Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Tesis de Licenciatura

Definición y Composición Dinámica de Metacomportamiento en Ambientes de Objetos con Clasificación

Alumna: María Alejandra De Bonis
LU 186/93
alejandradb@ciudad.com.ar

Directores:
Lic. Máximo Prieto
Lic. Hernán Wilkinson

Noviembre 2005

Agradecimientos

Quiero agradecer a Hernán Wilkinson por haberme propuesto el tema de esta Tesis y haberme dirigido y guiado para llegar a su conclusión. También agradezco a Máximo Prieto por haber aceptado participar en la dirección y haber hecho posible que pudiera hacer un trabajo de investigación en el área de la Programación Orientada a Objetos.

Agradezco a Carlos, mi marido, que con su amor, sus consejos, sugerencias y paciencia me ayudó en gran medida a cumplir este objetivo.

Doy gracias a mis padres, Luis e Isabel. Y a mis hermanos, Leila y Javier. Sin su gran cariño y contención me hubiera sido mucho más difícil llegar hasta el fin.

No puedo dejar de mencionar a Julia, que siempre estuvo dispuesta a escucharme y me ayudó a transitar el período que dediqué a la preparación de este trabajo con más confianza y seguridad.

Finalmente, agradezco a todos mis amigos y familiares que de una u otra manera me apoyaron y animaron para que esto pudiera concretarse.

Resumen

Los ambientes que soportan la programación orientada a objetos y en los que las clases son tratadas como objetos permiten, en mayor o menor grado, redefinir comportamiento de las clases trabajando a nivel de metaclases (clases cuyas instancias son clases).

Con las clases y sus relaciones de herencia se intenta modelar un dominio o aplicación particular. Identificaremos como propiedad de una clase (o *metapropiedad*) aquel comportamiento asignado a la misma que no sea esencial a este dominio particular, es decir, que pueda ser definido independientemente de la funcionalidad de la aplicación (por ejemplo: propiedad abstracta, singleton, inicialización automática, el traceo de determinados métodos, etc.).

No siempre es deseable que las propiedades de una clase se propaguen a sus subclases. Por el contrario, en muchos casos se necesita definir una propiedad para una clase sin pasarla a sus subclases. Este inconveniente se identifica como el *problema de la propagación de las propiedades de clase*.

Otro tema que surge al tratar las clases como objetos es el de la comunicación entre niveles: el nivel de las clases y el de sus instancias. Una característica deseable en todo modelo que implemente metaclases es la de garantizar la compatibilidad de la comunicación entre los distintos niveles al armarse las jerarquías de clases y de metaclases.

En este trabajo estudiamos las metapropiedades desde un punto de vista conceptual e implementativo.

Esto lo logramos mediante la categorización y reificación del concepto de metapropiedades, llegando a un modelo de objetos que tiene las siguientes características:

- **Asignación y combinación dinámica de propiedades.** Esto es, permite asignar una o más metapropiedades a una clase dinámicamente. Esto se logra por la existencia de objetos que modelan las propiedades simples y el uso del mecanismo de composición en contraposición al uso de herencia, tanto para asignar una propiedad como para combinarlas en una clase. De esta manera, contando con un conjunto de propiedades básicas y ensamblándolas de la manera adecuada, se consigue que una clase cuente con las propiedades buscadas.
- **Reuso.** La lógica de una determinada propiedad quede expresada en uno y sólo un lugar, pudiendo ser reusada.
- **Solución al problema de propagación de propiedades de clase.**
- **Compatibilidad de comunicación entre niveles**.
- **Validación dinámica de las propiedades.**

Dadas sus características de reflexividad, este modelo se implementó en Smalltalk, en el ambiente de objetos de Squeak y se desarrollaron herramientas para facilitar su uso.

Abstract

The definition and modification of the behaviour of classes by working at metaclass level, is a common practice in object oriented programming environments that have classes as *first class objects.*

Classes and inheritance are used for building models of business domains or particular applications. We identify as class property (or *metaproperty*) the class behaviour that is not essential for that particular domain, focusing on things that are not part of the application's functionality (e.g. abstract quality, singleton, automatic initialization, trace for some methods, etc.).

Propagation of properties of a class to its subclasses is not always desirable. Indeed, there are many cases in which defining a class property without passing it to its subclasses is needed. This is known as *class property propagation issue*.

When having classes as first class objects, another issue arises: communication between class and instance levels. Granting communication compatibility among the different levels when building class and metaclass hierarchies, is a sound and appealing characteristic for a metaclass model.

Our main purpose in the present work is studying metaproperties from conceptual and implementation viewpoints.

We achieve this goal by means of classification and reification of the metaproperty concept, thus obtaining an object model with the following features:

- **Dynamic binding and combination of metaproperties**. It allows assigning one or more properties to a class in a dynamic way. There are some objects that model simple properties and we use composition mechanisms -in opposition to inheritance- in order to bind or merge properties for a class. In our proposal, a class can acquire desired properties by assembling basic properties in a proper way.
- **Reusability**. Logic and implementation of a particular property reside in one -and only in one- place. Property reuse is a natural side effect.
- **Class property propagation issue successfully addressed**.
- **Full compatibility of communication among class and instance levels.**
- **Dynamic validation of property binding and combination.**

This model was fully implemented in Smalltalk, profiting its wide reflection capabilities. Squeak dialect was the choice, and tools for easy exploitation of metaproperties were developed and integrated into its environment.

Índice

1 Introducción 11
1.1 Motivación 11
1.2 Resultados 12
2 Conceptos básicos de los ambientes de programación orientada a objetos 13
2.1 Objetos y mensajes 13
2.2 Mecanismos de “sharing”: clasificación y prototipación 14
2.3 Herencia y composición 15
3 Metaclases y propiedades de clases 17
3.1 Metaclases 17
3.2 Propiedades de las Clases 17
3.2.1 Problemáticas a considerar relacionadas con las propiedades de clase 19
3.2.1.1 Combinación o composición de propiedades de clase 19
3.2.1.2 Problema de propagación de las propiedades de clase 19
3.2.1.3 Comunicación entre niveles de clase e instancia y problemas de compatibilidad 19
3.2.1.3.1 Comunicación entre niveles de clase e instancia 19
3.2.1.3.2 Problemas de Compatibilidad 20
3.3 Conclusiones 22
4 Modelos actuales de ambientes que manejan metaclases 23
4.1 Metaclases en Smalltalk 23
4.1.1 Diseño del kernel de clases / metaclases 24
4.1.1.1 Descripción y responsabilidades de clases que conforman el kernel de clases /metaclases 27
4.1.2 Cómo se asignan y combinan propiedades de clase 28
4.1.3 Cómo se asignan propiedades a clases de la jerarquía que modela lógica booleana 28
4.1.4 Compatibilidad de comunicación entre niveles 29
4.1.5 Propagación de propiedades de clase 30
4.2 Metaclases en CLOS (Common Lisp Object System) 31
4.2.1 Diseño del kernel de clases y metaclases 32
4.2.2 Cómo se asignan y combinan propiedades de clase 32
4.2.3 Cómo se asignan propiedades a clases de la jerarquía que modela lógica booleana 33
4.2.4 Compatibilidad de comunicación entre niveles 33
4.2.5 Propagación de propiedades de clase 33
4.3 Metaclases en ObjVlisp 35
4.3.1 Diseño del kernel de clases y metaclases 35
4.3.2 ClassTalk: Implementación de ObjVlisp en Smalltalk-80 35
4.3.3 Cómo se asignan y combinan propiedades de clase 36
4.3.4 Cómo se asignan propiedades a clases de la jerarquía que modela lógica booleana 36
4.3.5 Compatibilidad de comunicación entre niveles 36
4.3.6 Propagación de propiedades de clase 37
4.4 Metaclases en SOM (System Object Model) 38
4.4.1 Diseño del kernel de clases y metaclases 39
4.4.2 Cómo se asignan y combinan propiedades de clase 39

4.4.3 Cómo se asignan propiedades a clases de la jerarquía que modela lógica booleana ... 39
4.4.4 Compatibilidad de comunicación entre niveles ... 40
4.4.5 Propagación de propiedades de clase ... 40
4.5 Metaclases en modelo de compatibilidad implementado en NeoClasstalk ... 42
4.5.1 Diseño del kernel de clases y metaclases ... 42
4.5.2 Cómo se asignan y combinan propiedades de clase ... 43
4.5.3 Cómo se asignan propiedades a clases de la jerarquía que modela lógica booleana ... 44
4.5.4 Compatibilidad de comunicación entre niveles ... 45
4.5.5 Propagación de propiedades de clase ... 45
4.6 Modelo de mixins y metaclases en MetaclassTalk ... 46
4.6.1 Diseño del kernel de clases y metaclases ... 48
4.6.2 Cómo se asignan y combinan propiedades de clase ... 49
4.6.3 Cómo se asignan propiedades a clases de la jerarquía que modela lógica booleana ... 49
4.6.4 Compatibilidad de comunicación entre niveles ... 50
4.6.5 Propagación de propiedades de clase ... 51
4.7 Metaclases y el modelo de traits ... 52
4.7.1 Diseño del kernel de clases y metaclases ... 52
4.7.2 Cómo se asignan y combinan propiedades de clase ... 53
4.7.3 Cómo se asignan propiedades a clases de la jerarquía que modela lógica booleana ... 54
4.7.4 Compatibilidad de comunicación entre niveles ... 55
4.7.5 Propagación de propiedades de clase ... 55
4.8 Resumen del análisis de los modelos ... 56
5 Reificación del concepto de propiedad de clase ... 57
5.1 ¿Por qué crear un nuevo modelo de metapropiedades? ... 58
5.2 ¿Por qué es importante contar con un modelo de metapropiedades como el que se propone? ... 62
6 Elección del ambiente de trabajo e investigación ... 64
6.1 Metaprogramación ... 64
6.2 ¿Por qué Smalltalk? ... 65
6.3 ¿Por qué Squeak? ... 65
7 Definiciones y diseño de modelo de metapropiedades ... 66
7.1 Composición y aplicación de metapropiedades ... 66
7.1.1 Clasificación de propiedades y estrategias definidas por metapropiedades ... 66
7.1.1.1 Estrategia de creación de instancias ... 68
7.1.1.2 Estrategia de Recepción de Mensajes ... 69
7.1.1.3 Clases que modelan estrategias de propiedades y protocolo para manejar su composición y aplicación 75
7.1.2 Metapropiedades ... 77
7.1.2.1 Clases que modelan metapropiedades y protocolo para manejar su aplicación y ejecución ... 78
7.1.2.2 Metapropiedades de creación de instancias y su protocolo para manejar su aplicación y ejecución ... 79
7.1.2.3 Metapropiedades de recepción de mensajes y su protocolo para manejar su aplicación y ejecución .. 81
7.2 Asignación de metapropiedades ... 85
7.2.1 Construcción de estrategias de metapropiedades ... 85
7.2.1.1 Agregado de una metapropiedad ... 85

7.2.1.2 Eliminación de una metapropiedad ... 87
7.2.1.3 Definición y protocolo de *builders* de estrategias de metapropiedades ... 88
7.2.1.4 Comportamiento de estrategias de metapropiedades relacionado con la asignación y construcción ... 91
7.2.1.5 Comportamiento de metapropiedades relacionado con la asignación ... 93
7.2.2 Categorías de metapropiedades ... 96
7.2.2.1 Cómo se relacionan las propiedades y categorías ... 96
7.2.2.2 Clase que modela las categorías de metapropiedades ... 96
7.2.2.3 Relaciones de orden entre categorías de metapropiedades ... 97
7.2.2.4 Incompatibilidades entre categorías de metapropiedades ... 98
7.2.2.5 Cómo se definen relaciones de orden e incompatibilidades entre categorías de metapropiedades ... 99
7.2.3 Cómo se propagan las propiedades definidas a nivel de clases al subclasificar ... 101
7.3 Modificación de metapropiedades de un objeto usando modelo definido ... 103
7.3.1 Clases que modelan la modificación de metapropiedades ... 103
7.3.2 Protocolo de propiedades relacionado con los objetos de la jerarquía de *MetaPropertiesModification* . 105
7.4 Cómo se alcanzan los objetivos propuestos en este modelo ... 106
8 Interfaces de usuario para trabajar con metapropiedades en el ambiente de Squeak ... 109
8.1 Diseño de interfaces de usuario ... 109
8.1.1 Interfaces de propiedades de recepción de mensajes de un objeto ... 109
8.1.1.1 Agregado de propiedades ... 110
8.1.1.2 Eliminación de propiedades ... 111
8.1.2 Interfaces de propiedades de recepción de mensajes definidas para las instancias de una clase ... 112
8.1.3 Interfaces de propiedades de creación de instancias ... 113
8.2 Modelo de browsers de metapropiedades ... 115
8.2.1 Jerarquía de clases de *MetaPropertyBrowser* ... 116
8.2.2 Objetos y protocolo con los que se relacionan los browsers de metapropiedades ... 121
8.2.2.1 Jerarquía de clases de *MetaPropertiesCatalog* ... 121
8.2.2.2 Definición de atributos de metapropiedades ... 121
8.2.2.3 Atributos de metapropiedades ... 122
8.2.2.4 Editores de atributos de metapropiedades ... 124
8.2.2.5 Colaboraciones con objetos de jerarquía de *MetaPropertiesModification* ... 125
9 Cómo extender el modelo existente para implementar otras metapropiedades ... 126
9.1 Definición de una nueva metapropiedad relacionada con una funcionalidad ya reificada ... 126
9.2 Definición de una nueva estrategia de metapropiedades ... 128
10 Modificación de máquina virtual para reificar la recepción de mensajes ... 130
10.1 Máquina virtual de Smalltalk ... 130
10.2 Máquina virtual de Squeak ... 130
10.3 Modificación del formato de los objetos ... 131
10.4 Modificación de implementación de envío y recepción de mensajes ... 133
10.5 Límites de las modificaciones para reificar la recepción de mensajes ... 134
11 Conclusiones ... 136
12 Trabajo futuro ... 138
13 Bibliografía y referencias ... 140
14 Apéndice A – Notación gráfica utilizada ... 142

14.1 Diagrama de instancias (o de objetos) ... 142
14.2 Diagrama de secuencia (o de colaboración) ... 142
14.3 Diagrama de Clases ... 143
14.4 Diagramas de clases y metaclases ... 145

Índice de figuras

Figura 3.1: Diagramas que ejemplifican la necesidad de asegurar upward compatibility 20
Figura 3.2: Diagramas que ejemplifican la necesidad de asegurar downward compatibility 21
Figura 4.1: Paralelismo entre jerarquías de clases y metaclases en Smalltalk 24
Figura 4.2: Clase Metaclass y sus relaciones de instanciación con las clases y su metaclase en Smalltalk 24
Figura 4.3: Todas las metaclases son subclases de Class en Smalltalk 25
Figura 4.4: Jerarquía de clases que modela las metaclases 26
Figura 4.5: Modelo de clases y metaclases en Smalltalk 26
Figura 4.6: Asignación de propiedades a las clases que modelan lógica booleana en Smalltalk 28
Figura 4.7: Diagrama que refleja que en Smalltalk se asegura la compatibilidad upward 29
Figura 4.8: Diagrama que refleja que en Smalltalk se asegura la compatibilidad downward 30
Figura 4.9: Relación por defecto entre las clases y sus metaclases en Clos 31
Figura 4.10: Posible relación entre clases y sus metaclases en Clos al eliminar restricción por defecto 31
Figura 4.11: Asignación de propiedades a las clases que modelan lógica booleana en Clos 33
Figura 4.12: Cómo evitar propagación de la propiedad abstracta en Clos 34
Figura 4.13: Diseño del kernel de clases y metaclases en ObjVlisp 35
Figura 4.14: Asignación de propiedades a las clases que modelan lógica booleana en ObjVlisp 36
Figura 4.15: Definición de clases y metaclases en SOM 38
Figura 4.16:Creación de metaclase derivada en SOM 38
Figura 4.17: Asignación de propiedades a las clases que modelan lógica booleana en SOM 39
Figura 4.18: Compatibilidad upward por creación de metaclases derivadas en SOM 40
Figura 4.19: Metaclases de compatibilidad y de propiedades en NeoClasstalk 42
Figura 4.20: Diseño de propiedades de clases en NeoClasstalk 43
Figura 4.21: Asignación y combinación de propiedades en NeoClasstalk 44
Figura 4.22: Asignación de propiedades a las clases que modelan lógica booleana en NeoClasstalk 44
Figura 4.23: Las metaclases de propiedades no introducen problemas de incompatibilidad en NeoClasstalk 45
Figura 4.24: Metaclases de compatibilidad y mixins en MetaclassTalk 47
Figura 4.25: Jerarquía de herencia simple a partir de jerarquía de herencia múltiple de mixins en MetaclassTalk 48
Figura 4.26: Asignación de propiedades a las clases que modelan lógica booleana en MetaclassTalk 49
Figura 4.27: Jerarquía de herencia simple de modelo de lógica booleana a partir de jerarquía de herencia múltiple de mixins 50
Figura 4.28:Diseño del kernel de clases y metaclases en el modelo de traits 53
Figura 4.29: Asignación de propiedades a las clases que modelan lógica booleana en el modelo de traits 54
Figura 7.1: Estrategias de metapropiedades de una clase 67
Figura 7.2: aClass tiene definida una estrategia de creación de instancias 68
Figura 7.3: Ejecución y aplicación de una estrategia de creación de instancias 69
Figura 7.4: anObject tiene definida una estrategia de recepción de mensajes con propiedades definidas para determinados mensajes 70
Figura 7.5:anObject tiene definida una estrategia de recepción de mensajes con propiedades que se aplican a cualquier mensaje 71
Figura 7.6: Ejecución y aplicación de estrategia de recepción de mensajes definida con propiedad trace para el mensaje m1 71
Figura 7.7: Ejecución y aplicación de estrategia de recepción de mensajes definida con propiedades trace y halt para mensaje m1 72

Figura 7.8: aClass tiene definida una estrategia de recepción de mensajes para sus instancias.. 72

Figura 7.9: Determinación de estrategia de recepción de mensajes de un objeto cuya clase tiene definida una estrategia para sus instancias... 73

Figura 7.10: Creación de una estrategia de recepción de mensajes para un objeto en particular a partir de la estrategia definida para las instancias de la clase de este objeto 74

Figura 7.11: Jerarquía de MetaPropertyPolicy con protocolo para manejar su composición y aplicación........................ 75

Figura 7.12: Jerarquía de MetaProperty con protocolo para aplicación 78

Figura 7.13: Jerarquía de InstanceCreationProperty y su protocolo para aplicación 80

Figura 7.14: Jerarquía de MessageReceptionProperty y su protocolo para aplicación... 82

Figura 7.15: Agregado de una metapropiedad a la estrategia correspondiente de anObject... 85

Figura 7.16: Un builder que construye una estrategia de a partir de otro que incluye a una nueva propiedad 86

Figura 7.17: Eliminación de una metapropiedad de la estrategia de anObject 87

Figura 7.18: Un builder que construye una nueva estrategia a partir de otra eliminando una de sus propiedades 87

Figura 7.19: Diagrama de instancias de un builder de metapropiedades.. 88

Figura 7.20:Jerarquía de builders de estrategias de metapropiedades 89

Figura 7.21: Jerarquía de estrategias de metapropiedades y su protocolo para agregar y eliminar propiedades 91

Figura 7.22: Jerarquía de metapropiedades y su protocolo para asignación... 93

Figura 7.23: Clase MetaPropertyCategory 96

Figura 7.24: Clase MetaPropertyCategoryOrder.. 97

Figura 7.25: Clase MetaPropertyCategoryIncompatibility... 98

Figura 7.26: Jerarquía de MetaPropertyCategoryRelationshipDefinition 99

Figura 7.27: Clase ClassPropertyPolicyBuilderWhenSubclassing.. 101

Figura 7.28: Jerarquía de MetaPropertyModification 103

Figura 7.29: Jerarquía de MetaProperty y su protocolo relacionado con objetos de jerarquía de MetaPropertyModification 105

Figura 7.30: Asignación de propiedades a las clases que modelan lógica booleana en modelo propuesto 107

Figura 8.1: Jerarquía de clases de MetaPropertyBrowser... 116

Figura 8.2: Método addProperty definido en MetaPropertyBrowser 117

Figura 8.3: Método createPropertyFromClass: definido en MetaPropertyBrowser 117

Figura 8.4: Método getValuesOfAttributes: definido en MetaPropertyBrowser... 118

Figura 8.5: Método removeProperty: definido en MetaPropertyBrowser 118

Figura 8.6: Jerarquía de MetaPropertiesCatalog 121

Figura 8.7: Jerarquía de MetaPropertyAttributesDefinition 121

Figura 8.8: Jerarquía de AbstractMetaPropertyAttribute 123

Figura 8.9: Jerarquía de MetaPropertyAttributeEditor.. 124

1 Introducción

1.1 Motivación

En el paradigma de programación orientada a objetos en su forma pura, sólo existen objetos y los mensajes que se envían entre ellos.

En los ambientes de programación orientada a objetos basados en clasificación, todo objeto es instancia de una clase. Las clases son objetos cuyas responsabilidades son la de especificar la descripción e implementación de objetos con características similares, y la de crearlos.

Una clase, al ser un objeto, es instancia de otra clase. Las clases cuyas instancias son clases se conocen como metaclases. Por lo tanto, las metaclases describen tanto la representación interna como el comportamiento de las clases. Por ejemplo, son las metaclases las que definen cómo las clases responden al mensaje de creación de sus propias instancias.

En los ambientes en los que las clases son tratadas como objetos, trabajando a nivel de metaclases se puede, en mayor o menor medida, redefinir comportamiento de las mismas.

Con las clases y sus relaciones de herencia se intenta modelar un dominio o aplicación particular. Identificaremos como propiedad de una clase o metapropiedad aquel comportamiento asignado a la misma que no sea esencial a este dominio particular, es decir, que pueda ser definido independientemente de la funcionalidad de la aplicación [BS/00]. Las metapropiedades representan conceptos propios del dominio de las clases que pueden presentarse recurrentemente.

El interés principal de este trabajo es identificar y categorizar a las metapropiedades para lograr una reificación de este concepto con el objetivo de armar un modelo de objetos que represente este problema y que permita la asignación dinámica de estas propiedades.

Los objetivos que se quieren alcanzar con la existencia de un modelo que permita la asignación dinámica de metapropiedades son los siguientes:

- Que no se mezcle el código que implementa la funcionalidad del dominio particular con el que implementa las metapropiedades. Sin la existencia de este modelo, los objetos y colaboraciones agregados para implementar una metapropiedad enturbian el código, lo enredan y estorban cuando se trata de arreglar o extender algo de la funcionalidad original. El código relacionado con la responsabilidad del objeto está perdido entre la maraña de lo que se agregó. Este problema empeora cuando se trata de implementar más de una metapropiedad, hasta que se hace muy difícil recuperar la idea original de lo que debía hacer el objeto. Gran parte del esfuerzo puesto en el diseño para lograr objetos con funcionalidades y responsabilidades claras y jerarquías reusables y extensibles en un modelo claro, comprensible y autodocumentado, se pierde así en la implementación de las metapropiedades que no respetan la descomposición funcional.

- Que la lógica de una determinada metapropiedad quede expresada en uno y sólo un lugar, pudiendo ser reusada.

- Que el diseñador de una aplicación particular se focalice en la resolución de los problemas propios del dominio que está modelando y no se tenga que preocupar ni por la implementación de las metapropiedades que necesita usar (o por lo menos lo vea como un factor independiente a resolver) ni por cómo el uso de las mismas afectaría su diseño. Es decir, que se puedan determinar en forma transparente e independiente del diseño funcional realizado.

- Que exista, mediante la asignación dinámica de metapropiedades, una mayor libertad para poder agregarlas o quitarlas a voluntad, o inclusive graduarlas; o ponerlas en ciertas partes del sistema y en otras no.

1.2 Resultados

En este trabajo presentamos el diseño e implementación en Smalltalk (en el ambiente de objetos de Squeak) de un modelo de objetos de metapropiedades que tiene las siguientes características:

- **Asignación y combinación dinámica de propiedades.** Permite asignar una o más metapropiedades a una clase dinámicamente. Esto se logra por la existencia de objetos que modelan las propiedades simples y el uso del mecanismo de composición en contraposición al uso de herencia tanto para asignar una propiedad como para combinarlas en una clase. De esta manera, contando con un conjunto de propiedades básicas y ensamblándolas de la manera adecuada, se consigue que una clase cuente con las propiedades buscadas.
- **Reuso.** La lógica de una determinada propiedad quede expresada en uno y sólo un lugar, pudiendo ser reusada.
- **Solución al problema de propagación de propiedades de clase.**
- Compatibilidad de comunicación entre niveles.
- **Validación dinámica de las propiedades.**
- **Creación de una taxonomía de metapropiedades.**
- **Definición de metapropiedades a nivel de instancias.** Dependiendo del tipo de metapropiedad, ofrece la flexibilidad de permitir asignar metapropiedades para determinadas instancias de una clase y no para todo el conjunto, en los casos que así se desee.

Se desarrollaron herramientas para facilitar el uso del modelo implementado en el ambiente de Squeak.

La implementación del modelo resultante constituye un framework de metapropiedades, ya que el modelo fácilmente puede ser extendido para lograr la implementación de metapropiedades que no fueron desarrolladas en este trabajo.

2 Conceptos básicos de los ambientes de programación orientada a objetos

2.1 Objetos y mensajes

En el paradigma de programación orientada a objetos en su forma pura, sólo existen *objetos* y los *mensajes* que se envían entre ellos, es decir que sólo contamos con estas herramientas para modelar nuestros problemas o dominios de aplicación.

Un *objeto* es una abstracción de algún ente del mundo real, entendiendo como abstracción al proceso a través del cual se extraen las características esenciales de un ente y a partir de las cuales se lo puede modelar.

Cada objeto tiene una determinada responsabilidad o funcionalidad que provee a otros objetos. Esto está definido por el conjunto de mensajes que sabe responder y lo que devuelve como respuesta. Este conjunto de mensajes constituye el *protocolo* o *interface* del objeto.

Internamente, el comportamiento de un objeto está dado por los *métodos* a través de los cuales el objeto responde a los mensajes que recibe. Cada método es un conjunto de *colaboraciones* con otros objetos. Las colaboraciones son envíos de mensajes que el objeto manda a otros objetos (o a sí mismo) con la finalidad de producir la respuesta del mensaje.

Para hacer referencia al nombre de los métodos o mensajes se utiliza el término *selector*.

Esos otros objetos que cooperan con el receptor del mensaje para que pueda llevar a cabo su responsabilidad se llaman *colaboradores*. Se puede distinguir dos tipos de colaboradores: internos o habituales y externos o no habituales. Los colaboradores internos son aquellos que el objeto conoce por tenerlos definidos internamente y con los que habitualmente colabora. En general, los lenguajes de programación de este paradigma utilizan el mecanismo de *variables de instancia* para poder establecer estas relaciones de conocimiento. Los colaboradores externos son aquellos que cooperan ocasionalmente con el objeto. Son los que el objeto recibe como parámetros en un mensaje.

Respecto a la relación de conocimiento entre un objeto y sus colaboradores, cabe aclarar que el objeto no conoce directamente a sus colaboradores, sino que sólo conoce nombres a través de los cuales se puede referir a ellos. Sólo se requiere que el objeto que se encuentre tras el nombre sepa responder a los mensajes que el objeto emisor le envíe.

Pueden definirse métodos para acceder a los colaboradores de un objeto, tanto para obtener el objeto al que hacen referencia como para determinar esta referencia. A estos métodos se los suele identificar como *getters* y *setters* respectivamente, y para referirse a ambos se suele usar el término *accessors*.

Según [GR/86], *"un objeto consiste en una memoria privada y un conjunto de operaciones. La única forma de interactuar con un objeto es a través de su interface. Una propiedad esencial de un objeto es que su memoria privada puede ser manipulada solamente por sus propias operaciones. Una propiedad esencial de los mensajes es que son la única forma de invocar las operaciones de un objeto. Estas propiedades aseguran que la implementación de un objeto no pueda depender de los detalles internos de otros objetos, sólo de los mensajes a los cuales éstos responden. Los mensajes aseguran la modularidad del sistema porque especifican el tipo de operación deseada, pero no cómo debe ser efectuada"*.

La capacidad de los objetos de poseer tanto un comportamiento (a través de sus mensajes y métodos) como una representación interna (determinada por sus colaboradores internos) se conoce como *encapsulamiento*, mientras que la imposibilidad de otros objetos de acceder y /o manipular su representación o implementación interna, se conoce como *information hiding*.

Otra característica presente en un ambiente de objetos es el *polimorfismo,* la cualidad mediante la cual distintos objeto pueden responder a un mismo mensaje, pero cada uno a su propio modo. Esto posibilita que se puedan cambiar los colaboradores de un determinado objeto por otros, sin que tengan que modificarse los mensajes que envía a los objetos con los que colabora. Una de las formas a través de la cual se puede contar con polimorfismo es mediante el *binding dinámico*, que permite que la vinculación entre el envío de mensaje y el método que lo implementa se resuelva en tiempo de ejecución.

2.2 Mecanismos de "sharing": clasificación y prototipación

En las implementaciones de este paradigma es necesaria la presencia de algún mecanismo de *sharing*. Esto es, un medio que permita compartir definiciones y comportamiento para que se pueda definir un nuevo objeto en términos de otros que ya existen [SLU/92]. Es decir, una posibilidad de que los mensajes (y la forma de responder a ellos) que un objeto define puedan ser reusados por otro objeto. La creación de objetos nuevos a partir de otros ya existentes es una manera de compartir comportamiento.

Los esquemas más conocidos para compartir comportamiento son de dos tipos: basados en *clasificación* o en *prototipación*.

En los ambientes de objetos basados en clasificación, todo objeto es instancia de una *clase*. Las clases son objetos cuyas responsabilidades son la de especificar la descripción e implementación de objetos con características similares, y la de crearlos. Los objetos creados por una clase se llaman *instancias* de esta clase. Las instancias de una determinada clase comparten la representación interna, es decir tienen las mismas variables de instancia, también utilizan los mismos métodos para responder a los mensajes que reciben. El conocimiento común de los objetos se representa a través de las clases, constituyendo por esto un mecanismo de sharing. Entre las clases se pueden establecer relaciones de herencia, es decir definir a una clase como subclase de otra. La definición de estas relaciones, como se detalla más adelante, también constituye un mecanismo de sharing en estos ambientes.

En los ambientes basados en prototipación, la creación de objetos se realiza por clonación a partir de *prototipos*. Un objeto puede tomar comportamiento de cualquier otro y no existen objetos cuya responsabilidad específica sea la de crear otros objetos.

Los ambientes con prototipos son más flexibles, facilita la programación exploratoria y son más adecuados para dominios poco conocidos que se están investigando. En los ambientes con clases, las mismas constituyen herramientas de abstracción más organizadas que permiten

definir el conocimiento de manera más compacta y eficiente y que se consideran útiles cuando existe una mejor compresión del dominio.

El trabajo de esta tesis se desarrolla en el contexto de ambientes basados en clasificación, por esta razón se continuará con la descripción más detallada de conceptos que tienen sentido en este tipo de ambientes.

2.3 Herencia y composición

La *herencia* es otro mecanismo de sharing que se utiliza en los ambiente basados en clasificación.

La herencia es la capacidad de una clase de definir el comportamiento y la estructura de datos de sus instancias como un superconjunto de las definiciones de otra clase o clases. La herencia permite concebir una nueva clase como un refinamiento de otra, abstraer similitudes entre clases y diseñar y especificar sólo las diferencias para la nueva clase [WWW/90]. En esta relación entre clases, la nueva clase cumple el rol de *subclase* que se define por diferencia con respecto a la *superclase*.

La herencia permite establecer las diferencias entre las clases, así como también compartir lo qué es común entre ellas. Una subclase adquiere automáticamente el conocimiento que define su superclase, y puede redefinir partes de ese conocimiento o incorporar nuevos elementos, propios de la subclase. A la relación que existe entre una subclase A y su superclase B se la reconoce como *"toda instancia de A es un B"*.

Las relación de herencia debe usarse para modelar una relación *"es un"* que sea esencial al dominio del problema que se está modelando. El objetivo de usar la herencia no debería ser el de reusar código, ni aprovechar implementaciones, sino que esto debería ser una consecuencia de haber modelado bien el problema.

Las relaciones de herencia son relativamente estáticas, mucho más difíciles de modificar que las de colaboración. En los ambientes comunes no se pueden modificar las implementaciones heredadas de la clase padre en tiempo de ejecución (o por lo menos no se lo puede hacer de una manera muy directa).

Otro factor que atenta contra el dinamismo al usar herencia es el siguiente. Al usar herencia, la implementación y la estructura de la superclase son conocidas por sus subclases, por ello al reuso por subclasificación se lo suele identificar como *"reuso de caja blanca"* [GHJV/95]. Las dependencias implementativas existentes en esta relación, hacen que el cambio en la implementación de la superclase pueda implicar que se deba modificar las subclases.

Una técnica alternativa para el reuso de funcionalidad es la *composición* de objetos. En este caso, la nueva funcionalidad se obtiene ensamblando o componiendo objetos para obtener una funcionalidad de mayor complejidad. La composición se define dinámicamente, en tiempo de ejecución, haciendo que los objetos referencien o conozcan a otros.

Al usar composición, los objetos intervinientes sólo se conocen por sus interfaces, es decir sólo saben qué mensajes se pueden enviar unos a otros y no cómo están implementados. Por ello a esta forma de reuso se la conoce como *"reuso de caja negra"* [GHJV/95]. Existen en este caso menos dependencias implementativas, ya que un objeto referenciado o conocido puede ser reemplazado por cualquier otro que sepa responder a los mensajes que intervienen

en la relación de composición establecida, sin que esto implique que se deba modificar la implementación del objeto que los conoce.

Hay implementaciones del paradigma que soportan la herencia múltiple, es decir que una clase tenga más de una superclase. En estos casos se ve a la herencia múltiple como un medio para expresar la semántica del dominio en formas que la herencia simple no permite.

El uso de la herencia múltiple tiene la desventaja de generar modelos mucho más complejos y difíciles de entender. Además, se deben resolver conflictos cuando, por ejemplo, dos superclases de una misma clase definen un mismo mensaje, o variables de instancias con el mismo nombre.

La necesidad del uso de la herencia múltiple, en la mayoría de los casos, surge de un mal uso de la herencia como medio para resolver cuestiones implementativas. En la mayoría de los dominios no es común detectar relaciones de herencia que requieran de esta multiplicidad.

En general, todos estos casos se pueden resolver en un ambiente con herencia simple mediante el uso de la composición de objetos.

3 Metaclases y propiedades de clases

3.1 Metaclases

Una clase, al ser un objeto, es instancia de otra clase. Las clases cuyas instancias son clases se conocen como *metaclases*. Por lo tanto, las metaclases describen tanto la representación interna como el comportamiento de las clases.

Como ejemplos de las propiedades que determinan la estructura o representación interna de una clase se pueden mencionar: nombre de la clase, descripción de sus variables de instancias, información de sus relaciones de herencia, conjunto de métodos asociados a los mensajes definidos, etc.

Respecto al comportamiento, por ejemplo, son las metaclases las que definen cómo las clases responden al mensaje de creación de sus propias instancias.

3.2 Propiedades de las Clases

En los ambientes en los que las clases son tratadas como objetos, trabajando a nivel de metaclases se puede, en mayor o menor medida, redefinir comportamiento de las mismas.

Con las clases y sus relaciones de herencia se intenta modelar un dominio o aplicación particular. Identificaremos como *propiedad de una clase* o *metapropiedad* aquel comportamiento asignado a la misma que no sea esencial a este dominio particular, es decir, que pueda ser definido independientemente de la funcionalidad de la aplicación [BS/00]. Las metapropiedades representan conceptos propios del dominio de las clases que pueden presentarse recurrentemente.

Veamos algunos ejemplos que pueden ser considerados como propiedades de clases:

- ***Abstracta:*** Son clases que especifican el protocolo a ser respetado por la jerarquía de clases que ella encabeza. No tienen instancias porque en el dominio de problema representan conceptos abstractos.

  Otras propiedades que surgen a partir de esta definición serían aquellas que permiten verificar y garantizar que una jerarquía sea polimórfica. Es decir, que toda subclase concreta de la jerarquía provea una especificación y una implementación que respete la semántica definida por su superclase [Woo/97].

- ***Singleton:*** Son aquellas clases para las cuales se determina que no pueden tener más de una instancia [GHJV/95]. Esta restricción puede deberse a definiciones propias del dominio modelado, es decir que siempre debe usarse la misma instancia; o a cuestiones de performance debido a que es indistinto que exista una o más instancias y es caro crear una nueva instancia.

- ***Final:*** Una clase a la que se le asigna esta característica no puede ser subclasificada. Esta restricción puede surgir debido a que en el dominio modelado se detecta que no tiene sentido que se establezcan estas relaciones de herencia.
- ***Set:*** Una clase con este comportamiento conoce a todas sus instancias directamente. Este comportamiento puede ser necesario cuando una clase debe acceder recurrentemente a sus instancias y entonces, por cuestiones de performance, se hace que la clase tenga entre sus colaboradores a una colección que contiene a sus instancias activas.
- ***Inicialización automática:*** Una clase con este comportamiento envía un mensaje a una instancia recientemente creada para que inicialice a sus colaboradores de forma adecuada y de esta manera se encuentre en un estado consistente desde su creación.
- ***Chequeo de invariante:*** Se puede definir (a nivel de clase) invariantes que deben cumplir sus instancias. De esta manera cada vez que cada una de ellas recibe un mensaje, luego de la ejecución del método, se valida que la instancia cumpla con los invariantes definidos.
- ***Notificación de nuevas instancias creadas:*** Una clase con este comportamiento notifica a los interesados sobre la creación de una nueva instancia.

Otras propiedades se pueden asignar a los métodos definidos por una clase, por ejemplo:

- ***Traceable:*** Se define este comportamiento a una clase para loggear los llamados a métodos definidos para sus instancias y llevar registro de la información contextual de cada ejecución Este comportamiento puede ser asignado para analizar determinado funcionamiento de la aplicación, como ayuda para detectar errores, por determinadas restricciones en el dominio modelado como ser cuestiones de seguridad o auditoría, etc.
- ***Breakpoints:*** Este comportamiento se define para forzar una pausa en la ejecución de un método. Se usa esto como herramienta para analizar y corregir la implementación de un modelo en la etapa de desarrollo.
- ***Verificación de pre / post condiciones:*** Este comportamiento se define para que se verifiquen determinadas condiciones antes o después de la ejecución de un método. La validación de precondiciones se realiza con el objetivo de verificar que en el contexto se cumplan las condiciones que se deben dar para asegurar el correcto resultado de la ejecución. Las post condiciones se agregan con con el objetivo de validar el correcto funcionamiento del método en cuestión.
- ***Final:*** Al definirse este comportamiento en una clase respecto a un determinado método, se logra que el mismo no pueda ser redefinido en las subclases. Esta restricción se puede agregar porque en el dominio modelado no tiene sentido esta especialización.
- ***Contador de mensajes:*** Se agrega este comportamiento a una clase respecto a un determinado mensaje definido para sus instancias para contar el número de veces que se envía este mensaje. Esto puede ser útil cuando se necesita optimizar un proceso y analizar cuáles son los mensajes que más se envían.

3.2.1 Problemáticas a considerar relacionadas con las propiedades de clase

Los ambientes que manejan metaclases y la definición de propiedades de clase pueden ser analizados teniendo en cuenta cómo se soporta en cada uno de ellos los distintos aspectos relacionados con el concepto de propiedad de clase. Entre estos aspectos podemos mencionar:

- ***Combinación o composición de propiedades de clase.***
- ***Problema de propagación de las propiedades de clase.***
- ***Comunicación entre niveles de clase e instancia y problemas de compatibilidad.***

A continuación se detalla qué representa cada uno de estos aspectos.

3.2.1.1 Combinación o composición de propiedades de clase

Es probable que a una clase se le quiera asignar más de una propiedad, es decir que se necesita poder combinar o componer distintas propiedades. Por ejemplo, se puede definir que una clase sea singleton y a la vez determinar que su única instancia se inicialice automáticamente al crearse.

Cómo se soporta o se facilita esta combinación es un punto a analizar en los modelos que manejan metaclases y la definición de propiedades de clases.

3.2.1.2 Problema de propagación de las propiedades de clase

No siempre es deseable que las propiedades de clases se propaguen a sus subclases. Por el contrario, en muchos casos se necesita definir una propiedad para una clase sin pasarla a sus subclases. Podemos citar como ejemplo la propiedad de una clase de ser abstracta.

Este inconveniente, presente en algunas de las implementaciones actuales (como se detalla más adelante) se identifica como el *problema de la propagación de las propiedades de clases* [BLR/98] [BS/99].

3.2.1.3 Comunicación entre niveles de clase e instancia y problemas de compatibilidad

3.2.1.3.1 Comunicación entre niveles de clase e instancia

Otro asunto que aparece al tratar las clases como objetos es el de la *comunicación entre niveles*: el nivel de las clases y por otro lado el de sus instancias. Las clases envían mensajes a otros objetos, incluyendo a sus propias instancias. A la inversa, las instancias también pueden enviar mensajes a sus clases.

Un ejemplo del envío de mensaje de una clase a sus instancias es cuando una clase recibe un mensaje para crear una instancia. Es común que una clase inmediatamente después de crear una instancia le envíe a la misma un mensaje para que ésta se inicialice.

En general, en los distintos ambientes existen mensajes para que los objetos puedan acceder a sus clases y así enviarles mensajes a las mismas, por ejemplo para obtener el nombre de éstas.

3.2.1.3.2 Problemas de compatibilidad

Una característica deseable en todo modelo que maneje metaclases es la de garantizar la compatibilidad de la comunicación entre los distintos niveles al armarse las jerarquías de clases y de metaclases [BLR/98] [BS/99].

Existen dos casos de compatibilidad a considerar:

- ***Upward compatibility*:** la que se debe asegurar cuando las instancias envían mensajes a sus clases.

  Veamos un ejemplo:

  Sea una clase ***A*** que tiene implementado el mensaje ***i***.

  Sea ***c*** un mensaje implementado en la metaclase de ***A***.

  Sea que en la implementación de ***i*** se envía el mensaje ***c*** a la clase del receptor.

  Si se define a ***B*** subclase de ***A***, ***B*** hereda la implementación de ***i***. Luego si ***B*** no redefine la implementación de ***i***, al enviarse el mensaje ***i*** a una instancia de ***B***, se enviará el mensaje ***c*** a la clase ***B***.

  El modelo debería garantizar que la clase ***B*** sepa responder al mensaje ***c***, es decir que la metaclase de ***B*** implemente o herede el mensaje ***c***.

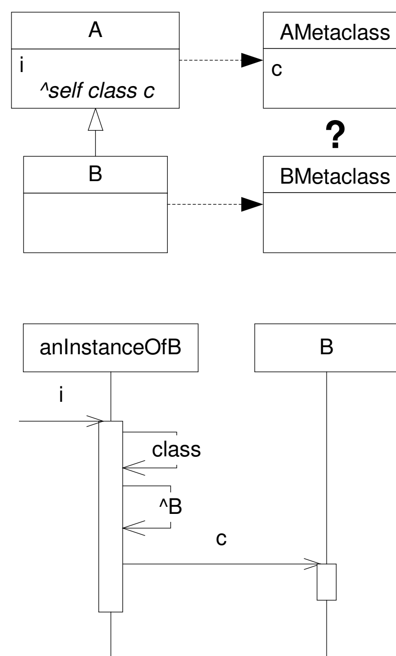


Figura 3.1: Diagramas[1] que ejemplifican la necesidad de asegurar upward compatibility

[1] La notación gráfica utilizada en los diagramas se detalla en el Apéndice de este informe.

- ***Downward compatibility*****:** la que se debe asegurar cuando las clases envían mensajes a sus instancias

  Veamos un ejemplo:

  Sea una clase ***A*** que tiene implementado el mensaje ***i***.

  Sea ***c*** un mensaje implementada en la metaclase de ***A***.

  Sea que en la implementación de ***c*** se envía el mensaje ***i*** a una instancia recientemente creada de la clase que recibe el mensaje ***c***.

  Si se define a la metaclase de ***B*** como subclase de la metaclase de ***A***, la metaclase de ***B*** heredará la implementación de ***c***. Luego, si la metaclase de ***B*** no redefine la implementación de ***c***, al enviarse el mensaje ***c*** a la clase ***B***, se enviará el mensaje ***i*** a la instancia recientemente creada de ***B***.

  El modelo debería garantizar que las instancias de ***B*** sepan responder al mensaje ***i***, es decir que la clase ***B*** implemente o herede el mensaje ***i***.

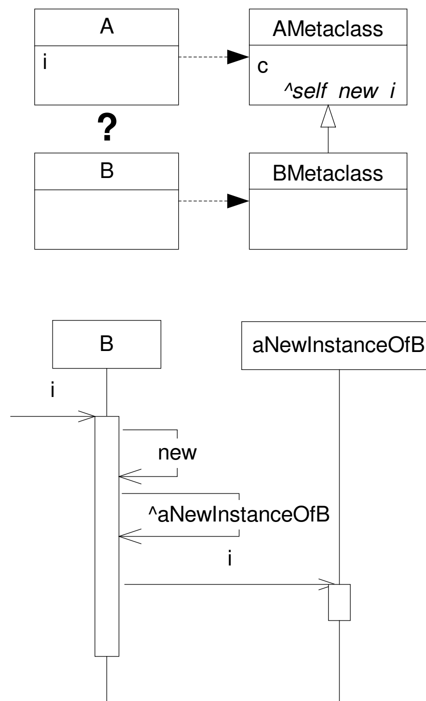


Figura 3.2: Diagramas que ejemplifican la necesidad de asegurar downward compatibility

3.3 Conclusiones

- Las *metaclases* describen estructura y comportamiento de las clases. Por ello son posibles lugares para definir propiedades de las clases.
- *Propiedad de clase o metapropiedad*: es un comportamiento de la clase que no es esencial al dominio que modela.
- Podemos considerar propiedades de las clases (*abstracta, singleton, inicialización automática,* etc.) o de los métodos de una clase (*traceable, verificación de pre / post condiciones, breakpoints,* etc.).
- Hay problemáticas que la implementación de metapropiedades debe resolver:
  - ✓ *Combinación o composición de propiedades de clase.*
  - ✓ *Problema de propagación de las propiedades de clase.*
  - ✓ *Comunicación entre niveles de clase e instancia y problemas de compatibilidad.*

4 Modelos actuales de ambientes que manejan metaclases

Existen distintos ambientes de objetos basados en clasificación en los que las clases se tratan como objetos y que manejan el concepto de metaclase y la definición de propiedades de clases. Entre estos ambientes, los más representativos son:

- *Smalltalk*
- *CLOS*
- *SOM*
- *ObjVlisp*
- *NeoClasstalk*
- *MetaclassTalk*
- *Traits*

A continuación se detallan características de estos ambientes y se los analiza considerando cómo soportan los distintos aspectos relacionados con el concepto de propiedad de clase que se mencionaron en el capítulo anterior.

4.1 Metaclases en Smalltalk

En *Smalltalk-80* [GR/86], cuando se crea una clase, se crea automáticamente una metaclase cuya única instancia es la nueva clase. Es decir, las metaclases son implícitas, no pueden ser definidas por el usuario y dos clases no pueden compartir la misma metaclase [BC/89] [Coi/87].

Una metaclase no tiene nombre propio y sólo puede ser accedida enviando a su única instancia el mensaje *class*.

La nueva metaclase se crea como una subclase de la metaclase de la superclase de la nueva clase. Por ejemplo, cuando se crea la clase *B*, como subclase de *A*, se genera la metaclase de *B* como una subclase de la metaclase de *A*. De esta manera, la jerarquía de metaclases resulta paralela a la jerarquía de clases. Esta jerarquía de metaclases es fija y queda determinada por la jerarquía de clases.

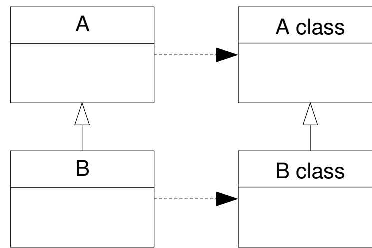


Figura 4.1: Paralelismo entre jerarquías de clases y metaclases en Smalltalk

Contar con una metaclase distinta por cada clase permite que se pueda determinar tanto la estructura como el comportamiento para cada una de ellas. Esto es posible dado que el usuario puede definir métodos a nivel de metaclase. Estos métodos implementan mensajes que se pueden enviar a la clase misma y se los llama *métodos de clase*. Además, para extender la estructura de las clases definidas, el usuario puede agregar variables de instancia a nivel de metaclases.

De esta manera, por ejemplo, se puede definir para cada clase distintos métodos de creación de instancias, lo que ofrece la posibilidad que las instancias se puedan inicializar de forma adecuada al crearse. *"El rol principal de una metaclase en Smalltalk-80 es el de proveer protocolo para inicializar variables de clases y para crear instancias inicializadas de la única instancia de la metaclase"* [GR/86]. En general, este esquema de múltiples metaclases permite la asignación de determinadas propiedades a las distintas clases.

4.1.1 Diseño del kernel de clases / metaclases

Las metaclases de Smalltalk-80 son similares al resto de las otras clases porque contienen los métodos que usan sus propias instancias y tienen relaciones de herencia con otras clases. Pero se diferencian debido a que no son instancias de metaclases implícitas, sino de una clase llamada *Metaclass*. De esta manera se logra tener la misma estructura y comportamiento para todas las metaclases implícitas.

Por otro lado, *Metaclass*, por ser una clase, es instancia de una metaclase, que a su vez es instancia de sí misma.

De esta manera, con la introducción de la clase *Metaclass* como metaclase e instancia de su metaclase, se logra implementar con una cantidad finita de elementos la relación de instanciación entre una clase y su metaclase, que es una definición potencialmente recursiva infinita.

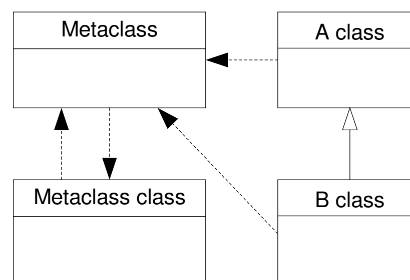


Figura 4.2: Clase Metaclass y sus relaciones de instanciación con las clases y su metaclase en Smalltalk

Existe una clase llamada *Class* que es abstracta y que, como su nombre lo indica, modela las clases. *Class* es superclase directa o indirecta de todas las metaclases y cada una de ellas especializa o agrega comportamiento para la clase específica que está representando

Todas las metaclases quedan definidas como subclases directas o indirectas de *Class* como consecuencia de las siguientes afirmaciones:

- Toda clase es directa o indirectamente subclase de la clase *Object* (que define el comportamiento común de todo objeto), excepto la clase *Object* misma que no tiene superclase[2].
- Como se expresó anteriormente, la jerarquía de metaclases es paralela a la de clases.
- La superclase de *Object class* (la metaclase de *Object*) es *Class*.

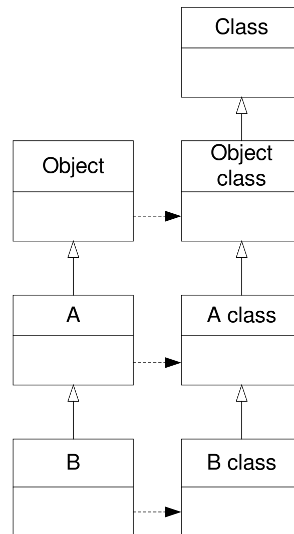


Figura 4.3: Todas las metaclases son subclases de Class en Smalltalk

De acuerdo a lo expresado anteriormente, vemos que Smalltalk-80 usa dos metaclases explícitas para describir clases:

- *Class*, que modela las clases que no son metaclases (es decir clases cuyas instancias no son clases).
- *Metaclass*, cuyas instancias son las metaclases.

Las clases *ClassDescription* y *Behavior* modelan comportamiento común de ambos tipos de clases. Es por esto, que tanto *Class* como *Metaclass* heredan de *ClassDescription*, que a su vez es subclase de *Behavior*.

[2] En algunos dialectos de Smalltalk, se define una superclase de *Object*, llamada *ProtoObject*. En esos casos, se puede enunciar estos hechos en relación a *ProtoObject* en vez de a *Object*, sin pérdida de generalidad.

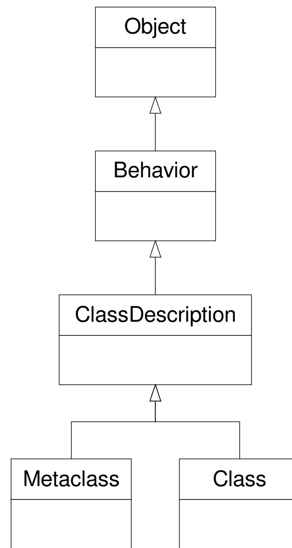


Figura 4.4: Jerarquía de clases que modela las metaclases

A continuación se expone un diagrama en el que se resumen todas las relaciones de herencia e instanciación que se definieron.

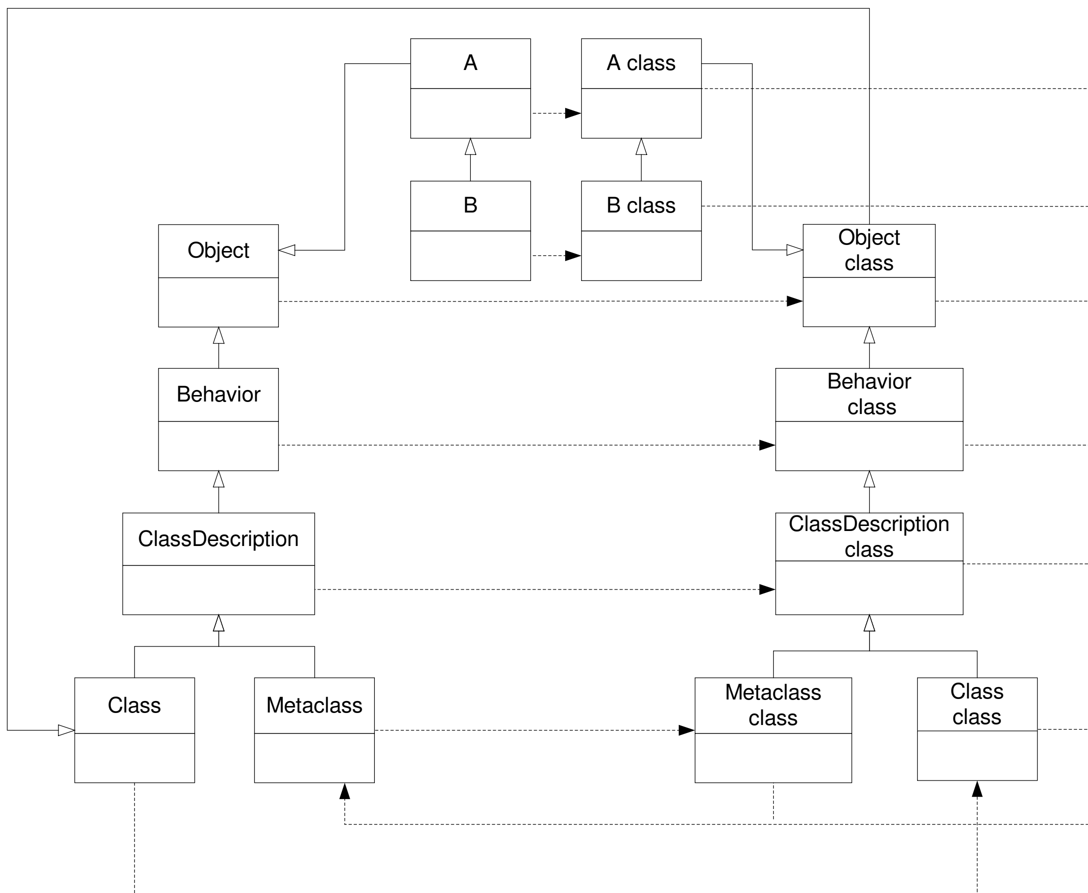


Figura 4.5: Modelo de clases y metaclases en Smalltalk

4.1.1.1 Descripción y responsabilidades de clases que conforman el kernel de clases /metaclases

Object

Define el comportamiento que todos los objetos comparten por defecto.

Behavior

Define el comportamiento mínimo necesario para los objetos que tienen instancias y métodos.

La estructura definida por *Behavior* incluye referencias para manejar relaciones de herencia, un diccionario de métodos y representación de las variables de instancias.

Ofrece protocolo para:

- crear, eliminar y compilar métodos, así como también para acceder tanto al código fuente como al método compilado
- crear instancias (mediante la definición de los métodos básicos *new y new:*), y accederlas
- establecer relaciones de herencia entre clases y acceder a esta información
- acceder al conjunto de nombres de las variables definidas.

ClassDescription

Agrega comportamiento adicional al básico provisto por *Behavior*, para facilitar una mejor estructuración de la descripción de una clase.

Ofrece protocolo para:

- dar nombre a una clase
- agregar un comentario textual a la clase
- nombrar, agregar y eliminar variables de instancias
- categorizar métodos
- guardar la descripción completa de la clase en un archivo
- registrar los cambios en la descripción de la clase.

Class

Define el comportamiento de las clases que no son metaclases. Las instancias de *Class* describen la representación interna y el comportamiento de objetos.

Respecto a la estructura interna, se agregan variables de instancias referidas al nombre, variables de clases (variables que la clase comparte con sus subclases) y variables compartidas (variables globales usadas por la clase).

Class agrega funcionalidades que complementan a las básicas provistas por *Behavior* y *ClassDescription*.

Entre estas responsabilidades adicionales se puede mencionar:

- representación para los nombres de variables de clases
- almacenamiento de la descripción de la clase en un archivo donde el archivo tiene el mismo nombre que la clase
- eliminación de la clase del sistema

- renombre de la clase.

Metaclass

Describe el comportamiento de una metaclase.

El mensaje principal que se agrega para las instancias de esta clase es justamente el que les permite crear a su propia y única instancia.

Respecto a la estructura interna, se agrega una variable de instancia para referenciar a la instancia de una metaclase (implementando así la característica de *singleton*).

4.1.2 Cómo se asignan y combinan propiedades de clase

La asignación y composición de propiedades se maneja en la implementación de los métodos asociados a los mensajes que saben responder las clases (definidos en su metaclase).

Por ejemplo, supongamos que se quiere que una clase sea *singleton*, que su *inicialización sea automática* y que sea *final*. En este caso, en la metaclase asociada se redefine el método de creación de instancias para controlar que no se cree más de una instancia y, en caso de crearla, hacer que se inicialice. Además se redefine el método con el cual se crea una subclase para impedir que esto ocurra.

Si queremos que alguna otra clase tenga estas propiedades, debemos redefinir los mismos métodos, con la misma implementación. De esta manera, cada vez que se asigna una propiedad a una clase, se duplica la lógica que la implementa.

4.1.3 Cómo se asignan propiedades a clases de la jerarquía que modela lógica booleana

Con el fin de ejemplificar los conceptos expuestos y las diferencias entre los distintos modelos respecto a cómo se diseñan y asignan las propiedades a las clases, se analiza cómo se hace esto en Smalltalk-80 respecto a la jerarquía de clases que modela la lógica booleana. Este mismo ejemplo se estudiará en cada uno de los modelos que se exponen más adelante.

En general, la lógica booleana se modela con tres clases: *Boolean*, superclase de *True* y *False*. *Boolean* es *abstracta*. Y *True* y *False* se manejan como clases *singleton*. Como además no parece tener sentido subclasificarlas, se les puede asignar también la propiedad de ser *finales*.

Veamos cómo se implementa esto en Smalltalk-80.

A continuación se muestran las relaciones entre clases y metaclases de esta jerarquía.

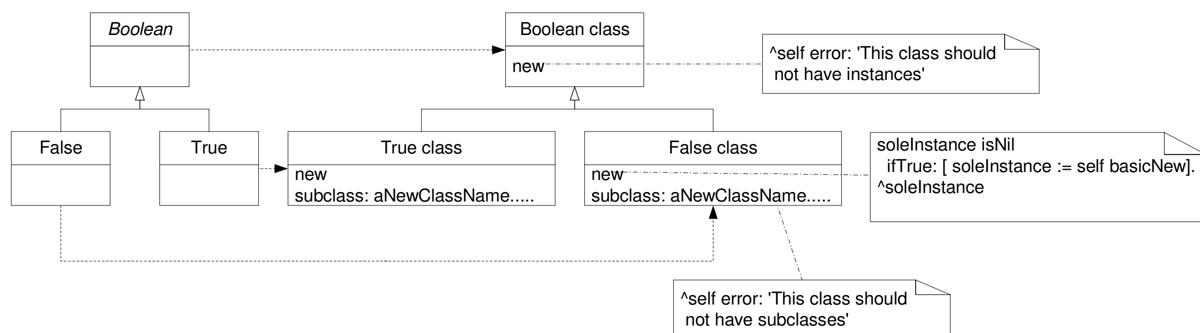


Figura 4.6: Asignación de propiedades a las clases que modelan lógica booleana en Smalltalk

Las propiedades *abstracta* y *singleton* se manejan en el método de creación de instancias, por eso en las tres metaclases se redefine el método *new*. En *Boolean class*, para manejar la propiedad *abstracta*, evitando que se creen instancias. Y en *True class* y *False class*, se redefine por dos motivos: Por un lado, evitar que se creen instancias usando el método *new* heredado de *Boolean class* que provocaría el error por querer instanciar una clase abstracta. Por otro lado, para manejar la propiedad *singleton*, es decir para evitar que se cree más de una instancia.

La propiedad *final* se implementa redefiniendo en las metaclases *True class* y *False class* el método que permite crear una subclase a partir de una clase existente, justamente para evitar las mismas sean subclasificadas.

4.1.4 Compatibilidad de comunicación entre niveles

La jerarquía de metaclases es fija, está determinada por la jerarquía de las clases que son sus instancias, y es paralela a la misma. Con esto se asegura la compatibilidad de comunicación entre niveles [BLR/98].

Veamos esto con ejemplos:

- Cómo se asegura la compatibilidad *upward*

Sea ***A*** una clase que tiene implementado el mensaje ***i***.

Sea ***c*** un mensaje implementado en la metaclase de ***A***.

Sea que en la implementación de ***i*** se envía el mensaje ***c*** a la clase del objeto que lo recibe.

Al definirse ***B*** como subclase de ***A***, se crea automáticamente la metaclase de ***B*** como subclase de la metaclase de ***A***. De esta manera, la clase ***B*** sabe responder al mensaje ***c***, ya que hereda la definición del mismo hecha para la clase ***A***.

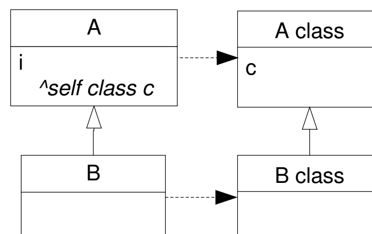


Figura 4.7: Diagrama que refleja que en Smalltalk se asegura la compatibilidad upward

Luego, si en ***B*** no se redefine la implementación de ***i***, al enviarse el mensaje ***i*** a una instancia de ***B***, se enviará el mensaje ***c*** a la clase ***B*** y ésta lo sabrá responder.

- Cómo se asegura la compatibilidad *downward*

Sea ***A*** una clase que tiene implementado el mensaje ***i***.

Sea ***c*** un mensaje implementado en la metaclase de ***A***.

Sea que en la implementación de ***c*** se envía el mensaje ***i*** a una instancia recientemente creada de la clase que recibe el mensaje ***c***.

Al definirse ***B*** como subclase de ***A***, se crea automáticamente la metaclase de ***B*** como subclase de la metaclase de ***A***. De esta manera, la clase ***B*** hereda la implementación del método ***c*** hecha para la clase ***A***.

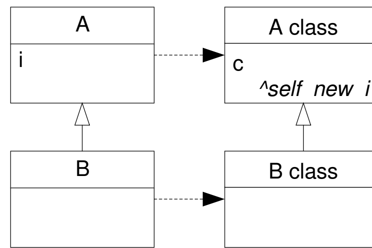


Figura 4.8: Diagrama que refleja que en Smalltalk se asegura la compatibilidad downward

Luego, si la metaclase de ***B*** no redefine la implementación de ***c***, al enviarse el mensaje ***c*** a la clase ***B***, se enviará el mensaje ***i*** a la instancia recientemente creada de ***B***. Y esta instancia de ***B*** sabe responder al mensaje ***i*** ya que la clase ***B*** se había creado como subclase de la clase ***A*** y por lo tanto hereda la definición de este mensaje hecha en ***A***.

4.1.5 Propagación de propiedades de clase

En esta implementación está presente el problema de la propagación de propiedades de clase [BLR/98].

Como la jerarquía de metaclases es paralela a la jerarquía de clases asociada, los mensajes definidos para una clase en su metaclase son heredados por sus subclases. En general, las propiedades se manejan en la implementación de los métodos asociados a estos mensajes. A partir de estas dos afirmaciones, se puede deducir que una subclase hereda las propiedades definidas por su superclase.

Analicemos esto respecto al ejemplo de la lógica Booleana:

La propiedad *abstracta* para la clase *Boolean* se implementa redefiniendo el mensaje *new* en su metaclase (*Boolean class*) para que se genere un error.

Como *True* hereda de *Boolean*, entonces *True class* hereda de *Boolean class*.

Por lo tanto si en *True class* no se hubiera redefinido el método *new*, por defecto, *True class* heredaría la implementación de *new* hecha en *Boolean class* y por lo tanto *True* heredaría la propiedad *abstracta*.

4.2 Metaclases en CLOS (Common Lisp Object System)

Clos [Mic] [MG/87] es un sistema orientado a objetos basado en clasificación, resultado de una extensión del modelo de Common Lisp.

En este modelo las clases, como en Smalltalk-80, también son objetos, instancias de sus metaclases. Pero a diferencia de aquel, las metaclases se definen explícitamente, es decir que no se crean automáticamente al crear una clase, sino que se definen de la misma manera que las clases que no son metaclases [BC/89]. Una clase se crea enviando explícitamente el mensaje de creación a su metaclase, que puede ser accedida como cualquier otro objeto.

En este esquema la cantidad de meta-niveles que se pueden definir no está acotada. Se pueden definir metaclases, cuyas instancias sean a su vez metaclases, y así sucesivamente.

La jerarquía de metaclases no está determinada por la jerarquía de clases, es decir la metaclase de una clase puede no ser subclase de la metaclase de su superclase. Aunque por defecto, una nueva clase debe tener la misma metaclase que la de su superclase, el programador puede modificar esto, y tiene la posibilidad de crear una nueva metaclase por cada clase creada, sin ninguna restricción respecto a sus relaciones de herencia [BLR/98] [Duc/00].

Las metaclases pueden ser compartidas, es decir que distintas clases pueden tener como clase a la misma metaclase.

En este modelo se maneja herencia múltiple de clases, tanto a nivel de metaclases como para aquellas que no lo son.

A continuación se exponen dos diagramas. En el primero se muestra un ejemplo de definiciones de clases y las relaciones con sus metaclases cuando se trabaja con la restricción por defecto sobre la determinación de metaclases. En el segundo se detalla otro ejemplo que se puede definir cuando se anula esta restricción.

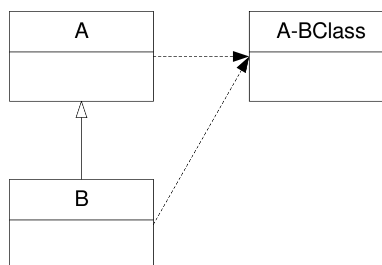


Figura 4.9: Relación por defecto entre las clases y sus metaclases en Clos

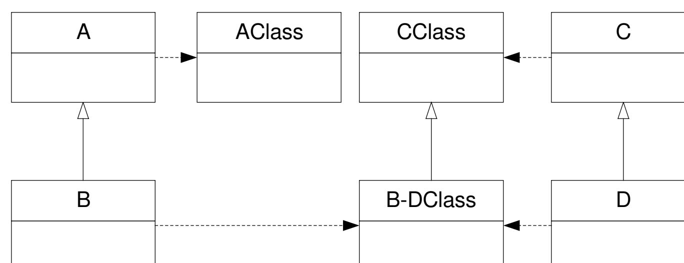


Figura 4.10: Posible relación entre clases y sus metaclases en Clos al eliminar restricción por defecto

4.2.1 Diseño del kernel de clases y metaclases

Por default las clases definidas por el usuario, incluidas las metaclases, son instancias de la clase *standard-class*. Se puede modificar esto indicando cuál es la metaclase de una nueva clase creada.

standard-class es una instancia de la clase llamada *class* y ésta es una instancia de sí misma. Ésta es una definición de referencia circular similar a la de Smalltalk-80, introducida también para implementar la regresión al infinito.

Cuando se define una clase en Clos, se ejecuta el método *validate-superclass*, antes de que se terminen de establecer las relaciones con sus superclases directas. Por default, *validate-superclass* devuelve *true* si la metaclase de la nueva clase es la misma que la metaclase de la superclase, esto implementa la restricción de que las clases y sus subclases deben tener la misma metaclase. De esta manera se evitan las incompatibilidades pero la programación queda restringida. Para permitir la definición de clases con diferentes comportamientos, los programadores pueden redefinir el método *validate-superclass* para hacer que acepte otras posibilidades como válidas. Esto proporciona la total libertad para usar una metaclase específica para nuevas clases.

4.2.2 Cómo se asignan y combinan propiedades de clase

Al poder definir metaclases explícitamente, se puede crear una metaclase por cada propiedad que se quiera implementar, redefiniendo los métodos que correspondan. Debido a que las metaclases pueden ser compartidas por distintas clases, se puede asignar a cada clase la metaclase adecuada según la propiedad que se le quiera asignar. Esto permite el reuso de la lógica que implementa cada propiedad.

Gracias al uso de herencia múltiple, la combinación de propiedades se podría lograr asignando a una clase una metaclase que sea subclase de cada una de las metaclases que implementan las propiedades requeridas.

Sin embargo, debido a que las propiedades se implementan redefiniendo métodos de las metaclases, al usar herencia múltiple se deben resolver conflictos de redefinición de métodos, que se pueden presentar cuando dos metaclases de las que se hereda redefinen los mismos métodos para implementar las propiedades respectivas.

En Clos, el tema de conflictos se encara definiendo un orden de precedencia entre las superclases. En los casos en que este orden no se pueda determinar debido a las relaciones de herencia definidas, se genera un error.

Para los casos en que dos metaclases implementen la propiedad respectiva redefiniendo el mismo método y ambas formen parte de una determinada jerarquía, existen mecanismos que permiten la combinación de métodos, por ejemplo para indicar que se ejecute la siguiente implementación. Si no se determina esto explícitamente, se ejecutará sólo la implementación provista por la clase que se encuentre antes en la lista de superclases ordenadas.

4.2.3 Cómo se asignan propiedades a clases de la jerarquía que modela lógica booleana

En Clos la asignación de propiedades a las clases que modelan la lógica booleana se implementa mediante la definición de las jerarquías de clases y metaclases que se exponen a continuación.

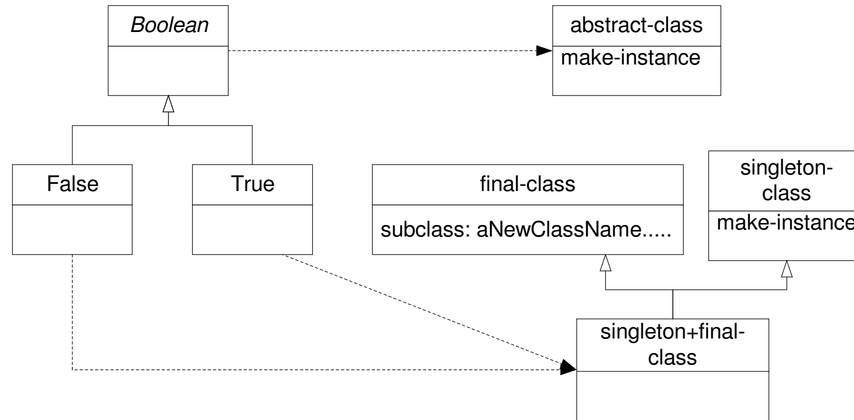


Figura 4.11: Asignación de propiedades a las clases que modelan lógica booleana en Clos

La propiedad *abstracta* se asigna a la clase *Boolean* creándola como instancia de la metaclase *abstract-class* que redefine el método de creación de instancias para implementar esta propiedad.

Las propiedades *singleton* y *final* de las clases *True* y *False* se determinan haciendo que ambas sean instancias de una metaclase que tiene como superclases a las metaclases que implementan cada una de estas propiedades redefiniendo los métodos correspondientes.

4.2.4 Compatibilidad de comunicación entre niveles

Debido a que se pueda anular la restricción por defecto respecto a la definición de metaclases, en este modelo no se puede asegurar la compatibilidad de comunicación entre niveles, y es el programador quien se debe responsabilizar de mantener este tipo de consistencia.

4.2.5 Propagación de propiedades de clase

Pero, por otro lado, es justamente usando esta libertad para definir metaclases y establecer las relaciones de herencia entre las mismas que se puede evitar la propagación de propiedades.

Veamos un ejemplo. Supongamos que se quiere que la clase *A* tenga la propiedad de ser *abstracta*, pero que su subclase *B* no lo sea. Así, se puede definir una nueva metaclase para la clase *A* que redefine el método de creación de instancias para evitar que se creen instancias de la misma. Por otro lado, se define que la metaclase de *B*, sea la metaclase standard, es decir *standard-class*. De esta manera al enviarse el mensaje de creación de instancias a la clase *B*, se ejecuta el método de creación de instancias por defecto y se crea efectivamente una nueva instancia de *B* y no se produce el error que se hubiera producido si se hubiera enviado el mensaje a la clase *A*.

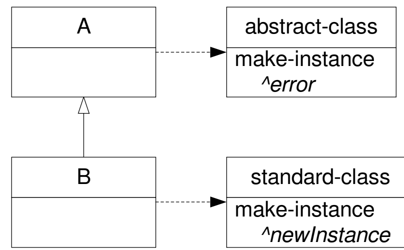


Figura 4.12: Cómo evitar propagación de la propiedad abstracta en Clos

4.3 Metaclases en ObjVlisp

ObjVlisp [Coi/87] es un sistema orientado a objetos basado en clasificación, resultado de una extensión del modelo de Vlisp, que es uno de los dialectos del lenguaje Lisp.

ObjVlisp define un modelo de metaclases con las mismas características que Clos. Pero en ObjVlisp no existe la restricción por defecto que existe en Clos respecto a la definición de metaclases.

4.3.1 Diseño del kernel de clases y metaclases

ObjVlisp queda autodefinido por sólo dos clases: la raíz del árbol de instanciación (*Class*) y la raíz del árbol de herencia (*Object*).

Class es la metaclase primitiva inicial y define el comportamiento standard de las clases.

Object es la clase que define el comportamiento común default para todos los objetos, sean clases o no.

Estos dos árboles quedan vinculados estableciendo las siguientes relaciones entre sus clases raíces:

- *Class* es subclase de *Object*
- *Object* es instancia de *Class*.

Class, al ser un objeto, también debe ser descripta e instanciada por una clase. Nuevamente, como en los modelos anteriores, se encuentra una definición con referencia circular, definiendo a *Class* como instancia de sí misma.

Todas las clases definidas terminan siendo directa o indirectamente subclases de *Object*. Todas las metaclases que se creen terminarán siendo directa o indirectamente subclases de *Class* y a su vez instancias de *Class* o de una subclase directa o indirecta de la misma. Y toda clase que no tenga a *Class* como superclase directa o indirecta, será una clase que no es una metaclase.

A continuación se expone un diagrama que resume los conceptos definidos.

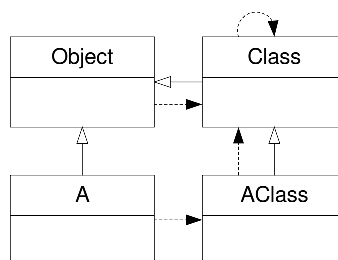


Figura 4.13: Diseño del kernel de clases y metaclases en ObjVlisp

4.3.2 ClassTalk: Implementación de ObjVlisp en Smalltalk-80

ClassTalk [LeCo] [BC/89] es una implementación en Smalltalk del modelo de ObjVlisp, que puede ser utilizada como una plataforma de experimentación para estudiar la programación con metaclases explícitas.

Se diseñaron browsers específicos para poder trabajar con las metaclases explícitamente.

Se ofrece una librería de metaclases explícitas que representan distintas propiedades. Estas metaclases pueden ser reusadas para crear metaclases más complejas (es decir para crear clases con más de una propiedad) mediante la combinación de las mismas, usando el mecanismo de herencia múltiple. Como ejemplos de estas metaclases se pueden mencionar:

- *AbstractClass*: modela la propiedad abstracta.
- *MemoClass*: modela las clases que conocen directamente a la colección de todas sus instancias.
- *AutoInitClass*: modela la propiedad de inicialización automática.
- *AccessClass*: modela las clases que crean automáticamente en sus clases instancias métodos para acceder a las variables de instancias (accesors).

4.3.3 Cómo se asignan y combinan propiedades de clase

Se resuelve de una forma similar a la de Clos.

Respecto a los conflictos que se presentan por el uso de herencia múltiple, en ClassTalk las metaclases que tienen más de una superclase se crean como instancias de una clase llamada *MIClass*. Ésta detecta los casos de métodos conflictivos, pero sigue siendo responsabilidad del programador la resolución de los mismos.

4.3.4 Cómo se asignan propiedades a clases de la jerarquía que modela lógica booleana

En ClassTalk, la definición de propiedades para este ejemplo resulta como la de Clos, con la diferencia que la metaclase de *True* y *False* es instancia de la clase: *MIClass*, que existe específicamente para metaclases que tienen más de una superclase.

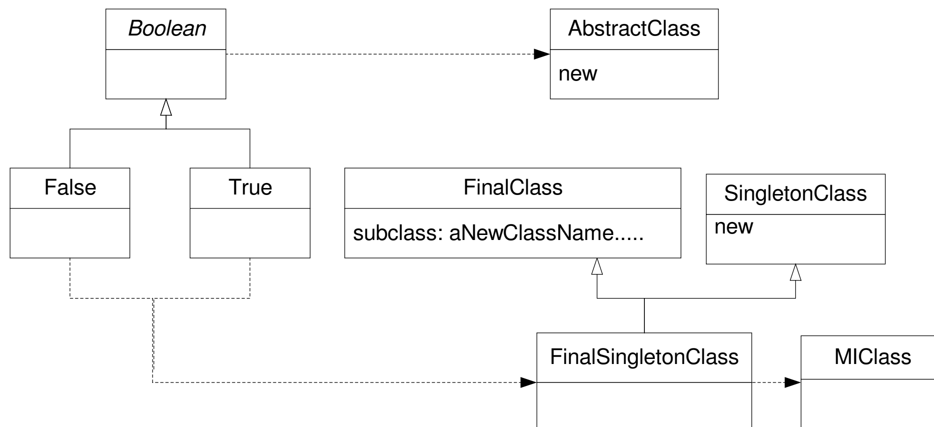


Figura 4.14: Asignación de propiedades a las clases que modelan lógica booleana en ObjVlisp

4.3.5 Compatibilidad de comunicación entre niveles

Como sucede en Clos, en este modelo tampoco se asegura la compatibilidad de comunicación entre niveles. Nuevamente es el programador quien debe asegurar este tipo de consistencias.

4.3.6 Propagación de propiedades de clase

Por las mismas razones que en Clos, aquí también se puede evitar la propagación de propiedades.

4.4 Metaclases en SOM (System Object Model)

SOM [FD/98] [FD] es un modelo de objetos basado en clasificación y en el cual las clases se manejan como objetos. Fue desarrollado en IBM y es compatible con el modelo de CORBA.

El kernel de SOM se basa en el modelo de ObjVlisp, tiene un modelo de metaclases con características similares. Las metaclases son explícitas y pueden tener más de una instancia. El programador tiene total libertad de organizar su jerarquía de metaclases, sin forzar ninguna relación con la jerarquía de las clases que son sus instancias. También, como en ObjVlisp, se soporta la definición de relaciones de herencia múltiple entre clases, sean éstas metaclases o no.

SOM introduce el concepto de *metaclase derivada*. Esto es, al definirse una clase, el sistema determina si la metaclase asignada es la misma (esto puede darse sólo en el caso que la clase tenga una sola superclase) o hereda de todas las metaclases de sus superclases En caso que esto no suceda, se crea una nueva metaclase como subclase de la metaclase definida y de las metaclases de sus superclases. A esta nueva metaclase se la llama derivada y la nueva clase pasa a ser instancia de la misma.

Veamos esto con un ejemplo. Supongamos que se quiere crear una clase *B*, instancia de *BClass*, y subclase de la clase *A*, que es a su vez instancia de *AClass*.

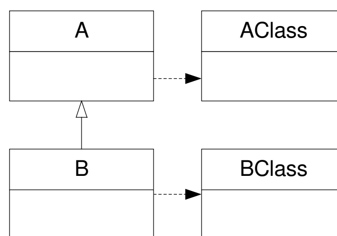


Figura 4.15: Definición de clases y metaclases en SOM

SOM detectará que *B* es subclase de *A* pero que *BClass* no es subclase de *AClass*, luego creará la metaclase derivada: *Derived*, subclase de *AClass* y *BClass*. Y la clase *B* será instancia de *Derived*.

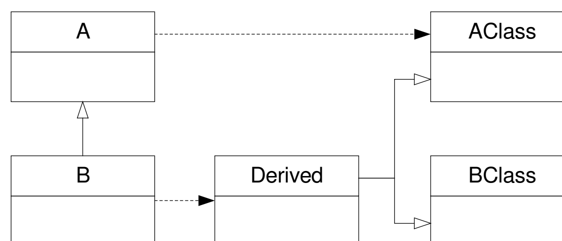


Figura 4.16:Creación de metaclase derivada en SOM

La semántica de las metaclases derivadas garantiza que la metaclase declarada tiene precedencia ante la resolución de ambigüedades que pueden darse por el uso de la herencia múltiple. En el ejemplo *BClass* precedería a *AClass.*

4.4.1 Diseño del kernel de clases y metaclases

El kernel del modelo de SOM queda definido conceptualmente de la misma forma que en ObjVlisp. Las clases primitivas que allí se llaman: *Object* y *Class*, aquí se denominan *SOMObject* y *SOMClass* respectivamente.

4.4.2 Cómo se asignan y combinan propiedades de clase

Se resuelve de forma similar a la usada en Clos y ObjVlisp: implementando propiedades de clase mediante la creación de distintas metaclases y combinándolas a través del uso de herencia múltiple.

Como en Clos, en este modelo se resuelven muchos de los conflictos de métodos por el uso de herencia múltiple estableciendo un orden lineal entre todos los ancestros de la clase, siendo la clase en cuestión la que encabeza la lista. La implementación del método que se ejecutará primero será la de la primer clase de la lista que defina esta método. Este orden se usa también cuando desde un método redefinido se determina que se ejecute la próxima implementación. Hay determinados casos en los que no se puede determinar este orden, debido a inconsistencias que se presentan por las relaciones de herencia y las redefiniciones de un mismo método existentes. En estos casos se analizan las implementaciones del método para determinar si las inconsistencias se pueden salvar (por ejemplo, si el orden de ejecución entre dos implementaciones no altera el resultado final).

4.4.3 Cómo se asignan propiedades a clases de la jerarquía que modela lógica booleana

En SOM, la definición de propiedades para este ejemplo resulta como la de Clos y ObjVlisp, pero las relaciones entre metaclases, luego de la creación de la metaclase derivada que se crea automáticamente a partir de esta definición, resulta como se muestra en el siguiente esquema.

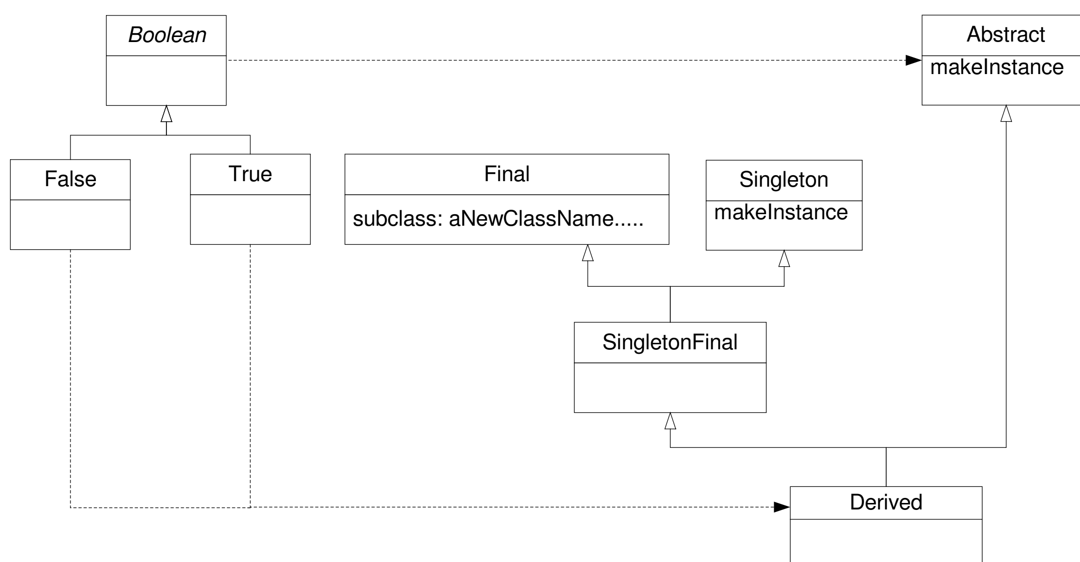


Figura 4.17: Asignación de propiedades a las clases que modelan lógica booleana en SOM

Aunque la metaclase de las clases *True* y *False* herede de la metaclase que implementa la propiedad *abstracta*, en este caso no traería inconvenientes. Esto es porque tanto la propiedad *singleton* como la *abstracta* se implementan redefiniendo el método de creación de instancias y, como se expuso anteriormente, las metaclases declaradas tienen precedencia sobre las que agrega el sistema. Entonces, primero se ejecuta el método de creación de instancias redefinido por la metaclase de la propiedad *singleton* y no se llega a ejecutar la implementación de creación de instancias de la metaclase de la propiedad *abstracta*.

4.4.4 Compatibilidad de comunicación entre niveles

Con la introducción de las metaclases derivadas, se evita la incompatibilidad upward.

Veamos esto con un ejemplo.

Sea ***A*** una clase que tiene implementado el mensaje ***i***.

Sea ***c*** un mensaje implementado en la metaclase de ***A*** (***AClass***).

Sea que en la implementación de ***i*** se envía el mensaje ***c*** a la clase del objeto que lo recibe.

Sea que se define a ***B*** como subclase de ***A***.

Si la metaclase de ***B*** (***BClass***) hereda de la metaclase de ***A*** entonces la clase ***B*** sabe responder al mensaje ***c***, ya que hereda la definición del mismo hecha para la clase ***A***.

En caso contrario, es decir si ***BClass*** no hereda de ***AClass***, se crea automáticamente una metaclase derivada (***Derived***) que hereda de ***AClass*** y ***BClass*** y se hace que ***B*** sea instancia de ***Derived***. De esta manera, nuevamente, la clase ***B*** hereda la definición de ***c*** hecha para la clase ***A***.

Luego, si en ***B*** no se redefine la implementación de ***i***, al enviarse el mensaje ***i*** a una instancia de ***B***, se enviará el mensaje ***c*** a la clase ***B*** y ésta lo sabrá responder.

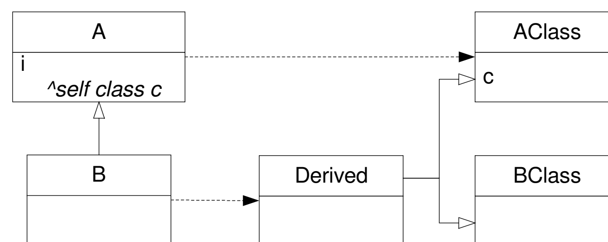


Figura 4.18: Compatibilidad upward por creación de metaclases derivadas en SOM

En cambio, la compatibilidad downward no se asegura, ya que está permitido crear una metaclase como subclase de otra sin la restricción de que exista la misma relación de herencia entre las clases que son instancias de las mismas.

4.4.5 Propagación de propiedades de clase

Debido a la creación de las metaclases derivadas, en SOM se da la propagación de propiedades de una clase a sus subclases [BLR/98] [Duc/00].

Supongamos que se quiere asignar a la clase *A* la propiedad de ser *abstracta*, entonces se crea como instancia de la metaclase que implementa la propiedad abstracta. Luego se crea la clase *B*, de la que sí se esperan crear instancias, como subclase de *A*. Entonces, se crea

automáticamente una metaclase derivada para *B* que tiene como superclase a la metaclase de *A* que es la que implementa la propiedad abstracta. De esta manera la clase *B* tiene asignada la propiedad abstracta cuando en realidad esto no era lo deseable.

4.5 Metaclases en modelo de compatibilidad implementado en NeoClasstalk

NeoClasstalk [BLR/98] [DSW/04] es un ambiente de objetos implementado en Smalltalk y en donde también se manejan metaclases explícitas.

El *modelo de compatibilidad* [BLR/98] es una extensión del modelo de Smalltalk y se usó NeoClasstalk para experimentar con el mismo. Existe una jerarquía de metaclases paralela a la jerarquía de clases que asegura la compatibilidad de comunicación entre niveles. Como en Smalltalk, esto se logra creando automáticamente una metaclase cada vez que se crea una nueva clase. Estas son las metaclases de compatibilidad. Las metaclases de esta jerarquía definen el comportamiento de las clases asociado al dominio particular que se está modelando y que debe propagarse a todas las subclases.

Se introduce un nivel adicional de metaclases para asignar localmente las propiedades a las clases. Las clases son instancias de las metaclases pertenecientes a este nivel (*metaclases de propiedades*). Una clase con una propiedad específica es instancia de una metaclase de propiedad que hereda de la correspondiente metaclase de compatibilidad.

Veamos estos conceptos con un ejemplo. Supongamos que se tienen dos clases: *A* y *B*, siendo *B* subclase de *A*. Se quiere asignar una propiedad distinta a cada clase. En el siguiente esquema se muestra cómo se define esto en NeoClasstalk.

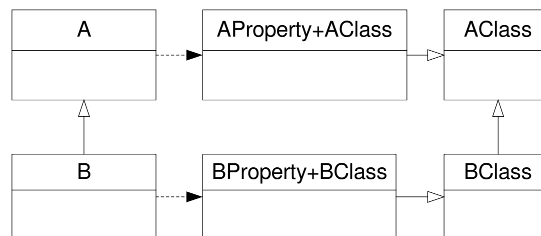


Figura 4.19: Metaclases de compatibilidad y de propiedades en NeoClasstalk

La metaclase *AClass* define el comportamiento de la clase *A* particular del dominio que se está modelando. La metaclase *AProperty+AClass* define una propiedad específica para la clase *A*. La clase *A* es instancia de *AProperty+AClass* y como ésta hereda de la metaclase *AClass*, la clase *A* tiene definido tanto el comportamiento asociado al dominio particular como el de la propiedad que se le quiere asignar.

Lo expuesto para la clase A se aplica de la misma manera para la clase *B*.

4.5.1 Diseño del kernel de clases y metaclases

Existe una clase llamada *PropertyMetaclass* que describe la estructura y comportamiento default de las metaclases de propiedades. Todas las metaclases de propiedades son instancias de una subclase de *PropertyMetaclass*.

Dado que las metaclases que implementan diferentes propiedades tienen diferentes comportamientos, se crea una subclase de *PropertyMetaclass* por cada propiedad que se quiera implementar.

Cuando una metaclase de propiedad se crea, su clase la inicializa con la definición de implementación de la correspondiente propiedad, determinando tanto sus variables de instancia como sus métodos. El código correspondiente a la definición de la propiedad se genera automáticamente [Duc/99].

De esta manera, las metaclases de propiedades que definen la misma propiedad de clase son instancias de la misma subclase de *PropertyMetaclass*, lográndose con esto el reuso de la implementación de las propiedades.

A continuación se expone un diagrama en el que se reflejan las relaciones entre las clases descriptas.

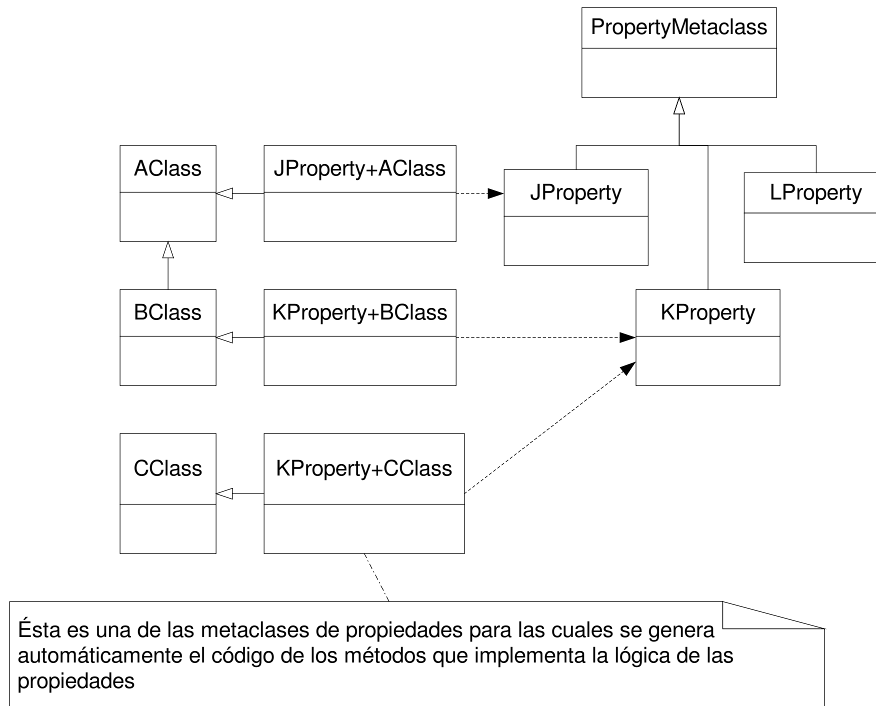


Figura 4.20: Diseño de propiedades de clases en NeoClasstalk

4.5.2 Cómo se asignan y combinan propiedades de clase

La asignación de una propiedad se logra creando una nueva metaclase, instancia de la subclase de *PropertyMetaclass* correspondiente. La nueva metaclase debe ser subclase de la metaclase original de la clase que queremos que tenga la propiedad. Finalmente la clase debe ser instancias de nuestra nueva metaclase.

La composición de propiedades de clases se obtiene por el encadenamiento de metaclases de propiedades de clases a través de una jerarquía de herencia simple, siendo la raíz de esta jerarquía una metaclase de compatibilidad. Cada vez que se asigna una nueva propiedad a una clase, se crea una nueva metaclase de propiedad. Si es la primera propiedad que se asigna se la crea como subclase de la metaclase de compatibilidad de la clase en cuestión, en caso que ya existiera una propiedad se la crea como subclase de la última metaclase de propiedad. Esta nueva metaclase pasa a ser la nueva clase de la clase a la que se le está asignando la propiedad.

A continuación se muestra un esquema en el que se ejemplifican estos conceptos. En el mismo se expone la clase *A*, a la que se le definen las propiedades *J* y *K*.

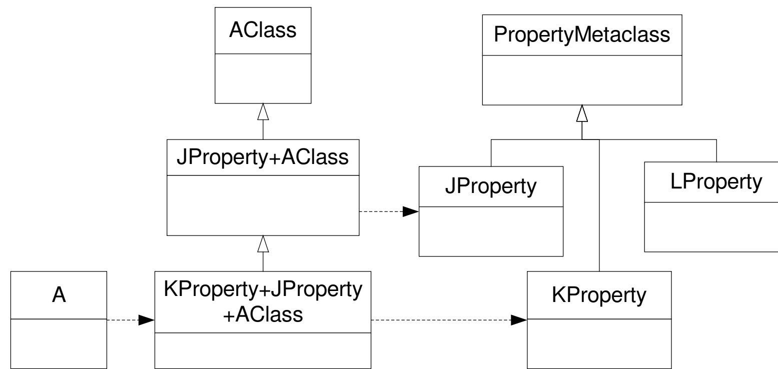


Figura 4.21: Asignación y combinación de propiedades en NeoClasstalk

Cuando se asignan dos o más propiedades que se implementan redefiniendo los mismos métodos, la aplicación efectiva de todas se logra determinando explícitamente en estos métodos que se ejecute la implementación de la superclase.

4.5.3 Cómo se asignan propiedades a clases de la jerarquía que modela lógica booleana

En NeoClasstalk la asignación de propiedades a las clases que modelan la lógica booleana se implementa mediante la definición de las jerarquías de clases y metaclases que se exponen a continuación.

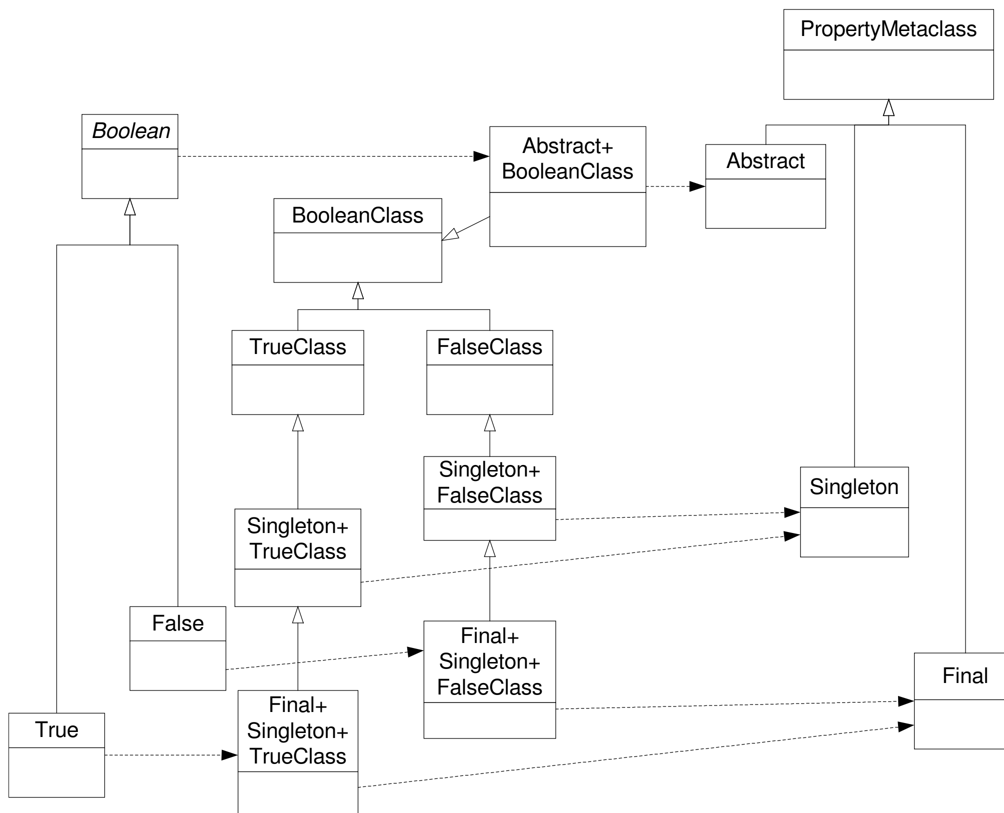


Figura 4.22: Asignación de propiedades a las clases que modelan lógica booleana en NeoClasstalk

La propiedad *abstracta* se asigna a la clase *Boolean* creando a esta clase como instancia de una metaclase de propiedad que implementa esta propiedad y que hereda de la metaclase de compatibilidad definida para esta clase.

Las propiedades *singleton* y *final* de la clase *True* se determinan de la siguiente manera. Primero se hace que una metaclase que implementa la propiedad *singleton* herede de su metaclase de compatibilidad. Luego se crea una metaclase que implementa la propiedad *final* como subclase de la metaclase de la propiedad anterior. Finalmente se crea a la clase *True* como instancia de la metaclase de propiedad final. De la misma manera se asignan estas propiedades a la clase *False*.

4.5.4 Compatibilidad de comunicación entre niveles

Se asegura de la misma manera que en Smalltalk debido a que se mantiene la jerarquía de metaclases paralela a la jerarquía de clases.

El agregado en la jerarquía de las metaclases que implementan las propiedades no trae problemas de compatibilidad. Estas metaclases heredan de la metaclase de compatibilidad de la clase a la que se quiere asignar la propiedad y no tienen ninguna relación con sus subclases de la jerarquía de compatibilidad y por lo tanto sólo tienen impacto en esta clase.

Veamos esto con un ejemplo.

Sea ***B*** subclase de ***A***.

Sea que a la clase ***A*** se le asigna la propiedad ***J***.

Sea que en la metaclase que implementa la propiedad ***J (JProperty+AClass)*** se define el método ***c***. Sea que en la implementación de ***c*** se envía el mensaje ***i*** a una instancia recientemente creada de la clase que recibe el mensaje ***c***.

El mensaje ***c*** se puede enviar a la clase ***A***, pero no a la clase ***B***, ya que la metaclase de ***B (BClass)*** no tiene ninguna relación de herencia con la metaclase ***JProperty+AClass*** y entonces no hereda la implementación de ***c***. Por lo tanto, no es necesario que ***B*** sepa responder al mensaje ***i***.

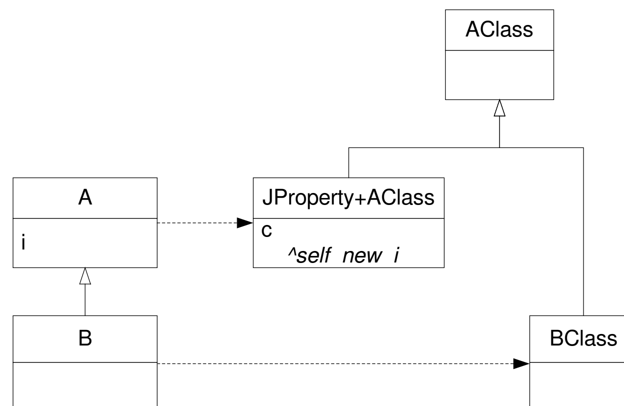


Figura 4.23: Las metaclases de propiedades no introducen problemas de incompatibilidad en NeoClasstalk

4.5.5 Propagación de propiedades de clase

En el modelo NeoClasstalk, se ataca este problema con la introducción de metaclases creadas específicamente para implementar propiedades. La propagación se evita ya que estas metaclases heredan de la metaclase de compatibilidad de una clase específica pero no están relacionadas con las metaclases de sus subclases.

El ejemplo expuesto en la sección inmediata anterior también sirve para ilustrar esto.

4.6 Modelo de mixins y metaclases en MetaclassTalk

MetaclassTalk [BS/00] [BS/02] [BS/03-1] es otra extensión de Smalltalk, donde las metaclases son explícitas. Este ambiente se usó para implementar el modelo de *mixins* que se detalla a continuación.

En este modelo se mantiene el modelo de compatibilidad implementado en NeoClasstalk y se siguen manejando los conceptos de metaclases de compatibilidad y metaclases de propiedades.

Se introduce el concepto de *mixin*. De acuerdo a [BC/90], *"Un mixin es una subclase abstracta que puede ser usada para especializar el comportamiento de distintas clases padres"*, es decir que a partir de una clase y un mixin se puede crear una subclase de la clase usando las definiciones del mixin.

Hay determinadas clases que son mixin y otras que no, se propone usar las clases mixin para modelar las propiedades de clases [Bou/03] [BS/03-2]. Una de las diferencias entre este modelo y el de NeoClasstalk, es que en aquel cuando se necesita una propiedad de clase se arma una metaclase a partir de definiciones guardadas en strings, es decir que no son compiladas en el momento de su armado sino en el uso. En cambio en este modelo, como los mixins son clases, el código provisto por los mismos se compila antes de usarlos.

Una clase (en particular una metaclase) puede ser subclase de una sola clase no mixin, pero puede heredar de una cantidad arbitraria de clases mixins.

Existen las metaclases de compatibilidad como en el modelo de NeoClasstalk. Se crean automáticamente al definirse una clase y siguen manteniendo las mismas relaciones de herencia respecto a la jerarquía de clases que son sus instancias.

Una clase es instancia de una metaclase que hereda tanto de su metaclase de compatibilidad como de los mixins que implementan las propiedades que se le quieren asociar.

Para enfrentar los posibles conflictos por el uso de herencia múltiple se establece un orden lineal entre una clase y sus superclases. La clase en cuestión encabeza esta lista. Las superclases que son mixins preceden a la superclase no mixin. A su vez, al definirse los mixins de una clase se establece también el orden de precedencia entre los mismos.

Veamos estos conceptos con un ejemplo. Supongamos que de acuerdo a las definiciones del dominio que se está modelando, la clase B hereda de la clase A. A la clase A se le quiere asociar la propiedad de clase asociada al mixin J. A la clase B se le quieren asignar las propiedades implementadas por los mixins K y L (mencionados en el orden de precedencia que se les quiere asignar). A continuación se expone un diagrama en el que se reflejan las relaciones definidas.

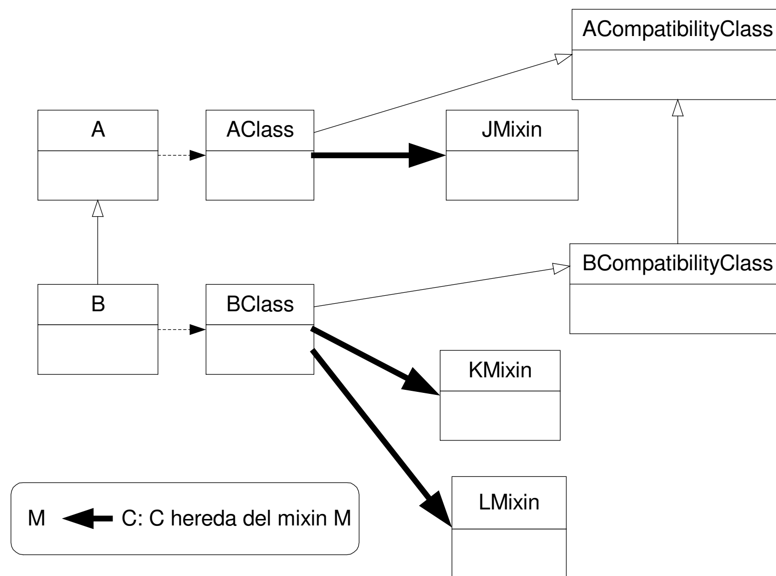


Figura 4.24: Metaclases de compatibilidad y mixins en MetaclassTalk

Conceptualmente se pueden establecer relaciones de herencia múltiple con los *mixins*. Sin embargo, la implementación del modelo está basada en el uso de la herencia simple. La obtención automática de la jerarquía simple a partir de las definiciones de herencia múltiples, está determinada por el orden lineal existente entre una clase y sus superclases.

La jerarquía de herencia simple se construye a partir de las clases que no sean mixins y clases generadas por los mixins que participan en las relaciones de herencia múltiple definidas. Se puede ver a los mixins como constructores de subclases ya que a partir de una clase crean una subclase de la misma con los mismos métodos y variables de instancias definidos por el mixin. Estas subclases son implícitas y no pueden ser modificadas manualmente.

La jerarquía de herencia simple que se obtiene a partir de la jerarquía que se expuso como ejemplo es la siguiente:

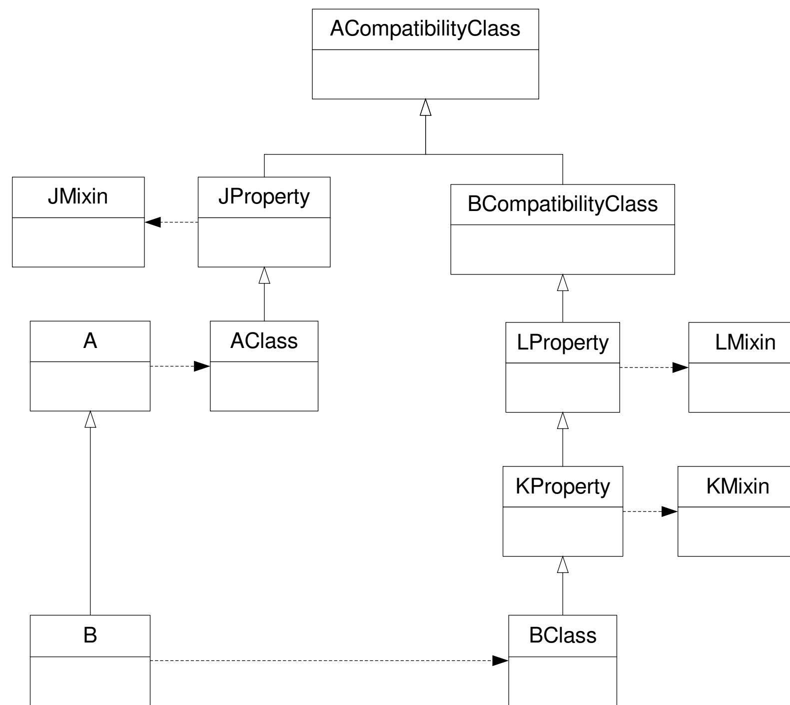


Figura 4.25: Jerarquía de herencia simple a partir de jerarquía de herencia múltiple de mixins en MetaclassTalk

4.6.1 Diseño del kernel de clases y metaclases

El kernel del modelo de MetaclassTalk queda definido conceptualmente de la misma forma que en ObjVlisp. Las clases primitivas que allí se llaman *Object* y *Class*, aquí se denominan *Object* y *StandardClass* respectivamente. El problema de la regresión infinita también se soluciona de la misma manera que en aquel.

La jerarquía basada en mixins está implementada mediante tres metaclases explícitas [Bou/03] [BS/03-2] que se detallan a continuación:

Mixin

Dado que se considera a los mixins como clases especiales, se describen los mismos usando esta metaclase. El conjunto de instancias de *Mixin* incluye metaclases que implementan propiedades de clase. Estas clases tienen las definiciones de variables de instancia y métodos que pasan a las clases que se generan a partir de las mismas. También tienen referencias a las clases generadas para actualizarlas cada vez que ocurre un cambio (por ejemplo el agregado o eliminación de métodos).

CompositeClass

Las instancias de esta metaclase son clases que conceptualmente pueden tener más de una superclase. Este es el caso de una metaclase de una clase que tiene asignada una o más propiedades de clase. Es responsabilidad de una *composite class* hacer que se cumplan las reglas de herencia de mixins. Prohíbe la herencia de más de una superclase no mixin o de dos superclases que tengan variables de instancia con el mismo nombre. Cuando se agregan o eliminan mixins, la *composite class* inserta o elimina de su jerarquía de herencia simple asociada, las clases generadas a partir de mixins.

GeneratedClass

Una *generated class* es una clase implícita construida por un *mixin*.

4.6.2 Cómo se asignan y combinan propiedades de clase

La asignación de una propiedad a una clase se logra haciendo que la metaclase de la misma herede del mixin que implementa la propiedad. Dos clases que tengan la misma propiedad tienen asociado el mismo mixin, lográndose con esto el reuso de la lógica de implementación.

La composición de propiedades de clases se logra mediante las definiciones de herencia múltiple entre los mixins y la metaclase de una determinada clase.

Los conflictos por nombres de métodos que pueden surgir por estas relaciones de herencia múltiple se enfrentan mediante el orden lineal que se establece entre las superclases de una clase y a partir del cual se arma la jerarquía de herencia simple que se mencionó anteriormente. Cuando se asignan dos o más propiedades que se implementan redefiniendo los mismos métodos, la aplicación efectiva de todas se logra determinando explícitamente en estos métodos que se ejecute la implementación de la superclase. Además, no se pueden asignar a una clase dos propiedades que utilicen para su implementación (a través de los mixins) variables de instancias con el mismo nombre.

4.6.3 Cómo se asignan propiedades a clases de la jerarquía que modela lógica booleana

En este modelo la asignación de propiedades a las clases que modelan la lógica booleana se implementa mediante la definición de las jerarquías de clases y metaclases que se exponen a continuación.

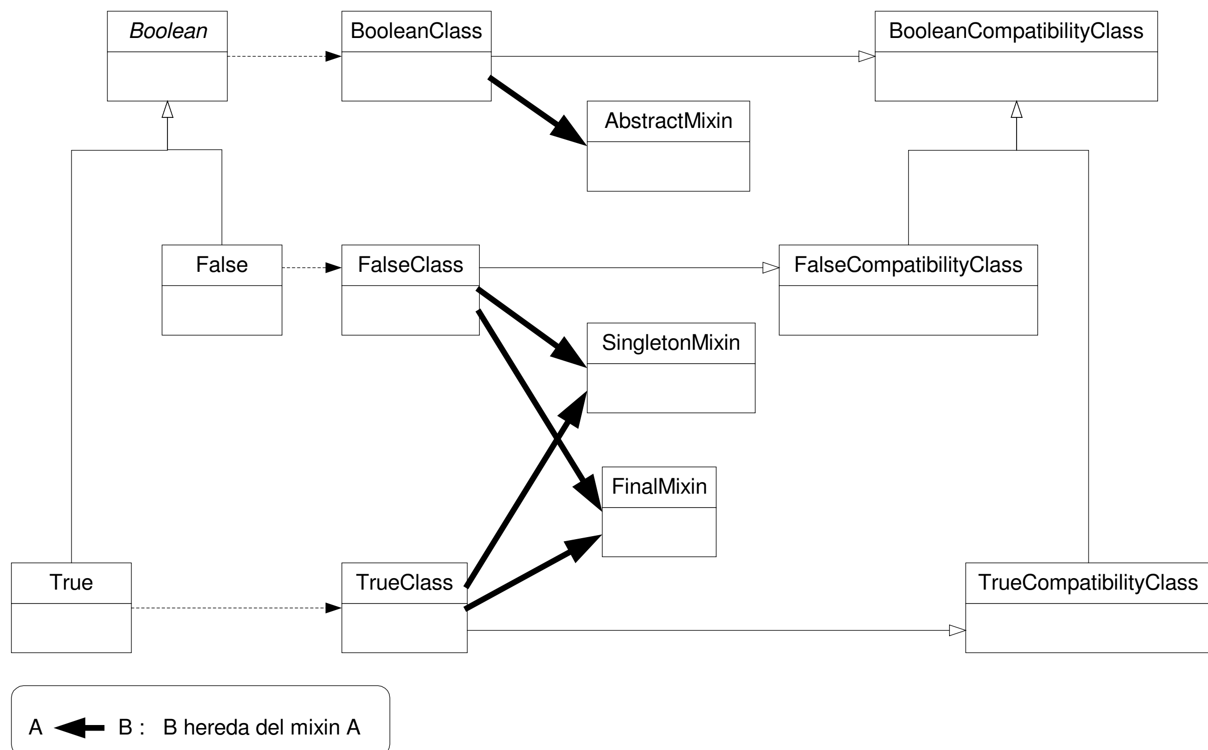


Figura 4.26: Asignación de propiedades a las clases que modelan lógica booleana en MetaclassTalk

La propiedad *abstracta* se asigna a la clase *Boolean* creando a esta clase como instancia de una metaclase que hereda tanto de la metaclase de compatibilidad asociada como de la clase mixin que implementa esta propiedad.

Las propiedades *singleton* y *final* de la clase *True* se determinan haciendo que esta clase sea instancia de una metaclase que herede de los mixins que implementan estas propiedades, además de la metaclase de compatibilidad asociada. De la misma manera se asignan estas propiedades a la clase *False*.

La compatibilidad se asegura en esta jerarquía manteniendo entre las clases las mismas relaciones de herencia que entre sus respectivas metaclases de compatibilidad.

A partir de esta jerarquía con relaciones (conceptuales) de herencia múltiple y teniendo en cuenta el orden lineal existente entre las superclases de una misma clase, se obtiene las siguiente jerarquía de herencia simple.

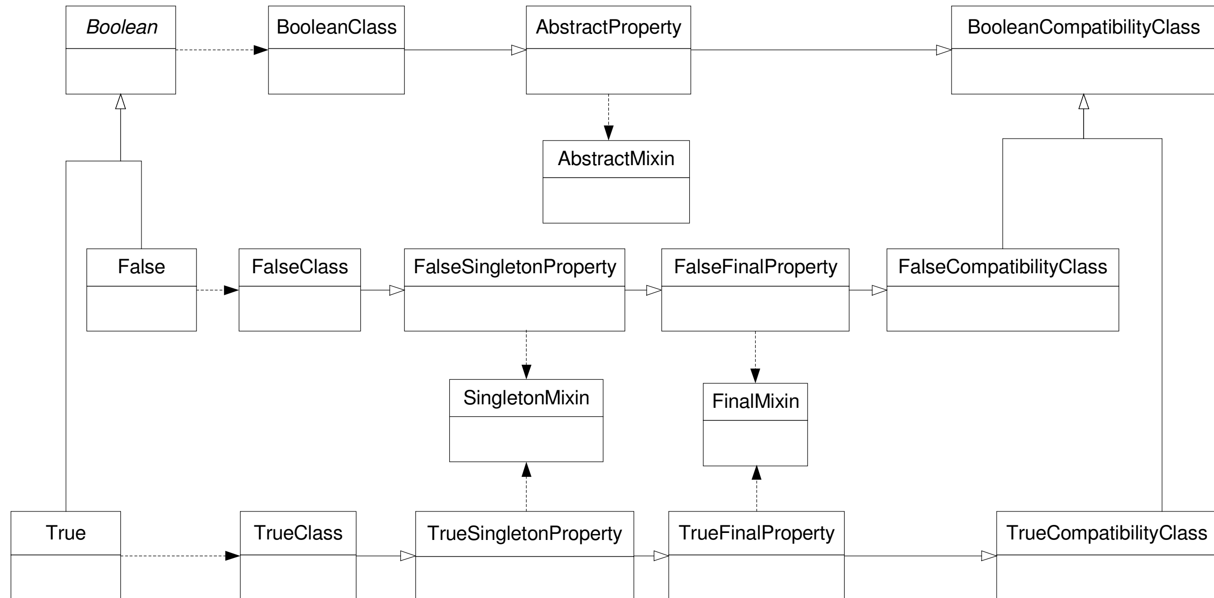


Figura 4.27: Jerarquía de herencia simple de modelo de lógica booleana a partir de jerarquía de herencia múltiple de mixins

Aquí se muestra la relación entre los mixins y las clases que implementan las propiedades, creadas a partir de los mismos y que son las que efectivamente intervienen en la jerarquía de metaclases resultante.

Se puede observar que en la jerarquía de cada una de las metaclases de la que son instancias las clases de esta jerarquía booleana, tienen precedencia las metaclases que implementan las propiedades respecto a las metaclases de compatibilidad. Además se debe notar que se siguen manteniendo las mismas relaciones de herencia entre las metaclases de compatibilidad que en la jerarquía anterior.

4.6.4 Compatibilidad de comunicación entre niveles

La metaclase de una clase, llamémosla *A*, hereda de los mixins de las propiedades que se le quieran asignar y de la metaclase de compatibilidad de *A*. Esta metaclase de compatibilidad a su vez hereda de la metaclase de compatibilidad de la superclase de *A*.

Teniendo en cuenta el orden lineal que se establece entre una clase y sus superclases, se puede verificar que la introducción de los mixins en la jerarquía no trae problemas de compatibilidad.

Consideremos el ejemplo de las clases *A* y *B* expuesto una sección anterior para mostrar esto.

El orden lineal entre la metaclase de *B* y sus superclases es el siguiente:

BClass , KProperty, LProperty, BCompatibilityClass, ACompatibilityClass.

El orden lineal entre la metaclase de *A* y sus superclases es el siguiente:

AClass , JProperty, ACompatibilityClass.

Esto muestra que las metaclases de compatibilidad de A y B se encuentran relacionadas de la misma manera que sus clases instancias.

4.6.5 Propagación de propiedades de clase

La asignación de propiedades de clase mediante las relaciones de herencia entre la metaclase de una clase y los mixins que implementan estas propiedades, permiten que se evite la propagación de propiedades a las subclases.

Esto se puede ver analizando las listas de superclases ordenadas de las metaclases. Observemos las listas de las metaclases de nuestro ejemplo, que se expusieron en la sección anterior. Se puede notar que la propiedad *J* asignada a la clase *A*, no se propaga a su subclase ya que la lista de superclases de la metaclase de *B*, no incluye a *JProperty*.

4.7 Metaclases y el modelo de traits

Un *trait* [SDNB/03] es una entidad funcional reusable que ofrece un determinado comportamiento. Se usan para definir partes del comportamiento de una clase. Al crearse una clase (en particular una metaclase) se la puede componer con un conjunto de *traits*. A su vez, un *trait* puede estar compuesto por otros *traits*.

Al igual que una clase, un trait provee un conjunto de métodos que implementan su comportamiento. Pero a diferencia de aquella, un trait puede requerir explícitamente la implementación de un determinado conjunto de métodos para poder efectivizar su comportamiento. Los requerimientos no satisfechos de los *subtraits* que componen un trait se convierten en métodos requeridos del trait compuesto. Estos métodos requeridos pueden ser implementados por la clase que usa el trait, por una superclase directa o indirecta o por otro trait que la clase incluya. Una de las razones de la necesidad de estos métodos es que en los *traits* no se definen variables de instancias, por lo tanto sus métodos nunca las acceden directamente, sino que, en caso necesario, lo hacen a través de métodos que requeridos.

El resultado de componer una clase con un trait es el mismo que si se hubiera construido a la clase con todos los métodos definidos en el trait (y en los traits de los que está compuesto). Además, cuando en algún método implementado en un trait se indica que se ejecute la implementación siguiente, esto tiene el mismo significado que en un método implementado en la clase, es decir que se comenzará la búsqueda del método en la superclase de la clase que usa el trait.

En este modelo, se propone la implementación de una propiedad de clase mediante la definición de un trait [DSW/04].

El modelo de metaclases se basa en el de Smalltalk, y se mantienen como en aquel, el paralelismo entre las jerarquías de clases y metaclases.

4.7.1 Diseño del kernel de clases y metaclases

El kernel de este modelo es una extensión del de Smalltalk. Las clases *Behavior*, *ClassDescription*, *Metaclass* y *Class* se siguen manteniendo, pero se les agrega funcionalidad para soportar la composición de clases con traits.

Para modelar los traits se usó un diseño análogo al que existe para modelar las clases. Así surgen las clases *TraitBehavior*, *TraitDescription* y *Trait*, que se relacionan entre sí de la misma forma que sus clases pares de la otra jerarquía mencionada. Como su nombre lo indica, la clase *Trait* modela los traits.

Se introduce también la clase *PureBehvavior* para factorizar el comportamiento común entre *Behavior* y *TraitBehavior*. Una de las diferencias principales entre estas dos clases es que *Behavior* tiene definiciones relacionadas con la superclase y las variables de instancia; esta funcionalidad no se necesita para los traits.

Se expone a continuación el diagrama detallado de las clases que conforman el kernel:

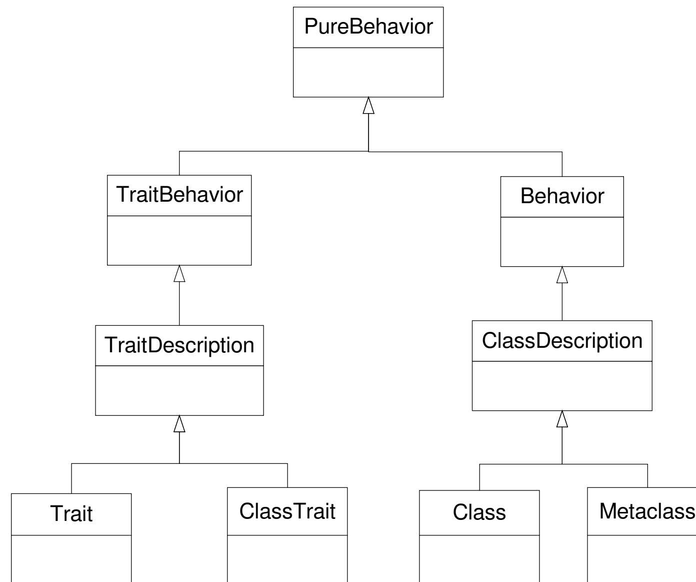


Figura 4.28:Diseño del kernel de clases y metaclases en el modelo de traits

Parte del comportamiento que definen las clases que modelan el kernel de este modelo se define mediante la composición con traits.

4.7.2 Cómo se asignan y combinan propiedades de clase

En este modelo la asignación y combinación de propiedades de una clase se logra componiendo a su metaclase con todos los traits que implementen tales propiedades. De esta manera se consigue el reuso de la lógica de implementación de las propiedades de clase, ya que todas las metaclases que necesiten definir una misma propiedad se componen con el mismo trait.

Los conflictos que puedan surgir por la composición de dos propiedades que implementen métodos con los mismos nombres se deben resolver explícitamente a nivel de la metaclase.

El orden de composición de los traits de una metaclase no se considera para resolver conflictos que pueden surgir por nombrar a métodos con un mismo selector en los distintos traits. Como se expuso anteriormente, un trait puede estar compuesto de otros traits, también en este caso es irrelevante el orden de composición de los subtraits para resolver los conflictos de métodos.

Estos conflictos se detectan automáticamente y se deben resolver explícitamente. Para solucionar estos casos se ofrecen los mecanismos que se detallan a continuación:

- Una solución posible es definir un método con el mismo selector que el de los métodos conflictivos en la clase o en el trait compuesto. Esto soluciona el conflicto debido a las reglas de precedencia (que se detallan más adelante) entre los métodos definidos en una clase o trait compuesto respecto a los que se definen en sus traits componentes.
- Para poder acceder a los métodos conflictivos, se cuenta con *alias*. Los alias se usan desde una clase o de un trait compuesto para hacer que el método de un trait componente se encuentre disponible bajo otro nombre.

- La composición de traits también soporta una operación de exclusión, que permite excluir un método de un trait cuando se lo compone y de esta manera el trait compuesto o la clase puede adquirir la implementación de un método con el mismo nombre de otro trait sin que surja conflicto alguno.

Por otro lado, sí existen reglas automáticas de precedencia para resolver conflictos entre los métodos definidos en una clase y los de sus traits. Estas reglas son:

- Los métodos definidos en una clase tienen precedencia respecto a aquellos definidos en los traits que usa esta clase.
- Los métodos definidos en un trait tienen precedencia respecto a los definidos en la superclase de la clase en cuestión.

Respecto a la relación entre un trait y sus subtraits, la regla es que los métodos definidos en el trait compuesto preceden a los de sus traits componentes.

4.7.3 Cómo se asignan propiedades a clases de la jerarquía que modela lógica booleana

En este modelo la asignación de propiedades a las clases que modelan la lógica booleana se implementa mediante la definición de las jerarquías de clases y metaclases que se exponen a continuación.

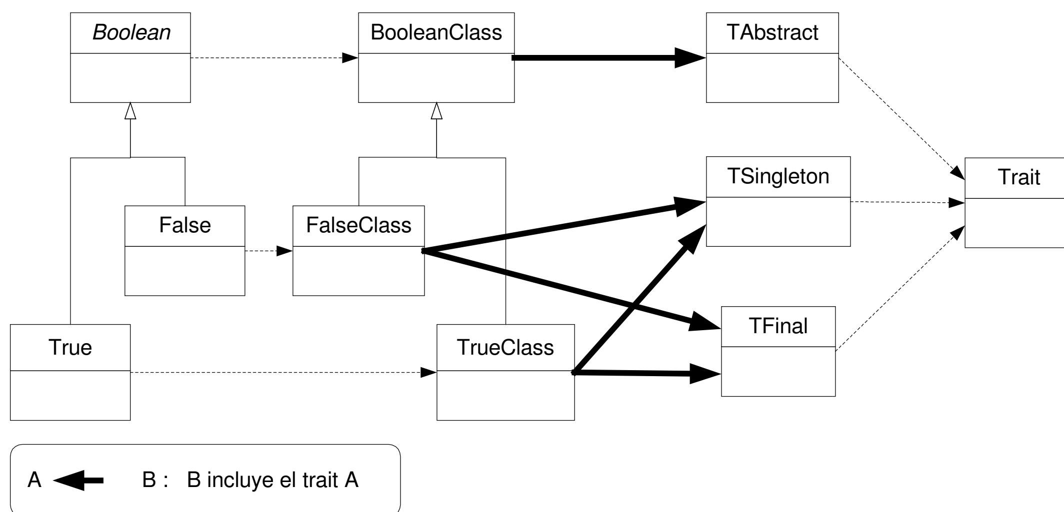


Figura 4.29: Asignación de propiedades a las clases que modelan lógica booleana en el modelo de traits

La propiedad *abstracta* se asigna a la clase *Boolean* creando a esta clase como instancia de una metaclase que está compuesta con el trait *TAbstract*.

Las propiedades *singleton* y *final* de la clase *True* se determinan haciendo que esta clase sea instancia de una metaclase compuesta con los traits que implementan estas propiedades. De la misma manera se asignan estas propiedades a la clase *False*.

Por lo explicado anteriormente, la propiedad *abstracta* definida para la clase *Boolean* se propaga automáticamente a las clases *True* y *False*. Pero en este caso particular, la propagación se anularía. Esto es debido a que las subclases tienen la propiedad *singleton* y tanto esta propiedad como la *abstracta* se implementan redefiniendo el método de creación de instancias, por lo tanto la redefinición del método de creación hecha en *TSingleton* anularía para las clases *True* y *False,* el efecto de la redefinición realizada en *TAbstract*.

4.7.4 Compatibilidad de comunicación entre niveles

Se puede ver a los traits como un medio para extender el comportamiento definido por una clase mediante el agregado de mensajes a su protocolo, lo que implica que el uso de traits por sí solo no puede introducir problemas de compatibilidad.

Por otro lado, considerando que en este modelo se mantiene el paralelismo entre las jerarquías de clases y metaclases como en Smalltalk, se verifica que este modelo asegura tanto la compatibilidad downward como upward.

4.7.5 Propagación de propiedades de clase

Las propiedades de una clase se definen mediante la asignación de traits a su metaclase. Como se explicó anteriormente, los métodos de un trait se tratan de la misma manera que los métodos definidos en la propia clase a los que están asociados, o sea que son heredados por las subclases. Luego, las subclases de una metaclase que tiene definida una determinada propiedad, heredan esta propiedad. Por lo tanto, en este modelo no se evita automáticamente la propagación de propiedades.

4.8 Resumen del análisis de los modelos

A modo de conclusión de este capítulo, se expone un cuadro comparativo en el que se resume cómo se manejan los distintos aspectos relacionados con el concepto de propiedad de clase en cada uno de los modelos analizados.

	Implementación de propiedad	***Reuso lógica***	***Asignación de propiedad***	***Combinación de propiedades***	***Compatibilidad de comunicación entre niveles***		***Propa-gación de propie-dades***
					Upward	***Down-ward***	
Smalltalk	En implementación de métodos definidos en metaclases implícitas	No	Modificación de método definido en metaclase implícita	Modificación de métodos definidos en metaclase implícita	Se asegura	Se asegura	Existe
CLOS	Creación de metaclase específica	Si	La clase es instancia de la metaclase que implementa la propiedad.	Herencia múltiple entre metaclases	No se asegura	No se asegura	No existe
ObjVlisp	Creación de metaclase específica	Si	La clase es instancia de la metaclase que implementa la propiedad.	Herencia múltiple entre metaclases	No se asegura	No se asegura	No existe
SOM	Creación de metaclase específica	Si	La clase es instancia de la metaclase que implementa la propiedad. Generación de metaclase derivada	Herencia múltiple entre metaclases. Generación de metaclase derivada	Se asegura	No se asegura	Existe
NeoClasstalk	Creación de subclase de *MetaProperty*	Si	La clase es instancia de una metaclase de propiedad.	Herencia entre metaclases de propiedades	Se asegura	Se asegura	No existe
MetaclassTalk	Definición de mixin	Si	Herencia entre metaclase y mixin. Generación de jerarquía de herencia simple entre metaclase y clase generada por mixin	Herencia múltiple entre metaclase y mixins. Generación de jerarquía de herencia simple entre metaclase y clases generadas por mixins	Se asegura	Se asegura	No existe
Traits	Definición de trait	Si	Composición de clase con trait. Se extiende el comportamiento de la clase.	Composición de clase con traits. Se extiende el comportamiento de la clase.	Se asegura	Se asegura	Existe

5 Reificación del concepto de propiedad de clase

Estableciendo como contexto de este trabajo a los ambientes de programación orientada a objetos basados en clasificación, el interés principal es identificar distintos aspectos del comportamiento de clases que puedan ser considerados como propiedades y armar una categorización de las mismas.

Esta categorización se tomará como base para reificar el concepto de propiedad de clase con el objetivo de armar un modelo de objetos que represente este problema y que permita la asignación dinámica de estas propiedades.

Nuestro modelo debería contar con las siguientes características:

- **Asignar y combinar dinámicamente las propiedades.** Debe permitir asignar una o más propiedades a una clase dinámicamente. Para lograr esto, se propone definir objetos que modelen las propiedades simples y el uso del mecanismo de composición en contraposición al uso de herencia, tanto para asignar una propiedad como para combinarlas en una clase. De esta manera, contando con un conjunto de propiedades básicas y ensamblándolas de la manera adecuada, se puede conseguir que una clase cuente con las propiedades buscadas.
- **Solucionar el problema de la propagación no deseada de propiedades de clase.**
- **Asegurar la compatibilidad de comunicación entre el nivel de las clases y el de las instancias**.
- **Asegurar dinámicamente la validez de propiedades asignadas.** Se debe asegurar dinámicamente que sean consistentes las asignaciones y las combinaciones de propiedades que se realicen. Ejemplo: no se debe permitir agregar propiedades duplicadas, se debe verificar consistencia entre una nueva propiedad y el estado o contexto de la clase a la cual se la quiere asociar y se deben detectar conflictos o incompatibilidades entre propiedades.

5.1 ¿Por qué crear un nuevo modelo de metapropiedades?

En secciones anteriores de este informe se analizaron modelos en los que se trata a las clases como objetos y en los que se maneja el concepto de propiedad de clase.

Creemos que ninguno de estos modelos cumple con todos los objetivos propuestos en este trabajo. A continuación se detalla la justificación de esta afirmación.

Ausencia de modelo de metapropiedades y propagación de propiedades en Smalltalk

En los ambientes actuales de Smalltalk, si bien se le pueden definir propiedades a las clases, no está reificado este concepto ni tampoco existen herramientas para este propósito. Comúnmente cuando se quiere lograr esto, se modifican los correspondientes métodos de clase para incorporar el comportamiento esperado. Por ejemplo, supongamos que se quiere que una clase sea *singleton* y que su *inicialización sea automática*. En este caso, en la metaclase asociada se redefine el método de creación de instancias para controlar que no se cree más de una instancia y en caso de crearla hacer que se inicialice. De esta manera, la lógica de una determinada propiedad se repite cada vez que se quiere que una clase cuente con la misma.

Por otro lado, como se explicó anteriormente, debido al paralelismo entre las jerarquías de clases y metaclases y a que en general la implementación de propiedades se realiza modificando los métodos definidos en las metaclases, en estos ambientes se produce la propagación de propiedades.

La propagación es un problema en este modelo que tiene su raíz en el uso de la herencia para resolver cuestiones que deberían atacarse mediante la creación de nuevos objetos que las modelen y componiendo a las clases con estos objetos.

La creación de objetos es un ejemplo de esto. En Smalltalk, esto habitualmente se lleva a cabo enviando el mensaje *new* o *new:* a la clase que corresponda. Al ser un mensaje, su implementación es heredada por todas las subclases. Con las clases y sus relaciones de herencia se intenta modelar un dominio particular. La jerarquía de metaclases es paralela a la de clases, es decir, está determinada por el modelo de ese dominio particular. Luego, si vemos a la herencia como un medio por el cual se expresan las semejanzas y diferencias de los objetos que conforman el dominio modelado, entonces la creación de objetos, que en general es ajena a lo esencial de este dominio, no tendría por qué ser determinada por el mecanismo de herencia en esta jerarquía de metaclases.

El mismo análisis podría aplicarse a la categoría de métodos de las clases que permiten determinar relaciones de herencia entre las mismas.

Creación de metaclases específicas para asignar propiedades

En los otros modelos presentados, se ofrecen mecanismos específicos para asignar propiedades a las clases y lograr el reuso de la lógica de implementación de cada una de las propiedades.

En general en todos ellos se ofrecen soluciones en las que, de una u otra manera, se terminan generando metaclases específicas para implementar cada una de las propiedades.

Creemos que conceptualmente no es adecuado que una propiedad de clase esté modelada como una metaclase, ya que su esencia no se corresponde con la que se define par los objetos metaclase, que es la de ser clase de otra clase. Más bien consideramos que el comportamiento de las propiedades es ortogonal al que se modela con las metaclases, ya que podrían verse como atributos de las clases.

El uso de metaclases para definir propiedades se da por falta de objetos que representen las propiedades y que sean independientes de la forma en que esté implementado el modelo de metaclases.

Por otro lado, al implementar propiedades con metaclases, no se consigue que la asignación se realice dinámicamente, ya que asignar una propiedad a una clase implica finalmente modificar de alguna manera a su metaclase. Y esto hace que se tenga que interactuar con el metamodelo y utilizar mecanismos propios de este metanivel, que en general son más lentos y tienen riesgo de producir efectos colaterales más amplios.

Uso de herencia para asignar y combinar propiedades

En la mayoría de los modelos analizados, permitiendo que las metaclases se manejen explícitamente, se propone el uso de relaciones de herencia entre metaclases para asignar una o más propiedades a una clase.

Por ejemplo en Clos, ObjVlisp y SOM, si se quiere que una clase tenga determinadas propiedades, se define su metaclase como subclase de las metaclases que implementan tales propiedades, es decir se usan relaciones de herencia múltiple.

En NeoClasstalk y MetaclassTalk, la combinación de propiedades de clases se obtiene mediante el encadenamiento de metaclases de propiedades de clases a través de una jerarquía de herencia simple, siendo la raíz de esta jerarquía la llamada metaclase de compatibilidad, es decir la metaclase donde se encuentra la funcionalidad de una clase relacionada con el dominio particular sobre el que se está trabajando.

En el modelo que usa traits, se utiliza el mecanismo de composición para asignar propiedades, pero al momento de su aplicación se las trata como un conjunto de métodos agregados a la metaclase. La clase no colabora con el trait para que la propiedad se aplique, sino que se usan las reglas comunes de herencia para determinar si un método definido en un trait asignado se debe ejecutar.

Las objeciones que planteamos al uso de la herencia para resolver las asignaciones y combinaciones de propiedades son las siguientes:

- El uso de la herencia atenta contra el dinamismo que se quiere lograr. Esto es así, ya que modificar estas relaciones implica que se tenga que interactuar con el metamodelo y utilizar mecanismos de este nivel que, como se mencionó antes, en general son más lentos y tienen el riesgo de tener efectos colaterales más amplios.

- Las relaciones de herencia no reflejan adecuadamente la forma en que se relaciona una clase con sus propiedades. Las relaciones de herencia establecen un vínculo de *"es un"* entre sus intervinientes, es decir definen que la subclase es una especialización más concreta de su superclase. Este vínculo debería establecerse cuando en el dominio modelado se detecta esta relación. Pero, analizando por ejemplo la propuesta de NeoClasstalk, se ve que este no es el caso. Si consideramos una propiedad y la funcionalidad de una clase, propia de un dominio particular, comprobamos que no es que la propiedad sea una especialización de esta funcionalidad, sino, como se mencionó anteriormente, que se trata de un comportamiento ortogonal al mismo.

Lo mismo sucede con la funcionalidad de las distintas propiedades entre sí, que también se relacionan con herencia simple al asignarlas a una misma clase, cuando en realidad se busca solamente un mecanismo de combinación, no de especialización.

- Un análisis similar se puede aplicar para cada uno de los modelos en que se usan relaciones de herencia múltiples y en los que, en forma contraria al modelo NeoClasstalk, las metaclases de propiedades son superclases de la metaclase de la clase en cuestión.

- El uso de la herencia deja como resultado modelos más difíciles de entender ya que provoca que se entremezclen conceptos de un dominio particular con el propio de las metapropiedades.

- En particular el uso de la herencia múltiple de por sí complica aún más los modelos e implica el surgimiento de conflictos para los cuales en general no existe solución que pueda ser automatizada. El uso de la herencia múltiple surge como recurso implementativo y no porque en el modelo se hayan detectado este tipo de relaciones. Si existieran objetos que modelen las metapropiedades en forma independiente al modelo de las metaclases, este uso de la herencia se podría reemplazar por una relación de composición entre las propiedades y la clase a la que se le quiere asignar las propiedades.

Ausencia de modelo integral de metapropiedades

En ninguno de los modelos analizados, el dominio de metapropiedades está modelado en forma integral. Veamos por qué decimos esto.

En primer lugar, no se lo considera como un dominio a modelar independiente del de metaclases.

Además no está propiamente reificado el concepto de metapropiedades ya que no se trata a las propiedades como verdaderos objetos de primera clase.

Por ejemplo, tanto en NeoClasstalk como en MetaclassTalk y Traits, para implementar la lógica de las propiedades se crean instancias o subclases de clases específicas (*PropertyMetaclass*, *Mixin* y *Trait*, respectivamente) pero en ninguno de estos casos se crean objetos que modelen las propiedades específicamente. Veamos por qué afirmamos esto.

En NeoClasstalk y MetaclassTalk se usa a estos objetos como un medio para armar o modificar las jerarquías de metaclases para introducir las metaclases de propiedades. En NeoClasstalk, con el fin de lograr el reuso de la lógica de las propiedades, se define una metaclase por cada propiedad distinta que se quiera implementar, estas metaclases son las clases de las metaclases de propiedades que se crean al realizarse una asignación particular. Pero cuando se definen estas metaclases que se usan como templates, no se crean métodos, sino que se definen strings con el código de los métodos que van a crearse cuando se asigne la propiedad a una clase y se cree la metaclase de propiedad correspondiente. Esto implica que la lógica de la implementación de una propiedad recién se verifica estáticamente y se compila cuando se usa. En MetaclassTalk, con la introducción de los mixins se logra un mejor modelado de los templates de las metaclases de propiedades, ya que la compilación de los métodos que implementan las propiedades se realiza al definir los mixins. Pero nuevamente aquí, se utilizan a los mixins asignados para modificar las jerarquías de metaclases con la introducción de metaclases de propiedades. O sea, que en ambos modelos, en lugar de modelar la propiedad, se construye *"un lugar"* para poner los métodos que la implementan.

En el caso de Traits se los usa para extender la funcionalidad de la clase. Pero, para asignar una propiedad no basta que se componga a la clase con el trait correspondiente, sino que la clase debe implementar los métodos requeridos del trait y definir las variables de instancia-necesarias para que la funcionalidad del trait pueda llevarse a cabo. Esto implica que la lógica de una propiedad no está encapsulada en el trait y su aplicación no es transparente para el objeto que lo usa.

Por otro lado, no están modelados todos los conceptos y relaciones que surgen en este dominio. Por ejemplo, relaciones de incompatibilidad entre propiedades y orden en que se deben aplicar. Esta ausencia implica que no se pueda automatizar las validaciones de consistencia de aplicación de propiedades y que tengan que manejarse manualmente cada vez que se asigna o aplica una propiedad.

5.2 ¿Por qué es importante contar con un modelo de metapropiedades como el que se propone?

Con el modelo propuesto se tiene como intención reificar el concepto de metapropiedad y establecer entre estos objetos y aquellos que quieran usarlos, una relación que refleje la ortogonalidad existente entre la funcionalidad propia de la propiedad y la propia del objeto que la use. Esto nos parece importante, ya que pretendemos que el modelo refleje la forma en que interpretamos al dominio conceptualmente.

Uno de los objetivos de contar con un modelo de objetos en el que se encuentren reificados los conceptos de metapropiedades es evitar que se mezcle la implementación de la funcionalidad de un dominio particular con la de los conceptos propios de metapropiedades. Sin la existencia de este modelo, los objetos y colaboraciones agregados para implementar una metapropiedad enturbian el código, lo enredan y estorban cuando se trata de arreglar o extender algo de la funcionalidad original. El código relacionado con la responsabilidad del objeto está perdido entre la maraña de lo que se agregó. Este problema empeora cuando se trata de asignar más de una metapropiedad, hasta que se hace muy difícil recuperar la idea original de lo que debía hacer el objeto. Gran parte del esfuerzo puesto en el diseño para lograr objetos con funcionalidades y responsabilidades claras y jerarquías reusables y extensibles en un modelo claro, comprensible y autodocumentado, se pierde así en la implementación de las metapropiedades que no encajan con la descomposición funcional del dominio original.

Otra de las ventajas de contar con un modelo de metapropiedades es que permite que el diseñador de una aplicación particular se focalice en la resolución de los problemas propios del dominio que está modelando y no se tenga que preocupar por la implementación de las metapropiedades que necesita usar (o por lo menos lo vea como un factor independiente) ni por cómo el uso de las mismas afectaría a su diseño. Es decir, que las propiedades se puedan determinar en forma transparente e independiente del diseño funcional realizado.

La existencia de un modelo así también es importante porque hace que la lógica de una determinada metapropiedad quede expresada en uno y sólo un lugar, pudiendo ser reusada en cada aplicación.

La propuesta de asignación dinámica de metapropiedades es importante, ya que al ser independiente de la compilación y poder realizarse en tiempo de ejecución, permite una mayor flexibilidad para poder agregarlas o quitarlas, o inclusive graduarlas; o ponerlas en ciertas partes del sistema y en otras no.

Este dinamismo se puede lograr usando el mecanismo de composición entre objetos, es decir haciendo que un objeto conozca a otro sólo a través de los servicios o protocolo que el objeto conocido ofrece. De esta manera, el uso de la composición por contraposición al uso de la herencia, hace que la asignación o modificación de propiedades tenga menos efectos colaterales en el contexto donde se están utilizando. Esto es así porque al cambiar una jerarquía de clases para asignar, modificar o eliminar una propiedad, esto puede tener impacto en las implementaciones de las subclases de la clase en cuestión, ya a que éstas conocen y pueden depender de la implementación de su(s) superclase(s). Con la composición sólo se requiere que el objeto que se compone cumpla con un determinado protocolo, independientemente de su implementación.

Además, la composición de objetos es más dinámica que la herencia. En cualquier ambiente de objetos, cambiar la composición de un objeto es una simple operación de

asignación. Esto no implica que se tenga que interactuar con el metamodelo, cosa que, como ya se explicó, sí sucede para el caso de la herencia.

Es importante también que el modelo garantice dinámicamente la validez y consistencia de las asignaciones y combinaciones de propiedades que se realicen. El objetivo es que estas restricciones estén definidas en forma genérica en el modelo, o a lo sumo se determinen en un solo lugar cuando se define una nueva propiedad, y que se verifiquen automáticamente en el momento de uso de las propiedades. Es decir, que a lo sumo la definición de restricciones sea responsabilidad de quien define una propiedad y no de quien la usa. De esta manera el que usa las propiedades no tiene que preocuparse por estas cuestiones recurrentemente cada vez que las asigna o combina, al mismo tiempo que se evita la duplicación de esta lógica.

Por razones que se expusieron en secciones anteriores, es importante que el modelo proponga una solución al problema de propagación de propiedades de clase y asegure la compatibilidad de comunicación entre el nivel de clases y de instancias. Respecto al problema de propagación de propiedades en particular, creemos que, la reificación del concepto de metapropiedad y la separación de implementación de los distintos dominios, junto con el uso del mecanismo de composición para asignar propiedades, son características que permiten solucionar este problema.

Contar con un modelo como el propuesto puede facilitar la tarea de diseñar una nueva propiedad. Con las abstracciones principales modeladas, se podría definir solamente lo específico de la nueva propiedad sin tener que volver a definir lo que es común a otras propiedades.

Finalmente, la existencia de un modelo de metapropiedades favorece el desarrollo de herramientas para facilitar tanto la definición de nuevas propiedades como su uso.

6 Elección del ambiente de trabajo e investigación

6.1 Metaprogramación

Antes de indicar el ambiente elegido para el desarrollo del presente trabajo, se explican algunos conceptos que ayudan a clarificar la justificación de la elección.

En un ambiente de objetos, la *metaprogramación* se refiere a la programación o diseño de los *metaobjetos* [Riv]. Se consideran *metaobjetos* a aquellos objetos que conforman el modelo de entidades que se usan para diseñar o programar a otros objetos. y que permiten que el ambiente se pueda ejecutar, es decir que los objetos se envíen y reciban mensajes. Así, como ejemplo de metaobjetos se pueden mencionar a las metaclases, los objetos que modelan a los métodos y sus activaciones, los mensajes, el compilador, los procesos, etc.

En este contexto, se entiende por *reificación* [Riv] al proceso por el cual se crea un modelo de objetos para un concepto presente en un determinado dominio.

Entendemos por *reflexión* [Met] a la habilidad de un sistema de actuar sobre sí mismo. Dependiendo de cuán reflexivo sea un sistema, puede en mayor o menor grado observar y analizar su propia ejecución y aún llegar a cambiar la forma en que se ejecuta. La reflexión es una característica interesante desde el punto de vista de la programación ya que por medio de la misma se logran ambientes de desarrollo más flexibles y adaptables a las necesidades del programador.

Contar con mecanismos de reflexión facilita el desarrollo de herramientas de programación tales como browsers y debuggers. La reflexión es también útil para separar los aspectos funcionales de aquellos que no lo son (tales como distribución, concurrencia, logging, etc). La adaptabilidad dinámica es otra característica importante que se puede conseguir en los sistemas reflexivos, por ejemplo los mecanismos de ejecución pueden evolucionar dinámicamente para optimizar el uso de recursos disponibles (memoria, uso del procesador, ancho de banda de la red, etc)

Finalmente, se pueden relacionar los conceptos definidos (*metaprogramación, reificación y reflexión*), determinando que los conceptos de *metaprogramación reificados* en un ambiente de objetos facilitan la implementación de capacidades *reflexivas*.

6.2 ¿Por qué Smalltalk?

Debido al modelo de metaclases de Smalltalk y sus capacidades de reflexión [Riv] [Duc/99], utilizaremos este ambiente de objetos para realizar nuestra investigación.

Por un lado, el modelo de metaclases ya soporta la compatibilidad de comunicación entre los niveles de clases y sus instancias, que es una de las características con las que se quiere contar en el modelo propuesto.

Además, parece correcto el uso del mantenimiento de las jerarquías paralelas de clases y metaclases para lograr este objetivo, ya que hay comportamiento de la aplicación particular cuya implementación se define al nivel de metaclases y es correcto heredar. Está bien que esto guarde un correlato con las relaciones definidas al nivel de las clases.

Por otro lado, usar Smalltalk como ambiente y herramienta de desarrollo permite tener a nuestra disposición un modelo de objetos muy potente respecto al dominio de la metaprogramación, que puede ser extendido tanto a nivel de diseño como de implementación. Las herramientas de programación están hechas en el mismo lenguaje y se pueden modificar, incluso la máquina virtual y el compilador son objetos con los que se puede interactuar.

Estas características hacen que Smalltalk sea considerado reflexivo y se consideran muy útiles para poder conseguir los objetivos descriptos.

6.3 ¿Por qué Squeak?

Squeak [IKMWK] es una implementación de Smalltalk altamente portátil, abierta y totalmente open-source. La máquina virtual de Squeak es la única que está escrita enteramente en Smalltalk, lo que permite que pueda ser modificada más fácilmente y generar nuevas versiones de la misma. Ésta es una de las características distintivas respecto a las otras versiones de Smalltalk en las que la implementación de la máquina virtual no puede ser modificada.

Squeak es actualmente el Smalltalk no comercial de referencia para la comunidad de objetos y sigue siendo revisado y corregido constantemente por ser un proyecto open-source.

Por estas razones consideramos que Squeak es la implementación de Smalltalk más flexible respecto a las modificaciones de metaprogramación que se pueden realizar.

Por otro lado, el hecho de que Squeak sea no comercial y open-source permite que nuestro aporte sea compartido más fácilmente con el resto de la comunidad relacionada con estos temas.

7 Definiciones y diseño de modelo de metapropiedades

A continuación se detalla el modelo de metapropiedades propuesto, que cumple con todas las características mencionadas.

7.1 Composición y aplicación de metapropiedades

7.1.1 Clasificación de propiedades y estrategias definidas por metapropiedades

Las propiedades de clase se pueden clasificar teniendo en cuenta a qué aspecto de la funcionalidad de una clase están asociadas. Como categorías de esta clasificación podemos distinguir, entre otras, a las siguientes:

- **Asociadas a la creación de instancias**. Ejemplos: abstracta, singleton, inicialización automática, notificación de nuevas instancias creadas, set.
- **Asociadas a la recepción de mensajes**. Ejemplos: traceable, breakpoints, verificación de pre y post condiciones, contador de mensajes.
- **Asociadas a la determinación de relaciones de herencia entre las clases**. Ejemplos: final.

En base al resultado de esta clasificación surge el concepto de *estrategia*[3] *definida por metapropiedades*, cuya razón de ser es la de especificar cómo debe reaccionar una clase cuando recibe alguno de los mensajes asociados a las funcionalidades de las categorías identificadas. De esta manera, por cada una de las categorías detectadas se debe modelar una estrategia de metapropiedades específica. Así, aparecen la estrategia de creación de instancias, la de recepción de mensajes y la de subclasificación [4]

Toda clase tiene como colaboradores a estas estrategias. La estrategia de creación de instancias determina qué debe hacer la clase al recibir el mensaje *new*, que tiene la responsabilidad de crear nuevas instancias. La estrategia de recepción de mensajes determina qué debe hacerse cuando cualquiera de las instancias de la clase recibe un mensaje. De esta manera, se logra la reificación de estas dos funcionalidades de una clase.

[3] Respecto al uso del término *estrategia*, se aclara que el mismo no interviene en los nombres en inglés de las clases y métodos que implementan el modelo, para que no parezca que se aplica el patrón de diseño *Strategy* [GHJV/95], cuando en realidad esto no es lo que sucede. Por este motivo en la implementación se encontrarán nombres en los que interviene el término *policy* en lugar de *strategy.*

[4] Respecto a esta última estrategia se aclara que no fue diseñada, por eso en lo sucesivo se hará referencia solamente a las otras dos.

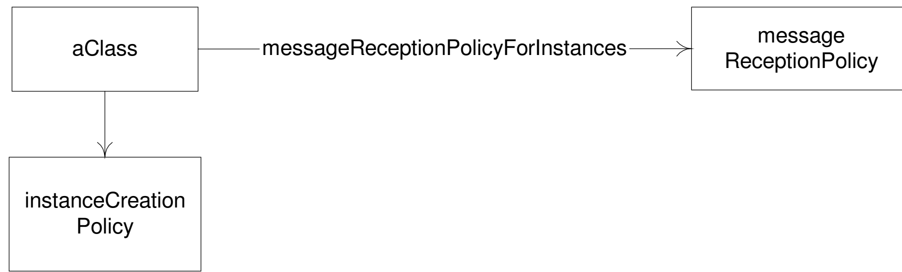


Figura 7.1: Estrategias de metapropiedades de una clase

Una estrategia se define mediante el agregado de propiedades que pertenezcan a la categoría correspondiente.

7.1.1.1 Estrategia de creación de instancias

Como se explicó anteriormente, toda clase tiene definida una estrategia de creación de instancias.

Esta estrategia determina qué se debe hacer cuando la clase recibe el mensaje *new* que se utiliza para crear nuevas instancias.

La estrategia se define mediante el agregado de propiedades de creación de instancias.

A continuación se presenta cómo está compuesta una estrategia de creación de instancias y cómo a partir de esta composición puede manejar las propiedades asociadas a la creación.

Definición y composición

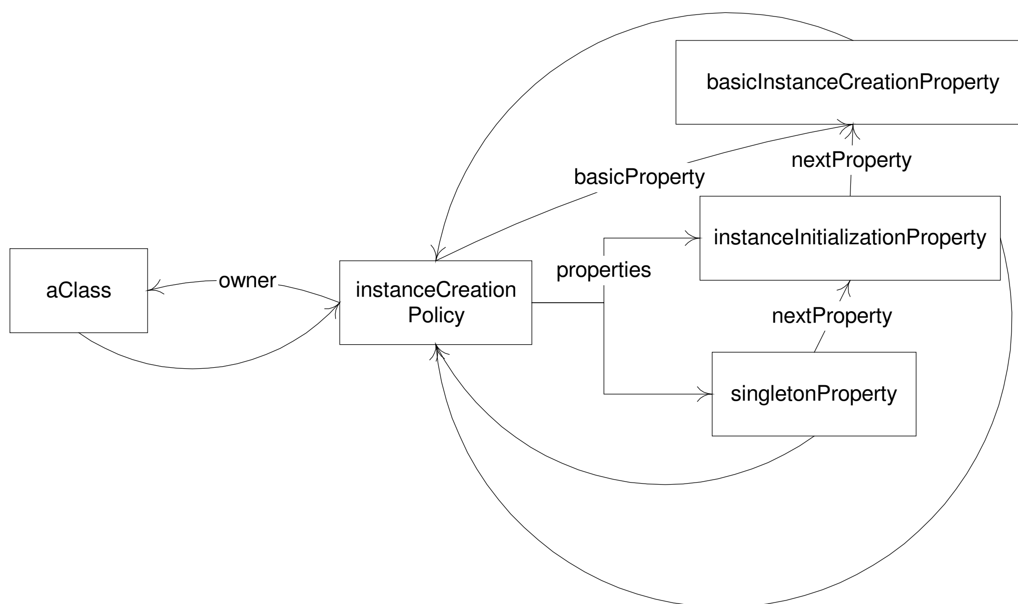


Figura 7.2: aClass tiene definida una estrategia de creación de instancias

En este diagrama se refleja que *aClass* tiene definida una estrategia de creación de instancias.

De acuerdo a las propiedades que tiene definida, esta estrategia determina lo siguiente:

- La clase es *singleton.*
- Al crearse una nueva instancia, ésta debe *inicializarse.*

La relación de orden que existe entre las propiedades se debe a restricciones respecto al momento de su aplicación, que, como se detalla más adelante, se definen en el momento en que se implementa una nueva propiedad.

La propiedad *basicInstanceCreationProperty* representa la propiedad que concretamente crea la nueva instancia. Se considera a ésta como la propiedad básica de la estrategia de creación de instancias. Cuando se crea una estrategia, nace con esta propiedad y siempre es la última propiedad en aplicarse.

Cómo se resuelve la creación de instancias cuando una clase tiene definida una estrategia de creación de instancias

En el siguiente diagrama se muestra cómo se resuelve la creación de instancias para el caso en que una clase tenga definida una estrategia como la que se describió en el diagrama anterior.

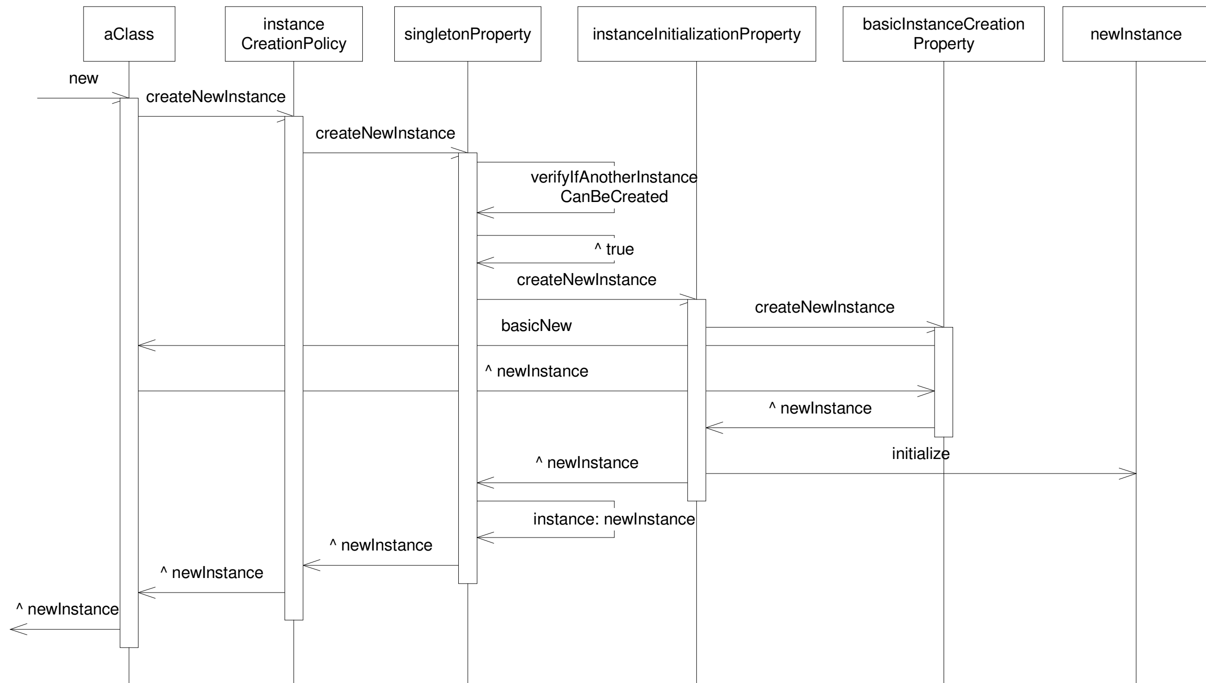


Figura 7.3: Ejecución y aplicación de una estrategia de creación de instancias

Como se puede observar *aClass* al recibir el mensaje *new* delega esta responsabilidad en su estrategia de creación de instancias, quien a su vez delega esto en la primera propiedad que deba aplicarse.

Luego de su ejecución cada propiedad, si corresponde (ya sea porque no es la última propiedad o porque su definición indica que debe hacerlo), reenvía el mensaje recibido a la siguiente propiedad para que realice lo que tenga que hacer.

7.1.1.2 Estrategia de Recepción de Mensajes

Como se explicó anteriormente, toda clase tiene definida una estrategia de recepción de mensajes.

Esta estrategia determina qué se debe hacer cuando alguna de las instancias de la clase recibe un mensaje.

La estrategia se define mediante el agregado de propiedades de recepción de mensajes.

Como estas metapropiedades caracterizan la recepción de mensajes de las **instancias** de la clase, surge naturalmente la idea de afectar la recepción de mensajes de **una** instancia determinada, de manera independiente de lo que definimos para las otras instancias de la clase.

De esto se desprende que se debe poder determinar una estrategia para un objeto en particular y que se debe reificar la recepción de mensajes a nivel de objeto. Además, es

adecuado que sea responsabilidad propia de una instancia determinar cuál es su estrategia de recepción de mensajes ya que es ella misma quien los recibe.

Pero esto no invalida que al nivel de una clase se pueda definir cómo deben manejar la recepción de mensajes todas sus instancias en general (o un conjunto de las mismas). En el momento en que un objeto recibe un mensaje se aplica su estrategia de recepción de mensajes ya sea definida en forma particular o armada a partir de lo definido a nivel de su clase para sus instancias.

Es por esto que modelamos la definición de recepción de mensajes para un determinado objeto. Luego, este modelo se extiende para poder definir una estrategia de recepción de mensajes para todas las instancias de una clase.

Esta flexibilidad de poder caracterizar la recepción de mensajes de una determinada instancia en forma particular no está presente en los otros modelos. En los mismos no se hace distinción de las propiedades de recepción de mensaje respecto a las otras en este sentido y se considera a la clase como única responsable de caracterizar a la recepción de mensajes de sus instancias.

A continuación se presenta cómo está compuesta una estrategia de recepción de mensajes y cómo puede cumplir con su responsabilidad.

Definición y composición

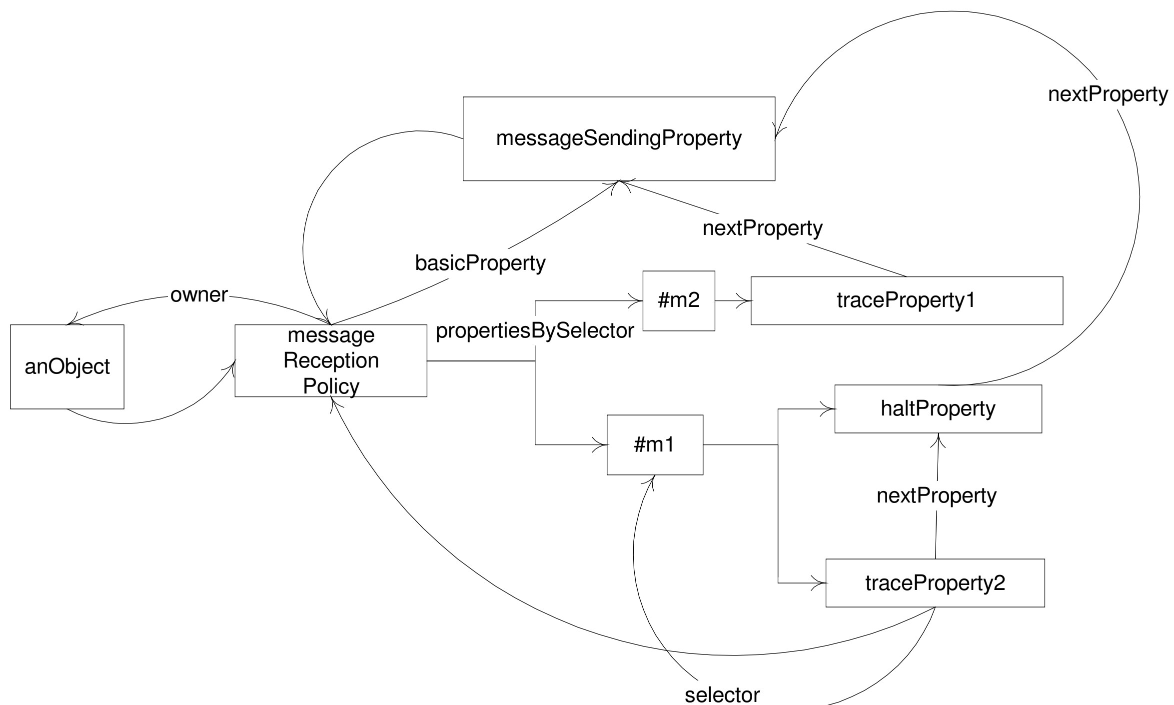


Figura 7.4: anObject tiene definida una estrategia de recepción de mensajes con propiedades definidas para determinados mensajes

En este diagrama se refleja que *anObject* tiene definida una estrategia de recepción de mensajes.

Esta estrategia determina que:

- se tiene que *trazar* el envío del mensaje *m1*
- se tiene que incluir un *halt* al recibir el mensaje *m1*

- se tiene que *trazar* el envío del mensaje *m2.*

La relación de orden que existe entre las propiedades se debe a restricciones respecto al momento de su aplicación. Como se detalla más adelante, este orden se define en el momento de implementación una nueva propiedad.

La propiedad *messageSendingProperty* representa la propiedad que ejecuta realmente el mensaje recibido. Se considera a ésta como la propiedad básica de la recepción de mensajes. Cuando se crea una estrategia, nace con esta propiedad y siempre es la última propiedad en aplicarse.

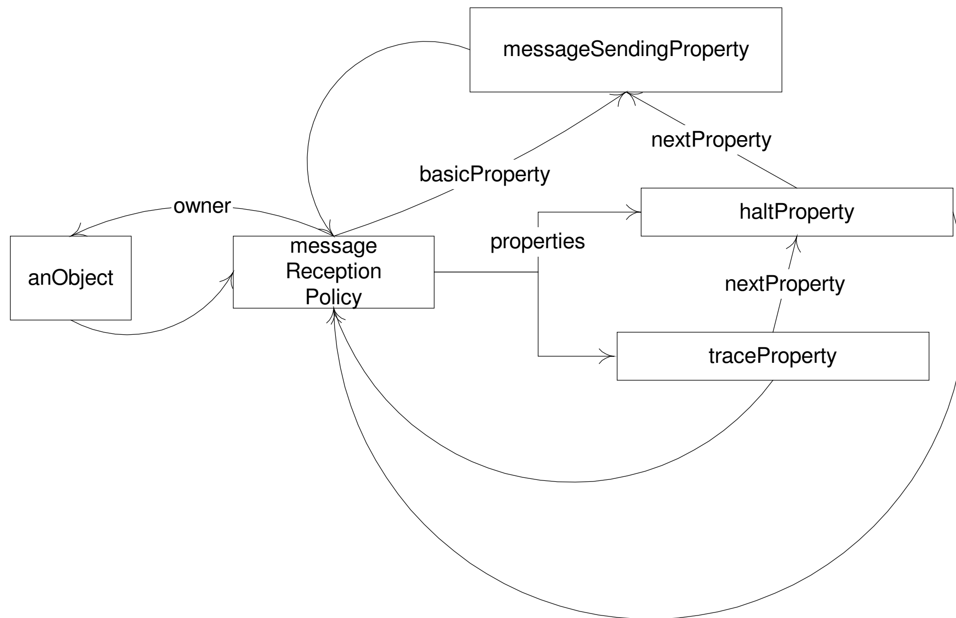


Figura 7.5:anObject tiene definida una estrategia de recepción de mensajes con propiedades que se aplican a cualquier mensaje

En este diagrama se muestra una estrategia de recepción de mensajes cuyas propiedades no fueron definidas para ningún mensaje en particular.

En este caso la estrategia determina que todos los mensajes que se envíen al objeto deben ser trazados y se les debe agregar un *halt*.

Cómo se resuelve la recepción de mensajes cuando un objeto tiene definida una estrategia de recepción de mensajes

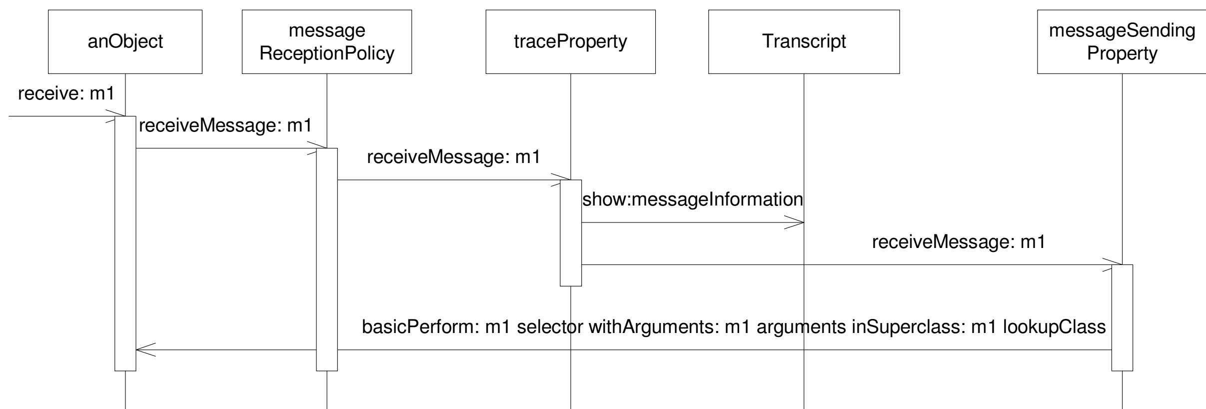


Figura 7.6: Ejecución y aplicación de estrategia de recepción de mensajes definida con propiedad trace para el mensaje m1

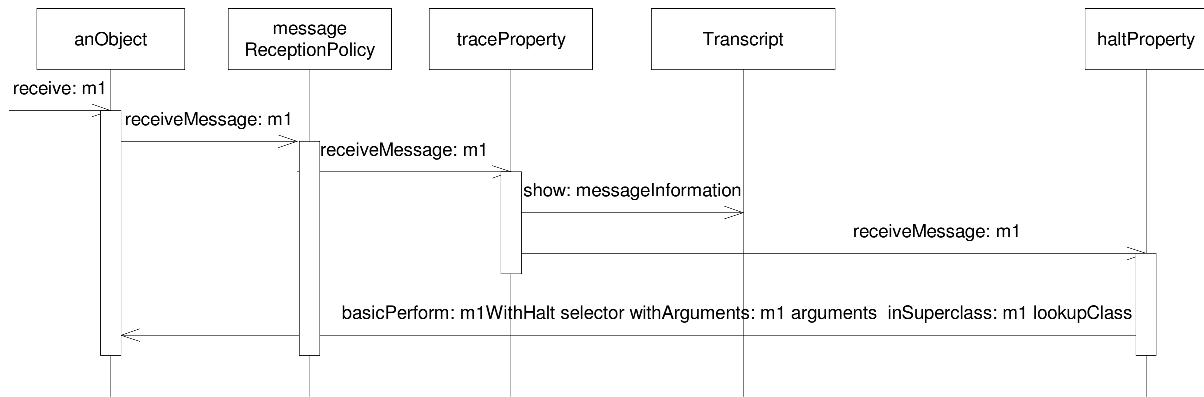


Figura 7.7: Ejecución y aplicación de estrategia de recepción de mensajes definida con propiedades trace y halt para mensaje m1

Aquí se muestra cómo *anObject* al recibir el mensaje *m1* delega esta responsabilidad en su estrategia de recepción de mensajes, quien a su vez delega esto en la primera propiedad que se encuentre definida para este mensaje.

Luego de su ejecución, cada propiedad reenvía el mensaje recibido a la siguiente propiedad para que realice lo que tenga definido. Esto puede cambiarse en la implementación de la propiedad y no se hace si es la última.

Cabe aclarar que se tuvo que modificar la máquina virtual de Squeak para lograr que a cualquier objeto se le pueda definir una estrategia de recepción de mensajes y para interceptar la recepción de mensajes para que se delegue esto en la estrategia definida. La forma en que se realizó esto se detalla más adelante.

Definición y composición de estrategia para instancias de una clase

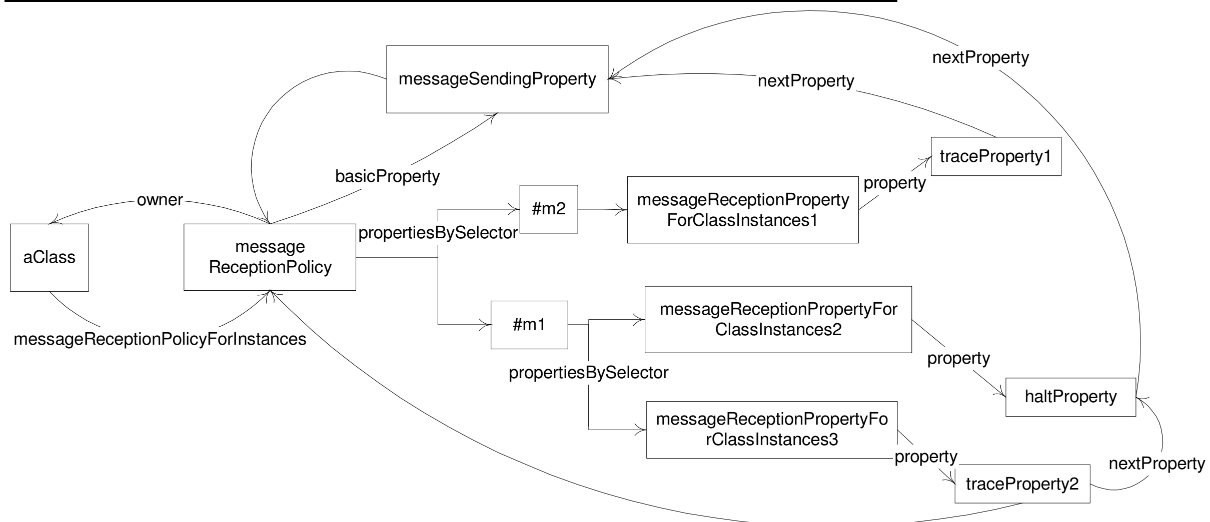


Figura 7.8: aClass tiene definida una estrategia de recepción de mensajes para sus instancias

Se puede observar que este diagrama es muy similar al diagrama de instancias en el que se representa a un objeto que tiene definida una estrategia de recepción de mensajes específica.

Una diferencia importante es que la clase tiene una estrategia de recepción **para sus instancias**, además de su propia estrategia, que puede tener como cualquier otro objeto.

Las otras diferencias se encuentran en los objetos que representan a las propiedades. Respecto a esto se puede observar que aparecen objetos *messageReceptionPropertyForClassInstances*.

Estos nuevos objetos conocen a propiedades que sí aparecieron en el diagrama anterior. La razón de que aparezcan estos nuevos objetos es que las propiedades definidas para una clase tienen atributos propios configurables. Uno de ellos sirve para determinar si la propiedad debe propagarse a las subclases, como se detalla más adelante.

Se observa que la propiedad básica de recepción de mensajes es de la misma clase que la de la estrategia definida para un objeto en particular. Esta propiedad no puede ser configurada, por lo tanto no necesita tener el comportamiento extra que se les agrega a las otras propiedades que se definen y configuran.

Cómo se determina la estrategia de recepción de mensajes asociada a un objeto

A continuación se explica cómo se determina la estrategia de recepción de mensajes de un objeto considerando que ésta puede estar determinada en forma particular para el objeto o bien que puede estar determinada por lo definido en su clase.

Caso 1: El objeto tiene definida una estrategia de recepción de mensajes específica

Si el objeto tiene definida una estrategia en forma particular, ésta es la estrategia que se aplica, independientemente de lo definido a nivel de su clase.

Caso 2: El objeto no tiene definida una estrategia de recepción de mensajes específica, pero su clase tiene definida una estrategia de recepción de mensajes para sus instancias

En este caso la estrategia del objeto se obtiene a partir de la estrategia para instancias definida en su clase.

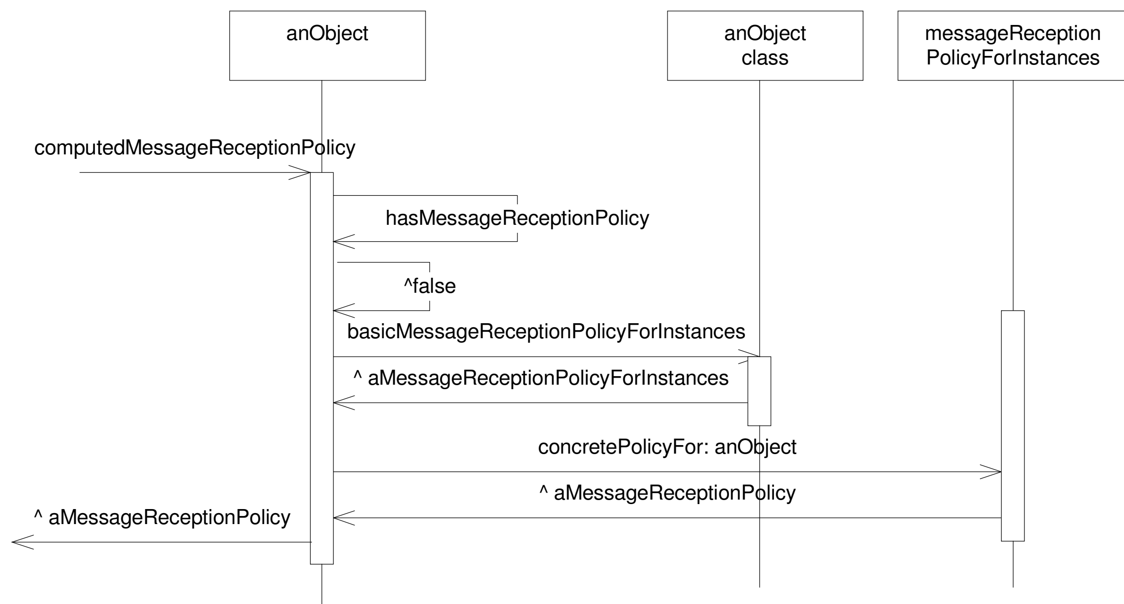


Figura 7.9: Determinación de estrategia de recepción de mensajes de un objeto cuya clase tiene definida una estrategia para sus instancias

Caso 3: El objeto no tiene definida una estrategia de recepción de mensajes específica y su clase tampoco tiene definida una estrategia de recepción de mensajes para sus instancias

En este caso, se determina que el objeto no tiene ninguna estrategia de recepción de mensajes definida y la recepción de mensajes se maneja sin delegarse a ninguna estrategia, en la forma tradicional.

Cómo se crea una estrategia de recepción de mensajes para un objeto en particular a partir de la estrategia definida para las instancias de la clase de este objeto

Cuando un objeto no tiene definida una estrategia en particular, pero su clase sí tiene definida una estrategia para sus instancias, la estrategia del objeto se obtiene a partir de la estrategia definida en la clase. Esto se logra enviando el mensaje *concretePolicyFor:* (que recibe como parámetro al objeto) a la estrategia para instancias. Este diagrama refleja cómo se lleva a cabo esta operación (básicamente una copia).

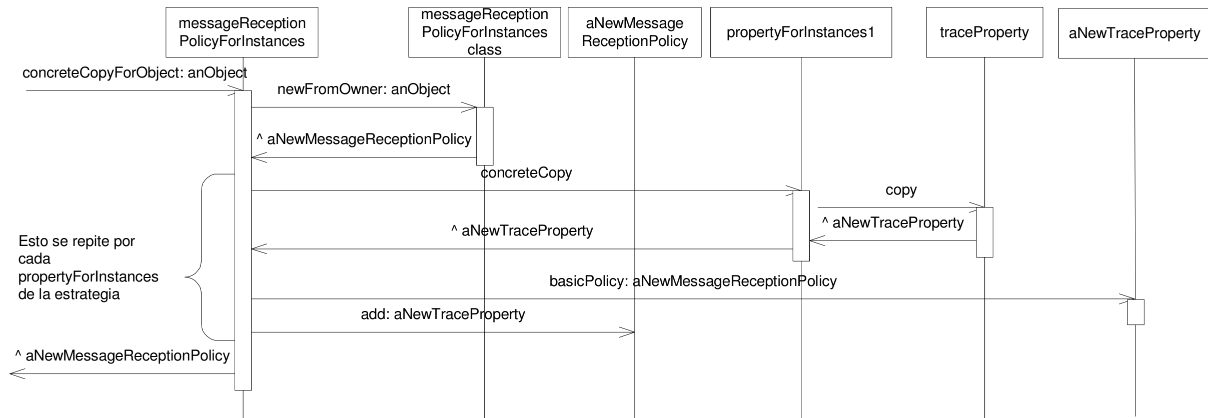


Figura 7.10: Creación de una estrategia de recepción de mensajes para un objeto en particular a partir de la estrategia definida para las instancias de la clase de este objeto

En el diagrama se muestra que primero se crea una nueva estrategia para el objeto recibido como parámetro. Luego, a esta nueva estrategia se le agregan copias de las propiedades concretas asociadas a cada una de las propiedades para instancias definidas en la estrategia original. Como se indica en el diagrama, estas copias se obtienen enviando el mensaje *concreteCopy* a cada una de las propiedades para instancias.

7.1.1.3 Clases que modelan estrategias de propiedades y protocolo para manejar composición y aplicación

A continuación, se muestra el diagrama de las clases que modelan las estrategias de metapropiedades que fueron detalladas. En el mismo se expone el protocolo necesario para que las estrategias resuelvan su aplicación[5].

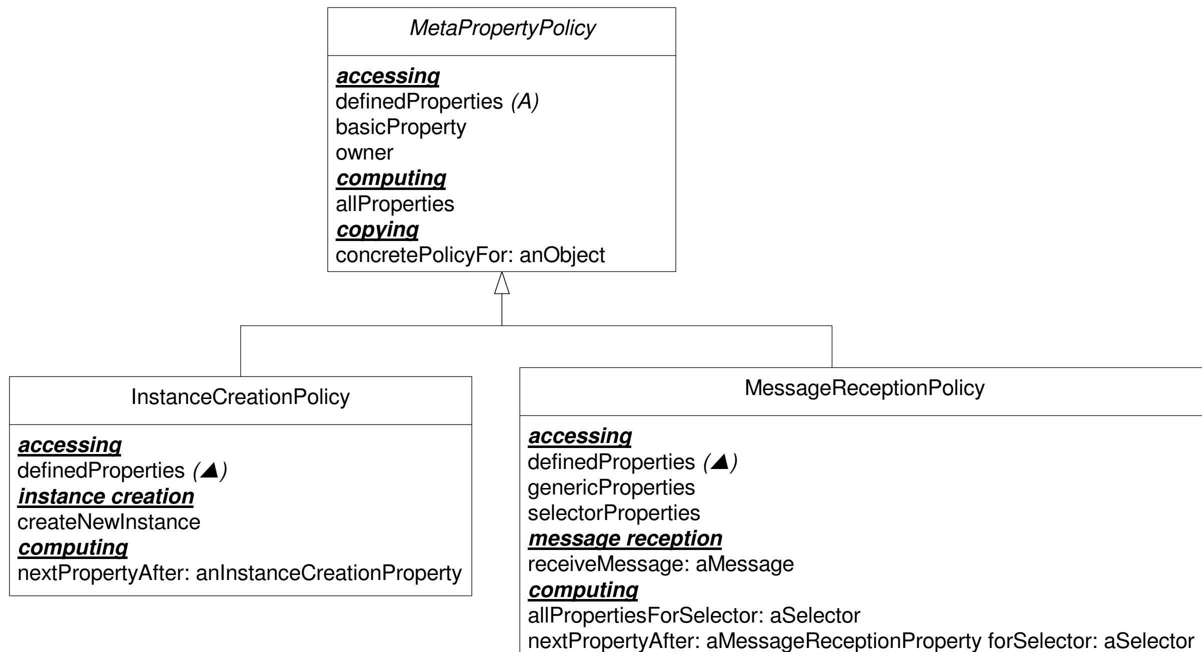


Figura 7.11: Jerarquía de MetaPropertyPolicy con protocolo para manejar su composición y aplicación

Clase MetaPropertyPolicy

Es la clase abstracta raíz de esta jerarquía y modela el concepto de estrategia definida por metapropiedades que ya fue definido. En ella se define a qué debe saber responder toda estrategia de metapropiedades.

accessing

- **definedProperties**

Retorna una colección con todas las propiedades que se agregaron a la estrategia. Esta colección está ordenada de acuerdo al orden de precedencia de aplicación de estas propiedades.

- **basicProperty**

Representa la propiedad básica de la estrategia. Para la estrategia de creación de instancias es la propiedad que crea a la nueva instancia. En el caso de una estrategia de recepción de mensajes es la propiedad que envía el mensaje recibido al objeto. Cuando se crea una estrategia ya nace con esta propiedad.

- **owner**

Es el objeto para el cual está definida la estrategia.

---

[5] Tener en cuenta que los diagramas de clases que se incluyen en estas secciones son incompletos respecto al protocolo ya que no incluyen el asociado a la asignación de propiedades. Éste se presenta en secciones posteriores donde se explica este tema.

computing

- **allProperties**

Retorna una colección con todas las propiedades. Incluye tanto: las propiedades definidas, como la *basicProperty*. Esta colección está ordenada de acuerdo al orden de precedencia de ejecución de estas propiedades.

copying

- **concretePolicyFor: anObject**

Devuelve una copia de la estrategia que efectivamente pueda ser aplicada, es decir compuesta por propiedades concretas.

Clase MessageReceptionPolicy

Es la clase que modela las estrategias de recepción de mensajes. Para que una de estas estrategias se aplique se le debe enviar el mensaje *receiveMessage: aMessage.*

accessing

- **genericProperties**

Representa las propiedades definidas para todos los mensajes. No se definen por un selector en particular. Están ordenadas de acuerdo a la precedencia para su ejecución.

- **selectorProperties**

Aquí quedan definidas la propiedades que se definen para selectores determinados. También están ordenadas teniendo en cuenta la precedencia de su ejecución.

message reception

- **receiveMessage: aMessage**

Maneja la recepción del mensaje recibido, delegando esta responsabilidad en la primera propiedad entre las que se aplican para este mensaje.

computing

- **allPropertiesForSelector: aSelector**

Retorna una colección con todas las propiedades que se aplican para el selector que se recibe. Incluye tanto las propiedades específicas para este selector como la *basicProperty* y las que se aplican en general (de estas últimas sólo aquellas para las cuales no exista una de la misma categoría definida en forma específica). Esta colección está ordenada de acuerdo al orden de precedencia de ejecución de estas propiedades.

- **nextPropertyAfter: aMessageReceptionProperty forSelector: aSelector**

Retorna la propiedad siguiente a la propiedad recibida, buscando entre todas las propiedades que se aplican al selector recibido.

Clase InstanceCreationPolicy

Es la clase que modela las estrategias de creación de instancias. Para que una de estas estrategias se aplique, se le debe enviar el mensaje *createNewInstance*.

instance creation

- **createNewInstance**

Resuelve la creación de una nueva instancia, delegando esta responsabilidad en la primera propiedad que se deba aplicar.

computing

▪ **nextPropertyAfter: anInstanceCreationProperty**

Retorna la propiedad siguiente a la propiedad recibida, buscando entre todas las propiedades.

7.1.2 Metapropiedades

Las estrategias presentadas quedan definidas a partir de las propiedades de las que están compuestas. Cuando una estrategia recibe el mensaje que determina que debe aplicarse, le delega a su primer propiedad esta responsabilidad y lo mismo hace cada propiedad respecto a la propiedad siguiente luego de haberse aplicado.

El conjunto de propiedades específico por el que está compuesto una estrategia define qué se debe realizar al enviarse el mensaje que hace que una estrategia se aplique. Lo que se hace concretamente al crearse una instancia o al recibirse un mensaje depende únicamente de las propiedades definidas en una estrategia.

De acuerdo a las categorías definidas, que dieron lugar a la identificación de las estrategias mencionadas, surgen los conceptos de metapropiedades de creación de instancias y de recepción de mensajes, que son especializaciones de una metapropiedad.

Por otra parte necesitamos contar con propiedades que se puedan definir en forma genérica para las instancias de una clase, como sucede para el caso de las propiedades de recepción de mensajes. Así, aparecen los conceptos de propiedad concreta y propiedad para las instancias de una clase.

7.1.2.1 Clases que modelan metapropiedades y protocolo para manejar su aplicación y ejecución

Se expone a continuación el diagrama de la jerarquía de clases que refleja las abstracciones de los conceptos de metapropiedades que se mencionaron, junto con el protocolo necesario para que se puedan aplicar.

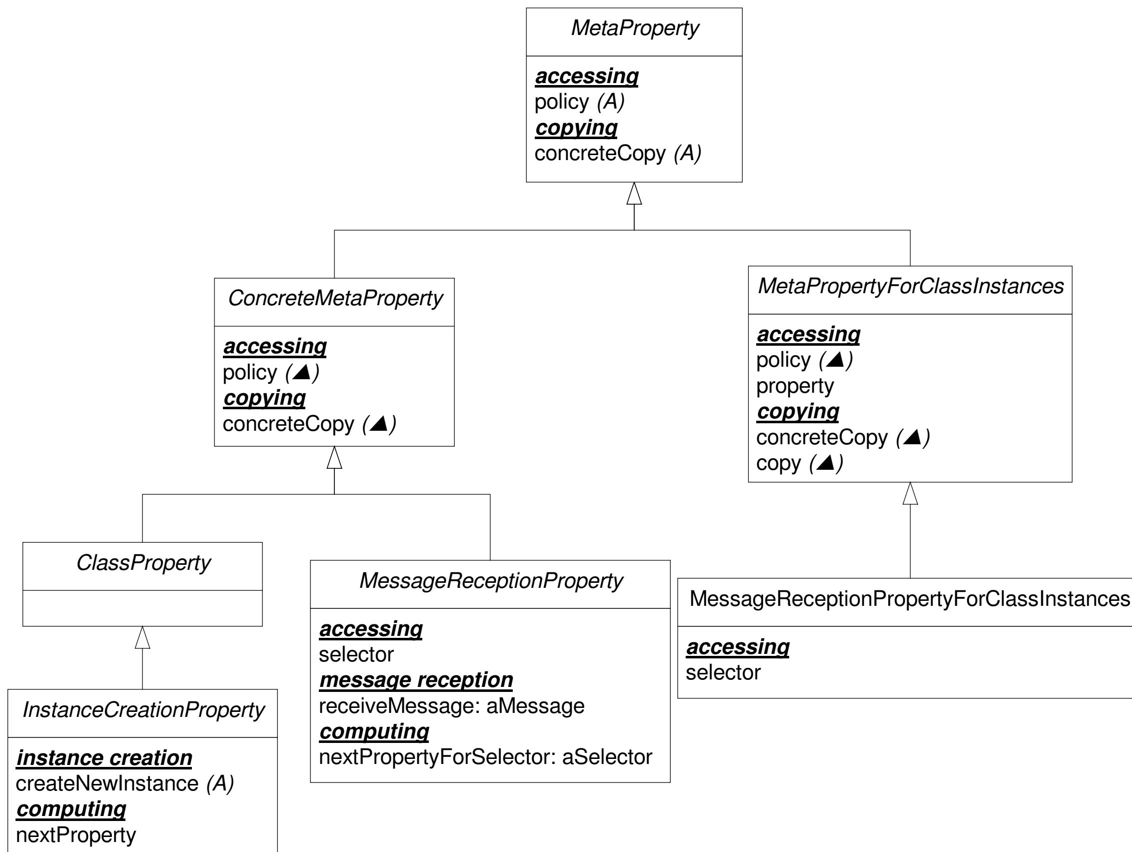


Figura 7.12: Jerarquía de MetaProperty con protocolo para aplicación

Clase MetaProperty

Es la clase abstracta raíz de esta jerarquía y modela el concepto de metapropiedad que ya fue definido.

accessing

- **policy**

Es la estrategia a la que se agrega la propiedad.

accessing

- **concreteCopy**

Retorna una copia de la propiedad concreta. En el caso de una propiedad concreta, es una copia de sí misma. Y en el caso de una propiedad para instancias, una copia de la propiedad concreta asociada.

Clase ConcreteMetaProperty

Es una clase abstracta que modela el concepto de metapropiedad que se aplica concretamente cuando la estrategia u otra propiedad le delega esta responsabilidad. Este

concepto surge en contraposición al de propiedades para instancias, que sólo se usan para definir a una estrategia, pero nunca se aplican.

Clase ClassProperty

Es una clase abstracta que modela el concepto de metapropiedad que se puede definir sólo para objetos que sean clases.

Clase InstanceCreationProperty

Es la clase abstracta que modela el concepto de propiedad de clase que se aplica en la creación de instancias.

Clase MessageReceptionProperty

Es la clase abstracta que modela el concepto de propiedad que se aplica en la recepción de mensajes.

Clase MetaPropertyForClassInstances

Clase abstracta que modela el concepto de metapropiedad para instancias que se contrapone al de propiedad concreta. Representa a las propiedades que se utilizan para determinar las metapropiedades que se deben aplicar a todas las instancias de la clase donde se definen.

accessing

- **property**

Es la propiedad concreta que sirve de prototipo cuando se necesita armar la estrategia que se debe aplicar para una instancia de la clase a la cual está asociada la *propertyForInstances.*

Clase MessageReceptionPropertyForClassInstances

Clase que modela las propiedades para las instancias de una clase, para el caso de la recepción de mensajes.

7.1.2.2 Metapropiedades de creación de instancias y su protocolo para manejar su aplicación y ejecución

Las propiedades de creación de instancias se definen como una especialización de la clase *InstanceCreationProperty* que, como se explicó antes, es la clase abstracta que modela el concepto de propiedad de clase que se aplica en la creación de instancias.

Éstas son las propiedades de las que está compuesta una estrategia de creación de instancias y a las que se delega la responsabilidad de resolver esta funcionalidad.

A continuación se expone el diagrama de clases de las propiedades de creación de instancias implementadas.

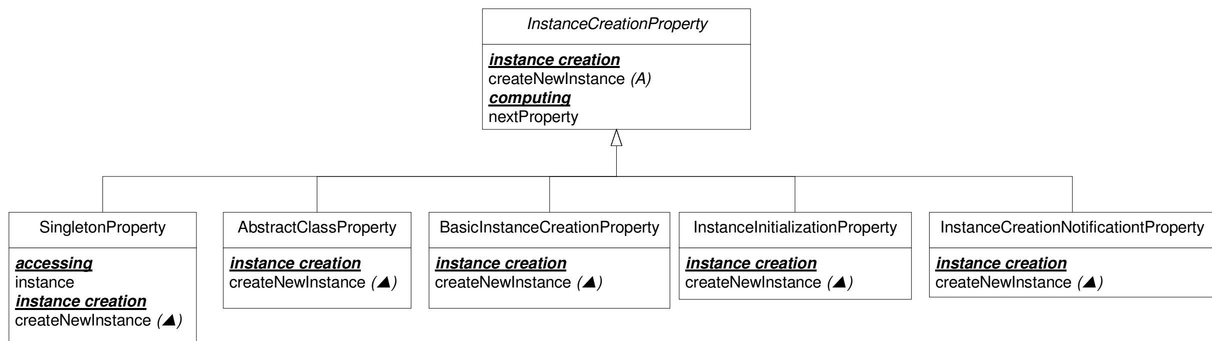


Figura 7.13: Jerarquía de InstanceCreationProperty y su protocolo para aplicación

Clase InstanceCreationProperty

Es la clase abstracta de esta jerarquía y modela el concepto de propiedad de creación de instancias que ya fue definido.

instance creation

▪ **createNewInstance**

Resuelve la creación de la nueva instancia. Luego, si corresponde, se pasa esta responsabilidad a la siguiente propiedad. Retorna la nueva instancia. La implementación de este método debe determinarse en forma particular en cada subclase.

computing

▪ **nextProperty**

Retorna la siguiente propiedad a la cual se le debe pasar la responsabilidad de manejar la creación de instancias. Para resolver esto se colabora con la estrategia.

Clase BasicInstanceCreationProperty

Esta clase modela la propiedad que realiza al modo tradicional provisto por el ambiente, la creación de la nueva instancia de la clase *owner*.

instance creation

▪ **createNewInstance**

Crea y retorna una nueva instancia de la clase *owner* de la estrategia a la cual pertenece la propiedad.

Clase SingletonProperty

Esta clase modela una propiedad que controla que exista sólo una instancia de una determinada clase.

accessing

▪ **instance**

Es la única instancia de la clase *owner*.

instance creation

▪ **createNewInstance**

Detecta si ya existe una instancia de la clase *owner* y si es así la retorna. De lo contrario, pasa la responsabilidad de manejar la creación de instancias a la siguiente propiedad y retorna el resultado del envío de este mensaje.

Clase AbstractClassProperty

Esta clase modela una propiedad que controla que no se creen instancias de una determinada clase.

instance creation

- **createNewInstance**

Genera una excepción que indica que no se pueden crear instancias de la clase *owner*.

Clase InstanceInitializationProperty

Esta clase modela la propiedad que inicializa cada nueva instancia de una clase determinada.

instance creation

- **createNewInstance**

Primero, pasa la responsabilidad de manejar la creación de instancia a la siguiente propiedad. Al objeto resultado de esto le envía el correspondiente mensaje para que se inicialice y luego lo retorna.

Clase InstanceCreationNotificationProperty

Modela la propiedad que notifica a los interesados sobre la creación de una nueva instancia de la clase *owner*.

instance creation

- **createNewInstance**

Primero, pasa la responsabilidad de manejar la creación de instancia a la siguiente propiedad. Luego, dispara el evento de notificación de la creación de la nueva instancia. Finalmente retorna la instancia creada.

7.1.2.3 Metapropiedades de recepción de mensajes y su protocolo para manejar su aplicación y ejecución

Las propiedades de recepción de mensajes se definen como una especialización de la clase *MessageReceptionProperty*, que como se explicó antes, es la clase abstracta que modela el concepto de propiedad que se aplica en la recepción de mensajes.

Estas son las propiedades de las que puede estar compuesta una estrategia de recepción de mensajes y a las que se les delega la responsabilidad de resolver esta funcionalidad.

A continuación se expone el diagrama de clases de las propiedades de recepción de mensajes implementadas.

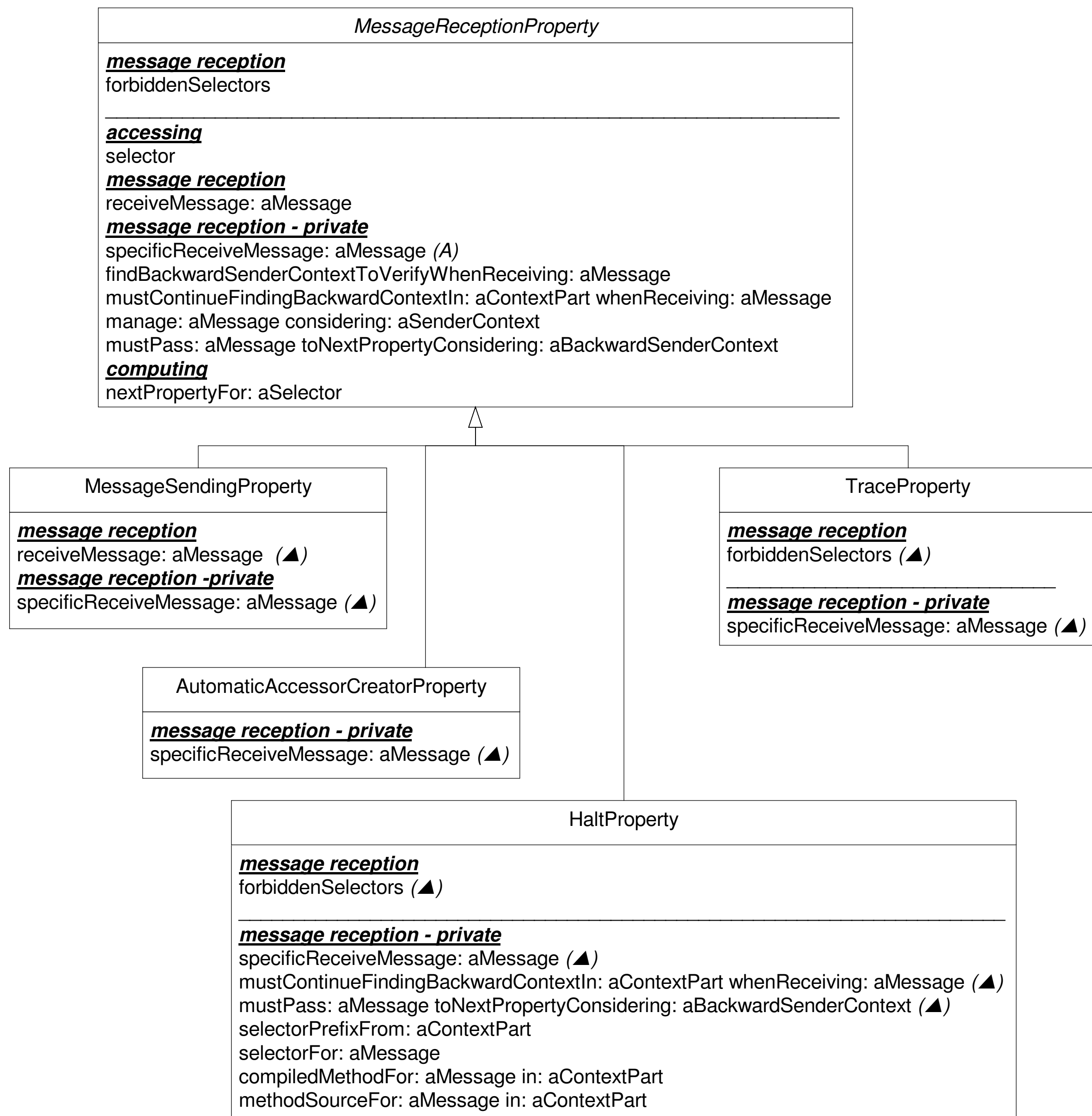


Figura 7.14: Jerarquía de MessageReceptionProperty y su protocolo para aplicación

Clase MessageReceptionProperty

Es la clase abstracta de esta jerarquía y modela el concepto de propiedad de recepción de mensajes que ya fue definido. Una propiedad de recepción de mensajes queda definida por un selector y la estrategia a la que pertenece. Si no tiene selector significa que se aplica a todos los mensajes que reciba el *owner* de la estrategia a la que pertenece.

accessing

▪ **selector**

Representa al selector para el que fue definida la propiedad. Determina al envío de qué mensajes se va a aplicar la propiedad. Si no se define significa que se aplica al envío de cualquier mensaje.

message reception

▪ **receiveMessage: aMessage**

Es el punto de entrada para manejar la recepción de mensaje recibido.

A partir de aquí se determina si la propiedad debe manejar concretamente la recepción de este mensaje o si debe delegar esta responsabilidad directamente a la propiedad siguiente.

En determinados casos aplicar una propiedad implica el riesgo de entrar en una recursión infinita. Así, se define que la propiedad no se debe aplicar si el selector del mensaje recibido se encuentra entre los definidos como prohibidos para la misma. Los selectores prohibidos en general están referidos a mensajes internos de nuestra implementación de la reificación de recepción de mensajes.

Tampoco se debe ejecutar la propiedad si se trata de un mensaje enviado por la propia ejecución de la propiedad.

El manejo de estas restricciones se resuelve con la implementación de los métodos privados de esta categoría (con excepción de *specificReceiveMessage: aMessage*).

message reception - private

▪ **specificReceiveMessage: aMessage**

Se encarga de manejar la recepción de mensaje recibido en forma concreta. Si corresponde, se forwardea la recepción del mensaje a la próxima propiedad. La implementación de este método debe determinarse en forma particular, en cada subclase.

computing

▪ **nextPropertyFor: aSelector**

Retorna la próxima propiedad dentro de las definidas en la estrategia para el selector recibido a la cual se le debe pasar la responsabilidad de manejar la recepción del mensaje. Para resolver esto se colabora con la estrategia.

Clase MessageSendingProperty

Esta clase modela la propiedad que realiza concretamente el envío del mensaje al *owner* de la propiedad. Es la propiedad más básica.

message reception

▪ **receiveMessage: aMessage**

Este método se redefine ya que el envío concreto del mensaje recibido no tiene ninguna de las restricciones mencionadas anteriormente para su ejecución. De esta manera se envía el mensaje para que esta propiedad se ejecute directamente.

▪ **specificReceiveMessage: aMessage**

Ejecuta el envío concreto del mensaje recibido al *owner* de la propiedad.

Clase TraceProperty

Esta clase modela la propiedad que traza el envío de un mensaje al *owner* de la propiedad.

message reception

▪ **specificReceiveMessage: aMessage**

Primero ejecuta la traza del mensaje recibido y luego delega la responsabilidad de manejar la recepción del mensaje a la próxima propiedad.

Clase AutomaticAccessorCreatorProperty

Esta clase modela la propiedad que crea accessors automáticamente cuando se envía al *owner* un mensaje no implementado y cuyo selector coincide con el nombre de una de sus variable de instancia.

message reception

- **specificReceiveMessage: aMessage**

Si el mensaje no está implementado y su selector coincide con el nombre de una de las variables de instancias del owner se crea el accessor correspondiente. Luego se delega la recepción del mensaje a la próxima propiedad.

Clase HaltProperty[6]

Esta clase modela la propiedad que agrega un halt (breakpoint) al método asociado al mensaje recibido.

message reception

- **specificReceiveMessage: aMessage**

Agrega el envío del mensaje *halt* en el código del método que implementa el mensaje recibido y crea un nuevo método en la clase del *owner* de la propiedad a partir de este código modificado. Se le envía este nuevo mensaje al *owner* y posteriormente se asegura la eliminación de este método luego de forzar su ejecución.

Todo esto se resuelve con la implementación de los siguientes métodos de instancia definidos en esta misma clase:

- **selectorPrefixFrom: aContextPart**
- **selectorFor: aMessage**
- **compiledMethodFor: aMessage in: aContextPart**
- **methodSourceFor: aMessage in: aContextPart**

- **mustContinueFindingBackwardContextIn: aContextPart whenReceiving: aMessage**
- **mustPass: aMessage toNextPropertyConsidering: aBackwardSenderContext**

Estos métodos se redefinen ya que por cuestiones de implementación de esta propiedad se deben extender las condiciones para determinar las restricciones de aplicabilidad de la propiedad.

[6] Por limitaciones de implementación esta propiedad no se puede definir para mensajes binarios, es decir aquellos cuyos selectores son operadores (por ejemplo: =, <, >, +, -, &, etc). Se propone esto como una mejora futura sobre la implementación actual.

7.2 Asignación de metapropiedades

La asignación de metapropiedades se logra agregando propiedades a la estrategia correspondiente. Cada vez que se agrega o elimina una propiedad de una estrategia se construye una nueva estrategia. Esto se hace para no interferir con el proceso de modificación en una estrategia que potencialmente puede estar en uso.

El proceso de construcción de nuevas estrategias se reificó. En este proceso, además de llevarse a cabo la construcción en sí, se realizan validaciones para garantizar la consistencia de la estrategia resultante.

También está modelado el manejo de incompatibilidades entre propiedades y la determinación de las relaciones de orden entre las propiedades asignadas, todo esto mediante categorías de metapropiedades.

7.2.1 Construcción de estrategias de metapropiedades

7.2.1.1 Agregado de una metapropiedad

Como se explicó antes, cada vez que se agrega una propiedad a una estrategia, se construye una nueva. Un objeto builder tiene la responsabilidad de esta construcción.

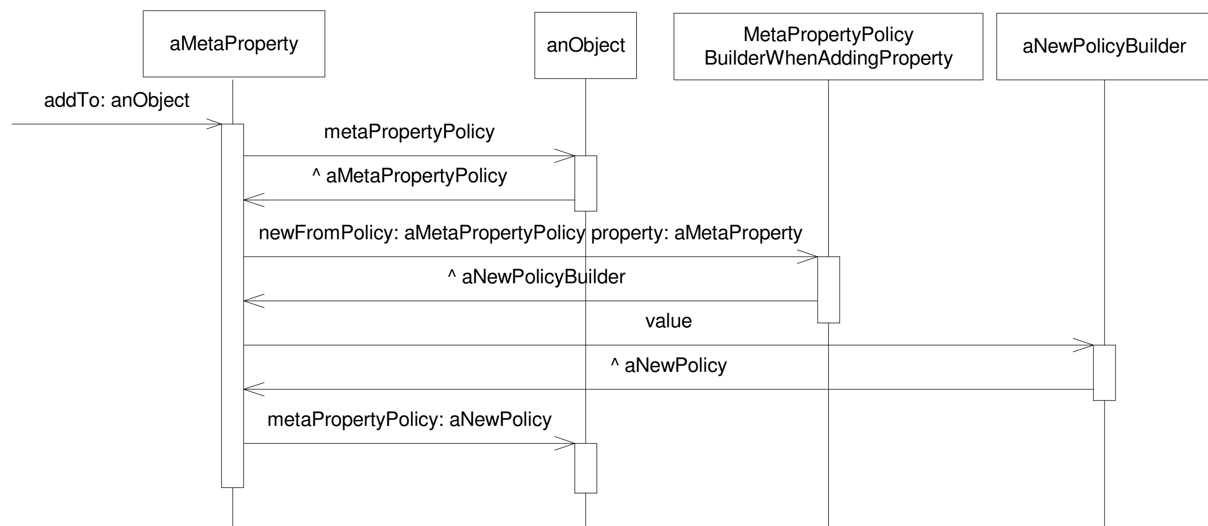


Figura 7.15: Agregado de una metapropiedad a la estrategia correspondiente de anObject

En este diagrama se puede observar que la propiedad tiene la responsabilidad de agregarse a la correspondiente estrategia de metapropiedades de *anObject*. La propiedad a su vez delega en el *builder* (creado a partir de ella misma y de la estrategia) esta responsabilidad. El *builder* finalmente devuelve una nueva estrategia que incluye a la propiedad que se quiso agregar y ésta pasa a ser la nueva estrategia de *anObject*.

Ahora veamos cómo un *builder* de estrategias de metapropiedades, a partir de una estrategia de metapropiedades y una propiedad, construye una nueva estrategia que incluye a esta propiedad.

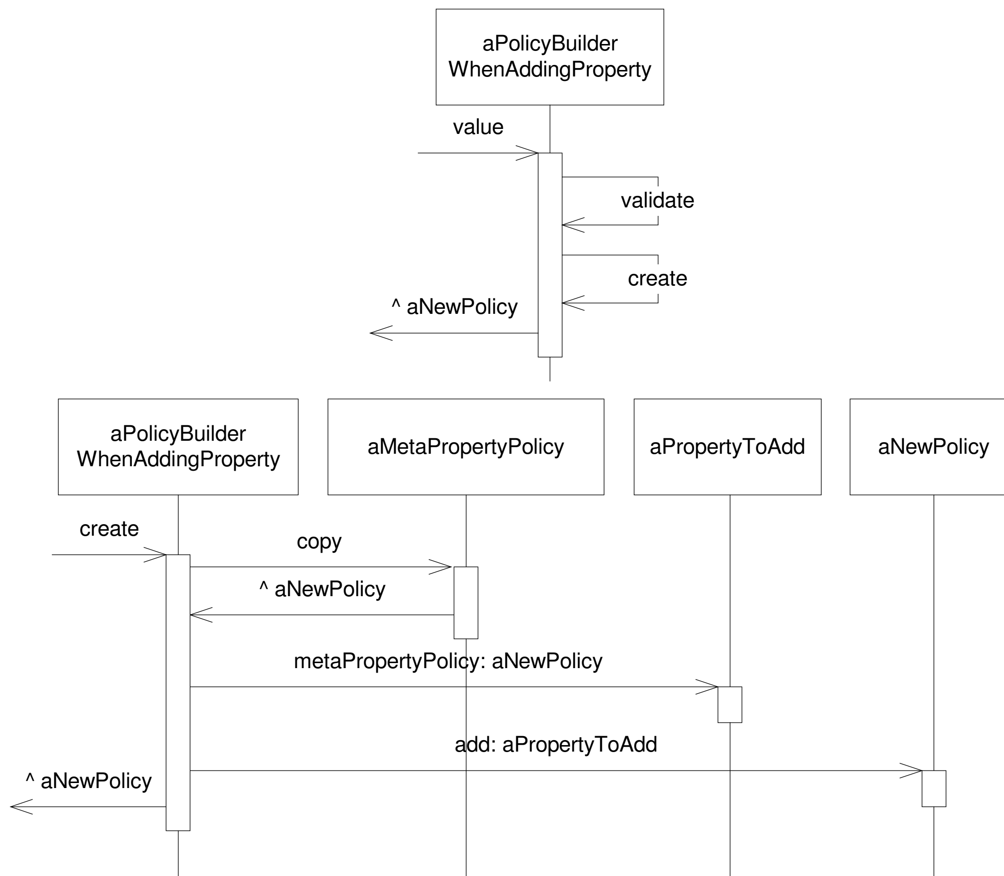


Figura 7.16: Un builder que construye una estrategia de a partir de otro que incluye a una nueva propiedad

En el primero de estos dos diagrama se puede observar que antes de construir la nueva estrategia se valida que el agregado de la propiedad se pueda llevar a cabo.

En el segundo diagrama se detalla cómo se construye la nueva estrategia a partir de la original. Primero se realiza una copia de ésta. Luego, se agrega la propiedad a esta nueva estrategia. La responsabilidad de este agregado se delega en la nueva estrategia para que lo realice en forma ordenada teniendo en cuenta la precedencia de ejecución de las propiedades.

Validaciones que se realizan al agregar una metapropiedad

Antes de construir la nueva estrategia, el *builder* determina si es válido agregar la propiedad a la estrategia. En caso que no sea así se genera una excepción y no se realiza la construcción. Las validaciones que se realizan son:

- Que no se esté agregando la propiedad básica. Como se mencionó anteriormente toda estrategia de metapropiedades ya nace con la propiedad básica correspondiente.
- Que no exista una propiedad similar o equivalente.
- Que la propiedad no sea incompatible con alguna de la propiedades relacionadas ya existentes en la estrategia.
  Por ejemplo, si en la estrategia de creación de instancias de una clase dada ya existe la propiedad *singleton*, no se podría agregar a esta estrategia la propiedad **abstracta**, debido a que ambas propiedades son incompatibles entre sí. Por un lado, la propiedad singleton determina que sólo puede existir una única instancia de la clase. Por otro lado, la propiedad abstracta determina que no se pueden crear instancias de esta clase.

7.2.1.2 Eliminación de una metapropiedad

De la misma manera que al agregar una propiedad, cada vez que se elimina una, se construye una nueva estrategia y, nuevamente, esta responsabilidad la tiene un *builder*.

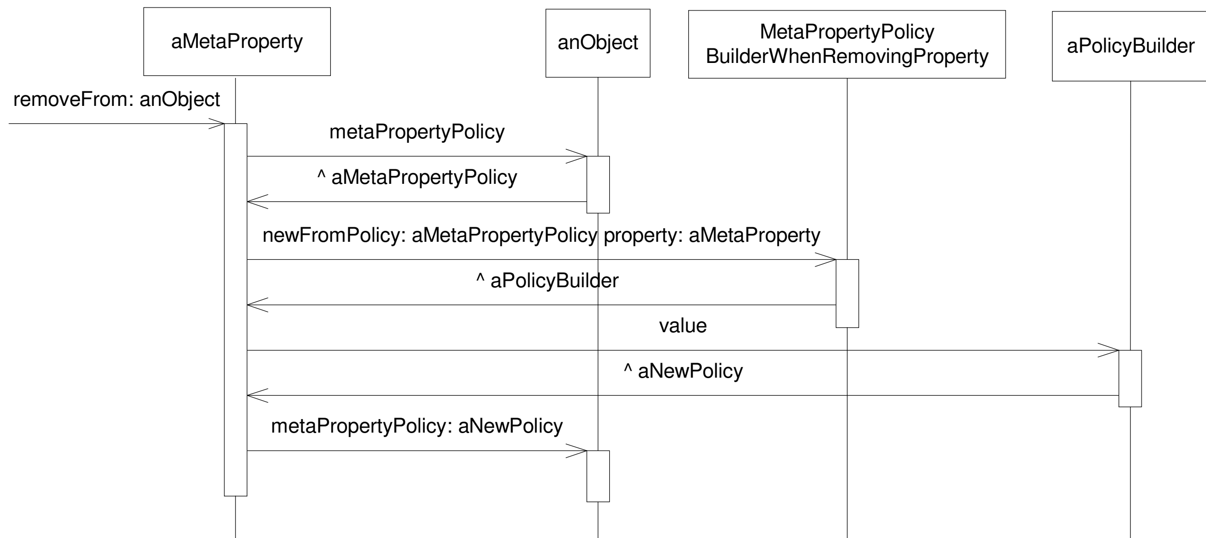


Figura 7.17: Eliminación de una metapropiedad de la estrategia de anObject

En este diagrama se puede observar que la propiedad tiene la responsabilidad de eliminarse de la estrategia de recepción de *anObject*. La propiedad a su vez delega en el *builder* (creado a partir de ella misma y de la estrategia) esta responsabilidad. El *builder* finalmente devuelve una nueva estrategia que no incluye a la propiedad que se quiso eliminar y ésta pasa a ser la nueva estrategia de *anObject.*

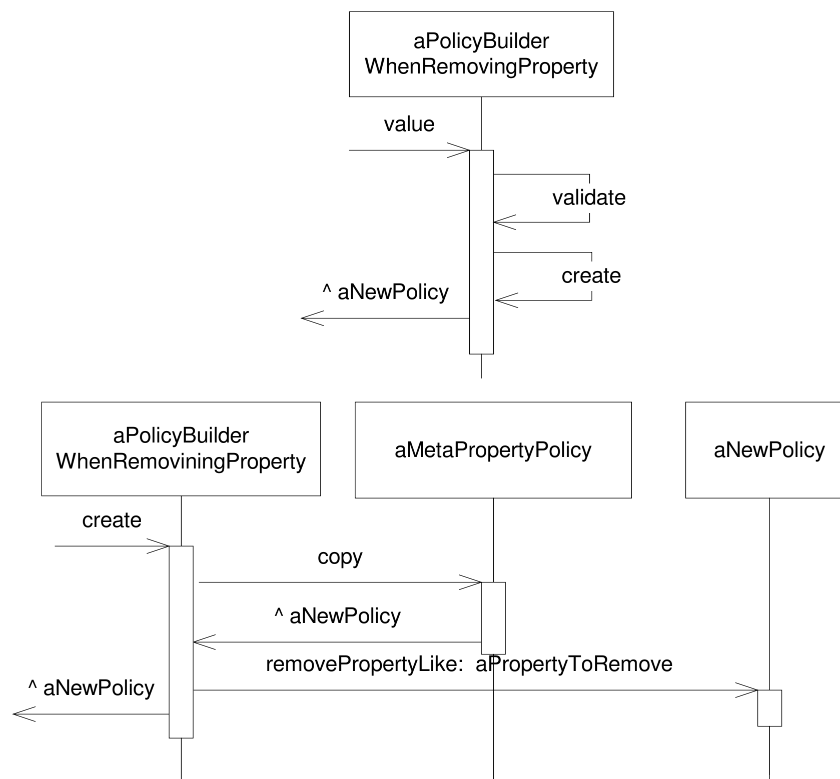


Figura 7.18: Un builder que construye una nueva estrategia a partir de otra eliminando una de sus propiedades

En el primero de estos dos diagramas se puede observar que antes de construir la nueva estrategia se valida que la eliminación de la propiedad se pueda llevar a cabo.

En el segundo diagrama se detalla cómo se construye la nueva estrategia a partir de la original. Primero se realiza una copia de la estrategia original. De esta copia se elimina la propiedad equivalente a la que se quiere eliminar. Esta responsabilidad se delega en la nueva estrategia.

Validaciones que se realizan al eliminar una metapropiedad

Antes de construir la nueva estrategia, el *builder* determina si es válido eliminar la propiedad de la estrategia. En caso de que no sea así, se genera una excepción y no se realiza la eliminación. Lo único que se valida es que exista una propiedad equivalente a la propiedad que se quiere eliminar. La eliminación de una propiedad no puede dejar una estrategia inconsistente.

7.2.1.3 Definición y protocolo de *builders* de estrategias de metapropiedades

Un *builder* que crea una nueva estrategia de metapropiedades a partir de otra cuando se quiere agregar o eliminar una propiedad, se define a partir esta propiedad y de la estrategia original.

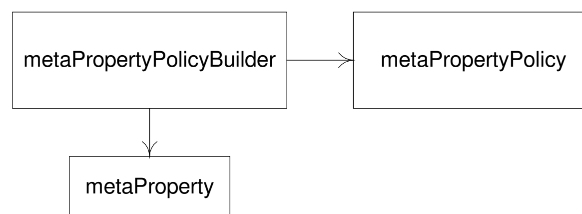


Figura 7.19: Diagrama de instancias de un builder de metapropiedades

A continuación se expone el diagrama de clases de la jerarquía que modela estos *builders*.

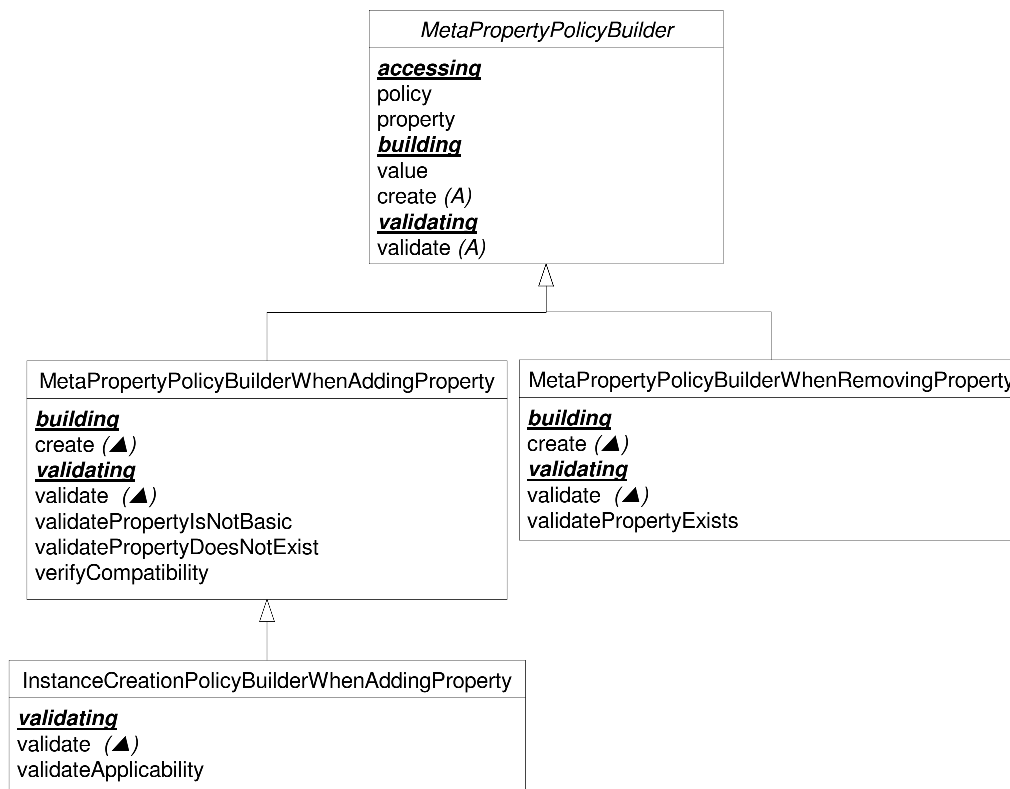


Figura 7.20:Jerarquía de builders de estrategias de metapropiedades

Clase MetaPropertyPolicyBuilder

Es la clase abstracta de esta jerarquía y modela los *builders* de estrategias de metapropiedades que ya fueron definidos.

accessing

▪ **policy**

Es la estrategia de metapropiedades a partir de la cual se construye la nueva.

▪ **property**

Es la propiedad a agregar o eliminar de la nueva estrategia.

building

▪ **value**

Se valida que efectivamente se pueda construir la nueva estrategia. En caso de que se cumplan todas las condiciones se construye la nueva estrategia.

▪ **create**

Construye concretamente la nueva estrategia. La implementación de este método se determina en cada subclase.

validating

▪ **validate**

Se validan todas las condiciones que deben darse para poder construir la nueva estrategia. La determinación de estas condiciones se realiza en las subclases.

Clase MetaPropertyPolicyBuilderWhenAddingProperty

Es la clase que modela la construcción de una nueva estrategia de metapropiedades cuando se quiere agregar una propiedad.

building

- **create**

Construye una nueva estrategia de recepción de metapropiedades creando una copia de la estrategia definida y agregándole la propiedad determinada.

validating

- **validatePropertyIsNotBasicProperty**

Valida que la propiedad definida no sea una propiedad básica.

- **validatePropertyDoesNotExist**

Valida que la propiedad no sea equivalente a una ya existente en la estrategia.

- **verifyCompatibility**

Valida que la propiedad no sea incompatible con alguna de las propiedades ya existentes en la estrategia.

Clase MetaPropertyPolicyBuilderWhenRemovingProperty

Es la clase que modela la construcción de una nueva estrategia de metapropiedades cuando se quiere eliminar una propiedad.

building

- **create**

Construye una nueva estrategia creando una copia de la estrategia definida y eliminando la propiedad equivalente a la propiedad indicada.

validating

- **validatePropertyExists**

Valida que exista en la estrategia una propiedad equivalente a la propiedad definida.

Clase InstanceCreationPolicyBuilderWhenAddingProperty

Es la clase que modela la construcción de una nueva estrategia de creación de instancias cuando se quiere agregar una propiedad.

validating

- **validate**

Se validan todas las condiciones que deben darse para poder agregar la propiedad a la estrategia. Además de las validaciones determinadas por la superclase, se realiza la validación que se detalla a continuación.

- **validateApplicability**

Valida que la propiedad pueda aplicarse a la creación de instancias de la clase *owner* de la estrategia a la cual se quiere agregar la propiedad.

La responsabilidad de la determinación de esta aplicabilidad se delega en la propiedad que se quiere agregar, que a su vez colabora con la clase *owner* para resolver esto.

7.2.1.4 Comportamiento de estrategias de metapropiedades relacionado con la asignación y construcción

A continuación se detalla el protocolo y comportamiento de las estrategias de metapropiedades necesario para agregar y eliminar propiedades.

Figura 7.21: Jerarquía de estrategias de metapropiedades y su protocolo para agregar y eliminar propiedades

Clase MetaPropertyPolicy

adding

- **add: aMetaProperty**

Agrega la propiedad recibida a la colección de propiedades relacionadas con la misma.

La propiedad se agrega a la colección que corresponda en forma ordenada, teniendo en cuenta el criterio de precedencia de aplicación de las propiedades.

removing

- **removePropertyLike: aMetaProperty**

Elimina de la estrategia la propiedad que sea equivalente a la propiedad recibida. Se delega a las propiedades existentes la responsabilidad de determinar esta equivalencia.

- **propertiesRelatedTo: aMetaProperty**

Devuelve todas las propiedades agregadas a la estrategia que estén relacionadas con la que se recibe. En este contexto se considera que dos propiedades están relacionadas si se determina que existen casos de uso de la estrategia para los que deban aplicarse ambas propiedades. La implementación de este método se realiza en cada una de las subclases.

Clase InstanceCreationPolicy
computing

- **propertiesRelatedTo: anInstanceCreationProperty**

Devuelve todas las propiedades agregadas a la estrategia, ya que en este caso todas las propiedades se aplican para todos los casos de uso.

Clase MessageReceptionPolicy
computing

- **propertiesRelatedTo: aMessageReceptionProperty**

En este caso las propiedades relacionadas con la propiedad recibida no son todas las propiedades agregadas, ya que la aplicabilidad para los distintos casos de uso depende de la definición del atributo *selector* de las propiedades.

Si la propiedad recibida no tiene definido un selector retorna todas las propiedades agregadas a la estrategia, tanto las genéricas como las específicamente asociadas a cualquier selector.

De lo contrario devuelve las propiedades agregadas que tengan definido el mismo selector que la propiedad que se recibe, más las definidas en forma genérica para todos los selectores.

7.2.1.5 Comportamiento de metapropiedades relacionado con la asignación

Ahora se pasa a detallar el protocolo y comportamiento de la jerarquía de metapropiedades relacionado con la asignación de las mismas a estrategias.

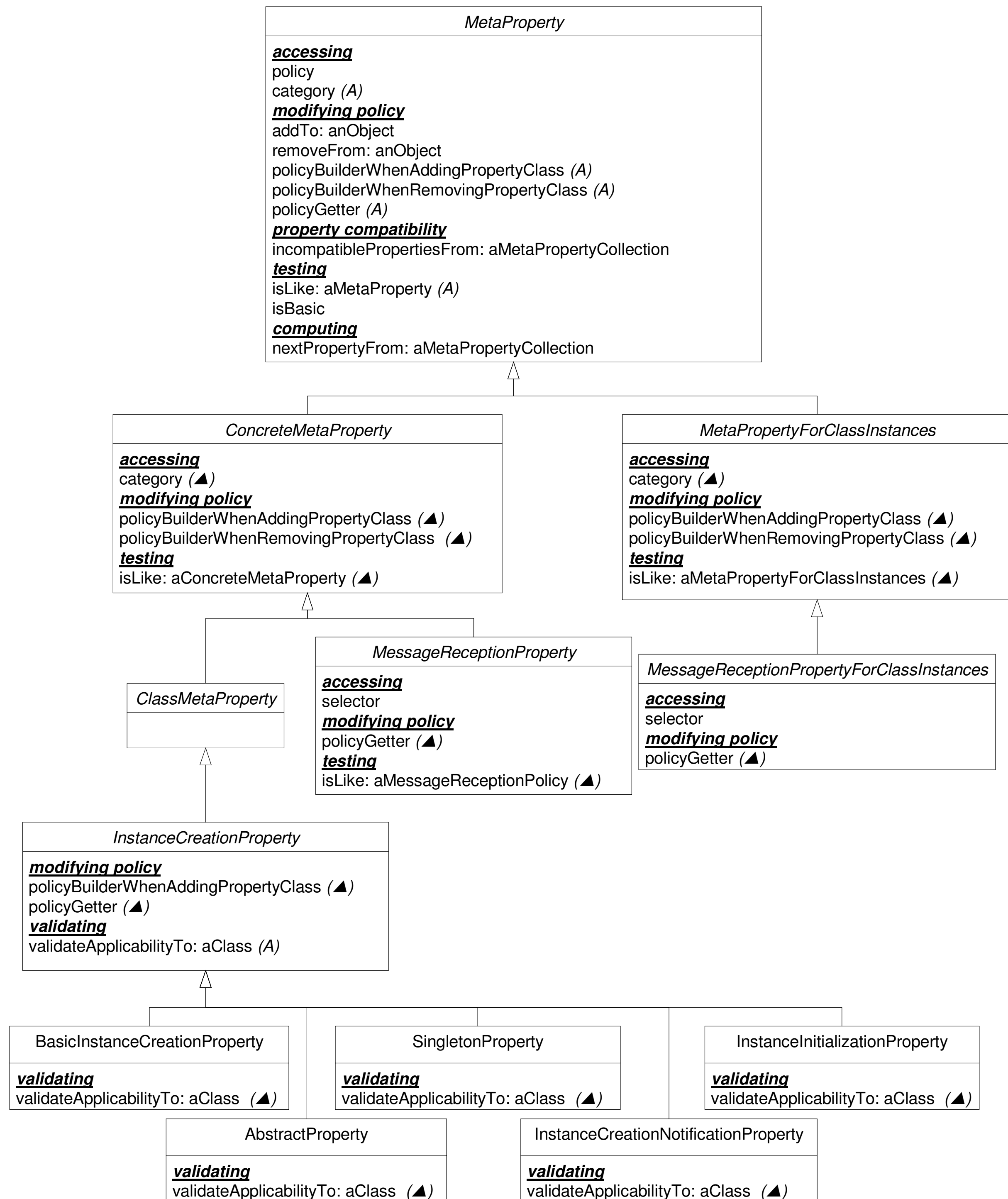


Figura 7.22: Jerarquía de metapropiedades y su protocolo para asignación

Clase MetaProperty

accessing

- **category**

Representa la categoría de la propiedad. En la misma se delega la responsabilidad de determinar las incompatibilidades y el orden de precedencia de aplicación de la propiedad respecto a las otras. Más adelante este concepto se expone más detalladamente.

modifying Policy

- **addTo: anObject**

Agrega la propiedad a la estrategia de metapropiedades asociada del objeto recibido.

Esta responsabilidad se delega en un *builder* que construye una nueva estrategia a partir de la estrategia actual del objeto y la propiedad.

Finalmente asigna esta nueva estrategia de metapropiedades al objeto recibido.

- **removeFrom: anObject**

Elimina la propiedad de la estrategia de metapropiedades asociada del objeto recibido.

Esta responsabilidad se delega en un builder que construye una nueva estrategia a partir de la estrategia actual del objeto y la propiedad.

Finalmente asigna esta nueva estrategia de metapropiedades al objeto recibido.

- **policyBuilderWhenAddingPropertyClass**

Devuelve la clase para obtener un builder que construye una nueva estrategia cuando se quiere agregar la propiedad a la misma. Esto puede variar de una propiedad a otra.

- **policyBuilderWhenRemovingPropertyClass**

Devuelve la clase para obtener un builder que construye una nueva estrategia cuando se quiere eliminar la propiedad de la misma. Esto puede variar de una propiedad a otra.

- **policyGetter**

Retorna el selector del mensaje que debe enviarse a un objeto para acceder a la estrategia asociada a la propiedad. Para el caso de una propiedad concreta se trata del mensaje a partir del cual se obtiene la estrategia definida para el objeto en particular. Para una propiedad para instancias (en este caso el objeto que recibirá el mensaje es una clase) es el mensaje que permite obtener la estrategia definida para las instancias de una clase.

property compatibility

- **incompatiblePropertiesFrom: aMetaPropertyCollection**

Retorna aquellas propiedades de la colección recibida que sean incompatibles con la propiedad. La responsabilidad de determinar estas incompatibilidades se delega en las categorías de las propiedades.

testing

- **isLike: aMetaProperty**

Determina si la propiedad recibida es equivalente a la recibida. Esto se determina tanto a partir de las categorías de las propiedades como de atributos propios de las propiedades

- **isBasic**

Determina si se trata de una propiedad básica. Esta responsabilidad se delega en la categoría.

computing

- **nextPropertyFrom: aMetaPropertyCollection**

Retorna la propiedad de la colección recibida que representa a la siguiente respecto a la propiedad que recibe este mensaje, teniendo en cuenta el orden de precedencia para su aplicación. La responsabilidad de determinar este orden se delega en las categorías de las propiedades.

Clase InstanceCreationProperty

modifying policy

- **policyBuilderWhenAddingPropertyClass**

Se redefine aquí este método debido a que, como se explicó antes, es otro el *builder* que arma una nueva estrategia cuando se agrega una de estas propiedades, como consecuencia de la validación extra que se realiza. Es justamente por la realización de esta validación que surge la necesidad de definir al mensaje *validateApplicabilityTo*: en esta clase.

validating

- **validateApplicabilityTo: aClass**

Valida que la propiedad pueda aplicarse teniendo en cuenta el contexto de la clase recibida. La implementación de esta validación se determina en cada una de las subclases porque depende de la propiedad de creación de instancias que reciba el mensaje.

Clase SingletonProperty

validating

- **validateApplicabilityTo: aClass**

Valida que la cantidad de instancias de la clase recibida sea a lo sumo una.

Si existe una única instancia, entonces se advierte que la instancia quedará asociada a la propiedad.

Clase AbstractProperty

validating

- **validateApplicabilityTo: aClass**

Valida que no haya instancias de la clase recibida.

Observación: El resto de las propiedades de creación de instancias no valida nada respecto a su aplicabilidad a una determinada clase.

Clase MetaPropertyForClassInstances

Los métodos definidos en esta clase, están implementados de forma tal que las responsabilidades asociadas se delegan a la propiedad concreta de la propiedad para instancias que recibe el mensaje.

7.2.2 Categorías de metapropiedades

Uno de los atributos de una metapropiedad es su categoría. Una propiedad delega en su categoría varias responsabilidades. La categoría determina el orden de precedencia y la incompatibilidad respecto a otras de propiedades. También define si las propiedades de esta categoría son básicas.

7.2.2.1 Cómo se relacionan las propiedades y categorías

Por cada tipo de propiedad se puede definir una categoría distinta y todas las instancias de un mismo tipo de propiedad comparten la misma categoría.

Las categorías definidas para las **propiedades de creación de instancias** son:

- *basicInstanceCreationCategory*
- *singletonCategory*
- *abstractClassCategory*
- *instanceCreationNotificationCategory*
- *instanceInitializationCategory*

Las categorías definidas para las **propiedades de recepción de mensajes** son:

- *automaticAccessorCreatorCategory*
- *haltCategory*
- *traceCategory*
- *messageSendingCategory*

Estas categorías son instancias de la clase *MetaPropertyCategory*, que modela a las categorías y que se detalla a continuación. Cada una de estas categorías es un objeto conocido al que se puede acceder enviando un mensaje a su clase que lo tiene registrado.

7.2.2.2 Clase que modela las categorías de metapropiedades

MetaPropertyCategory es la clase que modela las categorías de metapropiedades. Una categoría queda definida por su nombre y la determinación de si se trata o no de una categoría básica.

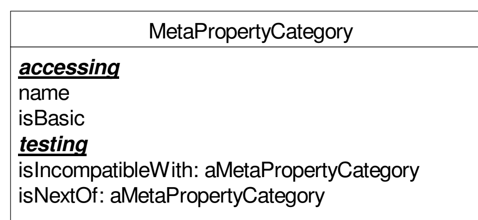


Figura 7.23: Clase MetaPropertyCategory

accessing

▪ **name**

Es el nombre de la categoría. En general guarda relación con la(s) clase(s) de propiedad(es) con la(s) que esté asociada la categoría.

- **<u>isBasic</u>**
  Determina si la propiedad con la que está relacionada la categoría es básica.

<u>testing</u>

- **<u>isIncompatibleWith: aMetaPropertyCategory</u>**
  Determina si la categoría es incompatible con la categoría recibida. Para determinar esto la categoría colabora con un objeto en donde se encuentran definidas las relaciones de incompatibilidad entre las categorías.

- **<u>isNextOf: aMetaPropertyCategory</u>**
  Determina si la categoría es la siguiente de la categoría recibida. Esto sirve para determinar el orden o la precedencia de aplicación de las propiedades a las que están asociadas. Para determinar esto, la categoría colabora con un objeto en donde se encuentran definidas las relaciones de orden entre las categorías.

7.2.2.3 Relaciones de orden entre categorías de metapropiedades

Las relaciones de orden entre las categorías quedan definidas en instancias de la clase *MetaPropertyCategoryOrder*.

Existen dos instancias conocidas de esta clase: una en la que quedan definidas la *relaciones de orden entre las categorías de creación de instancias* y otra para las *relaciones de orden entre la categorías de recepción de mensajes*. A estos dos objetos se puede acceder enviando mensajes a la clase *MetaPropertyCategoryOrder* que los tiene registrados.

MetaPropertyCategoryOrder
<u>accessing</u>
orderDefinitions
previousCategoriesOf: aCategory
<u>adding</u>
define: aCategory before: otherCategory
<u>testing</u>
is: aCategory relatedWith: otherCategory
is: aCategory before: otherCategory
<u>validating</u>
validateIf: aCategory canBeDeclaredBefore: otherCategory
validateNoneIsBasic: aCategory or: otherCategory
validateIf: aCategory wasDeclaredBefore: otherCategory
validateIf: aCategory isIncompatibleWith: otherCategory

Figura 7.24: Clase MetaPropertyCategoryOrder

<u>accessing</u>

- **<u>orderDefinitions</u>**
  Es la colección donde quedan definidas las relaciones de orden entre las categorías.

- **<u>previousCategoriesOf: aCategory</u>**
  Retorna las categorías definidas como anteriores a la categoría recibida.

<u>adding</u>

- **<u>define: aCategory before: otherCategory</u>**
  Valida que se pueda definir una relación de orden entre las categorías recibidas, y si corresponde, agrega esta relación a las definiciones de orden ya existentes.

<u>testing</u>

- **<u>is: aCategory relatedWith: otherCategory</u>**
  Determina si existe alguna relación de orden entre las categorías recibidas.

▪ **is: aCategory before: otherCategory**

Determina si *aMetaPropertyCategory* es una de las categorías definidas como anteriores a *anotherMetaPropertyCategory*.

validating

▪ **validateIf: aCategory canBeDeclaredBefore: otherCategory**

Valida que se pueda definir a la primer categoría como anterior a la segunda categoría recibida.

Para ello se verifica que se cumpla lo siguiente:

- Que ninguna de las categorías sea básica.
- Que no exista una relación de orden inversa a la que se quiere definir.
- Que las categorías en cuestión no hayan sido declaradas como incompatibles. Esto se determina colaborando con el objeto en donde se encuentran definidas las relaciones de incompatibilidad entre las categorías.

7.2.2.4 Incompatibilidades entre categorías de metapropiedades

Las relaciones de incompatibilidad entre las categorías quedan definidas en instancias de la clase *MetaPropertyCategoryIncompatibility*.

Existen dos instancias conocidas de esta clase: una en la que quedan definidas las *relaciones de incompatibilidad entre las categorías de creación de instancias* y otra para las *relaciones de incompatibilidad entre las categorías de recepción de mensajes*. A estos dos objetos se puede acceder enviando mensajes a la clase *MetaPropertyCategoryIncompatibility* que los tiene registrados.

MetaPropertyCategoryIncompatibility
accessing incompatibilityDefinitions incompatibleCategoriesWith: aCategory ***adding*** define: aCategory incompatibleWith: otherCategory ***testing*** is: aCategory incompatbleWith: otherCategory ***validating*** validateIf: aCategory canBeDeclaredIncompatibleWith: otherCategory validateNoneIsBasic: aCategory or: otherCategory validateIfExistOrderRelationBetween: aCategory and: otherCategory

Figura 7.25: Clase MetaPropertyCategoryIncompatibility

accessing

▪ **incompatibilityDefinitions**

Es la colección donde quedan definidas las relaciones de incompatibilidad entre las categorías.

▪ **incompatibleCategoriesWith: aCategory**

Retorna las categorías definidas como incompatibles con la categoría recibida.

adding

▪ **define: aCategory incompatibleWith: anotherCategory**

Valida que las categorías recibidas puedan ser definidas como incompatibles, y si corresponde, agrega esta relación a las definiciones de incompatibilidades ya existentes.

testing

▪ **is: aCategory incompatibleWith: otherCategory**

Determina si la categoría es incompatible con la recibida.

validating

▪ **validateIfCategory: aCategory canBeDeclaredIncompatibleWithCategory: otherCategory**

Valida que las categorías recibidas puedan ser definidas como incompatibles.

Para ello se verifica que se cumpla lo siguiente:

- Que ninguna de las categorías sea básica.
- Que no exista una relación de orden entre las categorías recibidas. Esto se determina colaborando con el objeto en donde se encuentran definidas las relaciones de orden entre categorías.

7.2.2.5 Cómo se definen relaciones de orden e incompatibilidades entre categorías de metapropiedades

La definición de las relaciones de orden e incompatibilidad entre las categorías de metapropiedades existentes es responsabilidad de objetos cuyas clases pertenecen a la jerarquía encabezada por *MetaPropertyCategoryRelationshipDefinition*. Ésta es una clase abstracta que establece que la responsabilidad de toda subclase concreta es la de definir las relaciones antedichas entre las categorías del grupo que esta clase esté representando. Estas definiciones se deben establecer antes de comenzar a trabajar con la asignación de metapropiedades, pero pueden ser modificadas posteriormente.

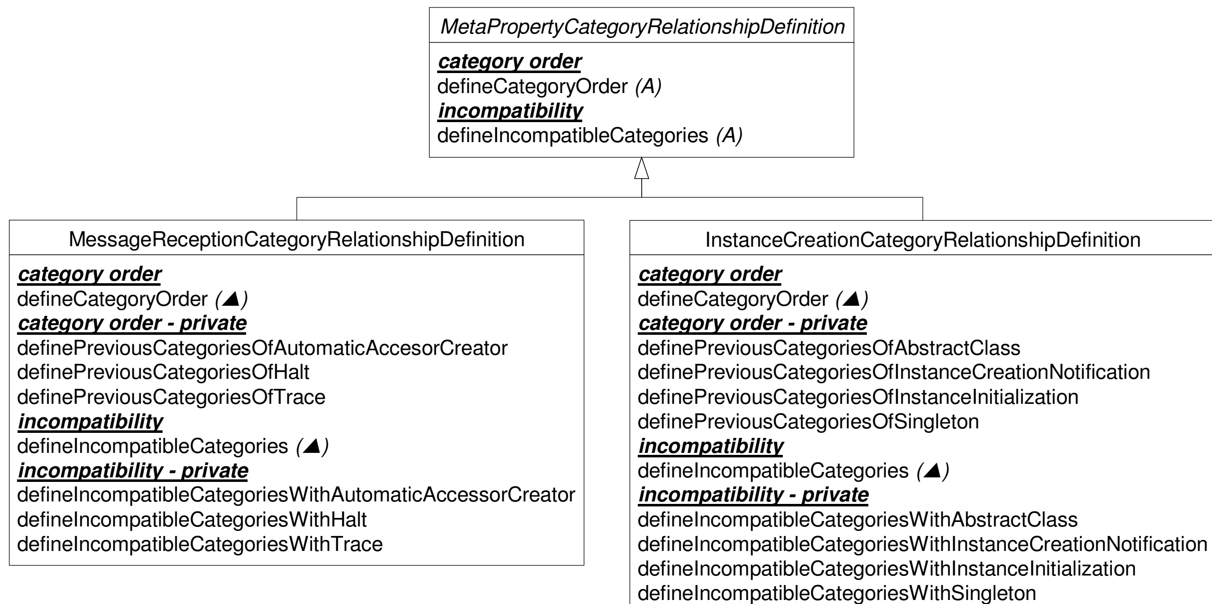


Figura 7.26: Jerarquía de MetaPropertyCategoryRelationshipDefinition

Clase MetaPropertyCategoryRelationshipDefinition

category order

- **defineCategoryOrder**

Define las relaciones de orden entre las categorías de metapropiedades del grupo que la clase representa.

Para lograr esto se colabora con el objeto que es responsable del registro de estas relaciones de orden (es decir la instancia de *MetaPropertyCategoryOrder* que corresponda).

La implementación de este método se debe realizar en cada subclase.

incompatibility

- **defineIncompatibleCategories**

Define las relaciones de incompatibilidad entre las categorías de metapropiedades del grupo que la clase representa.

Para lograr esto se colabora con el objeto que es responsable del registro de estas relaciones de incompatibilidad (es decir, la instancia de *MetaPropertyCategoryIncompatibility* que corresponda).

La implementación de este método se debe realizar en cada subclase.

Clase InstanceCreationCategoryRelationshipDefinition

Esta clase tiene la responsabilidad de definir las relaciones de orden y de incompatibilidad entre las categorías de creación de instancias existentes.

category order

- **defineCategoryOrder**

De acuerdo a las relaciones definidas, el orden entre las categorías de creación de instancias resulta ser:

1) *singletonCategory*
2) *instanceCreationNotificationCategory*
3) *instanceInitializationCategory*

incompatibility

- **defineIncompatibleCategories**

De acuerdo a las relaciones de incompatibilidad definidas, queda determinado que *abstractClassCategory* es incompatible con resto de las categorías.

Clase MessageReceptionCategoryRelationshipDefinition

Esta clase tiene la responsabilidad de definir las relaciones de orden y de incompatibilidad entre las categorías de recepción de mensajes existentes.

category order

- **defineCategoryOrder**

De acuerdo a las relaciones definidas, el orden entre las categorías resulta ser:

1) *automaticAccessorCreatorCategory*
2) *traceCategory*
3) *haltCategory*

incompatibility

▪ **defineIncompatibleCategories**

Entre las propiedades de recepción de mensajes implementadas hasta ahora no existe ninguna incompatibilidad, así que por el momento el conjunto de estas relaciones es vacío.

7.2.3 Cómo se propagan las propiedades definidas a nivel de clases al subclasificar

Las propiedades de clase y las definidas para las instancias de una clase tienen un atributo que determina si deben propagarse a las subclases.

Al crearse una nueva clase, las propiedades de su superclase que tengan definido que deben propagarse a las subclases, se agregan a la nueva clase.

En caso que se cambie la superclase de una clase, se agregan a la subclase (y a las subclases de la misma) las propiedades que correspondan de la nueva superclase.

Un *builder* especializado, modelado con la clase llamada: *ClassPropertyPolicyBuilder-WhenSubclassing*, tiene la responsabilidad de crear nuevas estrategias para una clase y sus subclases, a partir de las propiedades que ya tenían y agregando las que deben propagarse de su superclase.

ClassPropertyPolicyBuilderWhenSubclassing
accessing
subclass
superclass
policyOfOldSubclass
policyGetter
policySetter
building
value
propertiesToAddToSubclass
propertiesToAddToSubclassFromPolicyOfOldSubclass
propertiesToAddToSubclassFromSuperclass

Figura 7.27: Clase ClassPropertyPolicyBuilderWhenSubclassing

Clase ClassPropertyPolicyBuilderWhenSubclassing

Modela la creación de nuevas estrategias para una clase y sus subclases, a partir de las propiedades que ya tenían y agregando las que deben propagarse de su superclase.

accessing

▪ **subclass**

Representa a la nueva clase creada o a la que se le está cambiando la superclase.

▪ **superclass**

Representa la superclase una clase recién creada o la nueva superclase que adquiere una clase que ya existía.

▪ **policyOfOldSubclass**

Para el caso en que se esté cambiando la superclase de una clase ya existente, representa la estrategia que tenía la subclase antes de esta modificación.

- **policyGetter**
- **policySetter**

Selectores que permiten acceder a la estrategia de metapropiedades de que se debe modificar.

building

- **value**

Construye una nueva estrategia para la subclase indicada.

Las propiedades que se agregan a esta estrategia son aquellas propiedades de la superclase que tengan definido que deben propagarse a las subclases.

Si la clase ya existía también se agregan las propiedades que ya tenía en su estrategia.

Si la clase tiene subclases, se agrega a la estrategia de las mismas las propiedades de la nueva superclase que se deben propagar.

7.3 Modificación de metapropiedades de un objeto usando modelo definido

Existen objetos que, haciendo uso del modelo definido en las secciones anteriores, se encargan de agregar o sacar propiedades de un objeto. Manejan las excepciones que puedan generar estas acciones. En caso que se trate de metapropiedades de clases, agregan o sacan las propiedades de las subclases cuando corresponda.

Son objetos de un mayor nivel de abstracción que los previamente definidos. Éstos son los objetos con los que deben interactuar los clientes que quieran usar y asignar metapropiedades.

Las clases que modelan esto se encuentran en una jerarquía encabezada por *MetaPropertiesModification.*

7.3.1 Clases que modelan la modificación de metapropiedades

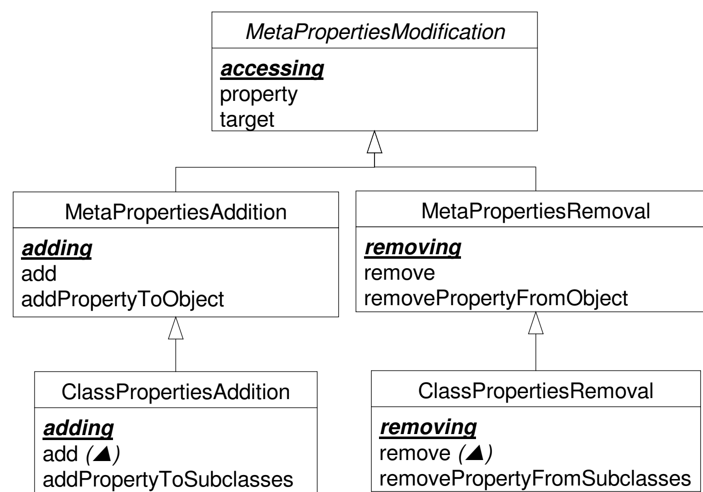


Figura 7.28: Jerarquía de MetaPropertyModification

Clase MetaPropertiesModification

Es la clase abstracta, raíz de esta jerarquía. Un objeto instancia de alguna de las clases de esta jerarquía queda definido a partir de una propiedad y el objeto al cual hay que agregarle o sacarle la propiedad.

accessing

▪ **property**

Es la metapropiedad que hay que agregar o sacar del *target*.

▪ **target**

Representa al objeto al cual hay que agregarle o sacarle la metapropiedad.

Clase MetaPropertiesAddition

Modela los objetos que se encargan del agregado de metapropiedades.

adding

- **add**

Agrega la metapropiedad definida al *target*. Esto lo hace delegando la tarea a la metapropiedad. Maneja las excepciones que eventualmente puedan ocurrir.

Clase MetaPropertiesRemoval

Modela los objetos que se encargan de la eliminación de metapropiedades.

removing

- **remove**

Elimina la metapropiedad definida del target. Esto lo hace delegándole esta responsabilidad a la metapropiedad. Maneja las excepciones que eventualmente puedan ocurrir.

Clase ClassMetaPropertiesAddition

Modela los objetos que se encargan del agregado de metapropiedades de objetos que sean específicamente clases.

adding

- **add**

Se agrega la metapropiedad a la clase y, si la definición de la propiedad así lo determina, también se agrega una copia de la misma a las subclases. Maneja las excepciones que eventualmente puedan ocurrir.

Clase ClassMetaPropertiesRemoval

Modela los objetos que encargan de la eliminación de metapropiedades de objetos que sean específicamente clases.

removing

- **remove**

Se elimina la metapropiedad de la clase y, si existe esta propiedad en alguna subclase también la elimina de la misma (siempre y cuando sea esto lo que se quiere). Maneja las excepciones que eventualmente puedan ocurrir**.**

7.3.2 Protocolo de propiedades relacionado con los objetos de la jerarquía de *MetaPropertiesModification*

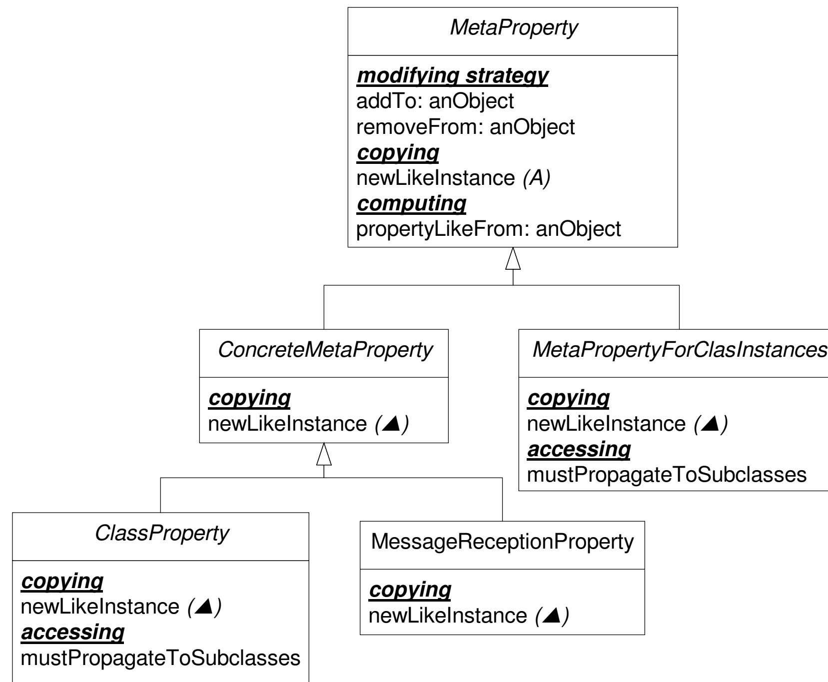


Figura 7.29: Jerarquía de MetaProperty y su protocolo relacionado con objetos de jerarquía de MetaPropertyModification

copying

▪ **newLikeInstance**

Devuelve una nueva metapropiedad que es instancia de la misma clase y con los mismos atributos de parametrización que la que receptora del mensaje. No se pasa a la nueva metapropiedad los atributos que relacionan a la propiedad con el objeto *owner* (como por ejemplo la estrategia o el atributo *instance* en el caso de la propiedad *singleton*)

La implementación de este método es responsabilidad de las subclases, ya que los atributos que se pasan a la nueva instancia dependen de la estructura de las mismas.

computing

▪ **propertyLikeFrom: anObject**

Devuelve una metapropiedad definida para el objeto parámetro que sea de la misma clase que la metapropiedad que recibe este mensaje y que haya sido configurada de la misma manera (por ejemplo en el caso de las propiedades de recepción de mensajes que tenga definido el mismo selector).

accessing

▪ **mustPropagateToSubclasses**

Para el caso de metapropiedades de clases, determina si al agregar la misma a una clase determinada también debe agregarse a sus subclases.

7.4 Cómo se alcanzan los objetivos propuestos en este modelo

Luego de haber definido las características y diseño del modelo de metapropiedades implementado, a continuación se resume de qué manera se alcanzaron los objetivos propuestos.

Identificación y categorización de las propiedades de clase

Se logró una taxonomía de metapropiedades, clasificando a las mismas de acuerdo a la funcionalidad de la clase a la que están asociadas.

Se reificaron las distintas funcionalidades de una clase que pueden caracterizarse por metapropiedades, como ser la creación de instancias, la recepción de mensajes, la subclasificación. Estas reificaciones se lograron mediante la creación de objetos que definimos como estrategias de metapropiedades, cuya responsabilidad principal es especificar modificaciones del comportamiento que se produce cuando una clase recibe alguno de los mensajes asociados a las funcionalidades de las categorías identificadas.

De esta manera, por cada una de las categorías detectadas se modeló una estrategia de metapropiedades específica. Así surgieron la estrategia de creación de instancias, la de recepción de mensajes y la de subclasificación.

Asignación y combinación dinámica de propiedades

En el modelo existen clases que modelan las propiedades simples, como las propiedades de creación de instancias (por ejemplo: *singletonProperty*, *instanceInitializationProperty*, *abstractProperty*, etc.) y las propiedades de recepción de mensajes (*haltProperty*, *traceProperty*, etc.).

Por otro lado, las clases tienen como colaboradores a las distintas estrategias de metapropiedades.

La asignación y combinación de propiedades para una clase se logra agregando estas propiedades simples a la correspondiente estrategia de la clase. Para conseguir que dos clases tengan la misma propiedad, se debe agregar un objeto distinto que represente a la misma propiedad en la estrategia respectiva de cada clase. Es decir, ambos objetos deben ser instancias de la misma clase (la que modela esta propiedad), lográndose con esto el reuso de la lógica de implementación.

De esta manera, contando con un conjunto de propiedades básicas y ensamblándolas en las estrategias de metapropiedades, se consigue que la clase cuente con las propiedades deseadas.

Estas asignaciones y combinaciones de propiedades se realizan dinámicamente, ya que para lograrlas sólo se modifican relaciones de colaboración o composición entre los objetos. Más concretamente, cuando se agrega una propiedad a una estrategia, se modifica el colaborador de la estrategia que representa sus propiedades para que conozca a la nueva propiedad.

Antes de continuar con el análisis del logro del resto de los objetivos, se expone el siguiente ejemplo en el que se reflejan estos conceptos y los que se mencionan en los otros puntos.

Cómo se asignan propiedades a clases de la jerarquía que modela lógica booleana

De la misma manera que se hizo cuando se estudiaron los otros modelos existentes, y con el fin de ejemplificar los conceptos de nuestro modelo y cómo se diferencia de los otros, se muestra cómo se realiza la asignación de propiedades en este modelo respecto a la jerarquía de clases que modelan la lógica booleana.

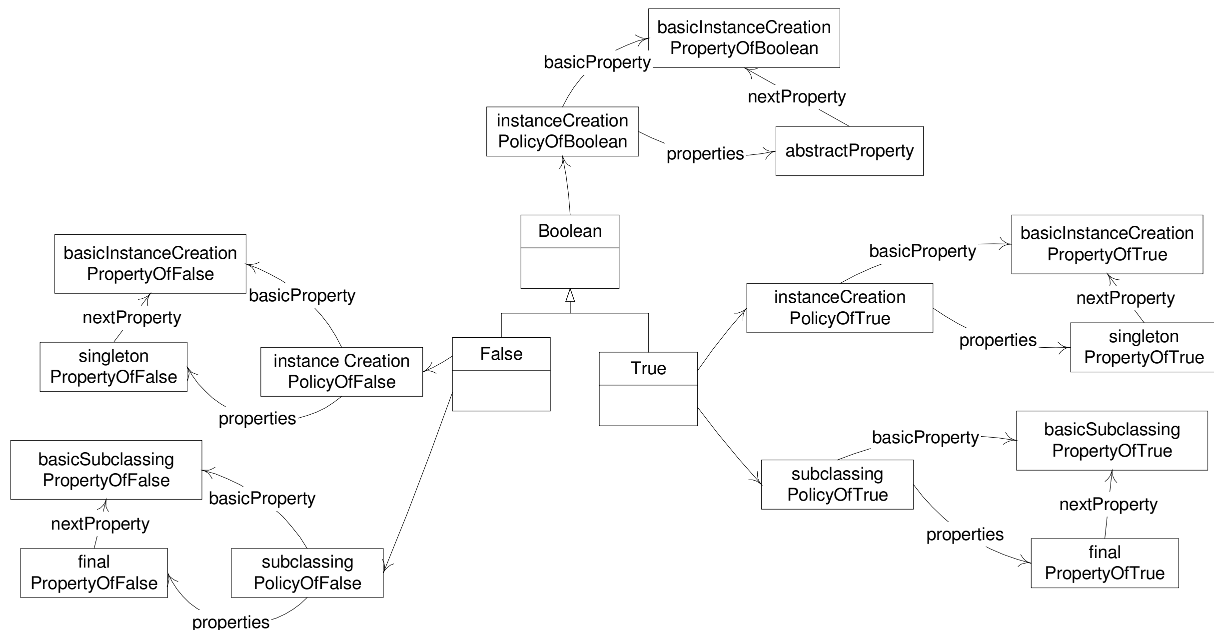


Figura 7.30: Asignación de propiedades a las clases que modelan lógica booleana en modelo propuesto

La propiedad *abstracta* se asigna a la clase *Boolean* agregando un objeto que represente a esta propiedad (*abstractProperty*) a la estrategia de creación de instancias de esta clase.

La propiedad *singleton* se asigna a la clase *True* agregando un objeto que represente a esta propiedad (*singletonPropertyOfTrue*) a la estrategia de creación de instancias de esta clase. La propiedad *final* se asigna a esta clase agregando un objeto que represente a esta propiedad (*finalPropertyOfTrue*) a su estrategia de subclasificación.

De forma análoga se logra la asignación de estas propiedades a la clase *False*.

En el diagrama se puede observar lo siguiente:

- Comparando este diagrama con sus análogos de los otros modelos analizados, se puede observar que en éste no aparecen metaclases. Esto es así debido a que en este modelo, a diferencia de aquellos, las propiedades no se implementan con metaclases ni tampoco se usan relaciones de herencia para asignar o combinar las propiedades. Aquí, estos objetivos se logran haciendo que las clases colaboren con estrategias definidas a partir de objetos que representan a las metapropiedades.
- Se puede observar que cada una de las estrategias tiene como colaborador a una propiedad básica. Como ya fue explicado, toda estrategia nace con esta propiedad, que siempre es la última en aplicarse.
- Respecto a la propiedad abstracta asignada a la clase *Boolean*, se observa que tiene una relación de *nextProperty* con la propiedad básica de la estrategia. Esto no significa que la propiedad básica vaya a aplicarse. Como se explico antes, una propiedad delega la responsabilidad pertinente a la siguiente propiedad para que se ejecute sólo si esto corresponde de acuerdo a su implementación. En el caso de la propiedad abstracta esta delegación no ocurre.

  Esto se aplica análogamente respecto a la definición de la propiedad *final* en las clases *True* y *False.*

Solución al problema de propagación de propiedades de clase

En este modelo no existe este problema. Veamos por qué sucede esto.

Las propiedades se asignan a una clase agregando objetos que las representen a la estrategia correspondiente de la clase.

Por otro lado, cada clase tiene definida sus propias estrategias de metapropiedades, que son independientes de las estrategias definidas para sus subclases. Esto es así debido a que las estrategias que se definen a una clase son colaboradores internos de la misma. Y los colaboradores internos de una clase, al igual que los de cualquier objeto, son propios de ésta y no se heredan. Esto se puede ver en el ejemplo de la jerarquía booleana: la clase *Boolean* colabora con una estrategia de creación de instancias que incluye la propiedad *abstracta*, mientras que la clase *True* colabora con otra estrategia de creación de instancias que incluye a la propiedad *singleton*, pero no la *abstracta.*

Considerando las afirmaciones de los dos párrafos anteriores, se puede ver que en este modelo no existe el problema de propagación de propiedades.

Compatibilidad de comunicación entre los niveles de clase e instancia

Esto está asegurado por haber usado el ambiente y el metamodelo de Smalltalk para implementar este modelo. Como ya fue explicado, en Smalltalk esta compatibilidad se asegura debido a que la jerarquía de metaclases está determinada por la de sus clases instancias y de esta manera ambas jerarquías resultan paralelas.

Validación dinámica de propiedades asignadas

Al implementar una propiedad deben definirse determinadas restricciones respecto a su relación con las otras propiedades que se pueden agregar a la misma estrategia (por ejemplo: de creación de instancias o de recepción de mensajes), con el fin de garantizar que la posterior combinación de las mismas sea consistente. Estas restricciones reflejan relaciones de incompatibilidad entre propiedades y relaciones de orden entre las mismas, considerando sus precedencias de aplicación. Para definir estas restricciones, primero se deben establecer las relaciones mencionadas entre los objetos que se definieron como categorías de metapropiedades. Luego, se debe determinar con cuál de estos objetos queda relacionado cada tipo de propiedad.

Cada vez que se asigna una propiedad, existen objetos que validan que se cumplan las restricciones definidas. Además, se valida que no se esté agregando una propiedad por duplicado. También, en los casos que corresponda, se verifica que la asignación que se quiere realizar sea consistente con el contexto de la clase en ese momento.

De esta manera se logra el reuso de la lógica de las validaciones, ya que las restricciones específicas sobre cada propiedad sólo deben determinarse cuando se define cada una de ellas. Por otro lado, el modelo asegura en todo momento la consistencia de las asignaciones realizadas, validando en cada asignación que se cumplan tanto las restricciones específicas definidas como los chequeos comunes. Así se libera de esta responsabilidad a quien usa las propiedades.

8 Interfaces de usuario para trabajar con metapropiedades en el ambiente de Squeak

8.1 Diseño de interfaces de usuario

Con el objetivo de facilitar el uso de metapropiedades se diseñaron e implementaron interfaces de usuario que permiten asignar propiedades a clases o a objetos en general según corresponda, así como inspeccionar las propiedades ya definidas.

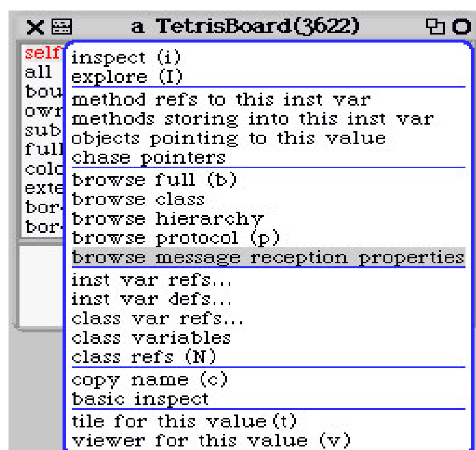
Por cada estrategia que se pueda definir para un objeto o para una clase en particular, se accede a una vista distinta. En estas interfaces se listan las metapropiedades que incluye la estrategia del objeto inspeccionada y se ofrece la posibilidad de agregar o eliminar metapropiedades.

Cuando se intenta agregar o eliminar metapropiedades de una estrategia, se informa acerca de eventuales validaciones no cumplidas y se solicitan confirmaciones o elecciones de alternativas que correspondan. Además, en el caso de agregar una metapropiedad que requiera parametrización, se solicita que se ingrese el valor de los parámetros necesarios.

A continuación se detalla cada una de las vistas implementadas.

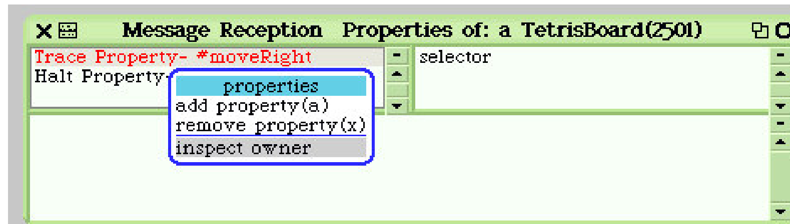
8.1.1 Interfaces de propiedades de recepción de mensajes de un objeto

En el menú contextual del inspector de cualquier objeto se agregó la opción: *browse message reception properties.*



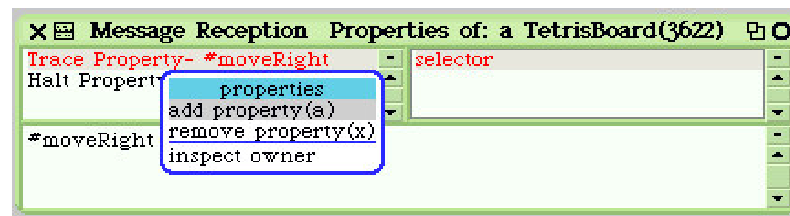
Al elegir esta opción se abre un *browser* que lista las propiedades de recepción de mensajes (como así también los atributos de cada una de ellas) que tiene el objeto que se está inspeccionando.

Desde este browser se puede inspeccionar al owner de la estrategia que se está mostrando.



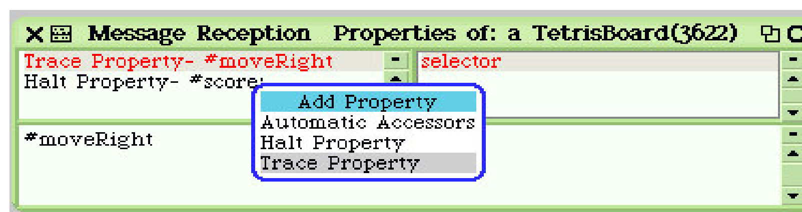
También cada atributo de las metapropiedades listado puede ser inspeccionado.

Desde este *browser* se pueden agregar o eliminar propiedades.

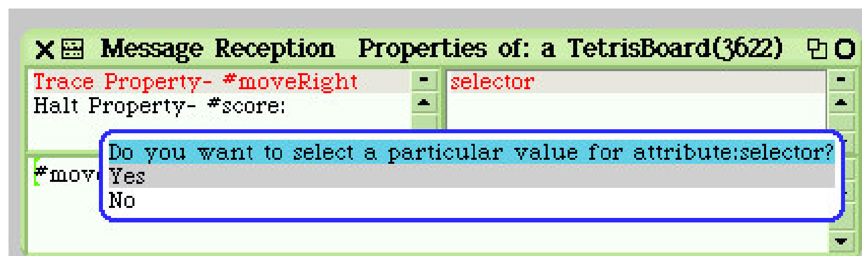


8.1.1.1 Agregado de propiedades

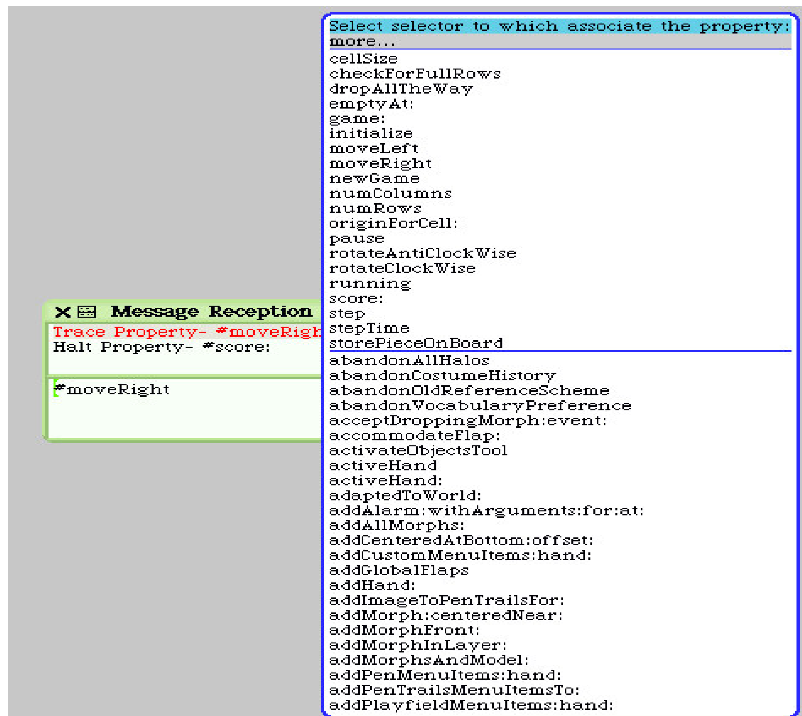
Para agregar una propiedad, primero se solicita la elección del tipo de la misma.



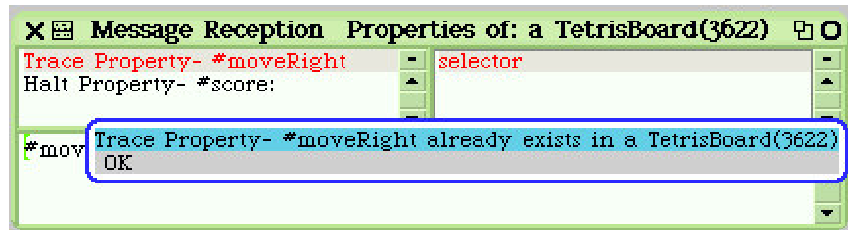
Luego se pregunta si se quiere asociar la propiedad a algún selector en particular.



En caso afirmativo se solicita se elija el selector.

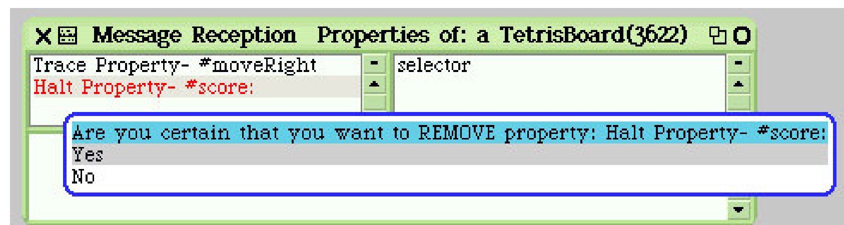


En caso que ya exista una propiedad equivalente sale un mensaje que avisa esto.



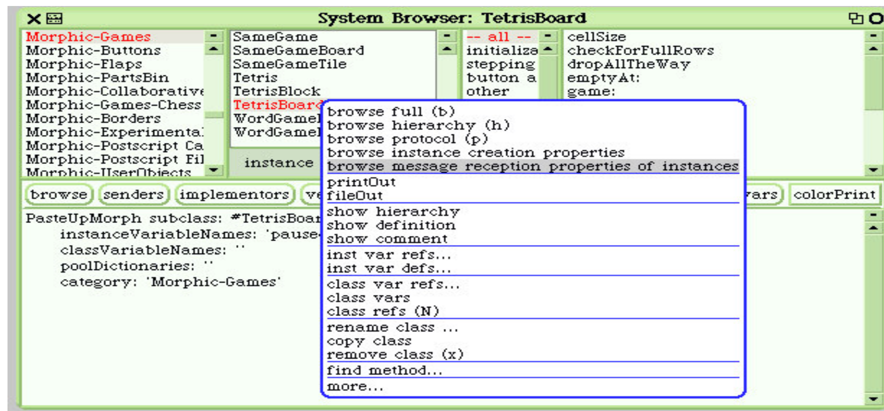
8.1.1.2 Eliminación de propiedades

Primero se debe seleccionar la propiedad que se quiere eliminar. Inmediatamente se pide una confirmación para realizar esta operación.



8.1.2 Interfaces de propiedades de recepción de mensajes definidas para las instancias de una clase

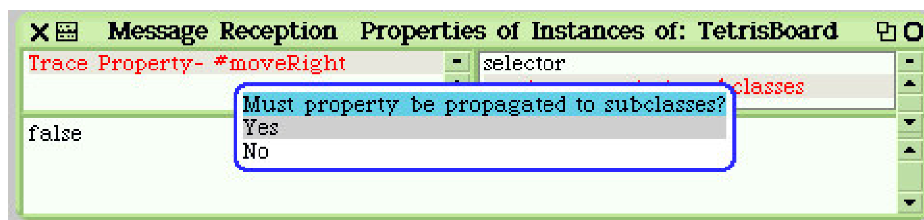
En el menú contextual que se abre al estar seleccionada una clase se agregó la opción: *browse message reception properties of instances.*

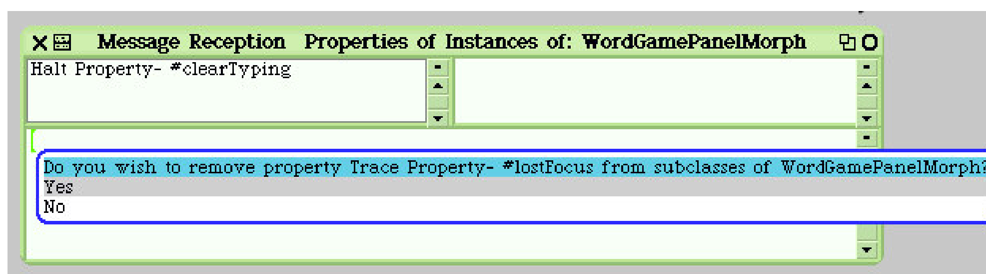


Al seleccionar esta opción se abre un *browser* que lista las propiedades de recepción de mensajes que se aplican en general para todas las instancias de la clase seleccionada. En este caso, se agrega como atributo de cada propiedad aquel que determina si la propiedad se debe propagar a las subclases.

De la misma manera que en el *browser* de propiedades para un objeto en particular, desde aquí se pueden agregar o eliminar propiedades de recepción de mensajes.

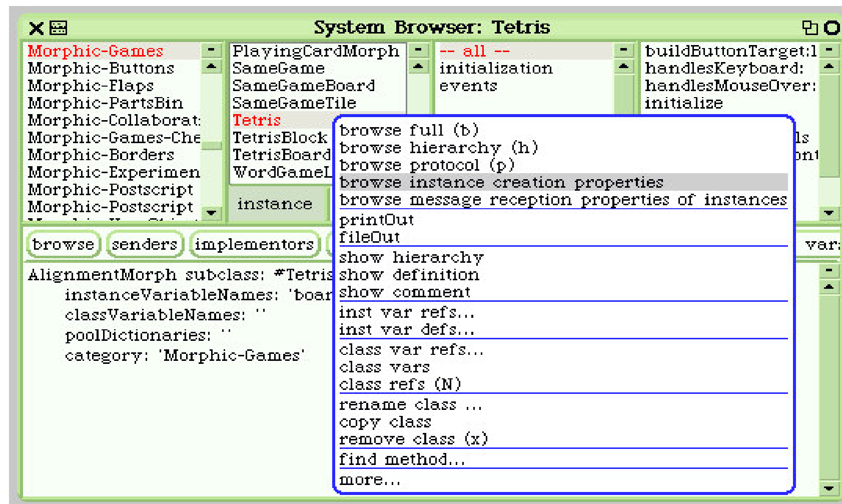
La interacción para realizar estas operaciones es la misma que la presentada en la sección anterior, pero se agrega una confirmación adicional para indicar si se quiere agregar o eliminar la propiedad de las subclases.





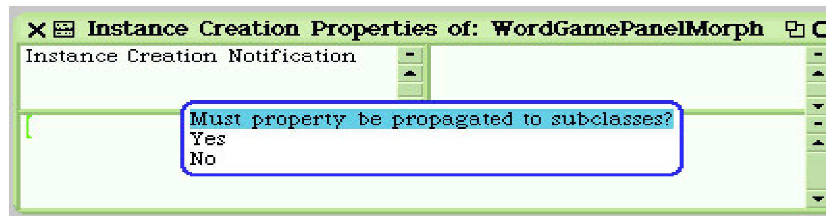
8.1.3 Interfaces de propiedades de creación de instancias

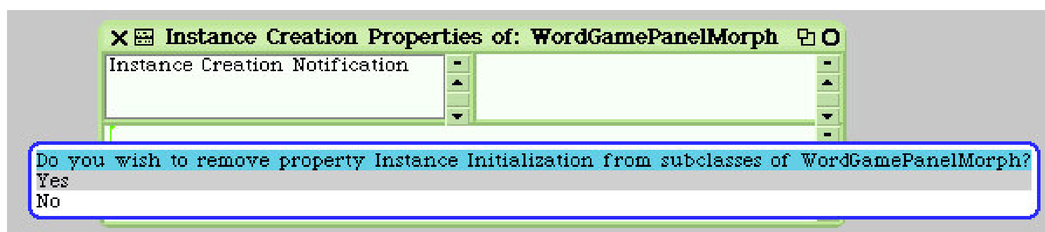
En el menú contextual que se abre al estar seleccionada una clase se agregó la opción: *browse instance creation properties.*



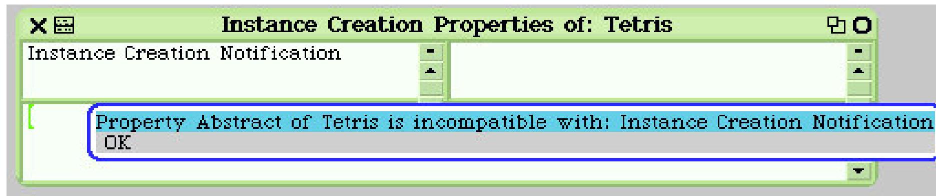
Al seleccionar esta opción se abre un *browser* que lista las propiedades de creación de instancias (y atributos de cada una de ellas) que tiene la clase seleccionada.

De forma análoga a los *browsers* de propiedades de recepción de mensajes para las instancias de una clase, desde aquí se pueden agregar o eliminar propiedades de creación de instancias para la clase del contexto. Como en el caso anterior, aquí también se pide que se determine si el alcance de estas operaciones llega a las subclases.

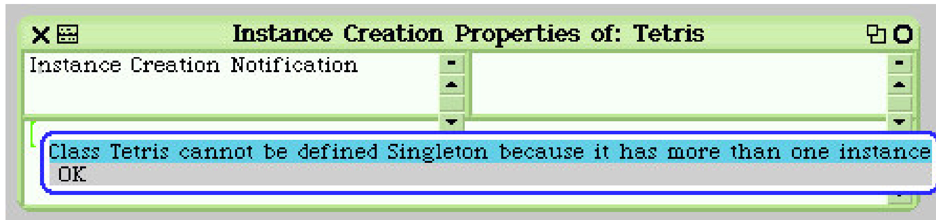




Tampoco se permite agregar una propiedad que sea incompatible con las ya existentes, y se indican las incompatibilidades.



Además se avisa si no se puede agregar una propiedad debido a que dicho agregado no es consistente con el contexto actual de la clase a la que se quiere aplicar.



8.2 Modelo de browsers de metapropiedades

Las interfaces de usuario están modeladas por una jerarquía encabezada por una clase que modela abstractamente a un *browser de metapropiedades*.Un *browser de metapropiedades* se abre sobre un determinado objeto y lista las propiedades de la estrategia de metapropiedades de este objeto que corresponda. Al seleccionarse una de las propiedades listadas muestra sus atributos, al seleccionarse uno de los atributos listados muestra el valor del mismo. Además, este browser permite agregar y eliminar metapropiedades de la estrategia.Por otro lado, se modelaron objetos con los que colaboran los browsers para cumplir sus responsabilidades. Estos objetos pertenecen a jerarquías que se pueden extender para modelar las interfaces de un grupo de metapropiedades en particular, sólo definiendo lo que es específico para el mismo.El diseño resultante puede ser extendido sin demasiada complejidad para implementar browsers para un nuevo grupo de metapropiedades. Para lograr esto se requiere:

- Crear una subclase de la clase abstracta de la jerarquía de browsers e implementar algunos métodos que se definen en la misma como responsabilidad de las subclases. En general la implementación de estos métodos consiste en la determinación de los objetos con los que debe colaborar el *browser*.
- Extender las jerarquías de clases con las que se relacionan los browsers para obtener el comportamiento particular para este grupo de metapropiedades.

A continuación detallamos el diseño de la jerarquía de browsers de metapropiedades y luego se exponen los objetos con los que colaboran los browsers.

8.2.1 Jerarquía de clases de *MetaPropertyBrowser*

A continuación se presenta la jerarquía de clases que modelan los *browsers* de metapropiedades que se presentaron.

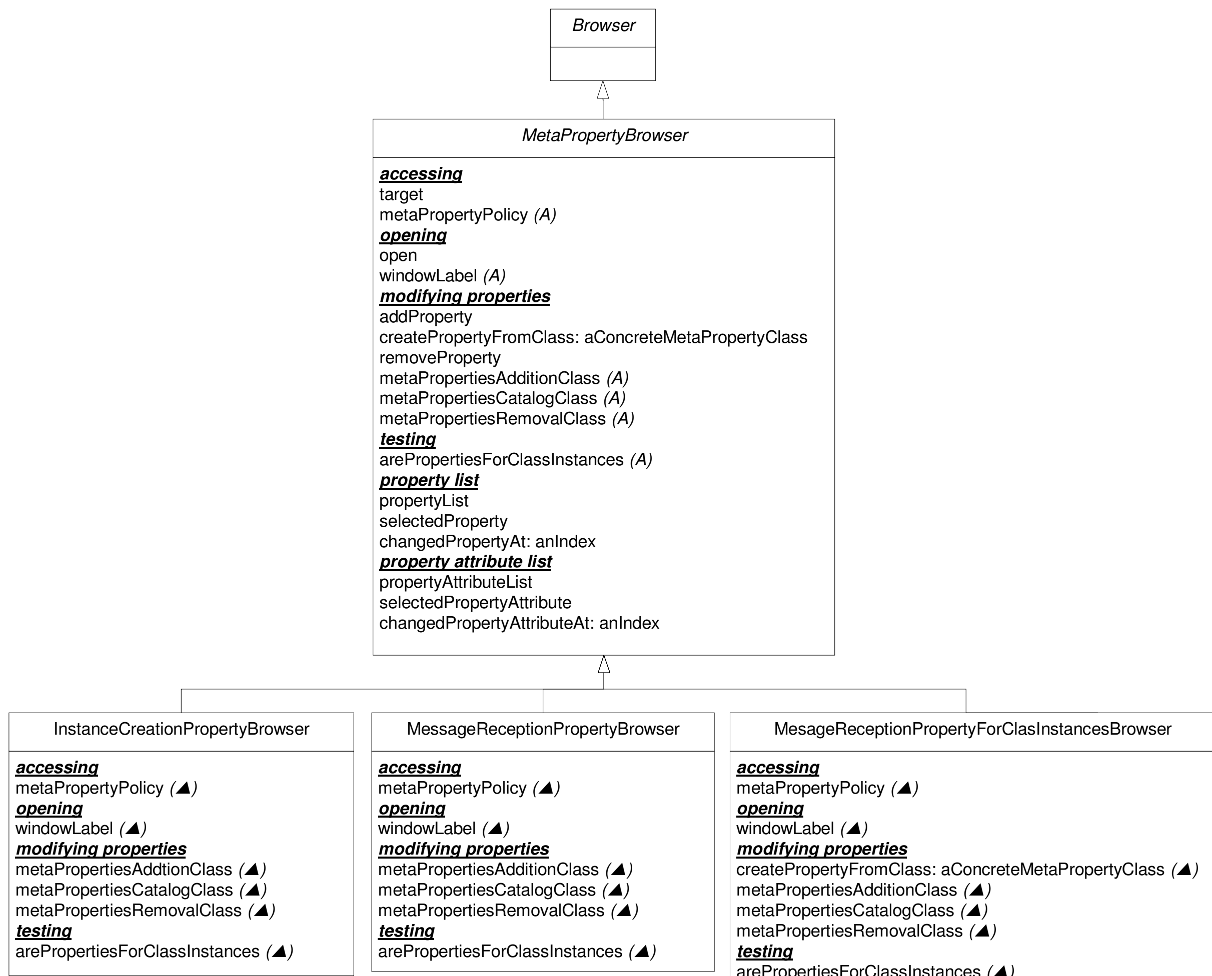


Figura 8.1: Jerarquía de clases de MetaPropertyBrowser

Clase MetaPropertyBrowser

Modela los *browsers* de metapropiedades. Es una clase abstracta.

accessing

- **target**

  Es el objeto sobre el que se abre el *browser* y cuyas metapropiedades son listadas

- **metaPropertyPolicy**

  Determina la estrategia de metapropiedades del *target* sobre la que se va a abrir el *browser*. Esto se define en cada subclase.

opening

- **open**

  Abre el *browser* sobre las metapropiedades de la estrategia indicada del *target*.

- **windowLabel**

  Determina el título de la ventana del *browser*. Esto se define en cada subclase.

modifying properties

- **addProperty**

  Permite agregar una propiedad a la estrategia del objeto.

  Las propiedades que se pueden agregar son determinadas por el objeto *metaPropertiesCatalog* asociado al *browser*. El agregado de la propiedad a la estrategia de metapropiedades del *target* se delega en el objeto que se encarga del agregado de metapropiedades que también determina el *browser*.

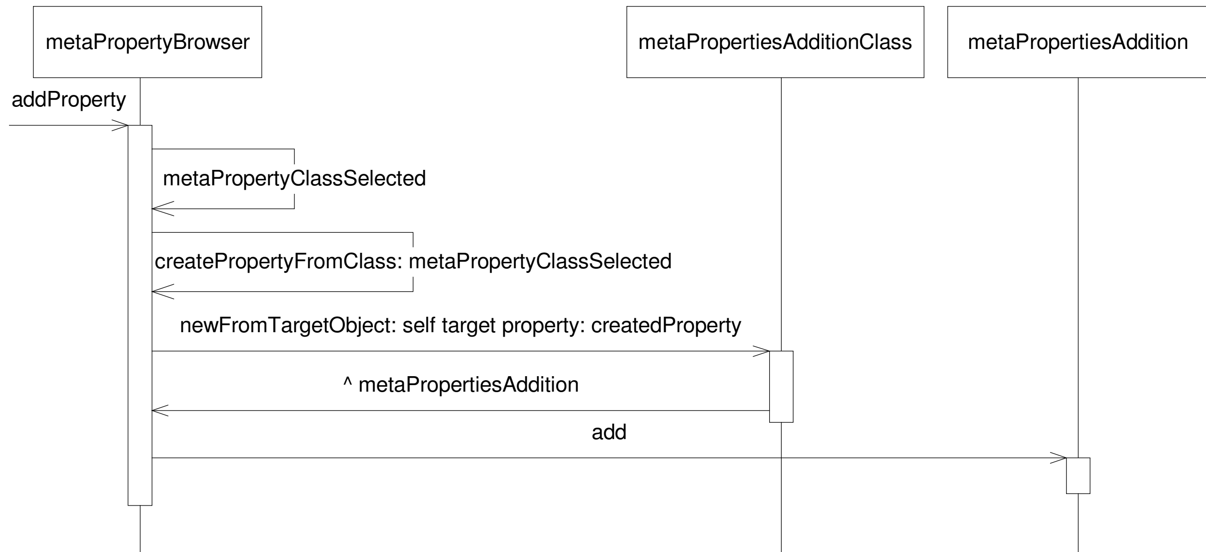


Figura 8.2: Método addProperty definido en MetaPropertyBrowser

- **createPropertyFromClass: aConcreteMetaPropertyClass**

  Crea y devuelve una nueva metapropiedad, instancia de la clase recibida a partir de valores de atributos determinados como necesarios para la creación de la nueva instancia. Para realizar esto colabora con el objeto de definición de atributos asociado a la clase de propiedad recibida.

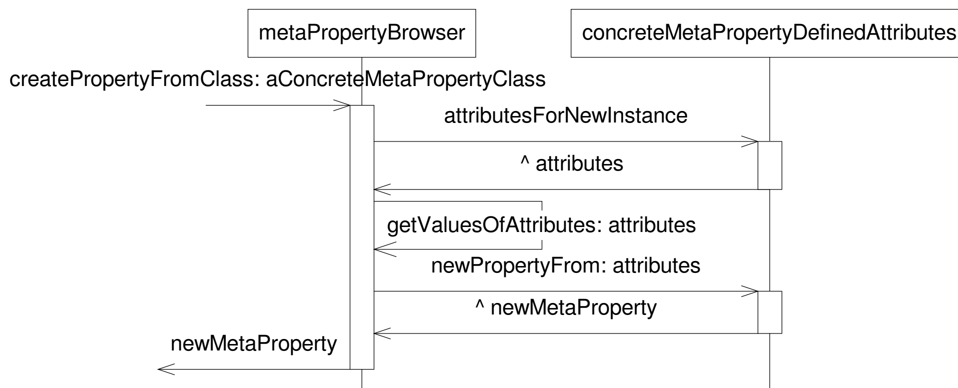


Figura 8.3: Método createPropertyFromClass: definido en MetaPropertyBrowser

▪ **getValuesOfAttributes: anAttributeCollection**
Obtiene el valor de los atributos de propiedades recibidos. Esto se hace delegando esta responsabilidad al editor de cada atributo.

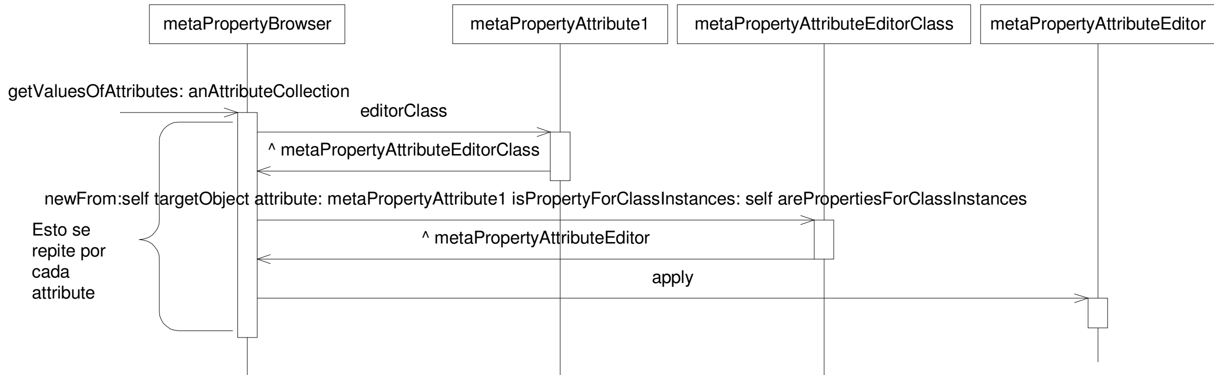


Figura 8.4: Método getValuesOfAttributes: definido en MetaPropertyBrowser

▪ **removeProperty**
Permite eliminar una propiedad de la estrategia del objeto determinado.

La eliminación de la propiedad en la estrategia de metapropiedades del *target* se delega en el objeto encargado de la eliminación de metapropiedades que determina el *browser.*

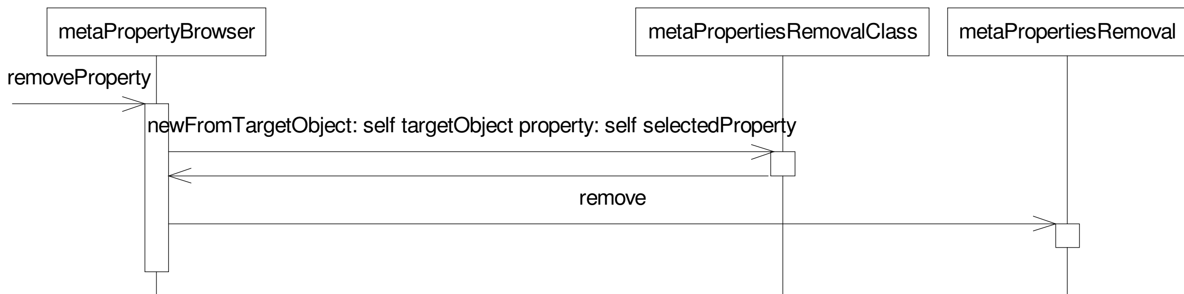


Figura 8.5: Método removeProperty: definido en MetaPropertyBrowser

▪ **metaPropertiesAdditionClass**
Determina la clase del objeto encargado del agregado de metapropiedades al *target* asociada al *browser*. Esto es definido en cada subclase.

▪ **metaPropertiesCatalogClass**
Determina la clase del objeto que determina las clases de metapropiedades que se pueden agregar a la estrategia que se está browseando. Esto es definido en subclase.

▪ **metaPropertiesRemovalClass**
Determina la clase del objeto encargado de eliminación de metapropiedades del *target* asociada al *browser*. Esto es definido en cada subclase.

testing

▪ **arePropertiesForClassInstances**
Determina si las propiedades que se muestran son propiedades determinadas para las instancias de una clase. Esta definición se necesita para crear el editor de atributos de propiedades, ver el diagrama de colaboraciones del mensaje *createPropertyFromClass:* donde esto queda expuesto. Este mensaje es implementado por cada subclase.

property list

- **propertyList**
- **selectedProperty**
- **changedPropertyListIndex: anIndex**

Los mensajes de esta categoría tienen la responsabilidad de determinar la lista de propiedades que se muestran y de detectar dentro de la misma al elemento seleccionado para así poder exhibir los atributos de la propiedad que correspondan.

property attribute list

- **propertyAttributeList**
- **selectedPropertyAttribute**
- **changedPropertyAttributeListIndex: anIndex**

Los mensajes de esta categoría tienen la responsabilidad de determinar la lista de atributos de la propiedad seleccionada que se muestran y de detectar dentro de la misma al elemento seleccionado para así poder exhibir el valor del atributo que corresponda.

Clase InstanceCreationPropertyBrowser

Modela los *browsers* que permiten mostrar y administrar las metapropiedades de creación de instancias de una clase.

accessing

- **metaPropertyPolicy**

Devuelve la estrategia de creación de instancias de la clase sobre la que se abre el *browser*.

modifying properties

- **metaPropertiesAdditionClass**

Devuelve la clase *ClassPropertiesAddition.*

- **metaPropertiesCatalogClass**

Devuelve la clase *InstanceCreationPropertiesCatalog.*

- **metaPropertiesRemovalClass**

Devuelve la clase *ClassPropertiesRemoval.*

testing

- **arePropertiesForClassInstances**

Devuelve *false*

Clase MessageReceptionPropertyBrowser

Modela los *browsers* que permiten mostrar y administrar metapropiedades de recepción de mensajes de un determinado objeto.

accessing

- **metaPropertyPolicy**

Devuelve la estrategia de recepción de mensajes del objeto sobre el que se abre el *browser*.

modifying properties

- **addingMetaPropertiesPolicyClass**

Devuelve la clase *MetaPropertiesAddition.*

- **metaPropertiesCatalogClass**

Devuelve la clase *MessageReceptionPropertiesCatalog.*

- **removingMetaPropertiesPolicyClass**
  Devuelve la clase *MetaPropertiesRemoval.*

testing

- **arePropertiesForClassInstances**
  Devuelve *false.*

Clase MessageReceptionPropertyForClassInstancesBrowser

Modela los *browsers* que permiten mostrar y administrar metapropiedades de recepción de mensajes para las instancias de una clase.

accessing

- **metaPropertyPolicy**
  Devuelve la estrategia de recepción de mensajes para las instancias de la clase sobre la que se abre el *browser*.

modifying properties

- **createPropertyFromClass: aMessageReceptionPropertyClass**
  Primero se crea una propiedad de recepción de mensajes a partir de la clase recibida, como está determinado en la superclase.

  Luego a partir de esta nueva propiedad y los atributos propios se crea y devuelve una propiedad de recepción de mensajes para instancias.

- **metaPropertiesAdditionClass**
  Devuelve la clase *ClassPropertiesAddition.*

- **metaPropertiesCatalogClass**
  Devuelve la clase *MessageReceptionPropertiesCatalog.*

- **metaPropertiesRemovalClass**
  Devuelve la clase *ClassPropertiesRemoval.*

testing

- **arePropertiesForClassInstances**
  Devuelve *true.*

8.2.2 Objetos y protocolo con los que se relacionan los browsers de metapropiedades

8.2.2.1 Jerarquía de clases de *MetaPropertiesCatalog*

Los *browsers* delegan en estos objetos la responsabilidad de determinar las metapropiedades que se pueden agregar al objeto sobre el que se abre el *browser*.

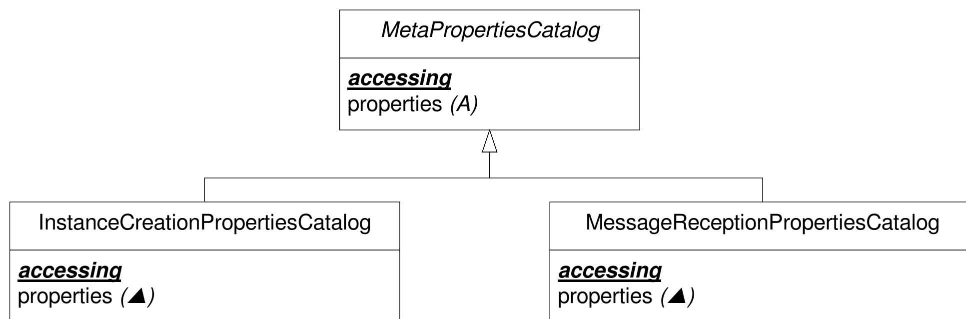


Figura 8.6: Jerarquía de MetaPropertiesCatalog

8.2.2.2 Definición de atributos de metapropiedades

El browser delega en objetos que manejan la definición de atributos de las metapropiedades, tanto la responsabilidad de determinar los atributos necesarios para crear una nueva metapropiedad, como la de determinar los atributos que se pueden mostrar. Estos objetos también tienen la responsabilidad de crear metapropiedades a partir de una colección de atributos con sus valores ya definidos.

La jerarquía de *MetaPropertyAttributesDefinition* define estos objetos.

Por cada clase de metapropiedad que se pueda agregar desde los browsers , se debe definir qué clase de esta jerarquía maneja sus atributos. Esto se hace definiendo en las clases concretas de esta jerarquía los tipos de propiedades manejan.

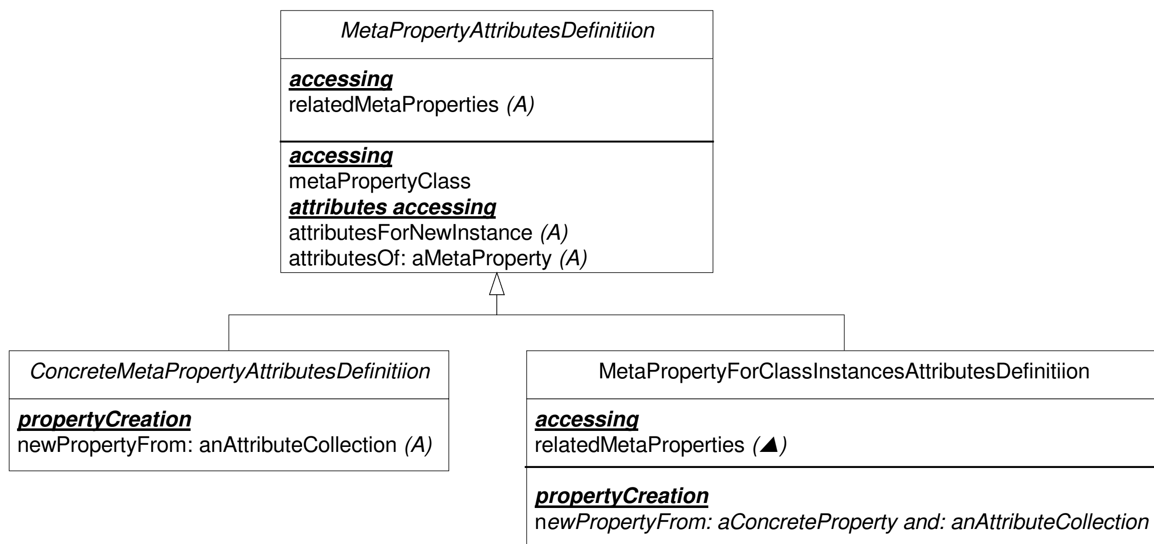


Figura 8.7: Jerarquía de MetaPropertyAttributesDefinition

Clase MetaPropertyAttributesDefinition

Modela la definición de atributos de las metapropiedades. Una definición de atributos se crea a partir de una clase de metapropiedades.

accessing

- **metaPropertyClass**

Representa la clase de propiedad a la que está asociada la definición de atributos.

attributes accessing

- **attributesForNewInstance**

Devuelve los atributos que necesita la clase de metapropiedad de la definición para poder crear una nueva instancia. Esto se define en cada subclase dependiendo de los tipos de metapropiedades a los que esté asociada. Por ejemplo, una definición de atributos que tenga asociada una metapropiedad de clase, determinará que uno de los atributos necesarios para crear una nueva propiedad es aquel que determina si la propiedad debe propagarse a las subclases. Por otro lado, una definición que tenga asociada a las metapropiedades de recepción de mensajes determinará que el atributo asociado a la determinación de selectores es necesario.

- **attributesOf: aMetaProperty**

Devuelve los atributos que se pueden mostrar de la metapropiedad que se recibe. Nuevamente, esto se define en cada subclase dependiendo de los tipos de metapropiedades a los que esté asociado. Por ejemplo, una definición de atributos que tenga asociada una metapropiedad de clase, determinará que uno de los atributos de la metapropiedad que se recibe es aquel que determina si la propiedad debe propagarse a las subclases. Por otro lado, una definición que tenga asociada a la propiedad singleton determinará que el atributo que determina la única instancia de la clase owner registrada se puede mostrar.

Clase ConcreteMetaPropertyAttributesDefinition

Modela la definición de atributos de las metapropiedades concretas.

property creation

- **newPropertyFrom: anAttributeCollection**

Crea una nueva propiedad instancia de la clase de la definición a partir de los atributos recibidos. Esto se implementa en cada subclase dependiendo de los tipos de metapropiedades a la los que esté asociado la definición.

Clase MetaPropertyForClassInstancesAttributesDefinition

Modela la definición de atributos de las metapropiedades definidas para las instancias de una clase.

property creation

- **newPropertyFrom: aConcreteProperty and: anAttributeCollection**

Crea una nueva propiedad para instancias enviando el mensaje de creación a la clase de la definición y pasando como colaboradores a la propiedad concreta y atributos recibidos.

8.2.2.3 Atributos de metapropiedades

Un atributo permite acceder y determinar el valor de un determinado accessor (o variable de instancia) de una metapropiedad.

Los atributos se usan para que una clase de metapropiedad determine qué valores son necesarios para la creación de una nueva instancia. En este caso el atributo posee protocolo para poder determinar el valor del accessor al que está asociado.

También se los usa para que una propiedad ya creada determine cuáles de sus atributos son públicos. En este caso se permite sólo acceder al valor de estos atributos y no modificarlos.

A continuación se expone la jerarquía de clases que modelan los atributos descriptos.

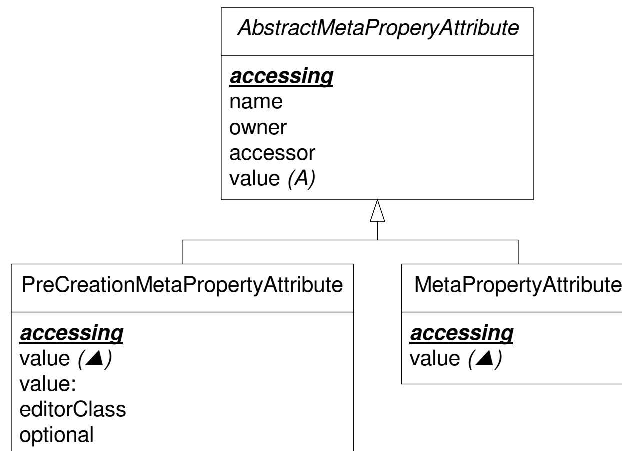


Figura 8.8: Jerarquía de AbstractMetaPropertyAttribute

Clase AbstractMetaPropertyAttribute

Modela los atributos de metapropiedades en general.

accessing

- **name**
  Nombre del atributo.
- **owner**
  Es la metapropiedad o clase de la misma sobre la cual se crea el atributo.
- **accessor**
  Es el selector a partir del cual se puede acceder al valor del atributo.
- **value**
  Representa el valor del atributo.

Clase PreCreationMetaPropertyAttribute

Estos son los atributos que usa una clase de metapropiedad para determinar los atributos necesarios para la creación de una nueva instancia.

En este caso el *owner* del atributo es una clase de metapropiedad.

accessing

- **value**
  En este caso el valor del *accessor* lo tiene el mismo atributo ya que en este contexto no existe la instancia de metapropiedad. Representa el valor que va a tener el *accessor* de la instancia a crear.
- **value:**
  Se usa para determinar el valor del *accessor* de la instancia de metapropiedad a crear.

- **editorClass**
  Representa la clase del editor que permite determinar el valor del *accessor*.
- **optional**
  Determina si es opcional que se defina el valor del atributo.

Clase MetaPropertyAttribute

Estos son los atributos que publica una metapropiedad. En este caso el *owner* del atributo es una metapropiedad.

accessing

- **value**
  En este caso el valor del *accessor* se obtiene accediendo a la metapropiedad *owner*.

8.2.2.4 Editores de atributos de metapropiedades

Los editores de atributos permiten, mediante interfaces de usuario, determinar el valor del *accessor* de un *preCreationMetaPropertyAttribute*. Tienen la responsabilidad de asegurar que los valores determinados sean válidos.

A continuación se expone la jerarquía de clases que modelan estos conceptos.

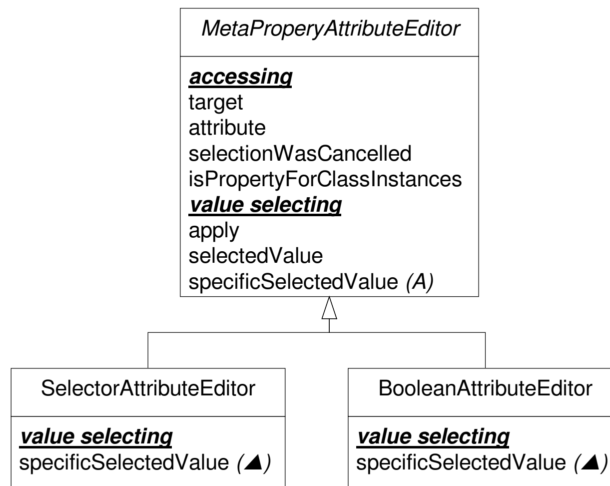


Figura 8.9: Jerarquía de MetaPropertyAttributeEditor

En lo implementado hasta ahora, los únicos atributos declarados como necesarios para la creación de una nueva instancia están asociados al *selector* de las propiedades de recepción de mensajes y al accessor *mustPropagateToSubclasses* de las propiedades de clases. Por esto, las únicas clases concretas que surgieron son: *SelectorAttributeEditor* y *BooleanAttributeEditor.*

Clase MetaPropertyAttributeEditor

Modela abstractamente los editores de atributos de metapropiedades y determina su protocolo genérico.

accessing

- **target**
  Es el objeto al cual se quiere agregar la propiedad.

- **attribute**
  Es el atributo de metapropiedad cuyo valor se quiere determinar.
- **selectionWasCancelled**
  Indica si se canceló la determinación del valor del accessor del atributo.
- **isPropertyForClassInstances**
  Determina si la propiedad cuyo accessor se quiere determinar es una propiedad para las instancias de una clase.

value selecting

- **apply**
  Asigna el valor seleccionado al atributo.
- **selectedValue**
  Permite que se determine el valor para el *accessor* del atributo. Si el atributo es opcional, se da opción a que no se determine este valor. Devuelve el valor seleccionado.
- **specificSelectedValue**
  Permite en forma concreta determinar el valor del *accessor* del atributo. La implementación de esto depende de cada subclase en particular.

Clase SelectorAttributeEditor

Permite determinar el valor de un atributo cuyo valor tiene que ser uno de los selectores definidos para el *target*.

Clase BooleanAttributeEditor

Permite determinar el valor de un atributo cuyo valor tiene que ser booleano.

8.2.2.5 Colaboraciones con objetos de jerarquía de *MetaPropertiesModification*

Los *browsers* de metapropiedades delegan en objetos de esta jerarquía la responsabilidad de agregar o eliminar una propiedad del objeto sobre el que se abren.

La clase de estos objetos con los que se colabora se define en los mensajes *metaPropertiesAdditionClass* o *metaPropertiesRemovalClass* del *browser*.

9 Cómo extender el modelo existente para implementar otras metapropiedades

La implementación del modelo resultante constituye un framework de metapropiedades.

En el modelo están reificados en forma genérica los mecanismos de asignación y el de aplicación de metapropiedades. Esto facilita la extensión del modelo para implementar otras metapropiedades que no fueron desarrolladas en este trabajo. Se puede lograr un mejor resultado gracias a las abstracciones modeladas, que permiten crear nuevas propiedades o estrategias definiendo solamente lo específico de las mismas sin tener que volver a definir lo que es común a otras propiedades o estrategias.

Veamos cómo se puede lograr esto.

9.1 Definición de una nueva metapropiedad relacionada con una funcionalidad ya reificada

En este caso se quiere definir una nueva metapropiedad asociada a una funcionalidad ya reificada mediante una estrategia de metapropiedades. Los pasos a seguir en esta situación son:

1) ***Crear una clase que modele la nueva propiedad***

Esta clase debe ser subclase de la clase abstracta que modela las propiedades asociadas a la funcionalidad en cuestión.

Por ejemplo, en caso de una metapropiedad asociada a la creación de instancias la nueva clase debe ser subclase de *InstanceCreationProperty*. En caso de tratarse de una metapropiedad asociada a la recepción de mensajes debe ser subclase de *MessageReceptionProperty*.

2) ***Implementar los métodos definidos como abstractos por las superclases de la nueva clase.***

Por ejemplo, en el caso de una metapropiedad asociada a la creación de instancias se deben implementar los siguientes métodos de instancia:

- ✓ *createNewInstance*
- ✓ *validateApplicabilityTo: aClass*
- ✓ *instanceCreationPolicyWasSet*

y los siguientes métodos de clase:

- ✓ *defaultPropagationToSubclasses*

- ✓ *propertyCategory*
- ✓ *propertyName.*

En cambio, para el caso de una propiedad de recepción de mensajes se debe definir el siguiente método de instancia:

- ✓ *specificReceiveMessage: aMessage*

y los siguientes métodos de clase:

- ✓ *propertyCategory*
- ✓ *propertyName.*

3) ***Crear una nueva categoría de metapropiedades***

En caso que se necesite tener una categoría específica para la nueva metapropiedad, se debe agregar en las inicializaciones de categorías que se hacen en *MetaPropertyCategory*, la inicialización de una categoría asociada a la nueva metapropiedad. Dicha inicialización consiste en crear una nueva instancia de esta clase como se hace para las otras propiedades implementadas.

4) ***Definir relaciones de orden de la nueva categoría de metapropiedad respecto a las otras ya existentes***

Se logra modificando la clase que modela cómo se relacionan entre sí las categorías del grupo al que pertenece.

Por ejemplo, si se trata de una categoría de metapropiedad asociada a la creación de instancias, se debe agregar la nueva categoría en las definiciones de relaciones de orden establecidas en la clase *InstanceCreationCategoryRelationshipDefinition*, y en el caso de una categoría metapropiedad de recepción de mensajes, se debe agregar a las definiciones que se realizan en *MessageReceptionCategoryRelationshipDefinition*.

5) ***De forma análoga a la definición de relaciones de orden, también se deben definir, en caso que existan, las relaciones de incompatibilidad con otras categorías.***

6) ***Agregar la clase que modela la nueva propiedad en el método de inicialización de instancias de la subclase de MetaPropertiesCatalog que corresponda.***

Por ejemplo, para el caso de una metapropiedad de creación de instancias se debe agregar en la inicialización de una instancia de *InstanceCreationPropertiesCatalog*, y para el caso de una propiedad de recepción de mensajes se debe modificar la inicialización de una instancia de *MessageReceptionPropertiesCatalog*.

7) ***Definir qué clase de la jerarquía de definición de atributos es adecuada para crear una instancia de la nueva propiedad.***

Para lograr esto se debe modificar el mensaje de clase *relatedMetaProperties* de la subclase de *MetaPropertyAttributesDefinition* que corresponda.

En caso que no exista una clase que modele adecuadamente la definición de atributos necesarios para crear una instancia de la nueva propiedad, se debe agregar una nueva subclase para la definición de los atributos que correspondan.

En caso de que los editores de atributos existentes no sirvan para determinar el valor de los atributos, se deberá crear uno adecuado subclasificando *MetaPropertyAttributeEditor* y definiendo el método *specificSelectedValue*.

8) ***Para poder usar a la nueva metapropiedad, correr scripts[7] que realizan las inicializaciones necesarias para que se pueda trabajar con las metapropiedades definidas.***

9.2 Definición de una nueva estrategia de metapropiedades

En este caso lo que se quiere es reificar otro aspecto de la funcionalidad de una clase u objeto al que se le pueden asociar metapropiedades mediante la definición de una nueva estrategia de metapropiedades. Los pasos a seguir en esta situación son:

1) ***Crear una clase que modele la funcionalidad que se quiere reificar, como subclase de MetaPropertyPolicy.***

En esta subclase se deben implementar los métodos abstractos definidos por la superclase, como por ejemplo los siguiente métodos de instancia:

- ✓ *definedProperties*
- ✓ *basicPropertiesRelatedTo: aConcreteMetaProperty*
- ✓ *propertiesRelatedTo: aConcreteMetaProperty*

y el siguiente método de clase:

- ✓ *relatedMetaPropertyClass*[8].

Por otro lado, se debe definir en la nueva clase el método que tenga como responsabilidad el manejo de funcionalidad reificada mediante el forwarding de esta responsabilidad a la primera de las metapropiedades que tenga definida la estrategia. Por ejemplo, en el caso de la estrategia de creación de instancias con este propósito se definió el método *createNewInstance*, y en el caso de la estrategia de recepción de mensajes se definió el método *receiveMessage: aMessage*.

2) ***Definir qué objetos (y desde qué métodos) van a usar la nueva estrategia de metapropiedades.***

Por ejemplo, en el caso de la estrategia de creación de instancias, una clase al recibir el mensaje *new* le pasa a su estrategia de creación de instancias esta responsabilidad. En el caso de la estrategia de recepción de mensajes, un objeto al recibir el mensaje *receive: aMessage* le pasa a su estrategia de recepción de mensajes esta responsabilidad.

---

[7] Se trata del *script* al que se hace referencia en el instructivo de implementación (readMeMetaProperties.txt) que se adjunta con este informe.

[8] Aquí se debe determinar la clase abstracta que encabeza la jerarquía que modela las metapropiedades relacionadas con la nueva estrategia definida.

3) ***Definir la clase abstracta que modela las metapropiedades de la nueva estrategia definida.***

Se debe crear una subclase de *ConcreteMetaProperty* o de *ClassProperty* según se trate de metapropiedades que se puedan aplicar a cualquier objeto o sean específicas de clases.

En esta nueva clase se debe definir en forma abstracta el mensaje que tiene la responsabilidad de resolver la funcionalidad reificada con estas propiedades. Esto es necesario, porque la estrategia de metapropiedades delega en sus propiedades esta responsabilidad usando el mismo mensaje. Cada subclase de esta clase abstracta determinará la forma particular en que resuelve esta responsabilidad.

Además, se deben agregar y redefinir lo mensajes que se crean necesarios para que estas metapropiedades puedan llevar a cabo su responsabilidad, siempre y cuando se consideren comunes para toda propiedad de este grupo.

4) ***Crear una subclase de la clase abstracta que modela las nuevas metapropiedades, que represente a la metapropiedad básica de la funcionalidad reificada.***

Ésta debe ser definida cuando se crea una nueva estrategia de metapropiedades, ya que la misma requiere que esté definida la clase que modela su propiedad básica (recordar que al crearse una estrategia de metapropiedades ya nace con su propiedad básica).

Por ejemplo, en el caso de la estrategia de creación de instancias la metapropiedad básica es la que realiza al modo tradicional y común la creación de la nueva instancia de la clase owner. Y en el caso de una estrategia de recepción de mensajes, su metapropiedad básica es la que realiza concretamente el envío del mensaje al owner de la propiedad.

La forma en que esta nueva propiedad básica resuelve la funcionalidad que se está reificando, se determina implementando en forma concreta los métodos que para este propósito fueron definidos por su superclase en forma abstracta.

Además se deben implementar todos los métodos definidos como abstractos en sus superclases.

Por otro lado, en caso que se necesite tener una categoría específica para esta metapropiedad básica, también se debe agregar en las inicializaciones de categorías que se hacen en *MetaPropertyCategory*, la inicialización de una categoría asociada a esta metapropiedad, creando una nueva instancia de esta clase como se hace para las otras propiedades implementadas.

5) ***Crear una subclase de MetaPropertyCategoryRelationshipDefinition.***

En esta clase se deben definir las relaciones de orden y de incompatibilidad entre las categorías de metapropiedades de la nueva funcionalidad reificada.

Se deben definir concretamente los métodos abstractos definidos por su superclase, éstos son: *defineCategoryOrder* y *defineIncompatibleCategories*.

6) ***Para definir una nueva metapropiedad asociada a esta funcionalidad se debe proceder cómo se indicó para el caso de nuevas definiciones de metapropiedades para una funcionalidad reificada ya existente.***

10 Modificación de máquina virtual para reificar la recepción de mensajes

10.1 Máquina virtual de Smalltalk

Los dos componentes principales que se necesitan para correr un sistema Smalltalk [GR/86] son:

- una imagen virtual (o memoria de objetos) que está compuesta de todos los objetos del sistema
- una máquina virtual (o intérprete) que permite ejecutar la imagen en algún sistema operativo.

La máquina virtual ejecuta métodos compilados que se encuentran en la imagen cuyo código está expresado en secuencias de instrucciones de 8-bits, llamadas *bytecodes*. Las secuencias bytecodes son instrucciones que son interpretadas y ejecutadas por la máquina virtual.

La ventaja de esta técnica respecto a la generación directa de código para un procesador particular es que el mismo código de bytes, es decir la misma imagen, se puede ejecutar en cualquier plataforma para la que haya una implementación de máquina virtual.

10.2 Máquina virtual de Squeak

En Squeak [IKMWK], la implementación de la máquina virtual está realizada completamente en Smalltalk. Para conseguir mayor performance, un traductor, también escrito en Smalltalk, produce un programa C equivalente. Esto, sumado al hecho de que se trata de código abierto, permite generar nuevas versiones de la máquina virtual.

El intérprete está modelado con una sola clase: *Interpreter*. Además, una subclase de ésta, *InterpreterSimulator*, permite que se ejecute el intérprete desde un ambiente Squeak y de esta manera puede correr su propia imagen. Tener un intérprete que corra dentro de Smalltalk es muy importante para poder analizar el diseño e implementación de la máquina virtual.

La imagen de objetos está modelada con la clase: *ObjectMemory,* que se encarga de almacenar y obtener los objeto de la imagen.

Cada objeto en la memoria está precedido por un *header* con un formato de tamaño variable, que en la mayoría de los casos tiene una sola *word* de 32 bits. El tamaño de estos headers puede tener 1, 2 o 3 words, dependiendo del formato de la clase del objeto. Los dos *bits* finales de cada word indican el tipo de header del que depende el tamaño del mismo. Este

header tiene información acerca del tamaño del objeto, su identificación, su clase y formato, entre otras cosas.

10.3 Modificación del formato de los objetos

Como se explicó anteriormente, una estrategia de recepción de mensajes se puede determinar para un objeto en particular.

La forma más usual, directa y óptima para establecer la relación que se da entre un objeto y su estrategia de recepción de mensajes sería definiendo a la estrategia como un colaborador interno del objeto. Esto se lograría agregando una variable de instancia que represente a la estrategia en la definición de la clase *Object*, que es la clase de la que hereda directa o indirectamente toda clase que modele algún objeto. De esta manera a cualquier objeto se le podría definir una estrategia de recepción de mensajes.

El inconveniente con el que nos enfrentamos es que en Squeak (como en la mayoría de los dialectos de Smalltalk), por cuestiones implementativas, no se puede agregar directamente una variable de instancia en la clase *Object*, como en la definición de cualquier otra clase.

Esto es así porque hay determinadas clases optimizadas que no soportan que se les agregue variables de instancias, ya que la implementación de la máquina virtual supone que las instancias de estas clases tienen un tamaño fijo predeterminado. Éstas son clases que modelan objetos básicos y muy usados dentro del ambiente. Ejemplos de las mismas son: las clases que modelan los números (*Number*, *SmallInteger*, etc), el objeto indefinido (*Nil*), las colecciones básicas (*ArrayedCollection*, *SequenceableCollection*, etc), los objetos asociados al modelo de ejecución (*ContextPart*, *InstructionStream*, *MethodContext*, etc).

Frente a esta limitación, una alternativa es establecer las relaciones existente entre los objetos y sus estrategias en un objeto externo en el que queden reflejados estos mapeos. Por ejemplo, un diccionario, cuyas claves de acceso sean objetos cualesquiera y cuyos valores sean las estrategias que se las haya definido a estos objetos. Esta opción tiene la ventaja de la sencillez de su implementación. La desventaja es que no es muy óptima, teniendo en cuenta que cualquier objeto puede tener definida una estrategia y que cada vez que se envía un mensaje se debe acceder al diccionario mencionado.

La otra opción que analizamos fue la de modificar la implementación de la máquina virtual para lograr que en el mismo objeto se encuentre la definición de la estrategia de recepción de mensajes. La ventaja que presenta esta alternativa es que es mejor en velocidad que la anterior.

Finalmente, se decidió encarar la solución a este problema mediante la modificación de la implementación de la máquina virtual, ya que Squeak da facilidades para ello.

Luego de analizar cómo encarar estas modificaciones, llegamos a la conclusión de que la forma más directa de hacerlas era extendiendo el header que determina el formato de los objetos en la imagen virtual, de forma tal que en este header se pudiera determinar también la referencia a la estrategia de recepción de mensajes.

Esto implicó la realización de los siguientes pasos:

1) ***Definición del header de objetos extendido.***

Para poder llegar desde el header de cada objeto a su estrategia de recepción, se debe incluir en el mismo la dirección de memoria del objeto que representa a la estrategia. Para almacenar

una dirección basta con una word de 32 bits. Pero dado que en todas las words del header se utilizan los dos últimos bits para determinar el tipo del header y esto está asumido en toda la implementación de la máquina virtual, se determinó agregar 2 words al header para referenciar a la estrategia de recepción. De esta manera, los 30 primeros bits de la dirección se guardan en la primera word y los 2 bits restantes en la otra word. Y para mantener la compatibilidad con las otras words ya existentes en el header, los dos últimos bits de cada word se utilizan para indicar el tipo de header.

Se determinó que las words agregadas sean la cabecera del header, es decir que sean las dos primeras.

2) ***Generar una imagen con el header extendido.***

Esto se pudo hacer mediante la funcionalidad provista por la clase *SystemTracer2*, que permite convertir imágenes. Es decir, dada una imagen de entrada, recorre sus objetos, los modifica según se indica y genera una imagen de salida con los objetos modificados.

Para lograr la generación de la imagen requerida, se creó una subclase de la clase mencionada, redefiniendo los métodos necesarios para hacer que en la imagen de salida los objetos quedaran con el header extendido.

En la imagen inicial generada se determinó que la estrategia de todos los objetos quedara indefinida, de esta manera en las nuevas words de todos los headers se almacenó la dirección del objeto *nil*.

3) ***Modificar la implementación de la máquina virtual para que soporten el header extendido.***

Se modificó la implementación de la clase *ObjectMemory* para que se tuviera en cuenta que el header de los objetos ahora tenía 2 words más. Así se tuvieron que hacer modificaciones en:

- Métodos que calculan tamaño de headers, para que se consideren las dos words agregadas.
- Métodos que se usan para alocar nuevos objetos, así como los relacionados con la copia de objetos, para que reserven 2 words más de memoria por objeto.
- Métodos relacionados con la creación de la imagen de objetos en memoria a partir de un archivo donde se guardó una imagen, para ajustar también las direcciones de las estrategias referenciadas desde el header, teniendo en cuenta la nueva dirección de memoria de inicio asignada para la imagen respecto a la dirección de inicio anterior indicada en el archivo.
- Métodos que se encargan de mover objetos y remapear direcciones de memoria por cuestiones de *garbage collection,* para que también se modifiquen las referencias a estrategias de recepción desde el header cuando se cambian las direcciones de memoria de estos objetos.

4) ***Agregar primitivas para acceder a las estrategias de recepción de mensajes de los objetos.***

Para poder acceder desde el código Smalltalk a las estrategias de recepción de mensajes definidas para los objetos fue necesario agregar primitivas en la implementación de la

máquina virtual. Las primitivas son funciones de la máquina virtual que pueden ser invocadas desde el código Smalltalk.

Para ello definimos nuevos métodos en la clase *Interpreter*:

- para que a partir de las direcciones almacenadas en las words agregadas en los headers, se obtengan los objetos que representan a las estrategias de recepción de mensajes
- para que cuando se defina una nueva estrategia para un objeto, se determine la dirección de la misma y se almacene en las correspondientes words del header del objeto.

5) ***Generar el nuevo programa C que representa a la máquina virtual modificada.***

Esto se hizo mediante las funcionalidades provistas por las clases *VMMakerTool* y *VMMaker* que se encuentran definidas en el ambiente de Squeak. *VMMakerTool* modela una herramienta e interface de usuario para facilitar el uso de la funcionalidad provista por un *vMMaker*, que es quien tiene la responsabilidad de generar el código C a partir de las definiciones de las clases de Smalltalk que modelan la máquina virtual.

Luego de haber obtenido el código fuente en C, éste se compiló y vinculó para obtener un programa ejecutable correspondiente a la nueva versión de la máquina virtual.

6) ***Levantar y ejecutar la nueva imagen obtenida con la nueva máquina virtual.***

De esta manera se puede trabajar con una imagen en la cual a cualquier objeto se le puede definir una estrategia de recepción de mensajes, y acceder a las estrategias definidas.

10.4 Modificación de implementación de envío y recepción de mensajes

Smalltalk separa la acción de enviar un mensaje de la de activar o ejecutar el método correspondiente, y en medio está el algoritmo de *method lookup*. Dicho algoritmo está implementado a nivel de la máquina virtual por cuestiones de performance. La máquina virtual está todo el tiempo interpretando métodos compilados. Cuando detecta el envío de un mensaje, realiza el method lookup e invoca al método correspondiente.

De esta manera, la estrategia de recepción de mensajes se ignoraría cuando un objeto recibe un mensaje. Se debió modificar la implementación de la máquina virtual para que al detectar el envío de un mensaje se delegue la responsabilidad de manejar esto a la estrategia definida para el objeto que recibe el mensaje.

Si no existiera la posibilidad de modificar la máquina virtual se podrían haber seguido otros caminos para controlar la recepción de mensajes, como por ejemplo los descriptos en [BFJR]. Estas técnicas principalmente ofrecen soluciones implementativas a este problema y su principal desventaja es que ninguna de ellas está orientada a soportar de manera transparente y natural un modelo que reifique la recepción de mensajes.

Las modificaciones que se hicieron fueron las siguientes:

- Al detectar un envío de mensaje, la máquina virtual, en lugar de invocar al método correspondiente del objeto, invoca al método *receive:* (que sabe responder cualquier objeto) pasándole como parámetro el mensaje original[9].

- El método *receive*: le forwardea esta responsabilidad a la estrategia de recepción de mensajes del objeto, enviándole el mensaje *receiveMessage*:, pasándole como parámetro el mensaje que recibió.

A partir de aquí la estrategia de recepción de mensajes resuelve su responsabilidad de la forma en que ya fue descripta anteriormente.

Se debe observar que pese a que se modifica el funcionamiento del envío y recepción de mensajes, desde el punto de vista de los objetos de la aplicación todo es transparente. La forma en que envían y reciben los mensajes es independiente de todo lo que se realice antes o después.

Para realizar estas modificaciones de la máquina virtual se tomaron como base las modificaciones que con el mismo propósito se hicieron en [AI/02], trabajo en el cual también se encara la reificación de recepción de mensajes.

10.5 Límites de las modificaciones para reificar la recepción de mensajes

Los problemas o limitaciones de implementación relacionadas con las modificaciones de la máquina virtual para lograr la reificación de la recepción de mensajes que no se resolvieron son:

- A los objetos de la clase *SmallInteger* no se le puede determinar estrategia. Se produce un error de primitiva. Esto es porque no tienen header.

- Se produce un error de primitiva al intentar grabar un proyecto de Squeak.

- Cuando se abre el debugger y se hace *step* o *send* sobre el envío de un mensaje que tiene asociada una propiedad de recepción, ésta no se detecta. Esto sucede porque en estas circunstancia no se envían los mensajes de forma standard.

- *Garbage collection de estrategias de recepción de mensajes*. Una de las desventajas de no haber implementado a la estrategia de recepción de mensajes de un objeto como una variable de instancia más, es que estas estrategias sólo están referenciadas desde el header de los objetos. Desde el punto de vista del algoritmo de garbage collection, se considera que no tienen referencias. Luego, pueden ser eliminadas de la memoria y así pasar a referenciar desde el header a un objeto que no representa a la estrategia definida o apuntar a una zona inválida de memoria.

Ante esto, lo que se debe hacer es modificar la implementación del algoritmo de garbage collection. Aunque esto es viable, se resolvió dejarlo como un trabajo futuro, ya que no hacía a la esencia de los objetivos de este trabajo. Para suplir esta falencia, se implementó un

[9] Esto sucede siempre y cuando el objeto no sea el objeto indefinido (*nil*)

diccionario en el que se mantienen referencias a las estrategias de recepción de mensajes creadas mientras éstas estén definidas como las estrategias de algún objeto.

Una consecuencia de esto es que los objetos referenciados desde el diccionario con el propósito mencionado, no son considerados por el garbage collector cuando no tienen referencias reales. Para solucionarlo, se extendió el algoritmo de garbage collection para que se detecten las estrategias en estas condiciones para eliminarlas por el garbage collector del diccionario y que así puedan ser consideradas como objetos no referenciados y recolectadas.

11 Conclusiones

En este trabajo se identificaron, analizaron y categorizaron distintos aspectos del comportamiento de clases que pueden ser considerados como propiedades de clases o metapropiedades.

Se pudo concluir que los conceptos asociados a este problema conforman un dominio en sí mismo que puede ser modelado en forma independiente del dominio de metaclases. Se consideró al dominio de metapropiedades ortogonal al de metaclases ya que pueden tratarse y combinarse sin interferencias.

Se analizaron modelos existentes de ambientes o lenguajes en los que se tratan a las clases como objetos y en los cuales se maneja el concepto de propiedad de clase, y se justificó por qué se considera que en ninguno de ellos se encuentra realmente reificado este dominio y que carecen de un modelo integral de metapropiedades.

Se justificó la importancia de contar con un modelo de objetos del dominio de metapropiedades que refleje la forma en que lo interpretamos conceptualmente y que permita la asignación dinámica de estas propiedades.

Se probó la factibilidad de un modelo con las características propuestas, realizando el diseño e implementación del mismo en el ambiente Squeak, que gracias a su características reflexivas favoreció el logro de los objetivos.

La implementación del modelo resultante cuenta con las siguientes características:

- **Permite Asignación y combinación dinámica de propiedades.** Se logra por la existencia de objetos que modelan las propiedades simples y el uso del mecanismo de composición en contraposición al uso de herencia, tanto para asignar propiedades como para combinarlas en una clase. De esta manera, contando con un conjunto de propiedades básicas y ensamblándolas de la manera adecuada, se consigue que una clase cuente con las propiedades buscadas.
- **Refleja la existencia de distintas categorías de metapropiedades.** Esto se logra mediante la reificación de las distintas funcionalidades de una clase, como ser la creación de instancias y la recepción de mensajes, a través de los objetos que se definieron como estrategias de metapropiedades.
- **El diseño y asignación de propiedades se realizan en forma transparente e independiente de un dominio particular.** Además, no se mezcla el código que implementa la funcionalidad del dominio particular con el que implementa las metapropiedades.
- **Facilita el reuso de propiedades.** La lógica de una determinada propiedad está expresada en uno y sólo un lugar, pudiendo ser reusada.
- **Soluciona el problema de propagación de propiedades de clase.**

- **Asegura la compatibilidad de comunicación entre niveles.**
- **Asegura dinámicamente la validez de propiedades.**
- **Permite la definición de metapropiedades a nivel de instancias.** Dependiendo del tipo de metapropiedad, ofrece la flexibilidad de permitir asignar metapropiedades para determinadas instancias de una clase y no para todo el conjunto, en los casos que así se desee. Esto se aplica para la recepción de mensajes, ya que se detectó que la misma se podía definir a nivel de objeto. Además se considera adecuado que sea responsabilidad propia de una instancia determinar cuál es su estrategia de recepción de mensajes ya que es ella misma quien los recibe.

  Se pudo aprovechar las capacidades de Squeak que permiten modificar la máquina virtual para una mejor implementación de la reificación propuesta. Las modificaciones que debieron hacerse a este nivel estuvieron relacionadas con el formato de los objetos en memoria y con el manejo de envío y la recepción de mensajes.

Además, se desarrollaron herramientas gráficas para facilitar el uso del modelo implementado en el ambiente de Squeak. Estas herramientas permiten tanto asignar nuevas propiedades a las estrategias de metapropiedades de clases u objetos cualesquiera, como inspeccionar y editar las que ya tienen asignadas.

La implementación del modelo resultante constituye un framework de metapropiedades. En el modelo están reificados en forma genérica tanto el mecanismo de asignación como el de aplicación de metapropiedades. Esto facilita su extensión para implementar otras metapropiedades que no fueron desarrolladas en este trabajo. Las abstracciones modeladas permiten crear nuevas propiedades o estrategias definiendo solamente lo específico de las mismas sin tener que volver a definir lo que es común a otras propiedades o estrategias.

Por eso consideramos que la implementación del modelo resultante constituye un ambiente que facilita la investigación sobre qué otros aspectos de la funcionalidad de una clase o de objetos cualesquiera pueden ser reificados mediante estrategias de metapropiedades y permite probar la factibilidad de su implementación.

12 Trabajo futuro

Algunas de las mejoras o extensiones que consideramos se pueden realizar sobre este trabajo son:

- Detectar qué otras categorías de metapropiedades se pueden detectar, teniendo en cuenta a qué otros aspectos de la funcionalidad de una clase o de los objetos en general se les puede asociar metapropiedades y pueden reificarse como estrategias de metapropiedades. Tratar de implementarlas a partir del modelo existente.

Como ejemplo, podemos mencionar una categoría de metapropiedades que se detectó pero no se implementó: las propiedades relacionadas con la determinación de relaciones de herencia, como la propiedad *final* que determina que una clase no puede tener subclases.

Esto serviría para verificar tanto la potencialidad como la flexibilidad y facilidad de extensión del modelo implementado.

- Analizar e implementar otras propiedades pertenecientes a categorías ya existentes. Como propiedades interesantes que no se implementaron, se puede mencionar a las que verifican pre y post condiciones que se deben cumplir antes o después de ejecutarse un método y que pertenecen al conjunto de metapropiedades que se pueden definir para una estrategia de recepción de mensajes.

- Hacer más configurables a las metapropiedades existentes. Por ejemplo:
  - ✓ Al asignar la propiedad *halt*, poder determinar en qué línea del código del método asociado se quiere pausar la ejecución.
  - ✓ Al agregar un *trace*, poder elegir dónde volcar la información de la traza.
  - ✓ Al asignar la *propiedad de notificación de creación de una instancia*, poder configurarla con el método o bloque que se debe ejecutar ante la ocurrencia de la creación.

- Resolver las limitaciones mencionadas resultantes de las modificaciones de la máquina virtual para reificar la recepción de mensajes.

- Analizar el overhead (respecto a performance) en envío y recepción de mensajes impuesto por las modificaciones de máquina virtual para reificar la recepción de mensajes.

- Implementar otras extensiones a los chequeos de consistencia que se realizan tanto para garantizar la consistencia de las definiciones determinadas al extender el modelo para crear una nueva estrategia o propiedad, como para asegurar dinámicamente la validez de propiedades asignadas. Por ejemplo:
  - ✓ Al definirse relaciones de incompatibilidad entre dos categorías de propiedades, validar que no existan estrategias que contengan a propiedades de ambas categorías. En forma análoga, al definir relaciones de orden entre categorías de

metapropiedades, validar que todas las estrategias definidas respeten las nuevas relaciones de orden.

- ✓ Cuando se borra un método, controlar si existe una propiedad de recepción de mensajes definida para ese selector.
- ✓ Verificar por cada grupo de metapropiedades que exista una y solo una categoría definida como básica.

- Complementar y mejorar las herramientas desarrolladas. Tenemos herramientas para asignar nuevas propiedades a las distintas estrategias de un objeto e inspeccionar propiedades ya asignadas. No se desarrollaron herramientas para facilitar o guiar la definición de nuevas metapropiedades o estrategias, es decir, para facilitar la extensión del framework. Se propone analizar la conveniencia del desarrollo de estas herramientas e implementarlas en caso que se crea conveniente.

- Resolver limitación de propiedad de recepción de mensajes *halt* relacionada con la imposibilidad de poder definirla para selectores binarios.

- Explorar la posibilidad de implementar *Aspectos* [KIC/97] usando metapropiedades.

13 Bibliografía y referencias

[AI/02]. Daniel Altman, Alejandro Isacovich. *"Trabajo Práctico: Reificación de la recepción de mensajes"*. Desarrollado en la materia Diseño Avanzado con Objetos en la Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, 2002

[BC/89] Jean Pierre Briot, Pierre Cointe. *"Programming wiith Explicit Metaclasses in Smalltalk-80"*. OOSPLA 1989

[BC/90] Gilad Bracha, William Cook. *"Mixinbased Inheritance"*. OOPSLA 1990

[BFJR] John Brant, Brian Foote, Ralph E. Johnson, Donald Roberts. *"Wrappers to the Rescue"*. University of Illinois at Urbana Champaign

[BLR/98] Noury M. N. Bouraqady-Saadmi, Thomas Ledoux, Fred Rivard. *"Safe Metaclass Programming"*. OOSPLA 1998

[Bou/03] Noury Bouraqadi. *"Metaclass Composition Using Mixin-Based Inheritance"*. Esug 2003

[BS/00] M. N. Bouraqadi-Saâdani. *"Concern Oriented Programming using Reflection"*. OOSPLA 2000

[BS/02] M. N. Bouraqadi-Saâdani. *"MetaclassTalk: a Testbed for Exploring Programming Paradigms"*. Ecole des Mines de Douai, France, 2002

[BS/03-1] M. N. Bouraqadi-Saâdani. *"MetaclassTalk & Metaclass Composition"*. Ecole des Mines de Douai, France, 2003

[BS/03-2] M. N. Bouraqadi-Saâdani. *"Safe Metaclass Composition Using Mixin-Based Inheritance"*. Ecole des Mines de Douai, France, 2002

[BS/99] M. N. Bouraqadi-Saâdani. *"Un MOP Smalltalk pour l'étude de la composition et de la compatibilité des métaclasses. Application `a la programmation par aspects"*. Tesis de Doctorado, Universidad de Nantes, Nantes, Francia, 1999

[Cas/00] Gabriel P. Casarini. *"Towards Transparent Strong Mobility using a Reflective Smalltalk"*. Tesis de Grado, Vrije Universiteit Brussel, Faculty of Sciences, Bélgica, 2000

[Coi/87] Pierre Cointe. *"Metaclasses are Fist Class: The ObjVlisp Model"*. OOSPLA 1987

[DSW/04] Stéphane Ducasse, Nathanael Scharli, Roel Wuyts. *"Uniform and Safe Metaclass Composition"*. ESUG 2004

[Duc/00] Dr. Stéphane Ducasse.*"Reflective Programming and Open Implementations"*. University of Bern, 2000/2001

[Duc/99] Stéphane Ducasse.*"Evaluating Message Passing Control Techniques in Smalltalk"*. Journal of Object Oriented Programming, Junio 1999

[FD] Ira R. Forman, Scott H. Danforth. "*Inheritance of Metaclass Constraints in SOM"*. IBM Object Technology Products

[FD/98] Ira R. Forman, Scott H. Danforth. *"Putting Metaclasses to Work"*. First Edition, Addison Wesley Longman, Inc, 1998

[GHJV/95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. "*Design Patterns: Elements of Reusable Object-Oriented Software"*. Addison-Wesley, 1995

[GR/86] Adele Goldberg, David Robson. *"Smalltalk-80: The Language"*. Second Edition, Addison-Wesley Publishing Company, 1986

[IKMWK] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, Alan Kay. "*Back to the Future. The Story of Squeak, A Practical Smalltalk Written in Itself*". Apple Computer

[KIC/97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier y John Irwin. "*Aspect-Oriented Programming"*. http://www.parc.xerox.com/spl/projects/aop/. Xerox PARC, 1997

[LeCo] Thomas Ledoux,Pierre Cointe. *"Design of Class Libraries"*.Ecole des Mines de Nantes, Francia

[Met] http://csl.ensm-douai.fr/MetaclassTalk

[MG/87] Linda G. DeMichiel, Richard P. Gabriel. *"The Common Lisp Object System: An Overview"***.** ECOOP 1987

[Mic] Linda G. DeMichiel. *"The Common Lisp Object System"*

[Riv] Fred Rivard. *"Smalltalk: A Reflective Language"*

[SDNB/03] Nathanael Scharli, Stéphane Ducasse, Oscar Nierstrasz, Andrew P. Black. *"Traits: Composable Units of Behaviour"*. ECOOP'2003

[SLU/92] L. Stein, H. Lieberman, D. Ungar. *"A Shared View of Sharing: Treaty of Orlando"*

[Woo/97] Bobby Woolf. "*Polimorphic Hierarchy"*. Smalltalk Report, January1997

[WWW/90] Rebecca Wirfs Brock, Brian Wilkerson, Lauren Wiener. *"Designing Object-Oriented Software"*. Prentice Hall, 1990

14 Apéndice A – Notación gráfica utilizada

En esta sección se detalla la notación gráfica utilizada para los diagramas de diseño expuestos en este trabajo.

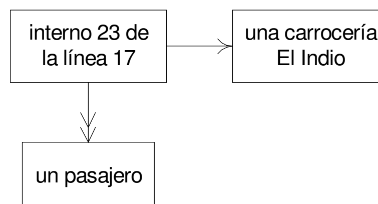
La notación utilizada está basada en la propuesta en la materia de *"Programación Orientada a Objetos"* que se dicta en la misma Facultad, en la que se encuentra contextualizado el presente trabajo.

A continuación se explican los distintos diagramas utilizados.

14.1 Diagrama de instancias (o de objetos)

Representa una vista estática del modelo, en él se incluyen los objetos intervinientes y las relaciones de colaboración entre ellos. No es indispensable representar todas las posibles relaciones de colaboración entre ellos, solamente las necesarias para transmitir lo que se desea. Si bien puede utilizarse para modelar de modo genérico, esto ocurre rara vez. La intención del diagrama es representar una situación particular, un ejemplo puntual.

Veamos un ejemplo:



- Cada "cajita" representa un objeto. Esto es, una instancia particular y única en el universo, y no una forma genérica de referirse a objetos como podría ser "Colectivo".

- Las flechas indican una relación estructural entre los objetos. Mas específicamente, indica que los objetos pueden colaborar, o sea, enviarse mensajes.

- La flecha simple hace referencia a una instancia particular, mientras que la doble indica una colección de objetos. Esto es un grupo de objetos, no necesariamente homogéneos. La relación indica conocimiento del grupo, algo distinto a conocer cada integrante por separado.

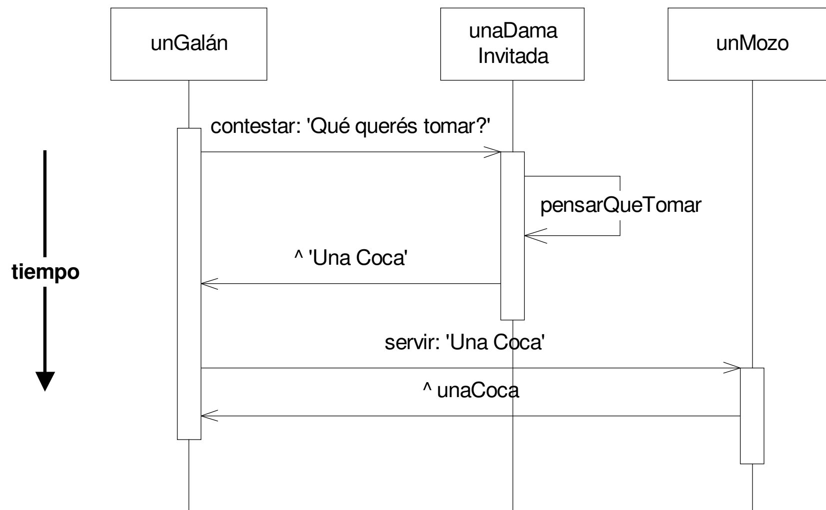
- Puede colocarse una etiqueta sobre la flecha, indicando el nombre con que el objeto origen conoce al objeto destino, si esto es importante para distinguirla de otras o para aclarar el significado de esta relación.

14.2 Diagrama de secuencia (o de colaboración)

Representa una vista dinámica del modelo, incluyendo la secuencia de envío de mensajes (colaboraciones), que tiene lugar entre algunos objetos del modelo para llevar a cabo una tarea. Al igual que en el diagrama de instancias, no es indispensable representar todos los

envíos de mensajes, solamente los necesarios para transmitir lo que se desea. La intención del diagrama es representar una situación particular, un ejemplo puntual.

Veamos un ejemplo:

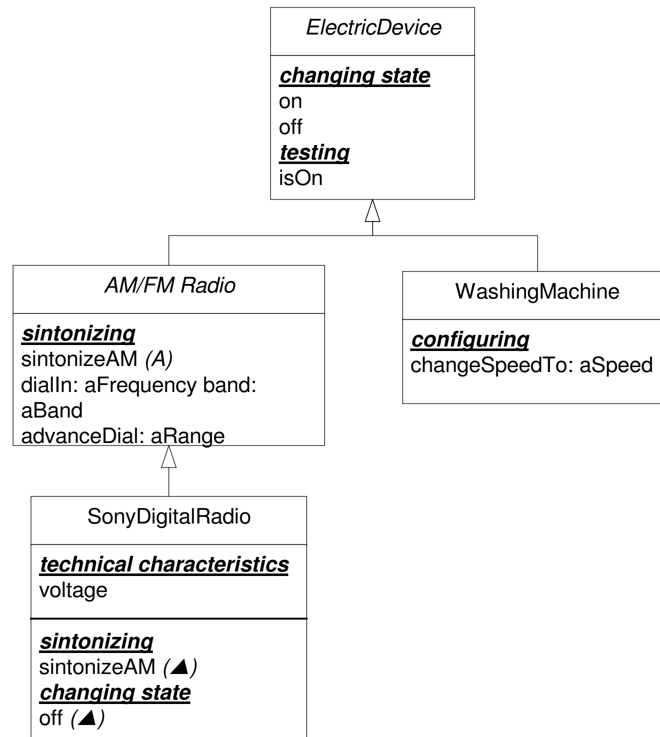


- Cada cajita, al igual que en el diagrama de objetos, representa un objeto.
- El sentido del avance del tiempo es de arriba hacia abajo.
- Cada flecha representa el envío de un mensaje de un objeto al otro, o el retorno de un objeto, que se denota mediante el símbolo ^.
- Un mensaje enviado por un objeto a sí mismo se indica con una flecha que baja y vuelve a la misma columna.

14.3 Diagrama de Clases

Representa las clases intervinientes en un modelo y el vínculo de herencia existente entre ellas. El diagrama incluye el protocolo que poseen los objetos instancias de las mismas.

Veamos un ejemplo:



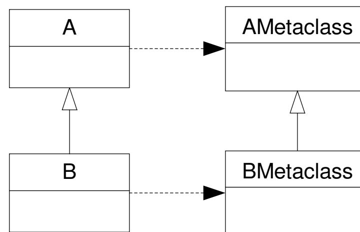
- Cada cajita representa una clase.
- La flecha indica relación de herencia y va desde la subclase hacia la superclase.
- El nombre de la clase en itálica indica que la misma es abstracta.
- Los métodos definidos en una clase abstracta, pero no implementados en la misma (ya que corresponde que esto se haga en cada subclase en forma específica), se indican agregando al final de la línea: *(A)*.
- El hecho de que en una subclase reaparezca un mensaje que estaba en una de sus superclases indica que la clase redefine el método que lo implementa, y además para resaltar esto se agrega al final de la línea el indicador: *(▲)*.
- No se incluyen los colaboradores internos, pero se indican los métodos de acceso a los mismos. De éstos, solamente se usan getters en representación de ambos.
- Los métodos de clase van arriba de los de instancia, separados por una línea.
- Los métodos se encuentran agrupados por categoría. La categoría se indica haciendo que el nombre de la misma (en negrita y subrayado) encabece a la lista de los métodos que pertenecen a la misma. La determinación de categorías no es esencial para definir un modelo de objetos, pero ayuda a la hora de transmitir un diseño ya que contribuye a revelar la intención de los métodos.
- Puede agregarse una línea de comentario bajo algún método utilizando comillas dobles (“”), o en su defecto pseudo código que ayude a determinar la intención del método.

14.4 Diagramas de clases y metaclases

Estos diagramas se utilizan para reflejar las relaciones entre las clases y sus metaclases.

Aquí se exhiben tanto relaciones de herencia entre clases como de instanciación entre las clases y sus metaclases.

Veamos un ejemplo:



- Las relaciones de herencia se reflejan de la misma manera que en los diagramas de clases.
- Las relaciones de instanciación se indican mediante flechas punteadas que van desde la instancia a su metaclase.
- Aunque en estos diagramas se reflejan relaciones entre instancias y sus clases, como todos los objetos intervinientes son clases, todas las cajas intervinientes tienen el formato de las que representan a las clases.