones de diseño pueden ser más o menos abiertas. Entonces surge la siguiente pregunta ¿cuál es el mínimo nivel de precisión que debe tener la *solución* de un patrón para que pueda automatizarse y validarse su uso?

A continuación se expone un análisis realizado en [Eden00b], en el cual tomando como criterio el nivel de precisión y formalidad, se clasifican las descripciones verbales de los patrones de diseño existentes en [Gamma95]. En dicho análisis se identifican 6 categorías de descripciones que se muestran con ejemplos en la siguiente tabla:

Categoría	**Ejemplos (tomados de [Gamma95])**
1) Precisa, singular	- *Participante Decorator*: "mantiene una referencia a un objeto Component" - *Participante ConcreteElement del patrón Visitor*: "implementa una operación Acept que tiene a un Visitor como argumento"
2) Alternativas enumeradas	- *Participante Creator del patrón Factory Method*: "puede llamar al factory method para crear un objeto Producto" - *Colaboraciones del patrón Strategy*: "alternativamente, el

	contexto puede pasarse a sí mismo como un argumento a las operaciones de las estrategias" - *Colaboraciones del patrón Decorator*: "el Decorator puede opcionalmente ejecutar operaciones adicionales antes y después de forwardear las operaciones"
3) Generalización precisa	- *Participante Visitor*: "declara una operación `visit` por cada clase de ConcreteElement en la estructura de objetos" - *Participante Decorator:* "define una interface que se ajusta a la interface del Componente"
4) Términos técnicos pero abiertos a varias interpretaciones	- *Participante Proxy*: "el tipo de proxy virtual puede puede **cachear** información adicional sobre el RealSubject para de esa forma posponer al acceso a éste" - *Participante Prototype*: "declara una interface para **clonarse** así mismo" - *Participante Memento*: "almacena el **estado interno** del objeto Originator"
5) Poco claro, informal o descripción teleológica	- *Participante Adaptee*: "define una interface existente que **necesita adaptación**" - *Participante Componente del patrón Composite*: "implementa el comportamiento default para la interface común a todas las clases como **sea apropiado**" - *Colaboraciones del patrón Observer*: "Después de que el ConcreteObserver es notificado de un cambio en el ConcreteSubject, aquel le puede pedir información adicional a éste. El ConcreteObserver usa esta información para **reconciliar** su estado con el del subject"
6) Omisión deliberada de detalle	- *Sección implementación del patrón State*: "El patrón State no especifica qué participante define los criterios para las transiciones de estado"

Las conclusiones más relevantes del trabajo [Eden00b] son:

- las 3 primeras categorías de la tabla anterior pueden considerarse relativamente *precisas.* Una descripción *precisa*, no necesariamente indica una implementación singular en cada lenguaje OO, sino que debe tener un conjunto "compacto" de interpretaciones. Por ejemplo, "el contexto puede pasarse a sí mismo como un argumento a las operaciones de las estrategias" tiene las siguientes interpretaciones en Smalltalk y C++ respectivamente:

```
aStrategy operation: self
aStrategy.operation: *this
```

- En la mayoría de los patrones de GoF, gran parte de las especificaciones de las soluciones son precisas, en el sentido que las descripciones caen en las categorías 1, 2 y 3. Este tipo de descripciones son las que Eden tomó como base para definir LePUS (ver sección 3.4)

Observamos que, si bien las categorías 1, 2 y 3 definen a priori una solución más determinada, no está claro como afirma Eden que la mayoría de las soluciones de los patrones de GoF sean precisas. Por otra parte, aunque la solución de un patrón pueda ser precisa, como por ejemplo podría serlo la solución del patrón Decorator, las

aplicaciones prácticas de un patrón normalmente presentarán las desviaciones discutidas en el punto 4.5

4.4 Las múltiples variantes de implementación de un patrón

Hemos destacado en el apartado anterior los problemas de aplicar los patrones de Diseño en forma mecánica y rígida. Para aplicar correctamente un patrón, un programador debería tener en cuenta las cuestiones de implementación que precisamente se plantean en la sección *Implementación* (ver sección 2.2.3). Las siguientes observaciones remarcan que esto no es la práctica habitual en nuestra industria:

- "Desafortunadamente, cuando los programadores miran el diagrama de Estructura que acompaña cada patrón en el libro de GoF, frecuentemente llegan a la conclusión de que el Diagrama de Estructura es ***la*** manera de implementar el patrón. Muchos programadores toman el libro de GoF, miran fijamente el Diagrama de Estructura y luego empiezan a codificar. El resultado es que el código imita exactamente el Diagrama de Estructura, en vez ser implementar el patrón adecuándose a las necesidades del problema." [Ker04]
- "En las implementaciones reales de un patrón, las desviaciones con respecto a la Estructura existente en GoF son inevitables e incluso deseables. Como mínimo usted va a renombrar los participantes de forma que sea apropiado para su dominio. Varíe los compromisos de implementación y su implementación comenzará a ser bastante distinta de la del Diagrama de Estructura" [Vlis98]

Para resolver cada problema de diseño debemos tener en cuenta las particularidades que éste presenta, y además, debemos tomar decisiones sobre varios aspectos de la implementación. Esto hará que las distintas aplicaciones de un mismo patrón puedan ser bastante diferentes entre sí. Esta característica de los patrones de diseño hace que sea complejo tener una herramienta que "automatice" el uso de patrones. Por ejemplo, para validar el "uso correcto" de un patrón, una herramienta debe reconocer en el código la *microarquitectura* asociada al patrón (un conjunto de participantes que colaboran de una forma determinada). El problema es que la microarquitectura de un patrón como observa [Vlis98] en 4.2 es sólo un *ejemplo* de la estructura que frecuentemente se ve en la práctica. Entonces, si una herramienta se limita a trabajar con la estructura de un patrón tal como se define en GoF, puede llegar a ser muy restrictiva y no tener valor práctico.

4.5 "Desviaciones" al aplicar patrones de diseño

En teoría, la utilización de un determinado patrón de diseño debería imponer un conjunto de restricciones sobre el código fuente. Ahora bien, generalmente las aplicaciones prácticas de un patrón no siempre siguen todas las restricciones. Por ejemplo, según el patrón Decorator (3.1.1), cada ConcreteDecorator debe forwardear directa o indirectamente los mensajes que recibe a su componente. Sin embargo, si analizamos el ejemplo de patrón Decorator que se muestra en la sección 3.1.2, vemos

que existen decoradores concretos que ***no forwardean*** el mensaje `'displayOn: aGraphicsContext'` a su componente:

```
StrokingWrapper>>displayOn: aGraphicsContext
"Display my component by stroking it."

lineWidth == nil
      ifFalse:[aGraphicsContext lineWidth: lineWidth].
component displayStrokedOn: aGraphicsContext


FillingWrapper>>displayOn: aGraphicsContext
"Display my component by filling it."

component displayFilledOn: aGraphicsContext
```

Estas desviaciones con respecto a la forma "ideal" o teórica de un patrón son un hecho en la aplicación práctica, y, por lo tanto, tienen que ser tenidas en cuenta por cualquier herramienta que pretenda facilitar la tarea del programador. Por ejemplo, si la herramienta detecta una de estas desviaciones, debería advertir al programador porque eventualmente puede tratarse de un error de programación, pero, a su vez, el programador debe poder informar que no se trata de un error sino de una decisión deliberada. En este caso, para que la herramienta sea útil y no se convierta en una molestia para el programador, tendría que incorporar de alguna forma esta información para no reportar más de una vez el mismo "error" o desviación.

4.6 Conclusiones

Un aspecto conceptual que suele generar confusión en la comprensión de los patrones de diseño es su naturaleza *dual*: un patrón abarca tanto la descripción de un problema como una de sus posibles soluciones.

Las soluciones de los distintos patrones de diseño se describen con un nivel de precisión muy variable. Los patrones que pertenecen a las categorías de descripciones más precisas (categorías 1, 2 y 3 de la sección 4.3) serían los más aptos para recibir soporte de herramientas. Para estos patrones, sería factible definir un conjunto compacto de implementaciones que permita instanciar, manipular y validar su uso.

Por otra parte, por más que la descripción de un patrón sea *precisa* (categoría 1), como observa Vlissides tenemos que tener en cuenta que la Estructura de un patrón es un *ejemplo* de implementación frecuentemente utilizado, pero *no* pretende ser una especificación. Por ejemplo, en el patrón Decorator (ver sección 3.1.1) que se expone en [Gamma95] el Decorator es una subclase del Component y el Decorator tiene una variable de instancia que referencia al Component. Sin embargo, existen otras alternativas de implementación:

- el Decorator no debe heredar necesariamente de Component, basta con que el Decorator y el Component sean polimórficos respecto del protocolo de Component. Esto permite que la existencia del Decorator sea transparente para el cliente.

- el Decorator no necesariamente debe referenciar al Component con una variable de instancia, basta con que pueda obtener al Component de alguna forma.

En síntesis, podemos concluir que para que un patrón de diseño pueda formalizarse y tener un soporte de herramientas, es condición necesaria que la descripción de su solución tenga un mínimo nivel de precisión. Para cada uno de estos patrones, es necesario identificar las variantes de implementación más utilizadas y darles a éstas un soporte realista en el entorno de desarrollo. Este soporte realista implica implementar un manejo adecuado de las "desviaciones" que normalmente ocurren al aplicar un patrón de diseño.

5 El Framework de Patrones

En esta sección se expondrá el Framework de Patrones construido para lograr los objetivos definidos en la sección 1.1. Como se observó anteriormente, el Framework se describirá usando el *"Caso de Estudio"* (3.1) existente en la sección de *"Revisión del Estado del Arte"* (3), para facilitar la comparación de los distintos enfoques. La descripción del Framework está organizada de la siguiente forma:

- En 5.1 se discutirán las razones por cuales se eligió como solución implementar el modelo de patrones como un Framework en Smalltalk
- En 5.2 se dará una visión general de los conceptos más importantes en los que se basa el Framework: las *definiciones* y las *instancias* de patrones. A los efectos de esta introducción y como primera aproximación, se puede decir que las *definiciones* pertenecen al Framework, y especifican la estructura y reglas comunes que comparten las *instancias* de patrones definidas por el programador
- En 5.3 se verá como el Framework, a través del modelo de roles, representa los aspectos estructurales de un patrón. Cada definición de patrón tiene asociado un conjunto de roles, y en el momento en el que un programador instancia un patrón, deberá especificar los elementos del sistema que *actúan* cada uno de dichos roles.
- La sección 5.4 tiene como objetivo discutir cómo se modelan las propiedades lógicas que deben cumplir las instancias de patrones. Se analizarán los distintos Predicados y Expresiones existentes en el Framework, y se verá como éstos se aplican para representar la formula lógica del patrón del *"Caso de Estudio"*
- Integrando los temas vistos en las secciones anteriores, en la sección en 5.5 se verá cómo se validan los patrones que instancia el programador.
- En la sección 5.6 se analizará el código con el cual se crean las definiciones e instancias de patrones. Se podrá observar que este código expresa claramente cuáles son las propiedades y restricciones que impone el uso del patrón.
- En la sección 5.7 se describe como el modelo de patrones interactúa con el entorno de desarrollo. Se muestra cómo el modelo propone acciones correctivas cuando detecta modificaciones realizadas por el programador, que si se produjeran, llevarían a los patrones utilizados a un estado inválido.
- Finalmente, en la sección 5.8 se detallan, en forma de referencia, algunas de las clases del Framework y ciertas características de su implementación. Para

mantener la simplicidad en la exposición del Framework, se remitirá al lector a puntos particulares de esta referencia para obtener mayores detalles e información

5.1 Enfoque y Metodología del trabajo

Hemos observado en la sección 1.3 que los problemas existentes en las herramientas de patrones, se deben a la falta de un *modelo* que represente los patrones de diseño adecuadamente, y que tenga una fuerte integración con el ambiente de desarrollo. Según definimos en nuestros objetivos (ver 1.1), el modelo debe soportar la instanciación, documentación, validación y manipulación del código de los patrones de diseño utilizados. En la sección 3 se han analizado los distintos enfoques existentes para modelar los patrones de diseño: construcciones del lenguaje de programación, componentes de diseño, lenguaje formal y Framework. En todos estos enfoques, se observó poca profundidad en la demostración de la aplicabilidad práctica de las ideas propuestas para automatizar el uso de patrones. En particular, no existe ninguna herramienta en Smalltalk que cumpla con los objetivos descriptos en 1.1 para al menos un patrón.

Los hechos nombrados anteriormente y las dificultades para modelar patrones de diseño discutidas en la sección 4, indican que crear un modelo de patrones que soporte la automatización de los mismos, es realmente un problema abierto y complejo. Como a priori no es claro que exista una solución que cumpla con los objetivos planteados, el trabajo se encaró de manera iterativa, definiendo un modelo de patrones inicial y poniéndolo a prueba para el patrón Decorator. Para implementar este modelo se siguió como estrategia crear iterativamente un Framework de Patrones en Smalltalk. A continuación se discutirán las razones por las cuales se adoptó esta estrategia de implementación.

5.1.1 Discusión de la estrategia de implementación

5.1.1.1 Necesidad de fuerte interacción con el entorno

Como se ha indicado, el modelo de patrones debe tener una fuerte integración con el entorno de desarrollo. Algunos ejemplos de requerimientos que el modelo necesita del entorno son los siguientes:

- **Conocimiento de las relaciones de herencia**. Dada una clase, se necesitan conocer todas sus subclases directas o indirectas. En el *“Patrón Decorator”* (3.1.1), una vez que el programador fija la clase que juega el rol de Decorator, todas sus subclases directas e indirectas jugarán también el rol de decorador. Se necesita entonces conocer todas las subclases de la clase Decorator para realizar determinadas validaciones sobre éstas.
- **Parseo de métodos**. El modelo debería reaccionar ante el agregado de un método que viole las restricciones sobre el código que define un determinado

patrón en uso. Para que esto se pueda hacer, el mecanismo de validación debe tener acceso directo al código mientras éste se modifica. Pero el código no debe ser visto como simple texto. En particular, para cada método se necesita su Abstract Syntax Tree (AST), ya que es necesario realizar validaciones sobre las colaboraciones existentes. Por ejemplo, el patrón Decorator define que cada método de un mensaje decorado debe forwardear el mensaje al objeto que juega el rol de Componente. Para poder realizar esta validación se necesita que el entorno permita acceder al AST de cada mensaje decorado [8]

- **Creación de nuevos métodos y clases**. Si el programador agrega un nuevo mensaje en el Componente de un patrón Decorator, el modelo podría, previo aviso al programador, agregar un método en la clase que juega el rol de Decorator que forwardeé el mensaje al Componente. O sea, el modelo debe tener la posibilidad de agregar nuevos métodos al sistema. También la instanciación de patrones de diseño puede requerir agregar nuevas clases en el entorno
- **Asociar información a elementos del sistema**. Como se discutió en 4.5, si el programador decide para un método particular de un Decorator no forwardear el mensaje al componente, el modelo debería registrar estas decisiones como desviaciones aceptadas con respecto a la forma ideal de usar un patrón. Esto implica entonces que el modelo de patrones sea capaz de mantener información adicional referida a ciertos elementos del programa. En este ejemplo, la información adicional estaría asociada a los métodos.

Una de las principales razones por las que se decidió implementar el modelo de patrones como un Framework, es que esta elección nos asegura que la interacción entre el modelo y el entorno de desarrollo cumpla con los requerimientos anteriores. Más adelante veremos otras razones por las cuales se tomó esta decisión, pero antes veremos por qué se eligió Smalltalk como lenguaje de implementación del Framework y entorno de trabajo.

5.1.1.2 Elección de Smalltalk como lenguaje de implementación

Las características de los requerimientos discutidos en el punto anterior, muestran que el modelo de patrones necesita acceder y manipular fuertemente el *metamodelo*. Por metamodelo, nos referimos a un modelo, que representa o modela el software en sí mismo. Desde su nacimiento, Smalltalk define este tipo de metamodelo. Por ejemplo, en Smalltalk las clases creadas por el programador, son a su vez instancias de otra clase llamada *Metaclass* que define el *comportamiento de las clases*. De esta forma, cada clase del sistema tiene un comportamiento rico y amplio que abarca mensajes para:

[8] En el *"Ejemplo de Patrón Decorator"* (3.1.2), `TranslatingWrapper` es una subclase del `Decorator Wrapper` y en su mensaje `displayOn:` forwardea correctamente el mensaje como lo establece el patrón:

```
TranslatingWrapper>>displayOn: aGraphicsContext
      aGraphicsContext translateBy: self translation.
      component displayOn: aGraphicsContext.
```

- Responder cuál es su superclase, cuáles son sus subclases, qué métodos tiene, etc.
- Compilar métodos y acceder su código fuente
- Crear instancias sí misma (la clase *Persona* debe crear instancias de *persona*). Para esto, una clase tiene que conocer cuáles serán las variables que deberán tener las instancias que crea

En síntesis, se observa que el modelo de patrones que se quiere construir, necesita una fuerte interacción con el entorno de desarrollo y su correspondiente metamodelo. Smalltalk provee un muy buen soporte del metamodelo, y es por ello, que se eligió como lenguaje para implementar el framework de patrones.

5.1.1.3 Accesibilidad para el programador

El usuario final de la herramienta construida sobre el modelo de patrones es el programador de aplicaciones. Se necesita entonces, que el modelo maneje conceptos conocidos para el programador. En la sección 3.5, vimos como el framework de Fragmentos, al apoyarse en un concepto nuevo y de escaso valor semántico como el Fragmento, termina siendo un framework de difícil comprensión y uso.

Las características del framework de patrones construido que lo hacen accesible para el programador son las siguientes:

- Todos los conceptos del framework (patrones, instancias de patrones, roles, mecanismos de validación, etc) están modelados dentro del mismo entorno de trabajo, usando el paradigma de objetos.
- Las propiedades lógicas que debe verificar una instancia de patrón definida por el programador, se modelan con fórmulas de lógica de primer orden, las cuales son familiares para el programador
- el código del framework es abierto y se testea automáticamente usando SUnit. Esto hace que el programador pueda ver el código, experimentar con él y tener ejemplos concretos del comportamiento del modelo

5.1.1.4 Feed-back de la aplicación práctica y construcción iterativa

Como se observó anteriormente, crear un modelo de patrones que soporte la automatización de los mismos, es un problema abierto y complejo. El soporte de la automatización que se busca en esta tesis y en los trabajos analizados, busca, en última instancia, hacer que para el programador sea más práctico y consistente usar los patrones de diseño. Por esto, es esencial que cualquier modelo propuesto tenga una instancia de validación de aplicabilidad práctica. En general, hemos observado en los trabajos discutidos que esta instancia de validación no se trata con la profundidad necesaria. Por ejemplo, Lepus (3.4) define un lenguaje para formalizar las microarquitecturas asociadas a los patrones, pero se plantean varios problemas al analizar la utilidad del formalismo (3.4.4.4)

Por la complejidad del problema que se encara y la necesidad de obtener un feed-back de la aplicación práctica, el framework de patrones se construyó siguiendo prácticas de XP como tests automatizados y refactoring. Estas prácticas son fundamentales para poder ir adaptando el modelo a medida que se va desarrollando, y también facilitarán futuras extensiones. Hubo 2 actividades principales durante el desarrollo del framework:

1) Desarrollo del modelo del framework.

Esta versión, que se expone más adelante, da soporte al *"Patrón Decorator"* (3.1.1) en su variante más utilizada que es la que está en el *"Caso de Estudio"* (3.1). Se eligieron este patrón y variante porque según se observó en 4.3 y 4.6, la descripción de la micro-arquitectura del patrón Decorator está dentro de las más precisas. Esto hace que el Decorator sea uno de los patrones más aptos para incluir en el framework y uno de los mejores candidatos para intentar automatizar su uso.

2) Desarrollo de la interacción básica entre Framework y el Entorno Desarrollo

La implementación entera de las funcionalidades discutidas en la sección 1.1, insumiría mucho tiempo de programación para contemplar todos los detalles de interacción entre el entorno y el framework. Las funcionalidades desarrolladas y el diseño correspondiente se describirán en la sección 5.7

5.1.1.5 Extensibilidad

Modelar los patrones de diseño es un problema complejo pero también es un problema extenso porque existen un gran número de patrones. Como se observó en el punto anterior, en la presente tesis el framework de patrones implementa el patrón del Caso de Estudio. En la sección 7 se analizará el impacto de incorporar otros patrones que tengan definiciones más precisas en el sentido visto en la sección 4.3. Se intentará entonces, obtener un modelo que soporte el agregado de más patrones de diseño (extensibilidad). Construir el modelo de patrones como un framework de objetos usando técnicas de testing automático, permite esta extensibilidad ya que el modelo se puede ir refinando y refactorizando al encarar la implementación de nuevos patrones de diseño

5.1.2 Observaciones sobre el alcance del modelo de patrones

Hemos visto en la sección 4.1, que un patrón define tanto un problema de diseño como una solución a dicho problema. Si bien la definición de un modelo que busque representar los problemas asociados a los patrones de diseño aparece como una idea interesante, la construcción de dicho modelo no se va a encarar en la presente tesis. Entonces, el modelo de patrones que se va a construir, representará de cada patrón su parte de *solución* (secciones Estructura, Participantes y Colaboraciones) pero no su parte de descripción de *problema* (secciones Intención, Motivación y Aplicabilidad).

5.2 Visión general

En esta sección se dará una visión de los conceptos y clases más importantes existentes en el Framework de Patrones. El Framework se apoya en dos conceptos centrales: las definiciones y las instancias de patrones. En la *"Figura 4 - Definición e Instancia de patrón"* se muestra como el framework modela el patrón del *"Caso de Estudio"*. A continuación, se introducirán las clases más importantes del framework y las relaciones entre las mismas, apoyándonos en el caso de estudio ilustrado en la figura:

- Los objetos `'aPattern'` y `'aPatternDefinition'` son instancias de las clases `Pattern` y `PatternDefinition` respectivamente.
- Cada instancia de `PatternDefinition` define el comportamiento y la estructura común que comparten todas las instancias de un determinado patrón. El objeto `'aPattern'` es una instancia de patrón definida por el programador que modela al *"Ejemplo de Patrón Decorator"* (3.1.2).
- Una de las principales responsabilidades de la clase `Pattern` es saberse validar. Para validar un pattern hay que enviarle el mensaje `#validate`. El pattern colabora entonces con su `patternDefinition`, y se termina evaluando el `Predicate` asociado a éste. Los predicates definen las propiedades lógicas que deben cumplir las instancias de patrones para considerarse válidas. El mecanismo de validación de patrones se describirá en detalle en la sección 5.5
- La clase `PatternDefinition` y sus colaboradores modelan la información incluida en las secciones Estructura, Participantes y Colaboraciones de la descripción de un patrón (ver sección 2.2.3). Los participantes que define un patrón se representan usando clases de la jerarquía de *roles*. En la figura se puede observar que `'aPatternDefinition'` tiene un mapa de roles en el cual se encuentran los participantes existentes en el patrón Decorator: `Component`, `Decorator` y `Decorators`.
- Cada `PatternDefinition` tiene la responsabilidad de verificar si sus instancias continúan siendo válidas a medida que se producen cambios en el código dentro del entorno de desarrollo. Para esto sabe responder al mensaje `#processRefactoryChange:change`, donde `'change'` representa un cambio de código. Para poder realizar esta tarea, cada `PatternDefinition` conoce todas las instancias de patrones de su "tipo" existentes dentro del entorno (variable `'patterns'`). El diseño de la interación entre el ambiente de desarrollo y el modelo de patrones se expondrá en la sección 5.7.2
- Una instancia de pattern conoce cuales son las clases del ambiente que "actúan" los roles existentes en su patternDefinition. Por esto, `'aPattern'` tiene un mapa llamado `'actors'` en el cual se asocia cada rol del patrón con el actor que juega dicho rol.

En la próxima sección se discutirá en detalle el *Modelo de Roles*, se analizarán los distintos tipos de actores para cada uno de los roles y las distintas formas en las que se puede calcular y/o definir el actor de un rol

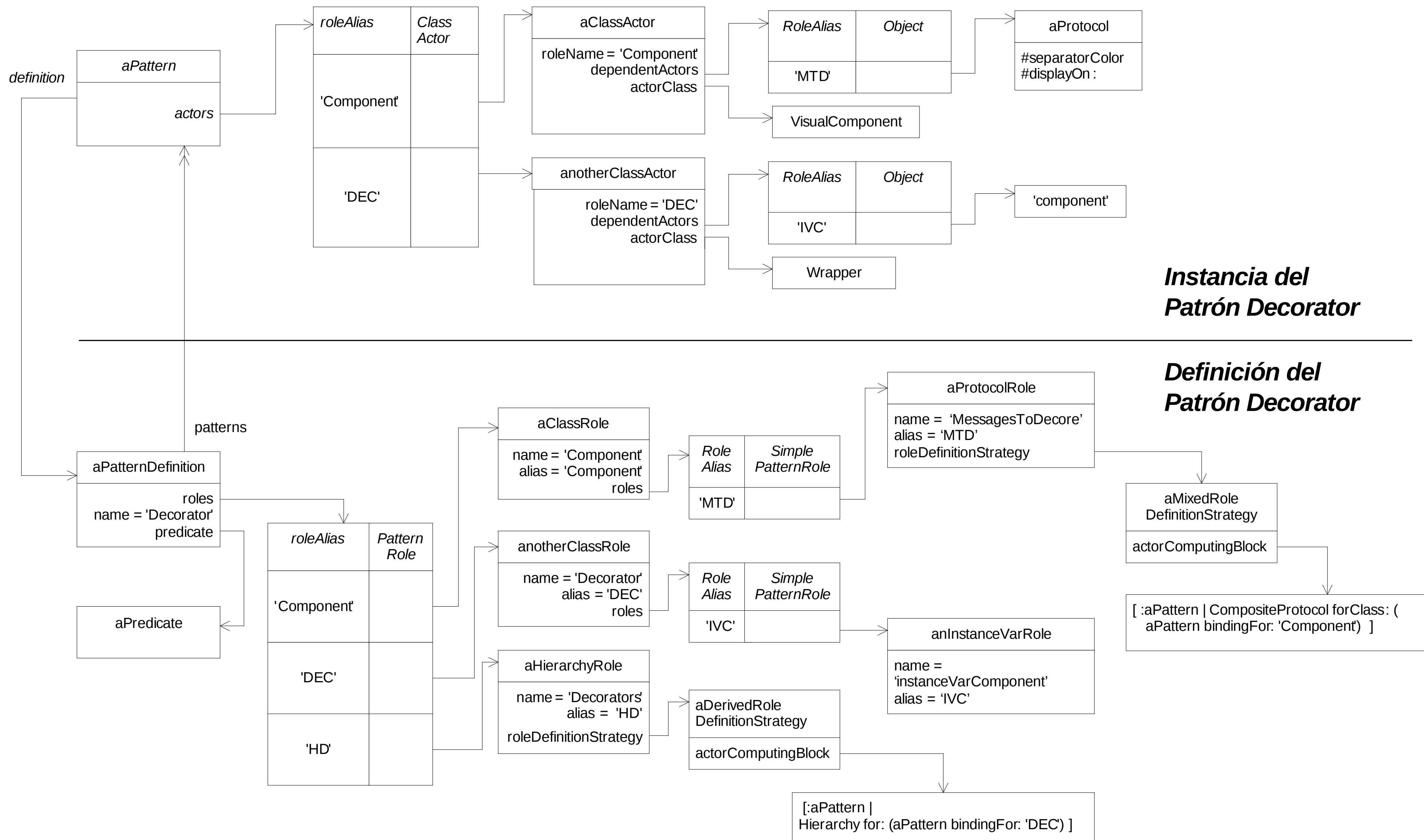


Figura 4 - Definición e Instancia de patrón

5.3 Modelo de roles

En esta sección se discutirá como el Framework, a través del modelo de roles, representa la información contenida en los apartados *Estructura* y *Participantes* de la descripción de un patrón (ver 2.2.3). Los aspectos dinámicos que se incluyen en el apartado *Colaboraciones* se representan usando los Predicados que serán descriptos en la próxima sección

Los participantes de un patrón se modelan asociando a la definición del patrón correspondiente, un conjunto de instancias de la jerarquía *PatternRole*. Por ejemplo, en la *"Figura 4 - Definición e Instancia de patrón"* se pueden observar los roles que tiene la definición del patrón Decorator (`'aPatternDefinition'`). Los elementos del sistema que cumplen o juegan un rol en una *instancia* de patrón se llaman actores. En la figura, la instancia de patrón `'aPattern'` tiene un mapa llamado `'actors'` en el cual se asocia cada rol del patrón con el actor que juega dicho rol.

Los distintos tipos de elementos del sistema que pueden tener un rol en un patrón de diseño son: las clases, las jerarquías, las variables de instancias y los mensajes. Para cada uno de estos tipos de elementos, existe una clase en la jerarquía *PatternRole* que se muestra en la siguiente figura:

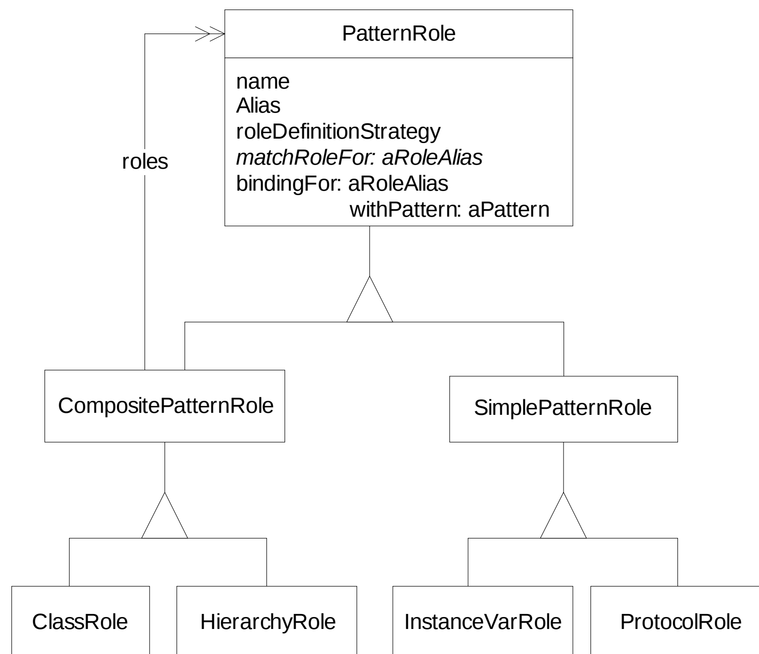


Figura 5 - Modelo de Roles

Como se observa, los roles pueden ser simples (*SimplePatternRole*) o compuestos (*CompositePatternRole*). Un rol compuesto se caracteriza por tener a su vez un conjunto de roles asociados. Por ejemplo, el rol 'Component' del patrón Decorator se modela con un ClassRole, que a su vez tiene relacionado un ProtocolRole para representar el conjunto de mensajes que se decoran. Para referenciar a un rol que está "anidado" dentro de otro, se califica el nombre del rol con el nombre del rol que lo

contiene. Por ejemplo, el rol *Component* contiene un ProtocolRole con alias *MTD* (MessagesToDecore) que se referencia como *Component.MTD*. Esta notación se utilizará en la sección 5.4.1 para construir el predicado del patrón Decorator.

En la siguiente tabla se muestran los principales mensajes que puede responder un *PatternRole*:

Mensaje	Tipo	Descripción
`Name`	`String`	El *nombre* de un rol sirve para identificar unívocamente al rol dentro de un patrón. Los roles existentes en el patrón Decorator tienen como nombres: `Component, Decorator y Decorators`.
`Alias`	`String`	Es un nombre corto que se utiliza para referirse al rol, generalmente dentro de los predicados. Los alias de los roles en el patrón Decorator son: `Component, Component.MTD, DEC, DEC.IVC y HD` (por jerarquía de decoradores)
`validate: aPattern`	`--`	Valida si el rol receptor está correctamente definido sobre el patrón `'aPattern'`
`roleDefinitionStrategy`	`Role Definition Strategy`	Cada rol tiene asociada una estrategia de definición que determina la forma en la que se calcula y/o especifica el actor del rol.
`matchRoleFor: aRoleAlias`	`PatternRole`	Devuelve el rol que matchea a '`aRoleAlias`' si el rol receptor o alguno de sus roles anidados matchea con este nombre. Devuele nil en caso contrario
`bindingFor: aRoleAlias withPattern: aPattern`	`Object`	Devuelve el actor para `'aRoleAlias'` en el patón `'aPattern'` Tiene como precondición que el rol receptor devuelva true a `matchRoleFor: aRoleAlias`
`bindingFor: aPattern`	`Object`	Devuelve el actor del rol receptor en el patrón `'aPattern'`. La resolución de este mensaje se delega en la estrategia de definición como se verá en la sección 5.3.2

La descripción del modelo de roles continuará analizando los actores correspondientes a los distintos tipos de roles existentes. Luego, se discutirán las posibles formas en las que se puede definir un rol en un patrón. Finalmente, en la sección 5.3.3 se detallará como el modelo de roles se aplica en patrón del *"Caso de Estudio"* (3.1)

5.3.1 Actores y Roles

Hemos visto que un actor es un elemento del sistema que cumple un rol en una instancia de patrón. Los distintos tipos de roles son actuados por diferentes tipos de actores. Esta relación entre roles y actores se detalla a continuación para cada una de las clases de la jerarquía `PatternRole`:

- ***ClassRole***: como se puede observar en la *"Figura 4 - Definición e Instancia de patrón"*, los actores de un `ClassRole` no se representan directamente con las clases del sistema, sino que se representan con instancias de `ClassActor`. Esto se modeló de esta forma para que un classActor pueda mantener la información sobre los actores de los roles anidados. Por ejemplo, el actor correspondiente al rol `'DEC'` (alias para el rol `'Decorator'`) es el objeto `'anotherClassActor'`. En la figura se puede ver que este objeto:
  - referencia a la clase `Wrapper` a través de la variable `'actorClass'`. La clase `Wrapper` pertenece al ambiente de Visual Works y juega el rol `Decorator` en el patrón de ejemplo. Por simplicidad, aunque en ocasiones es necesario tener en cuenta la existencia de los classActor's, diremos que la clase del sistema es el actor del rol (para el ejemplo diríamos que la clase `Wrapper` es el actor del rol `Decorator`)
  - tiene en su mapa `'dependentActors'` el nombre de la variable de instancia que referencia al Component (`'component'`). La variable `'component'` es el actor del rol `'IVC'` (alias para el rol `instanceVarComponent`).
- ***HierarchyRole***: este rol lo actúan objetos de tipo Hierarchy. La clase Hierarchy está definida en el Framework y representa una jerarquía de clases. Para construir una jerarquía que tenga como raíz a la clase `'aClass'` hay que usar el mensaje de clase `'Hierarchy>>for: aClass'` (para más detalles consultar la sección 5.8.3). Frecuentemente, la descripción de un patrón de diseño define propiedades que deben cumplir todas las clases que pertenecen a una jerarquía. Por ejemplo, en el *"Predicado del Patrón Decorator"* (5.4.1) se define que todas las implementaciones de mensajes decorados existentes en la *jerarquía de Decorators* deben forwardear el mensaje al componente. Para modelar este tipo de propiedades, resulta conveniente representar a las jerarquías como objetos a los cuales se les puede aplicar predicados. Esto se logra con la clase Hierarchy.
- ***InstanceVarRole***: el actor de un rol de variable de instancia es simplemente el string del nombre de la variable. En el ejemplo, el actor del rol `'IVC'` es `'component'` ya que el `Wrapper` referencia al `VisualComponent` a través de su variable de instancia `'component'`
- ***ProtocolRole***: los roles de tipo protocolo son actuados por instancias de Protocol. La clase Protocol (5.8.4) permite tratar de manera uniforme a los mensajes individuales y a los conjuntos de mensajes. En el ejemplo el actor del protocolRole *Component.MTD* es el objeto `'aProtocol'` que tiene los mensajes `#separatorColor` y `#displayOn:`

5.3.2 Estrategias de Definición de Roles

Como se observó anteriormente, cada rol tiene asociada una *estrategia de definición* que determina la forma en la que se calcula o se debe especificar el actor del rol. Los roles delegan en esta estrategia el mensaje `#bindingFor: aPattern`, el cual devuelve el actor del rol en el patrón `'aPattern'`. En la sección 5.4.3.1, se verá que las instancias de patrones implementan un algoritmo para hacer look-up de actores. En dicho algoritmo también participan las definiciones de patrones y los roles.

Cada estrategia de definición está modelada como una clase de la jerarquía *RoleDefinitionStrategy* que se muestra a continuación:

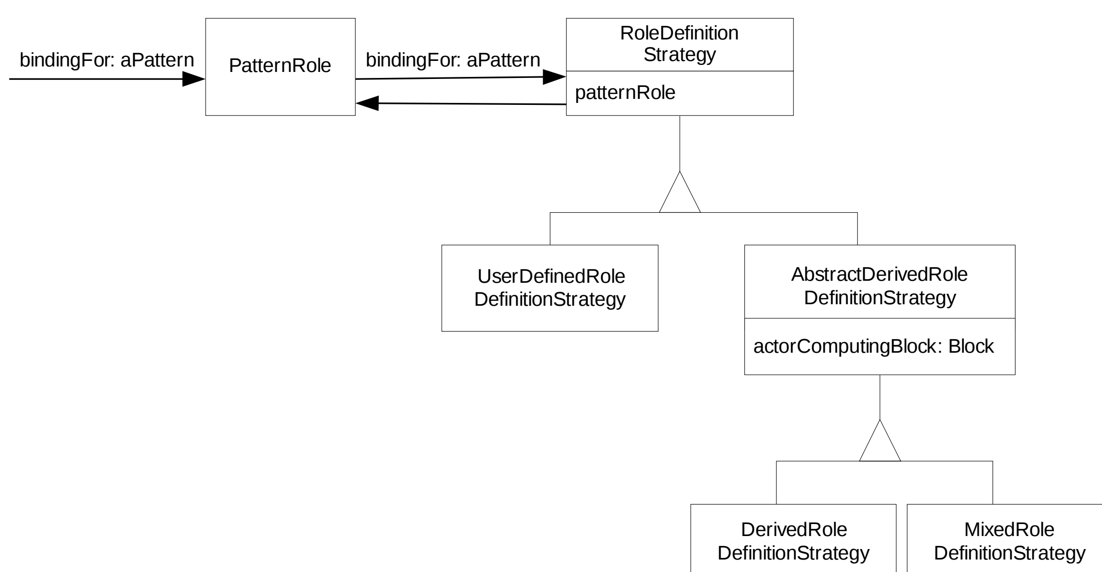


Figura 6 - Estrategia de Definición de Roles

La lógica de las distintas implementaciones de `#bindingFor: aPattern` es la siguiente:

- ***UserDefinedRoleDefinitionStrategy***: cuando en un patrón se define un rol con esta estrategia, el programador que instancia el patrón debe, necesariamente, especificar un actor para el rol. Esta estrategia busca el actor correspondiente para el rol dentro de `aPattern`. Si no existe, tira una excepcion
- ***DerivedRoleDefinitionStrategy***: esta estrategia determina que el actor del rol ***no*** puede ser especificado por el programador que instancia el patrón, sino que siempre es calculado (o derivado) por el Framework. Cuando esta estrategia recibe el mensaje `#bindingFor: aPattern` devuelve el resultado de evaluar su `'actorComputingBlock'` con `'aPattern'` como argumento.
- ***MixedRoleDefinitionStrategy***: esta estrategia es mixta en el sentido que permite que el actor lo especifique el usuario, y si esto no ocurre, calcula el actor como `DerivedRoleDefinitionStrategy`. Entonces, busca el actor correspondiente en

el patrón y si no existe lo calcula. La estrategia mixta, permite contar con la simplicidad de los roles derivados y con la flexibilidad de los roles definidos explícitamente por el programador (ver sección 3.2.1).

5.3.3 Roles en el patrón Decorator

En esta sección se detallarán los roles existentes en el patrón Decorator que se ilustran en la *"Figura 4 - Definición e Instancia de patrón"*. En esta figura, por simplicidad no se muestra la estrategia de definición de los roles

5.3.3.1 Component

La clase que actúa el rol Component debe ser definida por el usuario ya que la estrategia del rol es de tipo `UserDefinedRoleDefinitionStrategy`. Dentro del rol Component se define el `ProtocolRole` MessageToDecore *(* alias `'MTD'`*)* que se detalla a continuación.

5.3.3.2 Mensajes a decorar del Componente

El actor del rol `'Component.MTD'` determina, en una instancia de patrón, el conjunto de mensajes del Component que el Decorator debe decorar. Este rol tiene una estrategia de definición de tipo `MixedRoleDefinitionStrategy`. Como se mencionó anteriormente, esta estrategia permite que el actor del rol pueda obtenerse de dos formas:

- *el actor es especificado por el usuario*. Este caso se ejemplifica en la *"Figura 4 - Definición e Instancia de patrón"*, donde como actor del rol '`Component.MTD`' se definió un objeto `Protocol` que agrupa los mensajes `#separatorColor`, `#displayOn:`. Como se puede observar, el uso de una estrategia mixta permite que no se tengan que decorar necesariamente todos los mensajes del Component
- *el actor es calculado por la estrategia*. En el caso de que el usuario no defina explícitamente el conjunto de mensajes que desea decorar, la estrategia calculará el actor default usando el siguiente bloque:

```
[ :aPatternInstance |
     CompositeProtocol forClass: (
     aPatternInstance bindingFor:
     'Component')  ]
```

- El bloque anterior, al evaluarse sobre una instancia de patrón, devuelve un objeto Protocol que contiene todos los mensajes definidos en la clase que actúa el rol Component.

5.3.3.3 Decorator

La clase que actúa el rol Decorator debe ser definida por el usuario ya que la estrategia del rol es de tipo `UserDefinedRoleDefinitionStrategy`. En la variante del patrón Decorator que estamos usando como *"Caso de Estudio"* (3.1), se define que el participante Decorator *"mantiene una referencia a un objeto Component…"*. Esto se modela definiendo dentro del rol Decorator con un rol de tipo `InstanceVarRole` con nombre `'InstanceVarComponent'` *(*alias `'IVC'`*)*. En la *"Figura 4 - Definición e Instancia de patrón"*, el rol Decorator es el objeto `'anothetClassRole'`

5.3.3.4 Variable de instancia para referenciar al Component

El actor del rol `'InstanceVarComponent'` es un String con el nombre de la variable de instancia a través de la cual el Decorator referencia al Component . Este rol también debe ser definido por el usuario ya que su estrategia es de tipo `UserDefinedRoleDefinitionStrategy`

5.3.3.5 Decorators

El rol Decorators es un HierarchyRole con una estrategia de tipo `DerivedRoleDefinitionStrategy`. Esto implica, que el actor del rol no podrá ser definido por el usuario, sino que se calculará evaluando el bloque asociado a la estrategia:

```
[ :aPatternInstance |
Hierarchy for:(aPatternInstance bindingFor: 'DEC')]
```

Este bloque, dada una instancia de patrón `'aPatternInstance'`, devuelve un objeto de tipo `Hierarchy` que representa la jerarquía que extiende de la clase que actúa el rol `'DEC'`. En el *"Ejemplo de Patrón Decorator"* (3.1.2), el binding para el rol `'DEC'` es la clase `Wrapper` y entonces el bloque anterior tiene como resultado una jerarquía sobre `Wrapper`.

5.4 *Predicados y Expresiones*

En la sección anterior se ha visto que, cuando el programador instancia un patrón de Diseño, debe asignar a éste los actores correspondientes a los roles especificados en la definición del patrón. Estos actores tendrán que verificar ciertas propiedades lógicas para que la instancia de patrón pueda considerarse válida. Dichas propiedades son las que expresan informalmente como *reglas de roles* y *de colaboración* en [Hedin97] y como una fórmula LePUS en [Eden00]. En esta sección se verá cómo el Framework, a través del uso de Predicados y Expresiones, modela las restricciones que tienen que cumplir los actores de un patrón. El material de esta sección se organizará de la siguiente manera:

- En 5.4.1 se introducirán los Predicados y Expresiones, mostrando cómo se modela el predicado del patrón del *"Caso de Estudio"* (3.1)
- En 5.4.2 se dará una visión general de los distintos tipos de Predicados y Expresiones y los contextos que éstos necesitan para evaluarse.
- En 5.4.3 se discutirá el funcionamiento de los patrones como contexto de evaluación. Los Predicados y Expresiones necesitan, para poder evaluarse, obtener del contexto los actores que juegan determinados roles. Se verá entonces en esta sección, como un patrón realiza la búsqueda de actores
- Finalmente, en las secciones 5.4.4 y 5.4.5, se verá como los predicados y expresiones se evalúan individualmente

Una vez que se hayan analizado todos estos temas, se explicará en la próxima sección (5.5) cómo se realiza la validación de patrones

5.4.1 Predicado del Patrón Decorator

5.4.1.1 Fórmula lógica del patrón de Estudio

Las propiedades que deben cumplir los actores del patrón del *"Caso de Estudio"* se pueden representar con la siguiente fórmula:

```
[form-Decorator] ≡ ∀ m ∈ Component.MTD: MessageImplemented(m, HD) ∧
                   SendMessage(MessageImplementations(m, HD), DEC.IVC, m )
```

La fórmula anterior no pretende tener una interpretación formal; es una notación de carácter semi-formal, cuyo significado será detallado en breve, que sirve para:

- definir las restricciones que el Framework debe verificar sobre los actores de un patrón.
- servir como guía para lograr que la API de predicados del Framework sea lo más declarativa posible y maneje conceptos conocidos para el programador (ver sección 5.1.1.3).

Significado de la fórmula `[form-Decorator]`

- En la fórmula `[form-Decorator]` `'Component'`, `'Component.MTD'`, `'DEC'`, `'DEC.IVC'` y `'HD'` son los *alias* de los roles que se definieron en la sección *"Roles en el patrón Decorator"* (5.3.3)
- La fórmula establece que una instancia de patrón Decorator es válida si y solo si, para todo mensaje `'m'` incluido en el conjunto de mensajes a decorar (`'Component.MTD'`) se cumplen las siguientes propiedades:

    - **`MessageImplemented (m, HD)`**: este predicado es verdadero si en todas las clases de la jerarquía que actúa el rol `'HD'`, existe una implementación para el mensaje `'m'`. Obviamente, en la evaluación de este predicado se considera el hecho de que una clase herede la implementación de ciertos mensajes

    - **`SendMessage(MessageImplementations(m,HD),DEC.IVC,m)`**: expresa que todas las implementaciones existentes del mensaje `'m'` en la jerarquía de Decorators `'HD'` deben forwardear el mensaje `'m'` a la variable de instancia `'DEC.IVC'`. Observar que:

        - la expresión **`MessageImplementations(m,HD)`** devuelve un conjunto que contiene todos los métodos correspondientes a implementaciones del mensaje `'m'` en la jerarquía `'HD'` .

        - La relación **`SendMessage(method,instanceVarName,message)`** es verdadera si y solo si dentro del método `'method'` se le envía a la variable de instancia `'instanceVarName'` el mensaje `'message'`.

A continuación se verá como el Framework modela la formula `[form-Decorator ]`

5.4.1.2 Representación de fórmulas en el Framework

En la *"Figura 4 - Definición e Instancia de patrón"* se muestra que `'aPatternDefinition'` tiene como predicado `'aPredicate'`, pero por simplicidad, no se detallan los objetos que conforman dicho predicado. Los objetos que modelan la fórmula del patrón del *"Caso de Estudio"* (3.1) discutida en la sección anterior, se observan en el siguiente diagrama:

EMBED Visio.Drawing.11

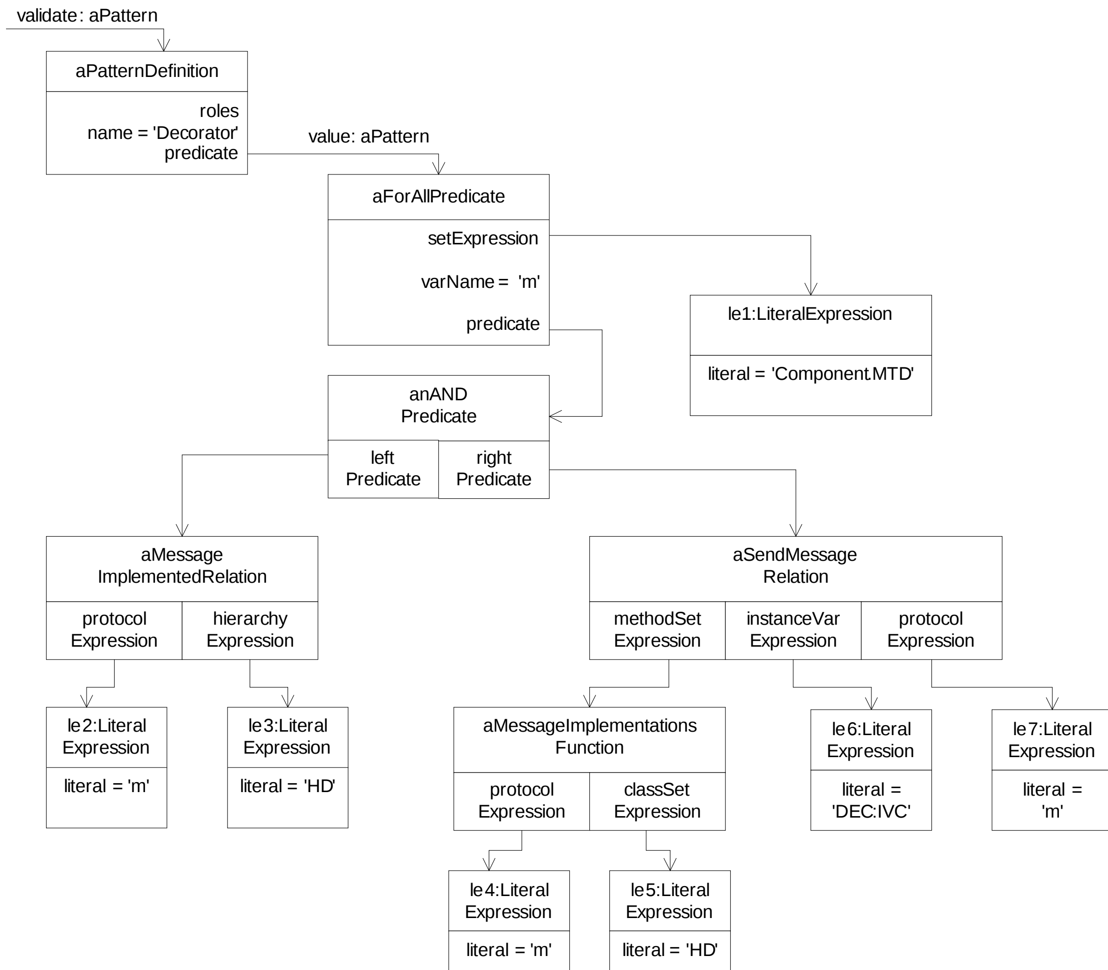


Figura SEQ Figura * ARABIC 7 - Diagrama de Instancias del Predicate Decorator

<u>Observaciones</u>

- La fórmula que tienen que cumplir los actores de un patrón, se representa con un árbol de predicados y expresiones. En el diagrama anterior, la raíz del árbol es un predicado de tipo `ForAllPredicate`.

- Las hojas del árbol siempre son expresiones del tipo `LiteralExpression`. El literal de una `LiteralExpression` es un string que puede tener dos tipos de valores :
  - el alias de un rol existente en el patrón (en el ejemplo `'Component'`, `'Component.MTD'`, etc)
  - el nombre de una variable cuantificada (en el ejemplo `'m'`)
- Los actores del patrón se referencian, desde las hojas del árbol, utilizando los alias de los roles
- Si bien el mecanismo de validación de patrones se va a discutir en la sección 5.5, se puede adelantar que cuando `'aPatternDefinition'` recibe el mensaje `#validate: aPattern` una de las cosas que hace es decirle a su predicate que se evalúe sobre el patrón con el mensaje `#value: aPattern`. Este mensaje a su vez se propaga recursivamente por los predicados del árbol para que todos se evalúen sobre el patrón.
- Los distintos componentes de la formula `[form-Decorator]`

```
∀ m ∈ Component.MTD: MessageImplemented(m, HD) ∧
SendMessage(MessageImplementations(m, HD), DEC.IVC, m )
```

se representan con las clases del framework que se indican en la siguiente tabla:

Componente de la formula	Clase del framework
Cuantificador ∀	ForAllPredicate
MessageImplemented(m,HD)	MessageImplementedRelation
MessageImplementations(m,HD)	MessageImplementationsFuntion
SendMessage(method,instanceVarName,message)	SendMessageRelation

Se puede observar que los distintos componentes de la fórmula están representados por clases específicas del Framework. Los detalles de cómo se evalúan los distintos predicados del Framework se encuentran en la sección 5.8.1

A continuación se verán los distintos tipos de Predicados y Expresiones que existen en el Framework y el contexto en el cual éstos se evalúan

5.4.2 Objetos Evaluables y Contextos de Evaluación

Los Predicados y Expresiones se implementan como clases en Smalltalk pertenecientes a la jerarquía de Evaluables. Los evaluables, como su nombre lo sugiere, tienen como principal responsabilidad saber evaluarse a sí mismos. Para esto, responden al mensaje `#value: anEvaluationContext` devolviendo el resultado de su evaluación. `EvaluationContext` es un *tipo*[9] que define un conjunto de mensajes para asociar nombres con objetos. En el siguiente diagrama se muestran los distintos tipos de Evaluables y contextos de evaluación existentes:

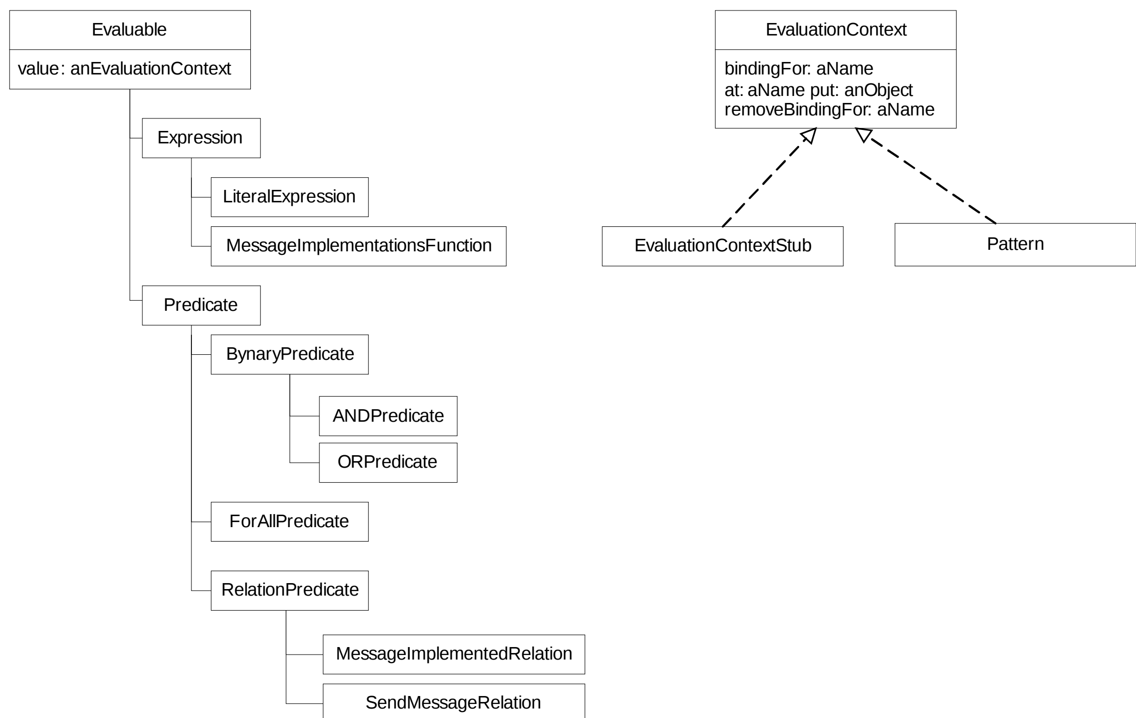


Figura SEQ Figura * ARABIC 8 - Evaluables y Contextos de Evaluación

`EvaluationContextStub` es una implementación de `EvaluationContext` que se utiliza para realizar test unitarios de los predicados. La implementación más importante de `EvaluationContext` la realiza la clase `Pattern`, y éste, será el tema que se tratará en la próxima sección..

[9] Estamos usando el concepto de tipo con la definición indicada en la sección 2.1.4

5.4.3 Los patrones como Contextos de Evaluación

Como se observó en la “”, los patrones implementan los mensajes definidos por `EvaluationContext`. Cuando los predicados y expresiones se evalúan sobre un patrón, deben resolver distintos nombres de roles que se los piden al contexto usando el mensaje `#bindingFor: aName`. Al recibir este mensaje, un pattern busca el actor asociado a `'aName'` con el algoritmo que se muestra y explica a continuación:

```
Pattern>>bindingFor: aName
| answer |

answer := self temporaryBindings at: aName ifAbsent: [].
answer isNil ifFalse: [ ^ answer ].


^ self definition bindingFor: aName withPattern: self .
```

En el método anterior, se puede observar que la búsqueda de un actor se realiza en el siguiente orden:

1. primero se busca en el mapa de bindings temporarios. El mapa `temporaryBindings` se usa durante la evaluación de los predicados de tipo `ForAllPredicate` para ligar el nombre de la variable cuantificada a los distintos valores que ésta puede tomar. En el caso de estudio, donde la fórmula es `∀ m ∈ Component.MTD: P(m)`, se liga sucesivamente la variable `'m'` a cada uno de los mensajes existentes en `Component.MTD`. Este mecanismo será visto en más detalle en la sección 5.5. Los mensajes de la clase `Pattern` para definir y eliminar bindings son `#at:varName put:anObject` y `#removeBindingFor: aName`

2. Si no se encuentra ningún binding dentro de `temporaryBindings`, la búsqueda se delega en la definición del patrón a través del mensaje: `PatternDefinition>>bindingFor: aRoleAlias withPattern: aPattern`.

3. La definición del patrón busca el rol cuyo alias matcheé con `aRoleAlias` y delega el *cálculo* del actor en dicho rol usando el mensaje `PatternRole>>bindingFor: aRoleAlias withPattern: aPattern`. Observar que, si la búsqueda de un actor llega a un rol, el actor se calculará con la estrategia de definición del rol.

A continuación se verán algunos ejemplos de este algoritmo de búsqueda de actores aplicados al caso de estudio.

5.4.3.1 Ejemplos de Resolución de Búsqueda de Actores

El siguiente diagrama es una versión simplificada de la *"Figura 4 - Definición e Instancia de patrón"* que se utilizará para analizar como un patrón resuelve la búsqueda de actores:

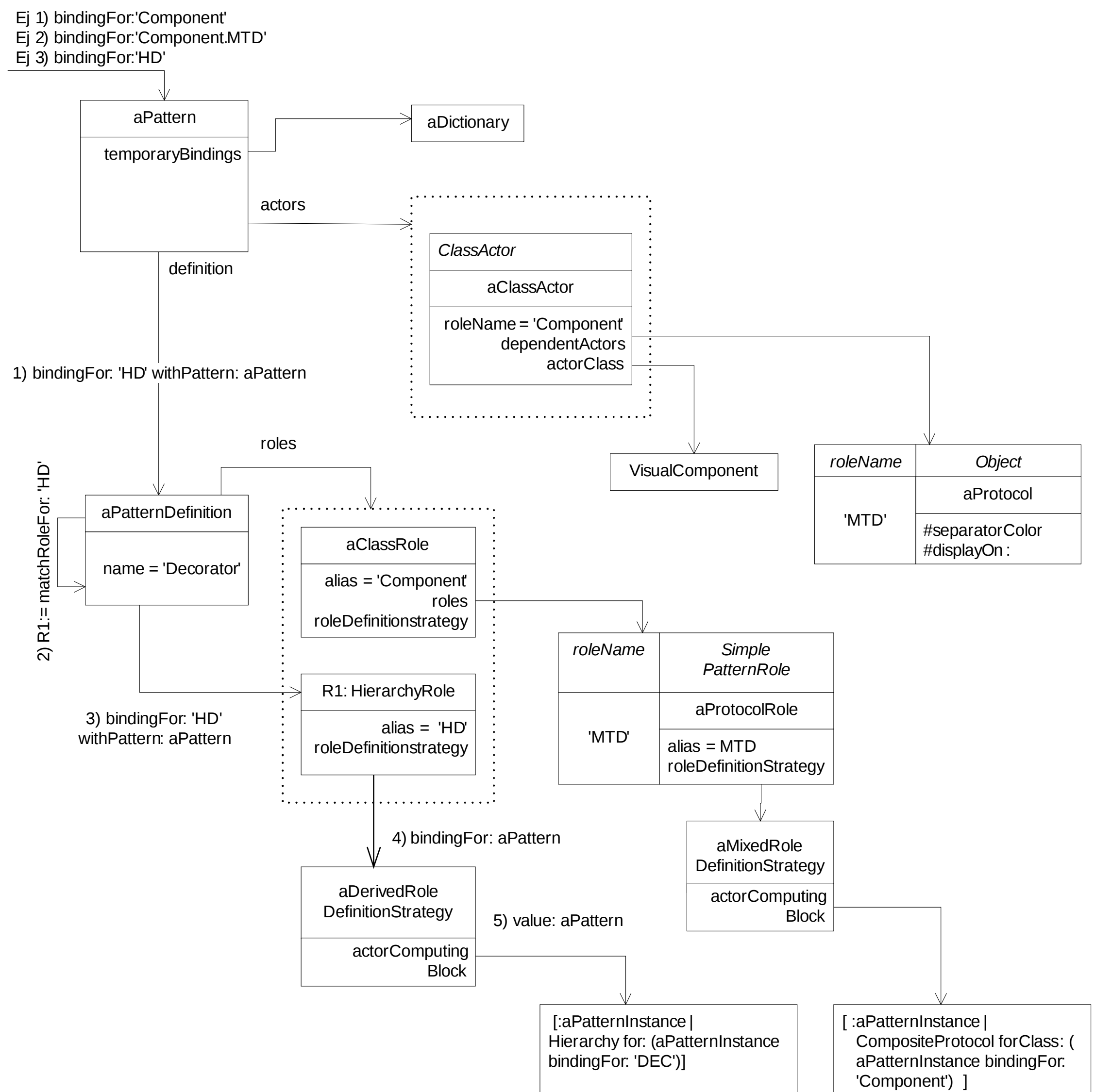


Figura 7 - Ejemplos de Búsquedas de Actores

A continuación se describe cómo `'aPattern'` resuelve el mensaje `#bindingFor:aName` para los tres ejemplos mostrados en la figura:

Ejemplo	Resolución
`bindingFor:'Component'`	La estrategia de este rol es del tipo `UserDefined.`. Entonces, esta estrategia busca el actor correspondiente en la colección de actores devolviendo la clase `VisualComponent`
`bindingFor:'Component.MTD'`	La estrategia para el rol `'Component.MTD' es aMixedRoleDefinitionStrategy.` Como el usuario definió un conjunto de mensajes a decorar específico, el actor es `'aProtocol' con los mensajes` (`#separatorColor y #displayOn:`). Si el programador cuando instancia el patrón, no definiera un actor para `'Component.MTD'`, este actor se calcularía evaluando el bloque de la estrategia
`bindingFor:'HD'`	Al resolver este mensaje, se desencadena la secuencia de colaboraciones que se muestra en los pasos 1-5 del diagrama El actor que se devuelve es el resultado de evaluar el bloque de la estrategia `'aDerivedRoleDefinitionStrategy':` `[:aPatternInstance | Hierarchy for: (aPatternInstance bindingFor: 'DEC')]` Como el binding para `'DEC'` es la clase `Wrapper`, el resultado de evaluar el bloque anterior es un objeto de tipo `Hierarchy` sobre la clase `Wrapper`.

Habiendo discutido como los patrones sirven como contexto de evaluación, se describirá a continuación como se evalúan las expresiones y predicados.

5.4.4 Evaluación de Expresiones

Las expresiones devuelven algún objeto o colección dependiendo del tipo de expresión que se trate. La expresión más sencilla es `LiteralExpression` que al evaluarse simplemente devuelve un objeto del contexto con un determinado nombre:

```
LiteralExpression>>value: anEvaluationContext
      ^ anEvaluationContext bindingFor: self literal
```

Como se observó en 5.4.1.2, las expresiones de tipo `LiteralExpression` siempre se encuentran en las hojas del árbol de Evaluables. Por estar en esta ubicación particular, deben detener la recursión de la evaluación y para ello devuelven un objeto del contexto.

La expresión `MessageImplementationsFunction` que aparece en la “ *EMBED Visio.Drawing.11* ” se reproduce a continuación:

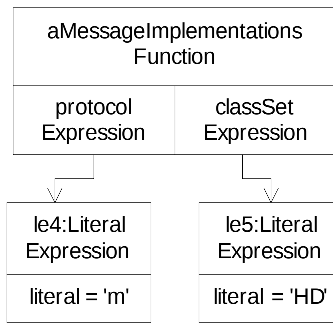


Figura 8 - MessageImplementationsFunction

Cuando la expresión anterior se evalúa, devuelve un conjunto de métodos en el cual existe un objeto de tipo Method por cada implementación particular del mensaje `'m'` en la jerarquía de decoradores `'HD'`. La clase Method (5.8.2) es una clase definida en el framework y representa precisamente un método particular.

5.4.5 Evaluación de Predicados

Cuando un `Predicate` recibe el mensaje `#value:anEvaluationContext` devuelve alguna subclase de `EvaluationResult`. En el siguiente diagrama se puede observar que existe una subclase de `EvaluationResult` por cada clase de `Predicate` (ver los predicates existentes en *"Objetos Evaluables y Contextos de Evaluación"* (5.4.2). Además, encerrados entre rectángulos, se muestran las clases específicas asociadas a cada `EvaluationResult`, cuyo funcionamiento se describirá en breve.

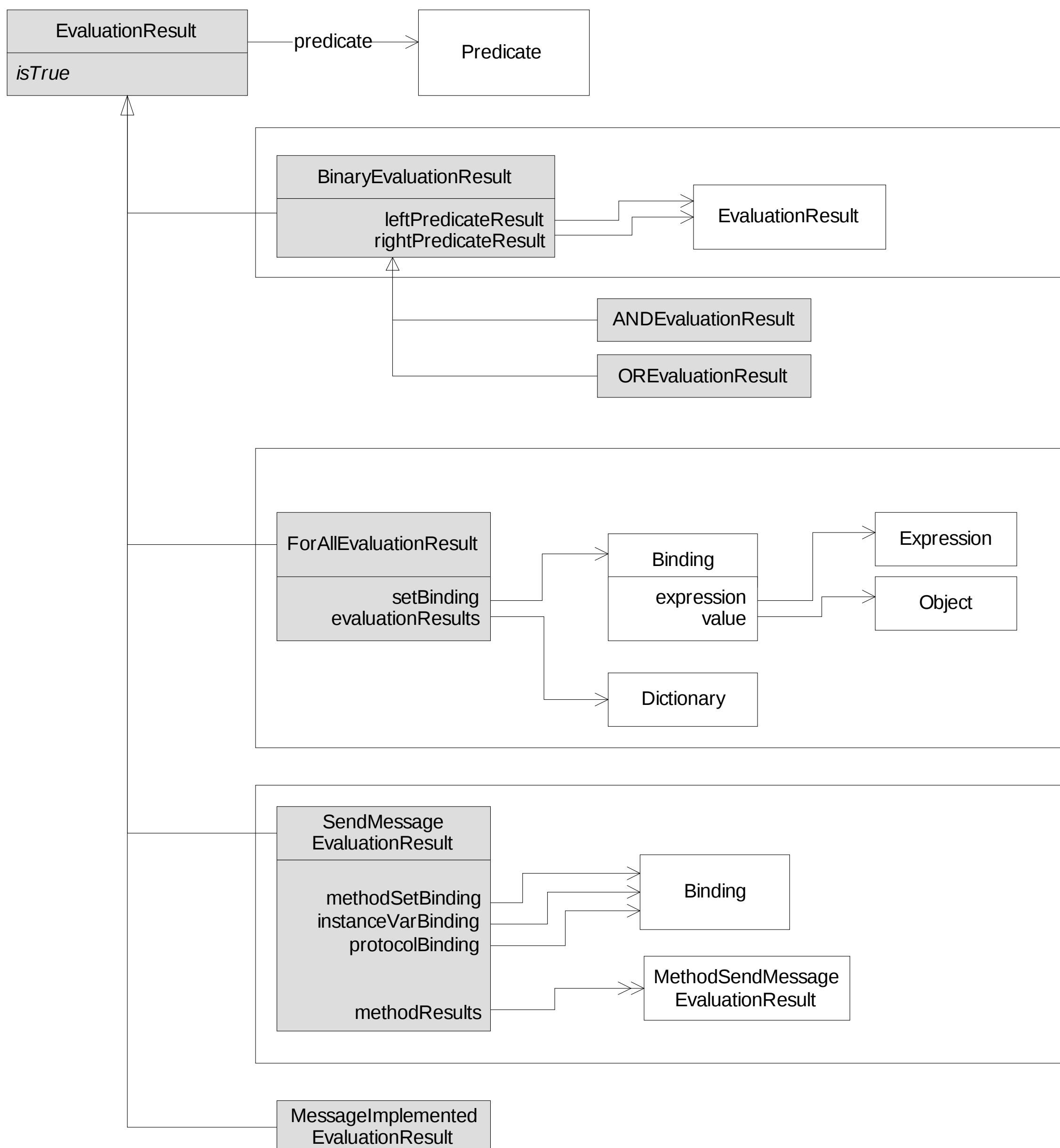


Figura 9 - Jerarquía de EvaluationResult

Una instancia de `EvaluationResult` conoce al predicado que se evaluó, y sabe responder si la evaluación fue verdadera o no con el mensaje `isTrue` . El mensaje

`#value:anEvaluationContext` devuelve un `EvaluationResult`, y no directamente un boolean, para poder describir en detalle los errores que se producen durante la evaluación. Para esto, cada `EvaluationResult` conoce, entre otra información, los bindings específicos de la evaluación. Un `Binding` es una tupla `<expression, value>` que asocia una expresión con el objeto que se obtiene al evaluar dicha expresión. Por ejemplo, veamos la relación entre un SendMessageRelation (5.8.1.1) y el resultado de su evaluación:

- Una instancia de `SendMesaggeRelation` se contruye de la siguiente forma:

```
SendMessageRelation class>>for:  aMethodSetExpression
                             with: anInstanceVarExpression
                             with: aProtocolExpression
```

- Cuando un `sendMessageRelation` se evalúa, el resultado de evaluar cada una de sus expresiones queda guardado en las variables `methodSetBinding`, `instanceVarBinding y protocolBinding`.
- Para cada método analizado, un objeto de tipo `MethodSendMessageEvaluationResult` describe si todos los mensajes del protocolo se envían a la variable de instancia

Cuando un predicado `P` se evalúa, el `EvaluationResult` que se obtiene sigue una estructura simétrica con respecto al árbol de evaluables que cuelga de `P`. Esta representación de los resultados permite conocer con todo detalle las fallas en la evaluación de predicados complejos. De esta forma, un programador puede saber todos los elementos del sistema que violan las restricciones asociadas al uso de un patrón.

Para detalles adicionales sobre la evaluación de predicados se puede consultar la sección 5.8.1

En esta sección, que ha llegado a su fin, se vio como el Framework usando predicados y expresiones, modela las propiedades lógicas que deben cumplir los patrones de diseño. Se ha descrito como los distintos tipos de Evaluables se evalúan dentro de un contexto, y cómo los patrones juegan este rol de contexto. Todos estos temas son los bloques básicos que utiliza el Framework para realizar la validación de patrones, que será el tema que se discutirá en la próxima sección

5.5 Validación de patrones

En esta sección se verán los distintos objetos del Framework que colaboran para verificar la validez de los patrones instanciados por el programador. A continuación, se muestran[10] y comentan las colaboraciones que se producen cuando el patrón del Caso de Estudio recibe el mensaje `#validate`:

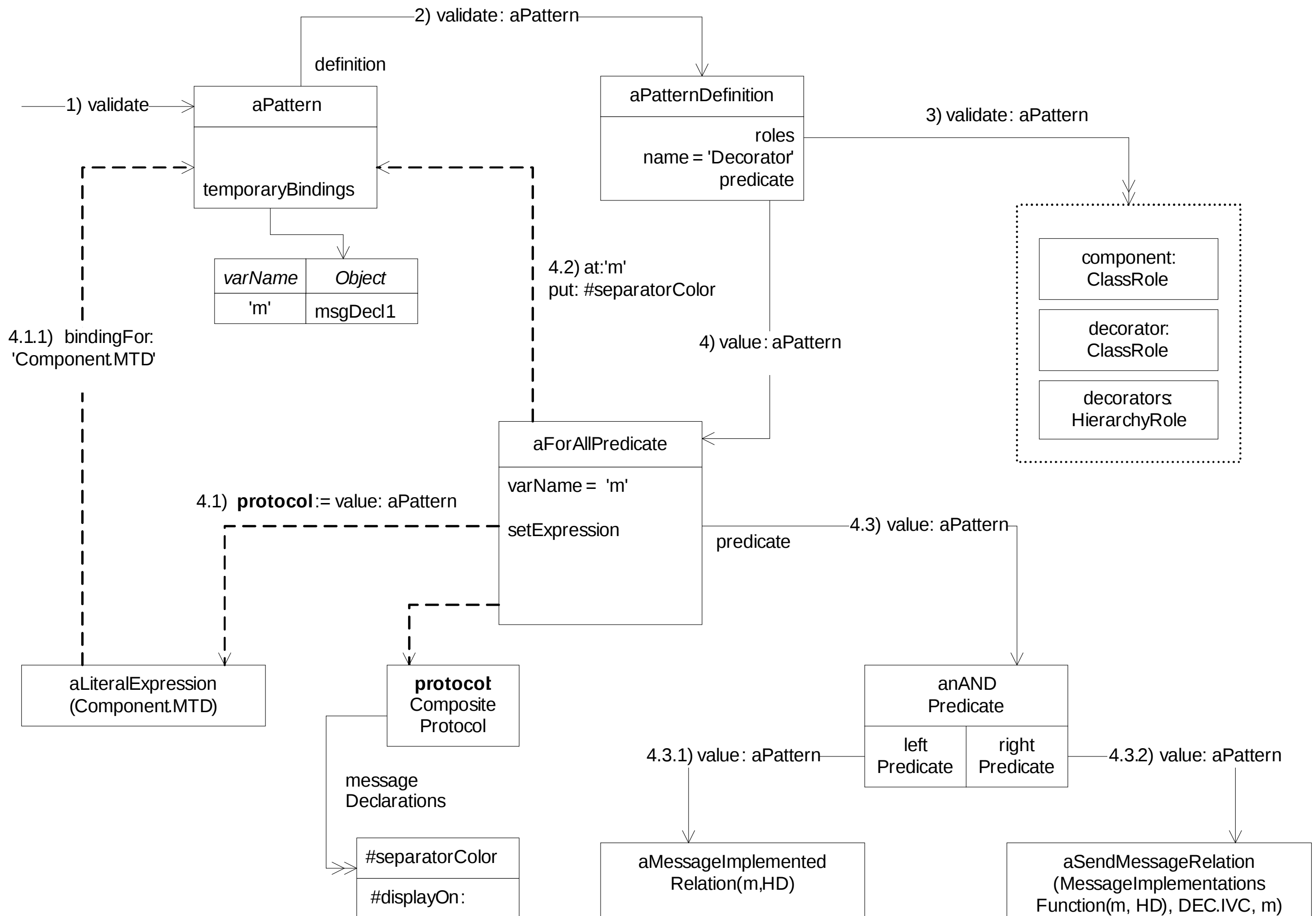


Figura 10 - Validación de Patrones

<u>Descripción de las colaboraciones</u>

1, 2) Al patrón que se quiere validar, en la figura `'aPattern'`, se le envía el mensaje `#validate`. Cuando un patrón recibe este mensaje, lo delega en su `PatternDefinition` asociada con el mensaje `#validate: aPattern`

3) `'aPatternDefinition'` les pide todos los roles existentes que se validen forwardeándoles el mensaje `#validate: aPattern`. Si alguno de los roles no está definido, o su definición es incorrecta, tira una excepción y se suspende la evaluación.

[10] Por simplicidad, en esta figura no se muestran todos los objetos del árbol de Evaluables (ver el árbol completo en la *“ EMBED Visio.Drawing.11 ”*)

4) `'aPatternDefinition'` le pide a su predicado que se valide sobre `'aPattern'` con el mensaje `#value: aPattern`. Entonces, el árbol de predicados que cuelga de `'aForAllPredicate'` se evalúa usando como contexto la instancia de patrón `'aPattern'` (ver patrones como contexto en 5.4.3). De esta forma, los nombres de roles de la fórmula se ligan a los actores correspondientes a la instancia `'aPattern'`, con lo se termina chequeando el cumplimiento de la fórmula sobre los actores del patrón

4.1, 4.1.1) Para evaluarse, un `ForAllPredicate` empieza evaluando su `'setExpression'`, que en este caso es una `LiteralExpression`. Cuando esta expresión se evalúa simplemente le pide al Contexto el binding para `'Component.MTD'`. Se obtiene así al objeto `protocol`, de tipo Protocol (5.8.4), que es el actor del rol `'Component.MTD'` Los objetos de tipo Protocol en realidad conocen una colección de MessageDeclaration (5.8.5), pero por simplicidad, en la figura se muestran sólo los selectores de los mensajes (`#separatorColor y #displayOn:`)

4.2) Para cada mensaje que tiene el objeto `protocol`, `'aForAlllPredicate'` agrega en el patrón `'aPattern'`, el binding entre la variable cuantificada `'m'` y el mensaje correspondiente. (en el diagrama sólo se muestra el caso `#at:'m' put: #separatorColor`). El patrón guarda los bindings que se agregan con el mensaje `#at: aKey put: aValue` en el diccionario `temporaryBindings`.

4.3, 4.3.1 y 4.3.2) Una vez hecho el binding de la variable cuantificada, `'aForAllPredicate'` evalúa su predicado, y esta evaluación, a su vez se propaga por el resto del árbol. Como se observó en la sección 5.4.5, la evaluación devuelve un `EvaluationResult` al cual se le puede preguntar `#isTrue`. En caso de que el resultado sea falso, el `EvaluationResult` permitirá conocer todos los errores que se produjeron durante la validación. En la sección 5.8.1.1, se dan más detalles de cómo se evalúa la relacion `SendMessageRelation` en particular y los predicados en general

En la próxima sección se verá la forma en la que se construyen las definiciones e instancias de Patrones

5.6 Construcción de Patrones

La construcción de las definiciones y las instancias de patrones de Diseño es una responsabilidad que tienen la jerarquía de `PatternBuilders` que se muestra a continuación:

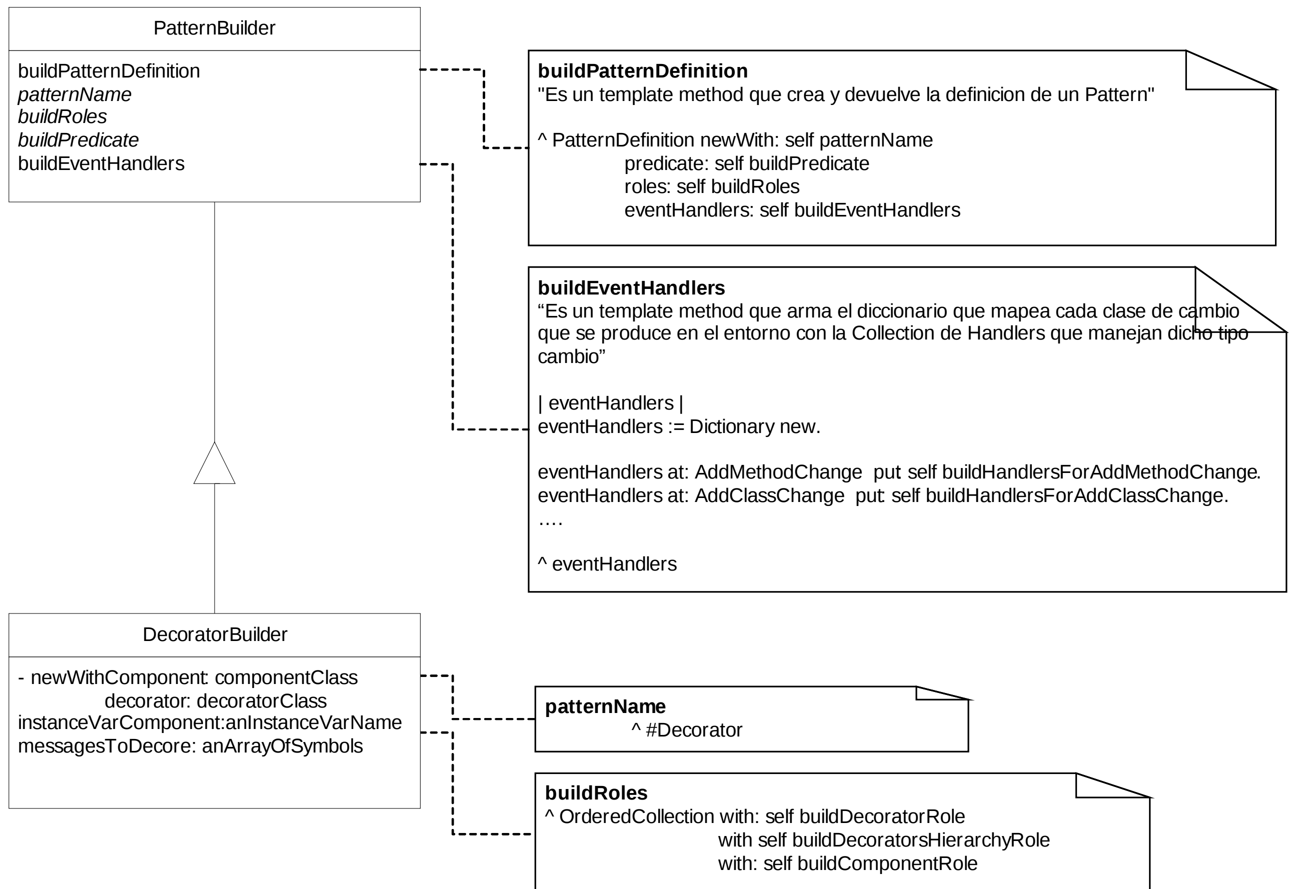


Figura 11 - Pattern Builders

Como se observa en el diagrama anterior, el mensaje `#buildPatternDefinition` es un Template Method, que llama a distintos mensajes en los cuales se construyen todos los objetos necesarios para armar una definición de un patrón (`PatternDefinition`):

- **`#patternName:`** devuelve el nombre de la definición del patrón
- **`#buildPredicate:`** devuelve el `Predicate` que define las propiedades lógicas que deben cumplir las instancias de patrones para ser válidas
- **`#buildRoles`**: devuelve el conjunto de roles que forman parte de la definición del patrón
- **`#buildEventHandlers`**:. los manejadores de eventos que devuelve este método determinan como el Framework de Patrones interactúa con el entorno de desarrollo. Si bien esta interacción se describirá en detalle en la sección 5.7.2, en este punto se puede observar que el método `#buildEventHandlers` es un Template Method, y cada subclase debe implementar los mensajes: `buildHandlersForAddMethodChange`, `buildHandlersForAddClassChange`, etc. En cada uno de estos mensajes se construyen los handlers que se activarán cuando dentro del entorno de desarrollo se produce una determinada *clase de cambio* de código. Algunos

ejemplos posibles de clases de cambio de código son `AddMethodChange y AddClassChange`, que modelan el agregado de un método y de una clase respectivamente. La forma en la que los cambios de código se modelan y ejecutan dentro de Visual Works se resume en la sección 5.7.2.1

En las secciones que siguen se verá la forma en la que el Framework construye las definiciones e instancias de patrones. Como siempre se seguirá el Caso de Estudio, y se mostrará entonces, la construcción de los objetos que aparecen en la *“Figura 4 - Definición e Instancia de patrón”*

5.6.1 Construcción de Definiciones de Patrones

En la sección 5.4.1.1, se vio que la fórmula asociada al patrón del Caso de Estudio es la siguiente:

```
[form-Decorator] ≡ ∀ m ∈ Component.MTD: MessageImplemented(m, HD) ∧
                   SendMessage(MessageImplementations(m, HD), DEC.IVC, m )
```

A continuación se muestra el método de `DecoratorBuilder` que se encarga de construir el predicado correspondiente a **`[form-Decorator]`**:

```
DecoratorBuilder Class>>buildPredicate
| p1 p2 methodSetExpression |


p1 := MessageImplementedRelation for:'m' in:'HD'.

methodSetExpression := MessageImplementationsFunction
                                 protocolExpression: 'm'
                                 classSetExpression: 'HD'.
p2 := SendMessageRelation for: methodSetExpression with: 'DEC.IVC' with: 'm'.

^ ForAllPredicate for: 'm'
                  in: 'Component.MTD'
                  predicate: ( ANDPredicate with: p1 with: p2 )
```

Como se puede observar, el Framework representa la fórmula `[form-Decorator]` de una manera muy directa. Esto se logra porque los mensajes para construir predicados y expresiones siguen la semántica y sintaxis de los componentes existentes en la fórmula. Por ejemplo, el predicado “`∀ m ∈ Component.MTD: P(m)`” se construye enviándole a la clase `ForAllPredicate` el siguiente mensaje:

```
ForAllPredicate for: 'm'  in: 'Component.MTD' predicate: aPredicate
```

Esta forma de construir los predicados y las expresiones permite que el programador pueda entender claramente cuáles son las reglas que se verifican en cada patrón. De esta forma se logran los objetivos de accesibilidad definidos en la sección 5.1.1.3. Además, el conjunto de Evaluables vistos en la sección 5.4 podría ser extendido, y los distintos Predicados y Expresiones podrían reutilizarse y combinarse de distintas maneras para verificar nuevos patrones de diseño.

Además del mensaje `#buildPredicate` que se vio anteriormente, `DecoratorBuilder` debe implementar el método `#buildRoles` para construir los roles que tiene la definición del patrón. Este método y los relacionados se muestran a continuación:

```
DecoratorBuilder Class>>buildRoles
^ OrderedCollection with: self buildDecoratorRole
                    with: self buildDecoratorsHierarchyRole
                    with: self buildComponentRole

DecoratorBuilder Class>>buildDecoratorRole

       | decorator instanceVarComponent |

       decorator := ClassRole newWithName: 'Decorator'  alias: 'DEC'.

       instanceVarComponent := InstanceVarRole newWithName:
                                'instanceVarComponent'  alias: 'IVC'.

       decorator addRole: instanceVarComponent.

       ^ decorator

DecoratorBuilder Class>>buildDecoratorsHierarchyRole
       |  roleDefinitionStrategy |

       roleDefinitionStrategy := DerivedRoleDefinitionStrategy
                    newWithActorComputingBlock:  [ :aPatternInstance | Hierarchy for:
             ( aPatternInstance bindingFor: 'DEC') ].

       ^ HierarchyRole newWithName: 'Decorators'  alias: 'HD'
                    roleDefinitionStrategy: roleDefinitionStrategy


DecoratorBuilder Class>>buildComponentRole
       | componentRole  |

       componentRole := ClassRole newWithName: 'Component'.
       componentRole addRole: self buildMessageToDecoreRole.

       ^ componentRole


DecoratorBuilder Class>>buildMessageToDecoreRole
       | mtdRole mtdComputingBlock roleDefinitionStrategy |

        mtdComputingBlock :=  [ :aPatternInstance  |
                               CompositeProtocol forClass: (aPatternInstance bindingFor:
                               'Component' ) ].

        roleDefinitionStrategy := MixedRoleDefinitionStrategy newWithActorComputingBlock:
                               mtdComputingBlock.

        mtdRole := ProtocolRole newWithName: 'MessagesToDecore'
                               alias: 'MTD'
                                 roleDefinitionStrategy: roleDefinitionStrategy.

       ^ mtdRole
```

Observar que los métodos anteriores construyen los roles que se discutieron en la sección 5.3.3.

En esta sección se ha terminado de ver cómo se construyen las definiciones de patrones. A continuación, se describirá como se realiza la instanciación de los mismos

5.6.2 Construcción de Instancias de Patrones

Cuando un programador instancia un patrón de diseño, debe asignar los actores a cada uno de los roles existentes en la definición del patrón. Como esto es específico del tipo patrón que se instancia, cada Builder tendrá uno o más mensajes destinados a la instanciación de patrones. Por ejemplo, `DecoratorBuilder` tiene el siguiente método que crea y devuelve una instancia de patrón para los actores pasados como parámetro:

```
DecoratorBuilder Class>>newWithComponent: componentClass
                       decorator: decoratorClass
                       instanceVarComponent: anInstanceVarName
                       messagesToDecore: anArrayOfSymbols
| actors |

actors  := OrderedCollection with: (self componentActorFor: componentClass
                                   selectors: anArrayOfSymbols )
                            with: (self decoratorActorFor: decoratorClass
                                   instanceVarRole: anInstanceVarName).

^ Pattern newWith: self patternDefinition actors: actors
```

Observaciones:

- **`'componentClass'` y `'decoratorClass'`** son las clases que actuarán los roles `'Component'` y `'Decorator'` respectivamente.
- **`'messagesToDecore'`** define el actor para el rol `'Component.MTD'`. Como se vio al discutir el modelo de roles del caso de estudio en 5.3.3.2, si no se especifica un conjunto de mensajes a decorar (`anArrayOfSymbols == nil`), el actor se calcula usando la estrategia del rol `'Component.MTD'`. Esta estrategia se construye en el método `#buildMessageToDecoreRole` trascripto en la sección anterior.
- Los métodos `#componentActorFor:selectors:` y `#decoratorActorFor:instanceVarRole:` son privados a `DecoratorBuilder` y sirven para construir los `ClassActor` con los que se crea la nueva instancia de `Pattern`.
- Para construir el pattern que se muestra en la *"Figura 4 - Definición e Instancia de patrón"* el mensaje que hay que enviarle a `DecoratorBuilder` es el siguiente:

```
DecoratorBuilder newWithComponent: VisualComponent
                 decorator: Wrapper
                 instanceVarComponent: 'component'
                 messagesToDecore: #( #displayOn: #separatorColor)
```

5.7 *Interacción entre el ambiente de Desarrollo y el framework de Patrones*

Como se observó en la sección 1.1, uno de los objetivos de la presente tesis es que el modelo detecte problemas en el uso de patrones y los reporte al programador, eventualmente sugiriendo acciones correctivas. En la siguiente sección, se describe como el framework reacciona ante modificaciones en el entorno que, de producirse, harían que el patrón del *"Caso de Estudio"* (3.1) se aplique de manera inconsistente. Posteriormente, se analizará el diseño de esta interacción.

5.7.1 Descripción funcional para el Caso de Estudio

La tabla siguiente describe las acciones tomadas por el modelo ante ciertas modificaciones en el entorno que pueden afectar la consistencia de las instancias existentes del patrón del *"Caso de Estudio"*. Las columnas de la tabla tienen la siguiente información:

- La columna *"modificación en el entorno"* describe el cambio que el programador está realizando sobre el código.
- La segunda columna describe una condición que el modelo chequea si se cumple o no. En el caso que se cumpla, el modelo realiza la acción indicada en la tercera columna.

Modificación en el entorno	Condición chequeada	Acción
Se agrega un nuevo método `M` a una clase que juega el rol de `Component`	`M ∈ Component.MTD`	Se le pregunta al programador *'Se agregó un mensaje a* `[ComponentClassName]` *que es un* Component *de un patrón Decorator: ¿Quiere agregar la implementación default en* `[DecoratorClassName]`*?'.* Si el programador responde que sí, se agrega un método en Decorator que forwardea el nuevo mensaje a la variable de instancia que referencia al Component
Se elimina un método `M` a una clase que juega el rol de `Component`	`M ∈ Component.MTD`	Se le pregunta al programador *Se va a eliminar un método de* `[ComponentClassName]` *¿Desea eliminar los métodos correspondientes de la jerarquía de Decoradores?*
Se agrega un nuevo método `M` a una clase que juega el rol de	`M ∈ Component.MTD` y dentro de la	Se advierte al programador:

Decorator	implementación de M no se forwardea el mensaje al Component	*"Se agregó un mensaje a un Decorador y no se está forwardeando dicho mensaje a la variable de instancia [instanceVarName]"* En este caso no se propone ninguna acción porque el programador podría no querer forwardear el mensaje al Component.. se trata de un warning porque el programador podría estar cometiendo un error.
Se elimina un nuevo método M a una clase que juega el rol de Decorator	M ∈ Component.MTD	Se advierte al programador *Se va a eliminar un método de [decoratorClassName] que pertenece al conjunto de mensajes decorados ¿Desea proceder de todas formas?*
Se elimina la clase C	C juega el rol de Decorator	Se advierte al programador: *Se va a eliminar la clase* [className] *que juega el rol Decorator. Si realiza esta acción el patrón relacionado quedará en un estado inconsistente. ¿Desea proceder de todas formas?*
Se elimina la clase C	C juega el rol de Component	Se advierte al programador: *Se va a eliminar la clase* [className] *que juega el rol Component. Si realiza esta acción el patrón relacionado quedará en un estado inconsistente. ¿Desea proceder de todas formas?*
Se modifica la definición de una clase C	C juega el rol de Decorator y se elimina o renombra la variable de instancia que según la definición del patrón debería referenciar al Component	Se advierte al programador: *Si realiza este cambio en la definición del* [decoratorClassName], *como no existiría más la variable* [instanceVarComponent] *que debería referenciar a* [ComponentClassName], *el patrón quedaría en un estado inconsistente. ¿Desea proceder de todas formas?*

A continuación se detallará el comportamiento del modelo para la primer modificación, usando como ejemplo el agregado del método `ComponentExample>>m1`. Como se describe en el *"Setup del entorno"* (8.2), la clase `ComponentExample` juega el rol de `Component` en uno de los patrones para probar el modelo:

```
TesisIS.DecoratorBuilder newWithComponent:  TesisIS.ComponentExample
                         decorator: TesisIS.DecoratorExample
                         instanceVarComponent: 'component'
```

Cuando el programador agrega el método `ComponentExample>>m1`, el framework chequea si `m1` pertenece al conjunto de mensajes a decorar (o sea, si `m1` ∈ `Component.MTD`). Si esto ocurre, el programador tendría que considerar si `m1` debería estar implementado en uno o más decoradores. Entonces, el framework le informa al programador: *'Se agregó un mensaje a* `[ComponentClassName]` *que es un* Component *de un patrón Decorator: ¿Quiere agregar la implementación default en* `[DecoratorClassName]`*?'* como se muestra en la siguiente pantalla:

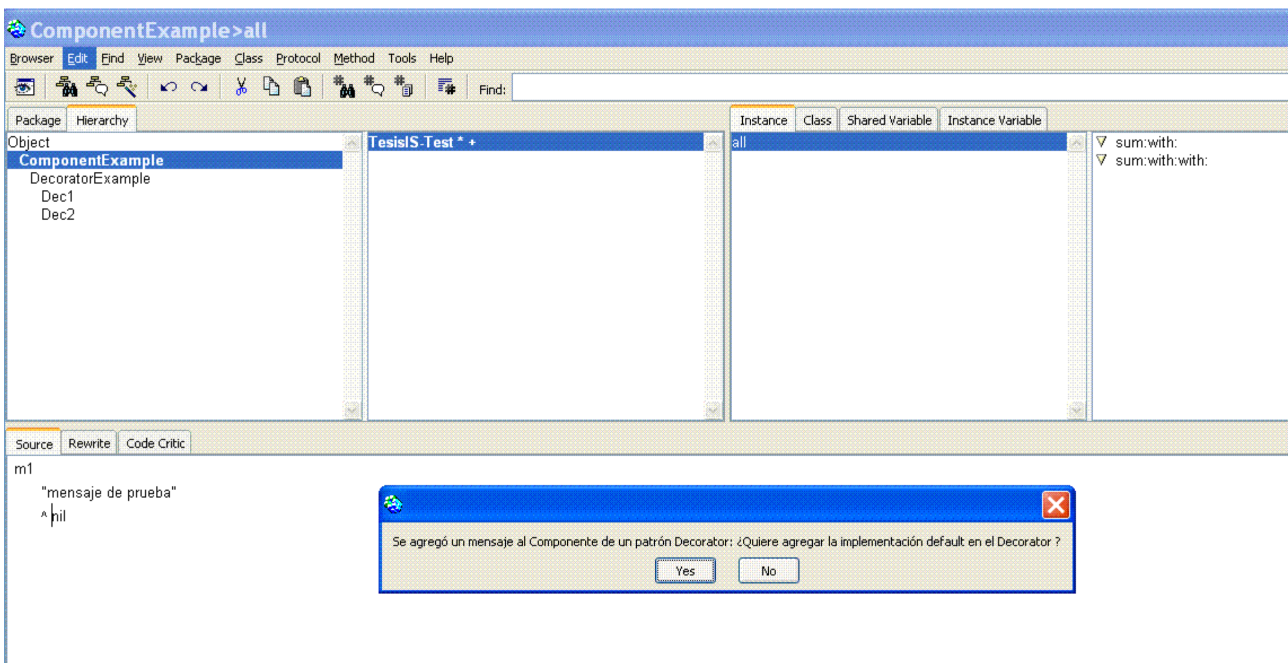


Figura 12 - Interacción con el Entorno. Agregar método a Component(1/2)

Si el programador elige que sí, entonces se agrega la implementación default en la clase que juega el rol de `Decorator` (`DecoratorExample`) como se muestra en la siguiente figura:

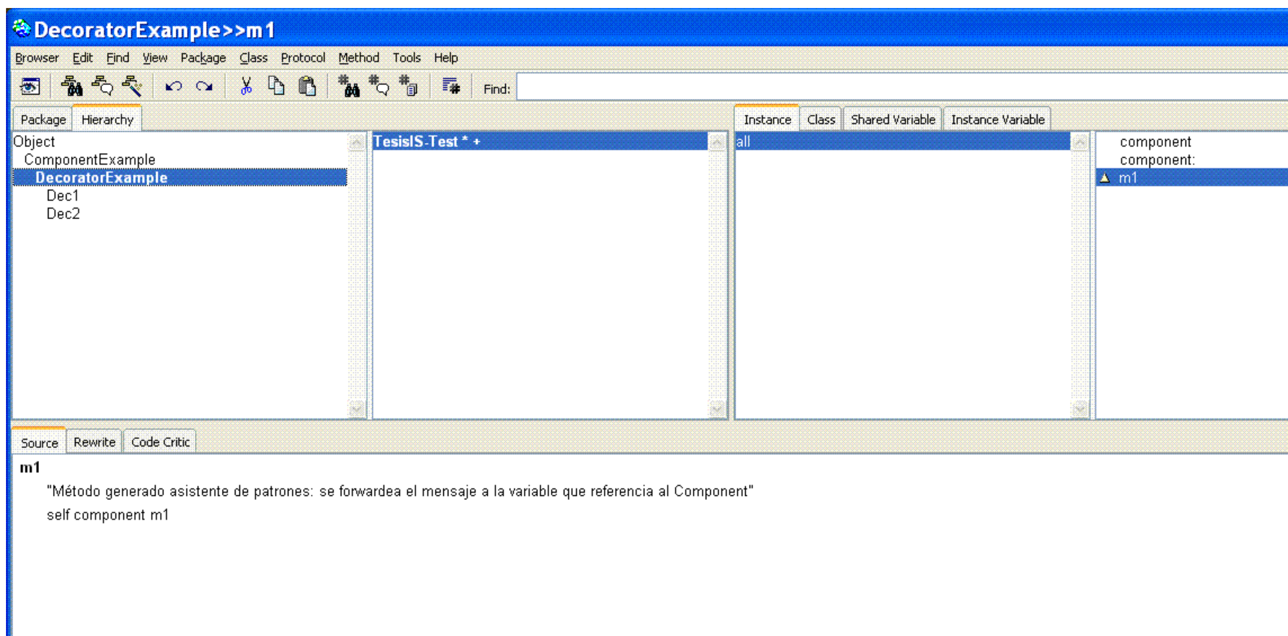


Figura 13 - Interacción con el Entorno. Agregar método a Component(2/2)

De esta forma, el framework detecta un cambio en el código que violaría las restricciones del patrón Decorator, y propone entonces, el agregado del método `DecoratorExample>>m1` mostrado anteriormente. Al agregarse este método se siguen cumpliendo las propiedades del patrón Decorator: los mensajes que se decoran del `Component`, tienen una implementación en jerarquía de Decoradores que a su vez envía el mensaje correspondiente a la variable de instancia que referencia al `Component`. Como se vió en la sección 5.4.1.1, estas propiedades se definen en la fórmula del patrón Decorator:

```
[form-Decorator] ≡ ∀ m ∊ Component.MTD: MessageImplemented(m, HD) ∧
                   SendMessage(MessageImplementations(m, HD), DEC.IVC, m )
```

A continuación se describirá el diseño de la interacción que se acaba de ver funcionalmente

5.7.2 Diseño de la interacción ambiente-framework de patrones

Para comprender el diseño de la interacción entre el ambiente de desarrollo y el framework, antes es necesario comprender cómo se implementan y representan dentro de Visual Works las modificaciones en el código que realiza el programador.

5.7.2.1 Representación de los cambios de código en Visual Works

Dentro de Visual Works, todo cambio que el programador realiza en el código (creación de un nuevo método, agregado de una nueva clase, modificación de una clase existente, etc) se representa como una instancia de la jerarquía `RefactoryChange`. Cada `RefactoryChange` tiene la habilidad de ejecutarse y, varios cambios individuales se pueden agrupar en un

`CompositeRefactoryChange` para ejecutarlos de manera transacccional. En la siguiente figura se muestran algunas clases de esta jerarquía:

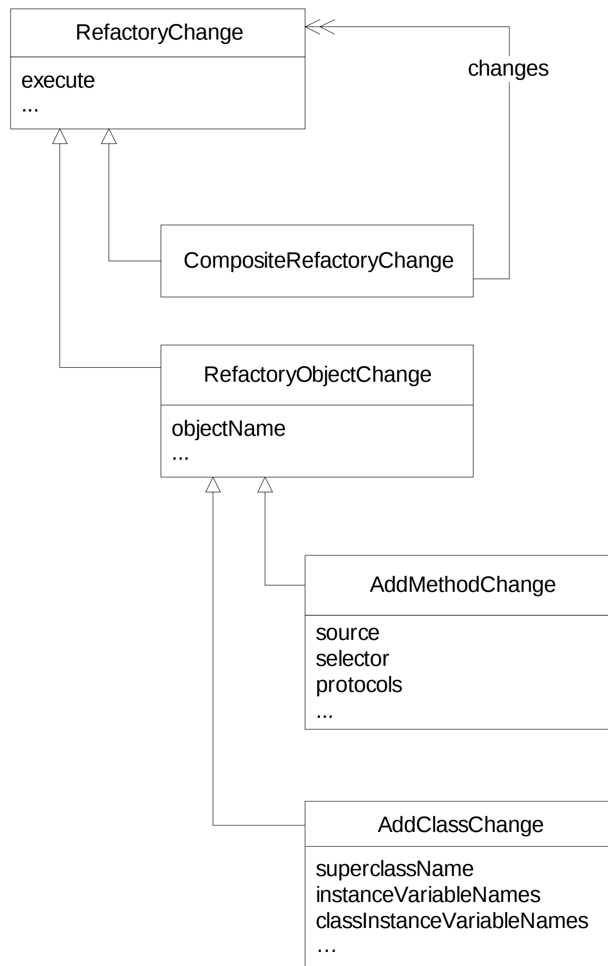


Figura 14 - Representación de los cambios de código en Visual Works

Por ejemplo, cuando el programador agrega un nuevo método se crea una instancia de la clase `AddMethodChange`. A través de este objeto se puede saber la clase que está modificando, el selector del nuevo método y el código fuente correspondiente. En la siguiente sección se verán las colaboraciones que se desencadenan en ejemplo detallado en 5.7

5.7.2.2 Colaboraciones entre el ambiente y el framework de patrones

En la siguiente figura se muestran las colaboraciones que se producen cuando se ejecuta el ejemplo detallado en la sección 5.7:

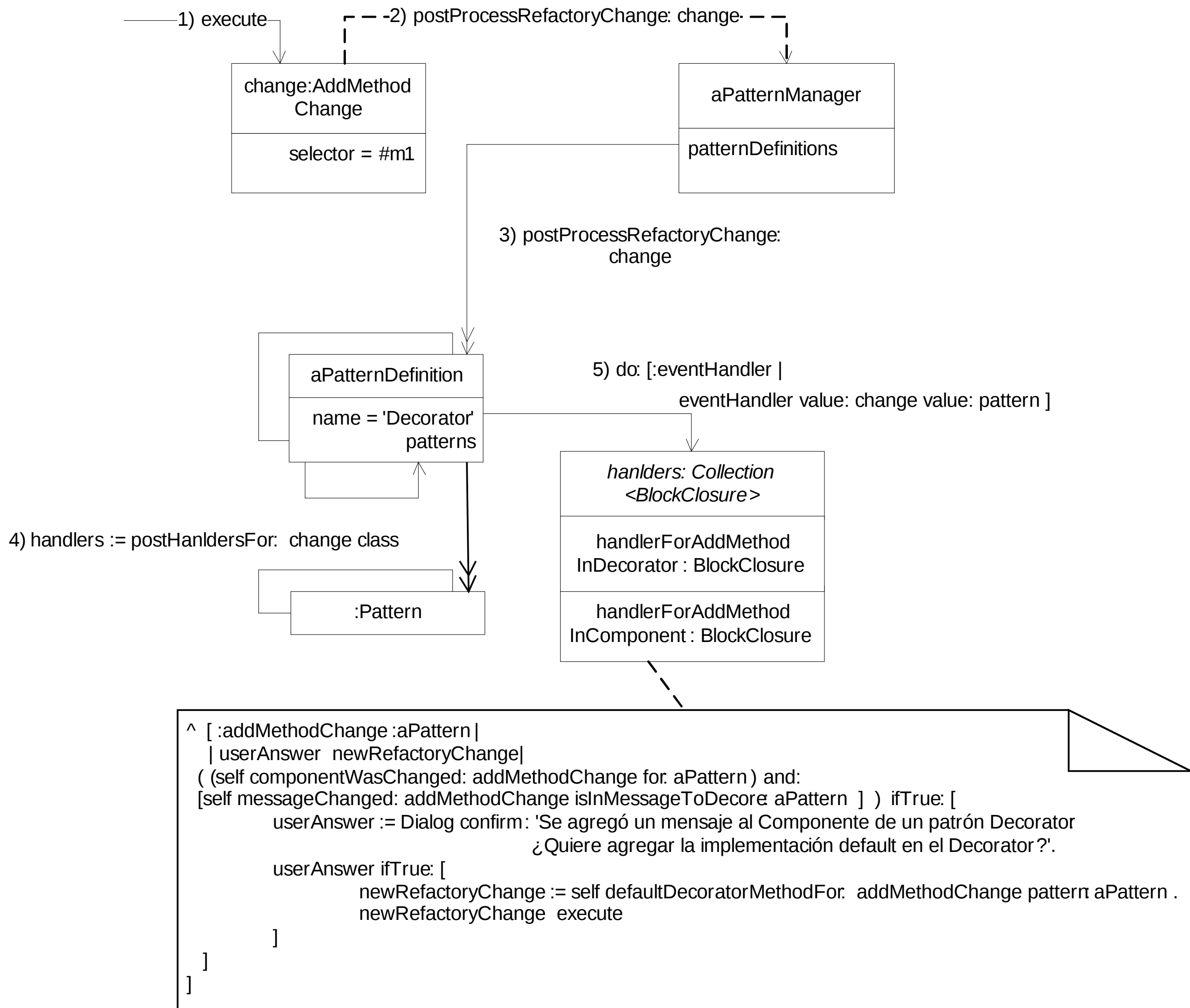


Figura 15 - Colaboraciones entre el ambiente y el framework de patrones

1) Al agregarse el método `#m1` a la clase `ComponentExample` se crea y se ejecuta el objeto `'change'`

2) La ejecución de todos los tipos de cambio se hace pasar por el `PatternManager`. Esta clase, que es un Singleton, funciona como punto de entrada al framework para procesar cambios. Para que todos los tipos de refactoryChange se validen por el `PatternManager`, se introdujo la siguiente modificación en el método `RefactoryChange>>execute`(clase de Visual Works):

```
RefactoryChange>>execute

    | answer |

    (TesisIS.PatternManager preProcessRefactoryChange: self) ifTrue: [
          answer := self executeNotifying: [].
          TesisIS.PatternManager postProcessRefactoryChange: self.
    ].

     ^ answer
```

Observar que existen handlers que se ejecutan antes de que el cambio se realice en el ambiente y otros que se ejecutan después. Esto es necesario porque

- Los handlers asociados a eliminaciones tienen que ejecutarse antes del RefactoryChange de VisualWorks. Entonces, si `PatternManager preProcessRefactoryChange: change,` devuelve false, el cambio de VisualWorks no se ejecuta. Por ejemplo, esto pasa si el usuario va a eliminar la clase que juega el rol Decorator. El `handler` detecta esta situación y le pregunta al programador si realmente quiere hacer eso. Si el programador dice que no, entonces el `handler` devuelve false, y el `change` no se ejecuta con lo que no se elimina la clase

- Algunas operaciones sólo pueden realizarse o verificarse una vez que el cambio se ha realizado. Por ejemplo, si los mensajes a decorar se calculan con el bloque que devuelve todos los mensajes del Component, y se agrega un nuevo mensaje al Component, es necesario que este nuevo método haya sido efectivamente agregado para detectarlo como un mensaje perteneciente a MTD.

3) ‘`aPatternManager`’ forwardea el mensaje `#postProcessRefactoryChange` a todas las definiciones de patrones existentes (`patternDefinitions`).

4) La definición del patrón Decorator (‘`aPatternDefinition`’), obtiene todos los `eventHandlers` asociados a la *clase de cambio* que se está produciendo en el entorno. En este caso, la clase del objeto ‘`change`’ es `AddMethodChange`, porque precisamente se está agregando un método nuevo en una clase. Un eventHandler es un bloque de Smalltalk que se evalúa sobre un cambio que se produce en el entorno y una instancia de patrón específicos. Los eventHanlders son creados y configurados por los `PatternBuilders` (ver sección 5.6).

5) Se evalúan todos eventHandlers para la clase de cambio `AddMethodChange:`

- Cómo se observó en 5.2, cada `PatternDefinition` conoce, a través de la variable ‘`patterns`’, a todas las instancias de patrones de su “tipo” existentes dentro del ambiente. Entonces, la evaluación de cada handler se realiza sobre cada una de estas instancias

- El handler que se detalla en la figura es el que maneja el agregado de nuevos métodos en el `Component` de un patrón `Decorator` (‘`handlerForAddMethodInComponent`’). Este handler verifica dos cosas:

- si la clase que se está modificando juega el rol de Component en el patrón

- si el mensaje agregado pertenece al conjunto de mensajes a decorar (o sea si `M ∈ Component.MTD`).

    Si ambas condiciones son verdaderas, como se indicó en 5.7.1, se le pregunta al programador si quiere agregar la implementación "default" en la clase que juega el rol de `Decorator`.

Para las distintas modificaciones del entorno indicadas en 5.7, existe un eventHanlder particular que se encarga de verificar la condición y de proponer la acción asociada en el caso de que corresponda.

Con lo expuesto en esta sección se tiene una visión funcional de cómo interactúa el modelo de patrones con el entorno de desarrollo y el diseño de la solución subyacente. La próxima sección es una referencia de las clases más importantes que constituyen el Framework de Patrones.

5.8 Referencia de las Clases del Framework

En esta sección se verán en forma de referencia, las clases más relevantes del Framework y ciertas características de su implementación. En la primera parte 5.8.1, se detallarán los distintos tipos de predicados existentes, profundizando el material de la sección 5.4.5. En particular, la sección del predicado SendMessageRelation(5.8.1.1) se utiliza como ejemplo para mostrar las siguientes aspectos de la evaluación de predicados:

- Colaboraciones entre los objetos participantes
- Los bindings entre expresiones y resultados
- Los objetos que registran las condiciones de un predicado no satisfechas

En el resto de la sección se describen las clases Method, ClassSet, Protocol y MessageDeclaration.

5.8.1 Predicates

5.8.1.1 SendMessageRelation

Como se mencionó en la introducción de esta referencia, la relación `SendMessageRelation` se va a usar como ejemplo para mostrar distintos aspectos de la evaluación de predicados. Esta sección se encuentra organizada de la siguiente manera:

- Inicialmente se detallarán las propiedades que se verifican con esta relación, y se verá que la semántica de la misma no presenta los problemas que se observaron en Lepus (3.4.4)
- Se describirá cómo se evalúa un `SendMessageRelation` con un diagrama de colaboraciones

Propiedades verificadas por un SendMessageRelation

Las instancias de `SendMessageRelation` se construyen con el siguiente mensaje:

```
SendMessageRelation Class>>for: aMethodSetExpression
                           with: anInstanceVarExpression
                           with: aProtocolExpression
```

Esta relación es verdadera si y solo si para todo `'method'` perteneciente a `'aMethodSetExpression'` y para todo `'message'` perteneciente a `'aProtocolExpression'` se cumple la relación `SendMessageRelation(method, instanceVarName, message)`. A su vez, esta última relación es verdadera si dentro del método `'method'` se le envía a la variable de instancia `'instanceVarName'` el mensaje `'message'`. Para evaluar si dento de un método se le envía un mensaje a una variable se instancia se consideran las tres posibilidades que se muestran a continuación. El código corresponde clases existente en el *“Ejemplo de Patrón Decorator”* (3.1.2):

1. **El mensaje se envía directamente**

```
TranslatingWrapper>>displayOn: aGraphicsContext
      aGraphicsContext translateBy: self translation.
      component displayOn: aGraphicsContext.
```

2. **El mensaje se envía indirectamente a través de accesor**

```
Wrapper>>displayOn: aGraphicsContext
      self component displayOn: aGraphicsContext.
```

3. **El mensaje se envía indirectamente pasando por mensajes intermedios**. El ejemplo siguiente muestra que `ReversingWrapper>>displayOn:` envía el mensaje al componente en forma indirecta, usando la implementación de `#displayOn:` que define la la superclase `Wrapper`.

```
ReversingWrapper>>displayOn: aGraphicsContext

"Display the receiver on aGraphicsContext. The receiver may alter
aGraphicsContext in any way it chooses."

reverse value ifTrue: [|gc box |
             box := container == nil
             ifTrue: [self bounds]
             ifFalse: [container compositionBoundsFor: self].
                   box left: box left + offset.
                   gc := aGraphicsContext copy.
                   gc paint: self selectionBackgroundColor.
                   gc displayRectangle: box .
                   gc paint: self selectionForegroundColor.
                   super displayOn: gc]
             ifFalse: [super displayOn: aGraphicsContext]
```

Definiendo la relación `SendMessageRelation` de esta forma, los problemas que se plantearon en las observaciones de Lepus 3.4.4 no se presentan porque:

- La relación `SendMessageRelation(method,instanceVarName,message)` se puede verificar estáticamente ya que dentro de un determinado método se verifica si se envía o no el mensaje `'message'` a la variable `'instanceVarName'`. Tampoco existen los problemas de las relaciones entre métodos que se observan en LePUS y además es explícito el receptor del mensaje.
- Las distintas posibilidades que se chequean para verificar el envío de mensajes abarcan los casos más comunes que se presentan en la práctica.

<u>Cómo se evalúa un SendMessageRelation</u>

Como se observó en la sección de "Construcción de Definiciones de Patrones" (5.6.1), cuando se construye el patrón del caso de estudio, la relación `SendMessageRelation` se crea con el siguiente mensaje:

```
SendMessageRelation for: (MessageImplementationsFunction protocolExpression: 'm'
                                                         classSetExpression: 'HD')
                   with: 'DEC.IVC'
                   with: 'm'.
```

A continuación, se muestran y comentan las colaboraciones que se producen cuando la instancia de `SendMessageRelation` que se construye con el mensaje anterior (en la figura `'aSendMessagRelation'`), recibe el mensaje `#value: anEvaluationContext` :

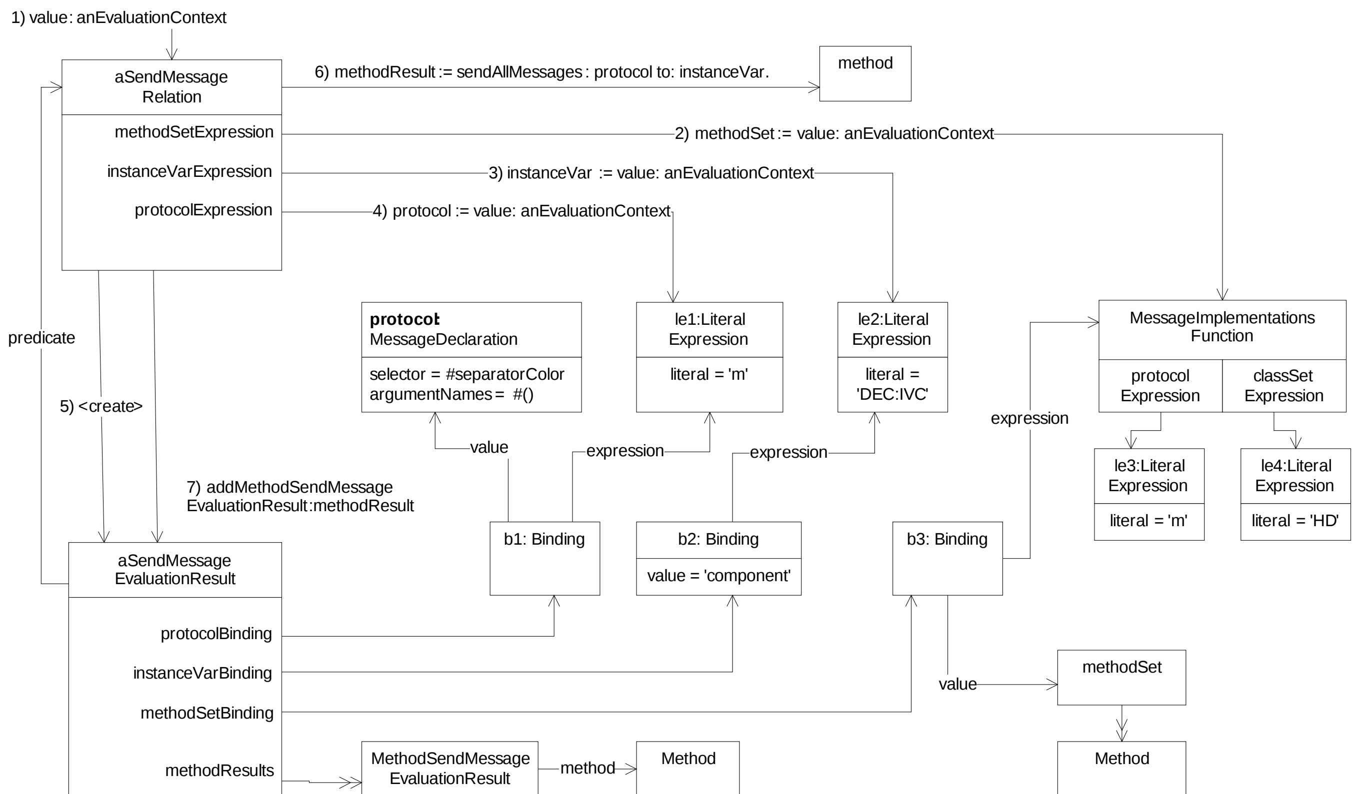


Figura 16 - Evaluación de SendMessageRelation

Descripción de las colaboraciones

1,2, 3 y 4) Para evaluarse una `SendMessageRelation`, evaluá su `methodSetExpression`, `instanceVarExpression` y `protocolExpression` (pasos 2 , 3 y 4 respectivamente).

5) `'aSendMessageRelation'` crea su resultado (`'aSendMessageEvaluationResult'`), usando los resultados de evaluar sus expresiones, con el siguiente mensaje:

```
SendMessageEvaluationResult newFor: self
                            methodSet: methodSet
                            instanceVar: instanceVar
                            protocol: protocol.
```

De esta forma, el resultado contiene toda la información de la evaluación:

- tiene una referencia al predicate que se evaluó, en este caso `'aSendMessagsRelation'`
- tiene todos los *bindings* de las expresiones del predicate asociado. Cada binding asocia una expresión con el resultado que se obtuvo al evaluar dicha expresión. Por ejemplo, el binding `'b1'` tiene el resultado de evaluar la literalExpression `'le1'` que es el protocolo `'protocol'`.

6 y 7) a cada `'method'` que pertenece a `'methodSet'` se le envía el mensaje

```
Method>>sendAllMessages: protocol to: instanceVar
```

La clase Method del Framework (5.8.2) tiene entre sus responsabilidades la de parsear el código fuente y responder preguntas como la anterior: si todos los mensajes del conjunto `'protocol'`, son a su vez forwardeados a la variable de instancia `'instanceVar'` (ver más detalles en 5.8.2). Como resultado de `Method>>sendAllMessages:to:` se obtiene un `'methodResult'` de tipo `MethodSendMessageEvaluationResult`. Este objeto, fijado un método `'method'`, sabe si la relacion `SendMessageRelation(method, instanceVarName, protocol)` se cumple o no para los mensajes de `'protocol'`.

7) se agrega el resultado de evaluar cada método a`'aSendMessageEvaluationResult'` y entonces este objeto concoce el resultado completo de evaluar la relacion construída con:

```
SendMessageRelation Class>>for: aMethodSetExpression
                          with: anInstanceVarExpression
                          with: aProtocolExpression
```

Para finalizar se observa que así como un `SendMessageEvaluationResult` conoce los bindings para las expresiones que contiene una `SendMessageRelation`, en general cada `EvaluationResult` conoce los bindings para las expresiones existentes en el predicado asociado.

5.8.1.2 ForAllPredicate

Las instancias de `ForAllPredicate` se construyen con el siguiente mensaje:

```
ForAllPredicate class>>for: varName in: setExpression predicate: aPredicate
```

Como se vió en la *"Construcción de Definiciones de Patrones"* (5.6.1), el predicado del patrón del caso de estudio se arma de esta forma:

```
predicate := ForAllPredicate for: 'm' in: 'Component.MTD'  predicate: P
```

Cuando a `'predicate'` se le envía el mensaje `#value:anEvaluationContext`, se obtiene un `ForAllEvaluationResult` con la siguiente información:

- **`setBinding`**: asocia la `LiteralExpression ('Component.MTD')` con el objeto Protocol que contiene el conjunto de mensajes que se decoran
- **`evaluationResults`**: un diccionario que tiene como clave cada uno de los bindings posibles de la variable `'m'`, y como valor el `EvaluationResult` que se obtuvo al evaluar el predicado `'P(m)'`. O sea, un `ForAllEvaluationResult` conoce cual es el resultado de evaluar su predicado interno P para cada uno de los posibles bindings de la variable cuantificada

Para más detalles sobre la evalución de un `ForAllPredicate` ver el diagrama de colaboración de la validación del patrón del Caso de Estudio (5.5)

5.8.1.3 MessageImplementedRelation.

Las instancias de `MessageImplementedRelation` se construyen con el siguiente mensaje:

```
MessageImplementedRelation class>>for: aProtocolExpression in: aClassSetExpression
```

El objetivo de esta relación es verificar si el conjunto de mensajes determinado por `'aProtocolExpression'` está implementado en el ClassSet asociado a `'aClassSetExpression'`. El concepto ClassSet abarca tanto un conjunto de clases arbitrario como una jerarquía de clases.

El resultado que se obtiene al evaluar un `MessageImplementedRelation` es un objeto de tipo `MessageImplementedEvaluationResult`, que tiene las siguientes variables de instancia:

- **`errors`**: es un diccionario cuyas claves son los selectores no implementados y como value para cada selector se especifica el ClassSet en el cual el selector no está implementado
- **`protocolBinding`**: es el binding correspondiente a `'protocolExpression'`
- **`classSetBinding`**: es el binding de `'classSetExpression'`

5.8.1.4 Predicados binarios

Los predicados binarios `ANDPredicate` y `ORPredicate` heredan de `BinaryPredicate`. Este predicate tiene un left y un right predicates. A continuación, se muestra la implementación del mensaje `BinaryPredicate>> value: anEvaluationContext`:

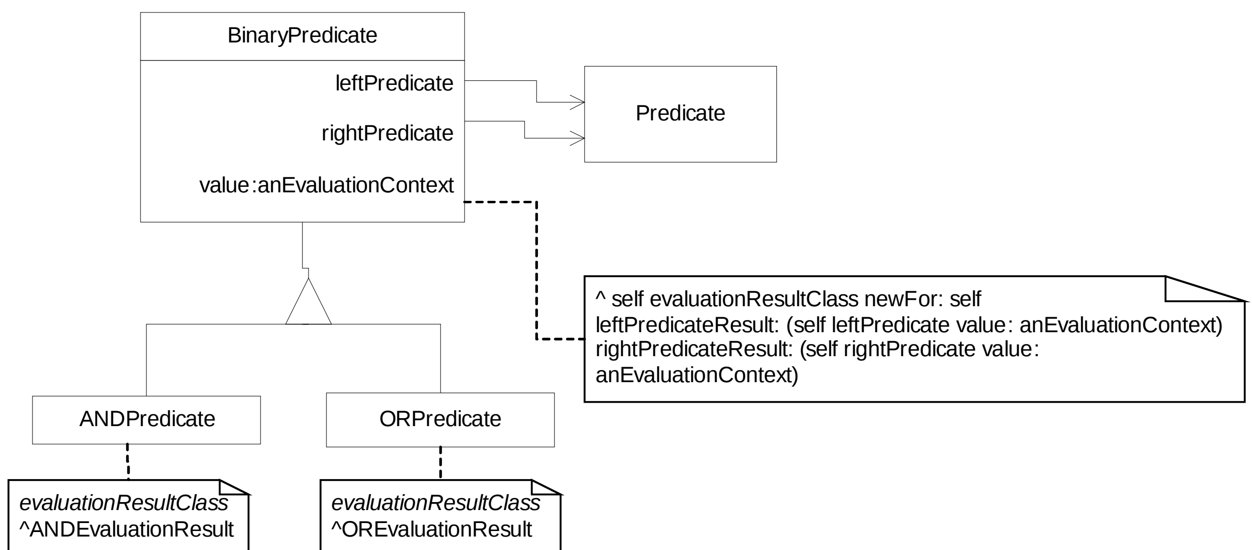


Figura 17 - Predicados binarios

Al evaluar un `ANDPredicate` se obtiene un `ANDEvaluationResult` y al evaluar un `ORPredicate` un `OREvaluationResult`. Estas clases se muestran a continuación:

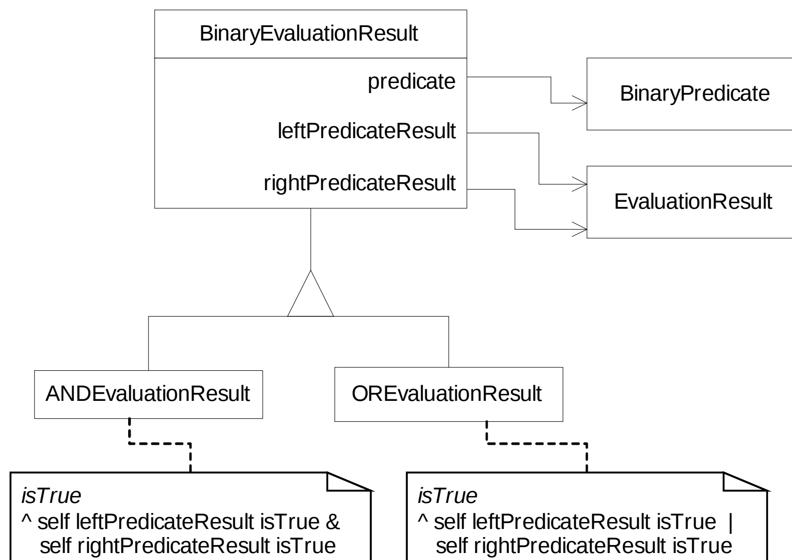


Figura 18 - Resultado de evaluar predicados binarios

5.8.2 Method

Como se establece en el paradigma de POO, un *mensaje* puede tener distintas implementaciones en distintas clases. Cada una de estas implementaciones es un *método*. La clase `Method` existente en el framework modela precisamente un método particular definido en una clase determinada. Cada `method` tiene como responsabilidad responder a mensajes que interrogan sobre propiedades de su código fuente. Estos mensajes son:

- **`Method>>messagesSendedToSelf.`** Devuelve una coleccion de `RBMessageNode`[11] que contiene todos los mensajes que en este método se envían a `'self'` incluyendo también los mensajes cuya búsqueda empieza por super.
- **`Method>>sendMessageUsingAccesor: aSelector to:anInstanceVarName`**. Devuelve true si el método receptor le envía el mensaje `'aSelector'` a la variable de instancia con nombre `'anInstanceVarName'` usando un accesor que tiene el mismo nombre que la variable de instancia.
- **`Method>>sendMessageWithOutAccessor: aSelector to:anInstanceVarName.`** Devuelve true si el método receptor le envía el mensaje `'aSelector'` a la variable de instancia con nombre `'anInstanceVarName'` sin usar un accesor"

Además, los methods responden al mensaje:

[11] Para análizar su código fuente cada método accede a su arbol sintáctico abstracto (AST). Dentro de VisualWorks cada nodo de un AST se representa con una clase de la jerarquía *RBProgramNode*. Por ejemplo, *RBMessageNode* representa un envío de mensaje y tiene las siguientes variables de instancia:

- **receiver**: es el receptor del mensaje y referencia a un *RBVariableNode*
- **selector**: es un Symbol corriente de Smalltalk que denota al mensaje envíado
- **selectorParts**: es una OrderedCollection de *RBKeywordToken*
- **arguments**: es una OrderedCollection de *RBVariableNode*

VisualWorks ofrece también clases como el ParseTreeSearcher que permite realizar búsquedas sobre el AST.

```
Method>>sendAllMessages: protocol to: instanceVarName.
```

Este mensaje permite saber si dentro del código del método se envían todos los mensajes de `'protocol'` a la variable de instancia cuyo nombre es `'instanceVarName'`. El resultado de este mensaje es un `MethodSendMessageEvaluationResult`. A continuación se muestra como las variables de instancia de `MethodSendMessageEvaluationResult` representan la evaluación:

Nombre	**Tipo**	**Descripción**
`Protocol`	`Protocol`	El conjunto de mensaje que se analiza
`instanceVarName`	`Symbol`	El nombre de la variable de instancia
`Method`	`Method`	El method al cual se le envió el mensaje `'sendAllMessages:...'`
`notSendedMessages`	`Collection (MessageDeclaration)`	una collection con todos los mensajes de `'protocol'` que no se envían a `'instanceVarName'`
`callSequences`	`Dictionary( MessageDeclaration → OrderedCollection)`	- la clave es un `MessageDeclaration`, `'messageDecl'` , que pertenece al protocolo referenciado por `'protocol'` - El value es una collection que tiene la secuencia de métodos que son llamados por `'method'` hasta que se produce el envío de `'messageDecl'` a `'instanceVarName'`

El mensaje `sendAllMessages: protocol to: instanceVarName` se utiliza en la evaluación de SendMessageRelation

5.8.3 ClassSet

Un `ClassSet` representa un conjunto de clases del sistema. `DefaultClassSet` es un conjunto de clases arbitrario. `Hierarchy` representa una jerarquía de clases centrada en una `'rootClass'`. A continuación se comentan los principales mensajes:

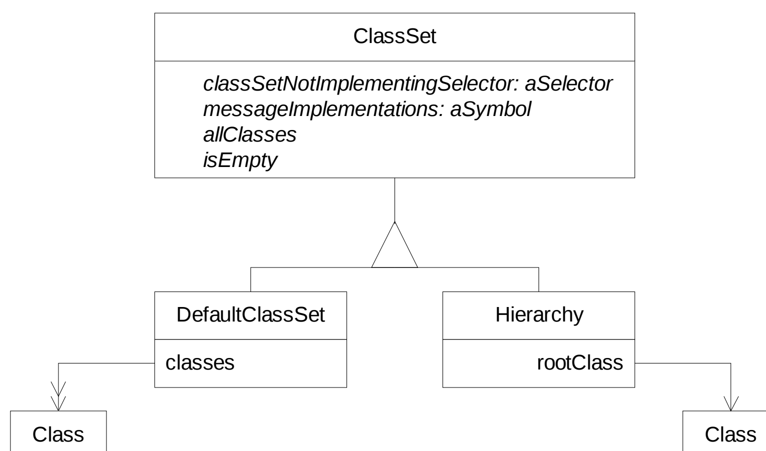


Figura 19 - Jerarquía de ClassSets

Mensaje	**Descripción**

classSetNotImplementingSelector: aSelector	Devuelve un ClassSet que contiene las clases del conjunto receptor que ***no*** implementan 'aSelector'. Si todas las clases implementan 'aSelector' devuelve nil. El objetivo de este método es determinar todas las clases que *deberían* implementar un determinado mensaje y no lo están haciendo. Por ejemplo, una clase interna de una jerarquía puede no implementar el mensaje #m y no va a estar en el conjunto de Hierarchy>>classesNotImplementingSelector:#m porque como es una clase interna no necesariamente debería implementarlo porque puede ser una clase abstracta
messageImplementations: aSymbol	Devuelve una colección de Method que contiene todos las implementaciones de 'aSymbol' en el classSet receptor. No se consideran 'implementaciones' aquellos métodos que como cuerpo del método tengan self subclassResponsibility
allClasses	Devuelve una collection con todas las clases existentes en este classSet
isEmpty	Devuelve false si el ClassSet receptor tiene al menos 1 clase

5.8.4 Protocol

La clase Protocol permite tratar de manera uniforme a los mensajes individuales y a los conjuntos de mensajes.. Un CompositeProtocol tiene un conjunto de MessageDeclaration. La clase Protocol tiene mensajes para acceder a todas las declaraciones y selectores. Con el mensaje do: aBlock se puede evaluar un bloque sobre todas las messageDeclarations

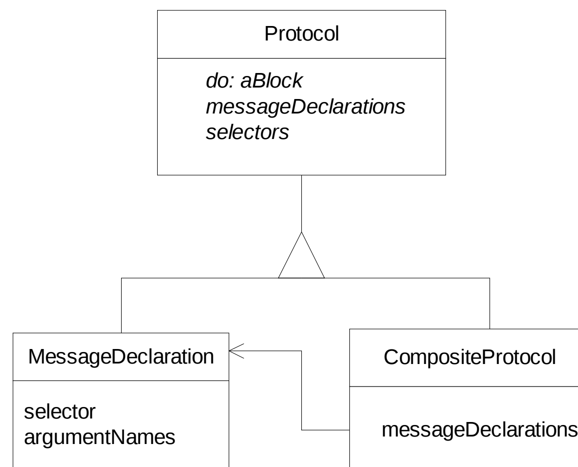


Figura 20 - Jerarquía de Protocols

5.8.5 MessageDeclaration

La clase MessageDeclaration se introdujo porque en el ambiente de Smalltalk cada mensaje se representa sólo con un Symbol y no se tiene acceso a los nombres de los parámetros. El Framework necesita conocer los nombres de parámetros de una declaración de mensaje para realizar determinadas operaciones como por ejemplo generar código (ver Protocol)

6 Conclusiones

Para presentar las conclusiones de la presente tesis, primeramente se analizarán los logros obtenidos con el Framework de patrones y se compararán con los objetivos planteados en 1.1. Luego, se analizará la factibilidad de generalizar y extender el enfoque propuesto para otros patrones de diseño

6.1 Resultados obtenidos con el Framework.

Como resultado del trabajo de desarrollo, se ha conseguido una versión del Framework que soporta satisfactoriamente el patrón Decorator del *"Caso de Estudio"* (3.1), y que tiene las siguientes características y funcionalidades:

- Permite crear nuevas definiciones de patrones e instancias patrones como se ha mostrado en las secciones 5.6.1 y 5.6.2 respectivamente
- Brinda una API que permite definir una manera sencilla las propiedades lógicas que tiene que cumplir un patrón de diseño. Como se observó en la sección 5.6.1 , la fórmula conceptual del patrón Decorator es la siguientes

```
∀ m ∈ Component.MTD: MessageImplemented(m, HD) ∧
SendMessage(MessageImplementations(m, HD), DEC.IVC, m )
```

  Esta fórmula se representa de una forma clara con los predicados y funciones del Framework

- **Validación de patrones**. Como vimos en la sección 5.5, cada instancia patrón de diseño se puede validar enviándole el mensaje `#validate`. Esto permite verificar si se cumplen todas las restricciones que el patrón define y, si no es así, se puede obtener información detallada de las "desviaciones". Estas desviaciones, o potenciales errores al aplicar un patrón, están modeladas con objetos dentro del framework lo que permite conocer los detalles de las posibles fallas. Eventualmente, el framework le daría al programador la opción de ignorar estas desviaciones, logrando la flexibilidad descripta en la sección 4.5

- **Integración con el entorno de desarrollo**. El framework se pudo integrar exitosamente con el entorno generando las recomendaciones y advertencias al programador que se describieron en la sección 5.7.1. Este resultado nos parece particularmente interesante ya que permite mostrar con ejemplos concretos las ventajas que le trae al programador tener a los patrones de diseños "vivos" dentro del entorno de desarrollo. Como observamos en la sección 1.3, estas características no las tienen otras herramientas observadas. Como ejemplos de esta integración podemos nombrar los siguientes casos:

- *Validación integrada con entorno*. Cuando el programador va a realizar un cambio que violaría las restricciones de un determinado patrón (ej, agregar un método en un Decorator que no forwardea el mensaje al Component), el framework detecta este potencial problema y le avisa al programador para que éste tome la acción necesaria si corresponde.

- *Soporte de operaciones de generación de código*. Por ejemplo, cuando se agrega un método en Component, el framework le propone al programador si quiere agregar la implementación default en el Decorator para mantener la consistencia del patrón

En el framework desarrollado en la presente tesis, se ha puesto el foco principalmente en los aspectos que se evaluaron como más complejos:

- la implementación del modelo de patrones (roles, actores, propiedades lógicas y mecanismo de validación)
- la integración en el entorno de desarrollo para entender y mostrar el tipo de beneficio que se puede obtener cuando los patrones están vivos dentro del ambiente.

Haciendo un balance de los objetivos planteados en 1.1 y teniendo en cuenta que el Framework actualmente sólo soporta el patrón del Caso de Estudio, podemos concluir que los objetivos más complejos se pudieron lograr satisfactoriamente. Estos objetivos más complejos que se lograron son:

- soportar la instanciación de patrones
- validar el uso correcto de patrones
- generar consejos al programador sobre posibles modificaciones a realizar en el código
- integrar el framework en el entorno de desarrollo, reaccionando frente a modificaciones que puedan implicar un mal uso de los patrones.

Sobre otros objetivos como la posibilidad de documentar el sistema a través de patrones, brindar facilidades de búsqueda y manipular patrones a través de operaciones específicas dependiendo del tipo de patrón, no se ha avanzado en el desarrollo, aunque podemos observar que, habiendo construido el modelo de patrones, es factible agregar estas funcionalidades.

Finalmente, como concluimos en la sección 4.6, para que un patrón de diseño pueda tener un soporte de herramientas, es condición necesaria que la descripción de su solución tenga un mínimo nivel de precisión. La variante del patrón Decorator del Caso de Estudio cumple con este requisito, y para ella, hemos podido integrarla dentro del Framework y dar un soporte dentro del entorno que nos resulta interesante (5.7.1). En qué medida esto podría realizarse para otros patrones de diseño es el tema que se discutirá en la próxima sección

6.2 Factibilidad de extender el Framework propuesto

El problema de analizar la factibilidad de extender el Framework de patrones propuesto a otros tipos de patrones, nos lleva al problema del nivel de precisión con el que cada uno de los patrones de diseño se describe. Como comentamos anteriormente, es condición necesaria que la descripción de la solución de un patrón tenga un mínimo nivel de precisión para que se pueda brindar un soporte automatizado. Nos parece entonces oportuno retomar la discusión planteada en la sección 4.2, donde recogimos diversas opiniones en relación al problema de si los patrones de diseño son formalizables o no. A continuación resumimos estas opiniones para luego plantear nuestra visión.

Con respecto a esta discusión de si los patrones son formalizables, si planteamos este problema para todos los patrones en su conjunto, concluimos que la respuesta es que no lo son. Coincidimos con las observaciones que realiza Vlissides [Vlis98] y, como vimos en la sección 4.3, el mismo Eden [Eden00b] reconoce que sólo algunos patrones tienen una descripción relativamente precisa.

Por otra parte, plantear el problema como una dicotomía sobre si los patrones de diseño son formalizables o no, o si es importante formalizarlos, no nos parece que conduzca a ningún resultado práctico. En función del trabajo realizado, nos parece más fructífero formularnos las siguientes preguntas:

- ¿Existe un conjunto de patrones y variantes asociadas que tengan el mínimo de precisión necesaria para ser automatizados, y que, al mismo tiempo, sean relevantes teniendo en cuenta su aplicación práctica y frecuencia de uso?
- ¿Cuáles son dichos patrones y variantes?

En base a la experiencia obtenida en la presente tesis, consideramos que no hay una respuesta cerrada a estas preguntas. Si bien los resultados obtenidos para el patrón del Caso de Estudio son interesantes, este patrón justamente tiene una descripción bastante precisa y no queda claro que los mismos resultados puedan lograrse para otros patrones que tengan relevancia práctica suficiente. Para continuar con esta línea de investigación en futuros trabajos, consideramos que algunas ideas y criterios importantes para responder a estas preguntas son los siguientes:

- Como observa Vlissides, cada descripción de un patrón es sólo un ejemplo de una variante que se observa en la práctica. Entonces, cuanto mayor sean las variantes de implementación que tenga un patrón, y más imprecisa sea su descripción, será menos factible dar soporte de herramientas a dicho patrón. Un ejemplo de este tipo de patrones podría ser el Observer
- Para otros patrones, con descripciones más precisas como el Visitor y el Composite, podría intentarse automatizar su uso de forma análoga a la que hizo para el patrón del caso de estudio. Esto se detallará más en la sección de Trabajos Futuro.
- Responder a estas preguntas implica, necesariamente, tener un feed-back de la aplicación práctica del Framework de patrones. Por esto, se podría imaginar que un framework como el desarrollado en la presente tesis, podría evolucionar como un proyecto Open Source ya que es necesario entender, en contacto con la práctica, si existen variantes de implementación de patrones que tengan aplicación real y frecuente, y que, a su vez, tengan el nivel de precisión necesario como para automatizarse.

7 Trabajos Futuros

Teniendo en cuenta las ideas delineadas en la sección anterior, pensamos que existen varias tareas que podrían realizarse para continuar explorando Framework de patrones como un enfoque que permita dar soporte automatizado a los patrones dentro del entorno de desarrollo. Dentro de estas tareas podemos destacar las siguientes:

- **Incorporar nuevos patrones**. Como observamos anteriormente, para que un patrón de diseño pueda tener un soporte de herramientas, es necesario que su descripción tenga un mínimo nivel de precisión. En esta categoría podríamos encontrar al *"Patrón Visitor"* (8.1),

al Composite, alguna variante del State y otros. Por ejemplo, el predicado del patrón Visitor verificaría, entre otras cosas, que todas las implementaciones del mensaje `Element>>accept:Visitor en la` jerarquía de `Elements`, cumplan ciertas propiedades. Para esto, se pueden utilizar varios predicados que ya están en el Framework y otros se pueden agregar extendiendo la jerarquía de Evaluables que se mostró en la sección 5.4.2 [12]

- **Incorporar variantes en los patrones.** Una forma de incorporar variantes para un mismo patrón puede ser agregando/especializando el `PatternBuilder` correspondiente. Por ejemplo, si quisiéramos agregar alguna variante del Patrón Decorator podríamos subclasificar la clase `DecoratorBuilder` que es la que tiene la lógica para crear la definición del Patrón Decorator.

- **Soportar el manejo de desviaciones al aplicar patrones**. Como se comentó en la sección de Conclusiones cuando el framework valida un patrón devuelve un objeto complejo que contiene información detallada de los lugares del código en los cuales hay posibles errores en el uso del patrón. La funcionalidad que se podría agregar es que cuando el Framework reporta los posibles errores, le permita al programador indicar cuáles de ellos son en realidad "desviaciones" conocidas y agregar un comentario aclarativo si el programador así lo desea. Luego de que el programador informa esto, las sucesivas validaciones del patrón, no deberían reportar errores para las desviaciones que ya fueron contempladas. Como vimos en la sección 4.5, es normal que se presenten estas desviaciones con respecto a la forma "ideal" de aplicar un patrón. En consecuencia, es importante que una herramienta no genere molestias al programador reportando falsos errores.

- **Crear, Modificar y buscar patrones a través la interface de usuario**. Estas funcionalidades, que potencialmente serían útiles para documentar y entender el sistema, podrían agregarse al modelo de patrones para facilitar su uso. Idealmente, sería bueno poder visualizar gráficamente el mapeo entre los actores de una instancia de un patrón y los roles existentes en la definición del patrón (de forma análoga a la que se muestra en el dibujo de la sección 5.2

---

[12] En el patrón Visitor habría 2 roles principales de tipo HierarchyRole (5.3): 'Elements' y 'Visitors'. Análogamente a cómo se definió la fórmula para el patrón Decorator en 5.4.1, la fórmula conceptual del patrón Visitor se podría expresar como una AND de las siguientes 3 condiciones:

- **`1) MessageImplemented('visit'+Elements, Visitors)`**: esta relación verifica que los mensajes visitElementA, visitElementB, etc estén implementados en la jerarquía de Visitors. La expresión 'visit'+Elements se podría modelar agregando al framework un objeto de tipo ProtocolExpression que reciba un String y una jerarquía de clases y que al evaluarse devuelva un objeto de tipo Protocol con los mensajes visitElementA, visitElementB, etc (ver sección 5.8.4)
- **`2) MessageImplemented(accept, Elements)`**: esta relación verifica que mensaje `accept` está implementado en la jerarquía de `Elements`
- **`3) ∀ met ∈ MessageImplementations(accept, Elements):`**
  `SendMessage(met, FirstArgument(met), 'visit' + Class(Met) )`

  Esta fórmula verifica para cada método `met` que implementa el mensaje `accept` en la jerarquía de `Elements se` tiene que cumplir con la siguiente estructura ilustrada para la clase `ConcreteElementA`

```
ConcreteElementA>>acceptVisitor: aVisitor
      ... "algún código opcional"
      aVisitor visitConcretElementA: self
      ... "algún código opcional"
```

- **Manipular instancias de patrones a través de operaciones específicas del tipo de patrón**. En el framework actual, las acciones que propone el framework surgen como respuesta a acciones previas tomadas por el programador. Este mecanismo que se describió en la sección 5.7 se utiliza para generar recomendaciones al programador y también podría utilizarse en futuros patrones que se agreguen al Framework. Por ejemplo, si el programador agrega una nueva clase que represente un estado en el patrón State, el framework podría detectar esta situación y proponer agregar a la nueva clase el conjunto de métodos que el Contexto delega en el State. Para implementar esto, habría que subclasificar el PatternBuilder, definiendo el StateBuilder y en este nuevo Builder habría que definir las acciones correspondientes para interactuar con el entorno (ver sección 5.7.2.2). Alternativamente, esta misma funcionalidad se podría hacer más visible al programador, a través de menús contextuales. Por ejemplo, si el programador puede navegar por los patrones existentes y ver los actores de dichos patrones, al pararse en una instancia del patrón State, le podría aparecer la opción para agregar un nuevo estado.

- **Incorporar validación de reglas de Diseño.** Como el Framework presentado implementa mecanismos para verificar propiedades lógicas sobre secciones arbitrarias del código, estos mecanismos se podrían utilizar no solo para validar el uso correcto de los patrones, sino también para verificar reglas de diseño. Por ejemplo, se podrían verificar reglas como que el mensaje `#initialize` sólo puede ser enviado a una instancia desde una clase.

- **Visualizar las colaboraciones asociadas a los patrones de diseño**. Si el framework de patrones se pudiera integrar de una manera adecuada con una herramienta que muestre gráficamente cómo los objetos colaboran, se podrían mostrar también las colaboraciones propias de la utilización de un patrón de diseño. Esto ayudaría a entender y describir más fácilmente la dinámica del sistema en general, y las colaboraciones definidas por la aplicación de un patrón de diseño en particular.

8 Apéndice

En esta sección se presentarán los siguientes materiales:

- **Resumen del patrón Visitor**. Este patrón es referenciado en algunos puntos de la presente tesis. Para una descripción del patrón Decorator se puede consultar la sección del *"Caso de Estudio"* (3.1). La descripción de los patrones que se realiza en la presente tesis es la misma que la existente en [Alper98], que a su vez coincide con la de [Gamma95], sólo que en [Alper98] se utiliza Smalltalk.

- **Setup del Entrono**. Se detallan los pasos a seguir para instalar el Framework de Patrones sobre el entorno de Visual Works

8.1 Patrón Visitor

Intención

Representar una operación a ser ejecutada sobre los elementos de una estructura de objetos. El uso del patrón *Visitor* permite definir nuevas operaciones sin cambiar las clases de los elementos sobre los cuales las operaciones trabajan.

Aplicabilidad

Usar el patrón Visitor cuando:

- una estructura de objetos contiene muchas clases de objetos con diferentes interfaces y se quiere ejecutar operaciones sobre estos objetos que dependen de su clase concreta
- cuando la extensión por subclasificación es impráctica. Algunas veces, es posible realizar un gran número de extensiones independientes y esto produciría una explosión de subclases para soportar cada combinación posible.

Estructura

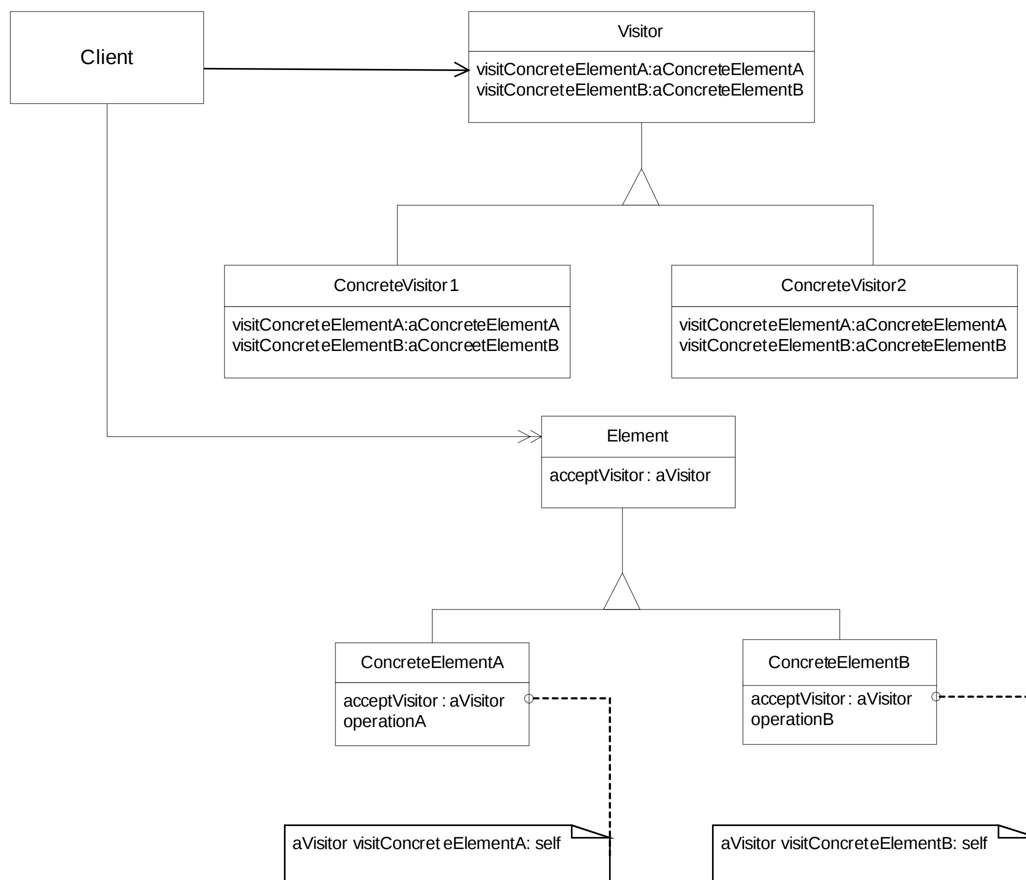


Figura 21 - Estructura patrón Visitor

Participantes

- **Visitor**: declara un mensaje *visitXXX* por cada clase existente en la estructura de objetos. El nombre del mensaje y la signatura identifican la clase que envía el requerimiento de visita al *Visitor*. Esto le deja al visitor determinar la clase concreta del elemento que está siendo visitado. De esta forma, el *Visitor* puede acceder al elemento directamente a través de su interface particular.
- **ConcreteVisitor**: cada ConcreteVisitor implementa una operación específica que se ejecuta sobre la estructura de Elementos. Cada método definido en un ConcretVisitor (visitConcretElementA, visitConcretElementB, etc) implementa el fragmento del algoritmo correspondiente a la clase pasada como parámetro. ConcreteVisitor proveé el contexto para la ejecución del algoritmo almacenando el estado de ejecución de éste. Este estado frecuentemente acumula resultados durante el recorrido de la estructura de elementos.
- **Element**: define un mensaje *acceptVisitor* que tiene un *Visitor* como parámetro.
- **ConcretElement**: implementa el mensaje *accepVisitor* enviándole al *Visitor* un mensaje cuyo nombre codifica la clase del Elemento que está siendo visitado.

Colaboraciones

- Un cliente que usa el patrón Visitor debe crear un *ConcreteVisitor* que será el encargado de visitar a los *Elements*
- Cuando un elemento es visitado, el elemento envía al *Visitor* el mensaje que corresponde a su clase. El elemento se pasa a sí mismo como parámetro del mensaje para dejar que el *Visitor* acceda a su estado si esto es necesario. El Visitor podría usar los mensajes operationA y operationB cuando visita a ConcreteElementA y ConcreteElementB respectivamente.

8.2 Setup del entorno

A continuación se detallan los pasos a seguilr para instalar el Framework de Patrones sobre el entorno de Visual Works 7.5 [13]

1) **Instalar Visual Works 7.5**. El instalador se encuentra en la carpeta *"Visual Works 7.5 installer"* del DVD entregado (installWin para Windows). Observar que el framework ha sido testeado únicamente en esta versión de Visual Works [14]

[13] La instalación más sencilla y recomendada es mediante el uso de la imagen provista. De todas formas, más adelante se describe una forma de instalación alternativa importando el código, asumiendo que ya se tiene Visual Works instalado

[14] Visual Works se pueden obtener de http://www.cincomsmalltalk.com/scripts/DownloadInstaller.ssp)

2) Copiar la imagen de la tesis. Copiar los archivos `tesis-is.im` y `tesis-is.cha` existentes en el directorio *"image"* del DVD entregado al directorio de imágenes de Visual Works. Por ejemplo

```
C:\Program Files\Cincom\vw7.5nc\image
```

3) Configurar acceso directo para ejecutar Visual Works con la imagen de la tesis

- Crear un acceso directo al ejecutable de Visual Works `"C:\Program Files\Cincom\vw7.5nc\bin\win\vwnt.exe"`

- Editar en el acceso directo las siguientes propiedades

  TARGET = `"C:\Program Files\Cincom\vw7.5nc\bin\win\vwnt.exe" "c:\Program Files\Cincom\vw7.5nc\image\tesis-is.im"`

  (Esto se hace para que Visual Works se inicie con la imagen de la tesis)

  START IN = `"c:\Program Files\Cincom\vw7.5nc\image"`

4) Ejecutar Visual Works. En el Transcript se encontrarán diversos Demos de código para interactuar con el Framework. Por ejemplo para crear y validar un patrón:

```
"==================  DEMO INSTANCIACION DE UN PATRON ========== "
TesisIS.PatternManager clear.
TesisIS.DecoratorBuilder newWithComponent:  TesisIS.ComponentExample
                                   decorator: TesisIS.DecoratorExample
                                  instanceVarComponent: 'component'


"================== DEMO VALIDACION DE UN PATRON  ============"
| results |
results := TesisIS.PatternManager  patterns collect: [ :pattern |
                                                    pattern validate ].
( (results at: 1) formattedStringDisplayingTrueEvaluationResults: false )
inspect
```

Instalación alternativa importando código fuente

Requiere Visual Works 7.5 instalado

1. **Cargar parcela de expresiones regulares**

   - Entrar al `Parcel Manager` desde el menú `System`
   - En el tab `Suggestions` esta parcela aparece en la categoría `AdvancedUtilities` y su nombre es `Regex11 [1.2.2]`

2. **Importar código fuente**. Hacer `fileIn` de los archivos `TesisIS.st y TesisIS-Test.st` existentes en el directorio "src" del DVD entregado, ejecutando desde el `Transcript` comandos análogos a los siguientes

```
(Filename named: 'C:\Tesis\TesisIS.st') fileIn
(Filename named: 'C:\Tesis\TesisIS-Test.st') fileIn
```

El archivo `TesisIS.st` contiene todo el código del framework de patrones y `TesisIS-Test.st` contiene los test unitarios y clases para mostrar la interacción del framework con el ambiente de desarrollo

3. **Modificaciones en clases de Visual Works**. Para que el framework funcione correctamente es necesario realizar ciertas modificaciones en clases de Visual Works

- Cambiar la implementación del método `RefactoryChange>>execute por`

```
RefactoryChange>>execute
   | answer |¡
   (TesisIS.PatternManager preProcessRefactoryChange: self) ifTrue: [
         answer := self executeNotifying: [].
         TesisIS.PatternManager postProcessRefactoryChange: self.
   ].
    ^ answer
```

- Agregar el siguiente método en la clase `AddMethodChange`

```
AddMethodChange>>source
   ^ source
```

- Agregar la variable de instancia `doPatternValidation` en la clase `RemoveMethodChange`

- Agregar los siguientes accesors para dicha variable

```
doPatternValidation
         doPatternValidation isNil ifTrue: [^ true ]
                                   ifFalse: [^ doPatternValidation ]

doPatternValidation: aBoolean
   doPatternValidation := aBoolean
```

- Agregar el siguiente método de clase en `RemoveMethodChange`

```
remove: aSymbol from: aClass doPatternValidation: aBoolean
   ^(self new)
         changeClass: aClass;
         selector: aSymbol;
         doPatternValidation: aBoolean
         yourself
```

9 Referencias

Bibliografía básica sobre el Paradigma de POO

[Aho86]	Alfred Aho, Ravi Sethi, Jeffry Ullman, *"Compilers: principles, techniques and tools"* Addison-Wesley 1986
[Alper98]	Sherman R. Alpert, Kyle Brown, Bobby Woolf *"The Design Patterns SmallTalk Companion"* , Addison-Wesley 1998
[Fowl99]	Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts *"Refactoring Improving the Design of Existing Code"*, Addison-Wesley 1999
[Gamma95]	E. Gamma, J.Vlissides, Jonhson, Helm *"Design Pattern, Elements of Reusable Object-Oriented Software"*, Addison-Wesley 1995
[Gold86]	Adele Goldberg y David Robson. *"Smalltalk-80: The Language"*. Second Edition, Addison-Wesley Publishing Company, 1986
[Kiczalez]	G. Kiczales, CSC-517 *Object Oriented Languages and Systems*, North Carolina State University, Fall 1996, Notas del curso.
[Parmas]	D. Parmas, *On the Criteria to Be used in Decomposing Systems into Modules*, en Classics in Software Engineering, Yourdon Press, 1979.
[Rumb91]	Rumbaugh, Blaha, Premerlani, Edy, Lorensen *"Object-Oriented Modeling and Design"*, Prentice Hall,1991
[Stein et. al]	L. Stein, H. Liberman, D. Ungar, *"A Shared View of Sharing: The Treaty of Orlando"*
[Ungar87]	David Ungar, and Randall B.Smith, "*Self: The power of simplicity*", in OOSPLA '87 Conference Proceedings, ACM Press, Orlando, 1987 La home page de Self en http://research.sun.com/research/self/
[Wirf90]	Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener. *"Designing Object-Oriented Software"*. Englewood Cliffs, NJ, Prentice Hall, 1990.

Bibliografía y específica sobre Patrones de Diseño

[Alex79]	Alexander, Christopher. *"A Timeless Way of Building"* New York, Oxford University Press, 1977
[Ager97]	Ellen Agerbo, Aino Cornilis *"Theory of Language Support for Design Patterns"* Computer Science Department, AarhusUniversity, Denmark, 1997
[Ager98]	Ellen Agerbo, Aino Cornilis *"How to preserve the benefits of design patterns"*
[Bud96]	F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu *"Automatic code generation from design patterns"*. En http://www.research.ibm.com/journal/sj/352/budinsky.html

[Eden98]	Ammon H. Eden, Technical report 326/98 department of computer science, Tel Aviv University *"LePUS - A Declarative Pattern Specification Language".*
[Eden00]	Ammon H. Eden *"Precise Specification of Design Patterns and Tool Support in Their Application"* (http://www.eden-study.org/publications.html)
[Eden00b]	Ammon H. Eden *"Giving 'The Quality' a Name - Precise Specification of Design Patterns: A Second Look at the Manuscripts".* Journal of Object Oriented Programming, Guest Column, Vol. 11, No. 3 (Jun. 1998) (http://www.eden-study.org/publications.html)
[Eden02]	Ammon H. Eden *"Theory of Object-Oriented Design"* (http://www.eden-study.org/publications.html)
[Meij96]	Marco Meijers, Gert Florijn *"Tool Support for Object-Oriented Design Patterns"* april 1996. (http://www.serc.nl/people/florijn/work/patterns.html)
[Grui97]	Dennis Gruijs, *"A Framework of Concepts for Representing Object-Oriented Design and Design Patterns"* Masters Thesis, Utrecht University, CS Dept., INF-SCR-97-28, November 1997. (http://www.serc.nl/people/florijn/work/patterns.html)
[Hedin96]	Gorel Hedin *"Enforcing programming conventions by attribute extension in an open compiler".* Lund Institute of Technology, Lund University, May 1996 (http://www.cs.lth.se/Research/ProgEnv/LSDF.html)
[Hedin97]	Gorel Hedin *"Language Support for Design Patterns Using Attribute Extension".* Computer Science Department, Aarhus University Ny Munkegade, May 1997 (http://www.cs.lth.se/Research/ProgEnv/LSDF.html)
[Hedin00]	A. Cornils and G. Hedin. *"Tool Support for Design Patterns based on Reference Attribute Grammars"* . Proc. of WAGA'00, Ponte de Lima, Portugal, July 2000 (http://www.cs.lth.se/Research/ProgEnv/LSDF.html)
[Hedin00b]	A. Cornils and G. Hedin. *"Statically Checked Documentation with Design Patterns"* . Proc. of TOOLS Europe 2000, Mont st. Michel, France, June 2000 (http://www.cs.lth.se/Research/ProgEnv/LSDF.html)
[Ker04]	Joshua Kerievsky, *"Refactoring to Patterns"* Addison-Wesley Agosto 2004
[Same02]	J. Sametinger y R. Keller *"Compositional Design Reuse"* . CACIC 2002, VIII Argentinean Conference on Computer Science, Universidad de Buenos Aires, Argentina, October 15-18, 2002. (http://www.swe.uni-linz.ac.at/publications/abstract/TR-SE-02.08.html)
[Tab00]	Tabita Enig, Henrik Kjaer Nielsen, Lars Mellergaard *"DPDOC - a user guidance and evaluation."* Institute of Computer Science University of Aarhus, May 2000
Vlis98]	John Vlissides, *"C++ Report"* abril 1998 http://www.reearch.ibm.com/designpatterns/pubs/ph-apr98.pdf