Departamento de Informática
Universidad de Valladolid
Valladolid – España

Definición de un soporte estructural para abordar el problema de la independencia del lenguaje en la definición de refactorizaciones

Carlos López Nozal, Yania Crespo González-Carvajal
Departamento de Informática, Universidad de Valladolid, España
clopezno@ubu.es
yania@infor.uva.es

Resumen En el presente trabajo se presenta un soporte para representar la información de un sistema software orientado a objetos con genericidad. Para ello se modelan las características comunes de las distintas familias lenguajes de programación a través de abstracciones definidas en un lenguaje modelo MOON. Las características variables serán soportadas introduciendo en las abstracciones del núcleo puntos de extensión que tendrán definiciones diferentes en familias distintas de lenguajes. El objetivo perseguido en la definición de esta solución de soporte es posibilitar la definición de refactorizaciones sobre las abstracciones del núcleo, consiguiendo así una cierta independencia del lenguaje en la definición de refactorizaciones.

Informe Técnico No DI- 2003 - 3

1 Introducción

Refactorizar se define como el proceso de cambiar un sistema software para mejorar su estructura interna, preservando el comportamiento externo de código. El proceso de refactorización busca mejorar el diseño de código una vez que este ha sido escrito, de esta forma se puede hacer frente al mantenimiento de un código de calidad en las continuas reestructuraciones al que se expone en la etapa de mantenimiento.

Actualmente, las refactorizaciones, se muestran como un área de interés: integrándose como etapa en recientes métodos de desarrollo software como Programación Extrema (Extreme Programming) [Beck 1999], aumentando y mejorando los catálogos de refactorizaciones [Fowler et al., 1999], incorporando las refactorizaciones como una funcionalidad en entornos integrados de desarrollo del software.

Los componentes clave que deben poseer las herramientas capaces de asistir el proceso de refactorización son: un repositorio donde se acumule una base del conocimiento sobre el sistema software candidato a refactorizar, y un motor de refactorizaciones donde se pueda almacenar la definición de las refactorizaciones soportadas. La arquitectura del repositorio requiere un modelo (también llamado metamodelo) donde se establezca la estructura e identifique los elementos del sistema software que pueden ser transformados a través de las refactorizaciones. En el caso de un sistema software orientado a objetos estos elementos se corresponden con: clases, métodos, atributos.... Desafortunadamente las transformaciones de código suelen tener una dependencia sobre el lenguaje de programación utilizado para desarrollar el sistema software que impide que una refactorización pueda ser definida independiente del lenguaje.

En este sentido en [Crespo 2001] se presenta un lenguaje modelo MOON que contiene el suficiente poder expresivo para representar las construcciones abstractas necesarias en la definición y análisis de refactorizaciones y avanzar hacia la posibilidad de establecer una cierta independencia del lenguaje en este campo. La esencia del lenguaje modelo radica en ser minimal, recogiendo únicamente las construcciones necesarias de los lenguajes para refactorizar. También se intenta que MOON sea lo más general posible para dar cabida a una amplia familia de lenguajes. La adaptación de los distintos lenguajes de programación pasa por incorporar sus características concretas a través de extensiones de las abstracciones definidas en MOON. Los beneficios obtenidos de esta traducción es poder definir una la refactorización de manera general sobre el lenguaje modelo.

La adaptación de las construcciones de los distintos lenguajes pasa por estudiar las características concretas de estos y analizar su incorporación sobre el lenguaje minimal. El estudio de una adaptación sintáctica sobre el lenguaje Eiffel es incluido en [Crespo 2000]. El estudio de más lenguajes de programación permitirán validar el lenguaje modelo respecto al conjunto de construcciones relevantes para las refactorizaciones que en él se representan y avanzar en definiciones independientes del lenguaje.

El presente trabajo tiene como objetivo definir una estructura para soportar las abstracciones sintácticas definidas en el lenguaje modelo (sección 2). Una vez definido el soporte común a las construcciones comunes de los lenguajes orientados a objetos estáticamente tipados se pretende definir los puntos de extensión para incorporar características concretas del los lenguaje

Java (sección 3) y su extensión para soportar tipos genéricos GJ (Generic Java) (sección 3.1).

1.1 Trabajos relacionados

En [Demeyer et al., 1999] se presenta FAMIX como un metamodelo para almacenar información en un repositorio que permita la integración de diferentes entornos de desarrollo de software con soporte para la refactorización. Por otro lado se presenta un estudio de factibilidad a partir del análisis de dos lenguajes, Java y Smalltalk, para validar su propuesta de metamodelo que abstrae las características necesarias para realizar refactorizaciones.

El objetivo perseguido en la propuesta [Demeyer et al., 1999] es la definición de una refactorización de manera general sobre los elementos del metamodelo FAMIX. La diferenciación principal sobre este trabajo es que el modelo MOON se centra en las características de los lenguajes estáticamente tipados y con genericidad , haciendo especial énfasis en la consideración de características avanzadas en cuanto a herencia y a genericidad.

Hasta donde se conoce, éste es el único trabajo que se ha publicado con un propósito final igual: modelar de las características comunes de los lenguajes de programación para definir las refactorizaciones de manera universal.

Existen varios trabajos relacionados respecto a la representación de las características de los sistemas software orientados a objetos a través de abstracciones, el modelo más destacado es el metamodelo de UML [OMG, 2003]. UML es un lenguaje de modelado, por esto no necesita representar toda la información contenida en el código fuente de un programa, este es el caso de las instrucciones contenidas en los métodos. Algunas refactorizaciones de código necesitan tratar las instrucciones asociadas a un método para poder ser definidas. Además en UML se tiene un pobre modelado de la genericidad donde no se recoge las distintas formas de acotación de los parámetros formales.

En UML se pueden definir transformaciones de modelos y no de código. En [Judson et al, 2003] se describe una técnica que describe el soporte de la definición de un tipo de transformaciones denominadas refactorizaciones de modelos (model refactoring) indicando como este tipo de transformaciones pueden ser definidas a través del metamodelo de UML. Esta familia de transformaciones se basan en incorporar patrones de diseño [Gamma et al, 2003] a los modelos fuentes. Este tipo de estudios incorpora a UML un lenguaje candidato a considerar en la validación del lenguaje modelo definiendo la su transformación cómo sus posibles extensiones.

Existen otros trabajos [Mens et al, 2002] de modelado de código a nivel de refactorizaciones en los cuales se utilizan grafos para representar los aspectos del código fuente que pudiera ser refactorizado y expresa las transformaciones como reescritura de reglas del grafo.

En la clasificación de reestructuraciones propuestas en [Crespo 2000], existe una categoría denominada método donde se distinguen dos tipos de refactorizaciones según el método en que se basan para desencadenar las transformaciones. Las transformaciones de inferencia desencadenan la reestructuración de manera automática al detectarse el cumplimiento de ciertas condiciones. Al definir un metamodelo independiente del lenguaje se abre el camino para poder definir un conjunto de métricas, independientes del lenguaje, que sirvan para poder detectar las condiciones de lanzamiento

de la refactorización. En este sentido existen trabajos de definición de métricas sobre FAMIX [Lanza & Ducasse, 2002], y sobre el metamodelo basado en grafos [Mens & Lanza 2002].

2 Modelado conceptual MOON para capturar la información de Sistemas Software

En esta sección se parte de las reglas sintácticas y semánticas de un lenguaje modelo MOON definido en [Crespo 2000] y presentado en [Crespo 2001]. El objetivo perseguido en esta sección es conceptualizar el lenguaje MOON a través de un conjunto de clases y sus relaciones que permitan capturar la información contenida en un sistema software. Además se pretende definir el conjunto de métodos selectores y modificadores sobre los atributos identificados en el modelo así como los métodos que permitan navegar por las relaciones definidas sobre él.

El análisis a posteriori de distintos lenguajes de programación O.O. permitirá mostrar la capacidad de extensión de MOON para poder adaptarse a las características concretas de distintos lenguajes OO estáticamente tipados y con genericidad. En este sentido previendo una futura adaptación a Java se mostraran ejemplos de código tanto en Java como en MOON sin entrar en características concretas del lenguaje Java (static, clases internas, final, syncronized, control de acceso a atributos y métodos...) que serán tratadas en las siguientes secciones.

Este es el objetivo previo para poder establecer la funcionalidad soportada por un marco (framework) completo de trabajo que persigue definir un conjunto de refactorizaciones independientes del lenguaje. La solución basada en framework necesita definir unos puntos de extensión que vendrán dados por las características concretas de los lenguajes analizados.

2.1 Guía de estilo en los diagramas presentados

- Para representar de forma gráfica las clases que aparecen en más de un diagrama de clases, éstas se han coloreado de azul.
- La representación gráfica de los objetos presentados en los distintos diagramas de objetos se basa en la notación UML (*nombreobjeto:nombreclase*). Para mostrar las múltiples clases de las cuales un objeto es una instancia se utiliza una lista de nombres de clase separadas por dos puntos (*:*). Los nombres de las clases deben corresponderse con múltiples jerarquías de herencia a las cuales puede pertenecer el objeto representado. Sólo se permite una implementación de clase, la correspondiente al último nombre de la lista, pero se permiten múltiples tipos.

  La Figura 1 muestra la representación de un objeto cuyo nombre es (A,a), su implementación de clase es ATT_DEC y sus posibles tipos son: PROPERTY, ENTITY, SIGNATURE_ENTITY, ATT_DEC.

Figura 1 Representación de objetos

- Las clases conceptuales identificadas para representar las características del lenguaje MOON han sido agrupadas conforme a

la estructura mostrada en la Figura 2. Cada clase presentada en los distintos diagramas muestran esta información indicando el nombre del paquete debajo del nombre de la clases.

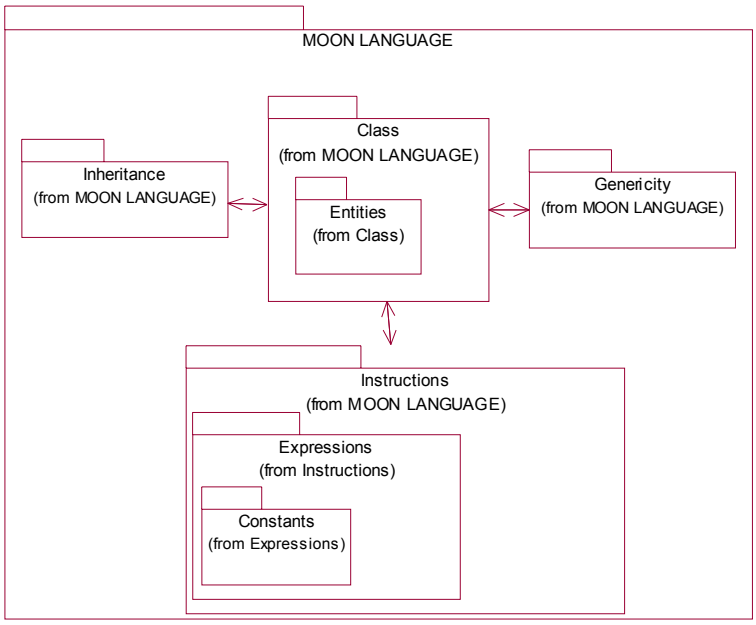


Figura 2 Agrupación de clases en paquetes

- Se ha tomado como conjunto de tipos básicos utilizados en la descripción de los métodos los predefinidos en OCL [OMG, 2003]. Al igual que los tipos básicos se utilizan las distintas colecciones definidas en OCL añadiéndolas propiedades genéricas para mejorar la expresividad de las mismas. Se trabaja fundamentalmente con el tipo abstracto *Collection* y con la colección *Sequence* que se caracteriza por ser una colección cuyos elementos están ordenados.
- Se ha utilizado el lenguaje OCL para definir invariantes sobre los modelos presentados.

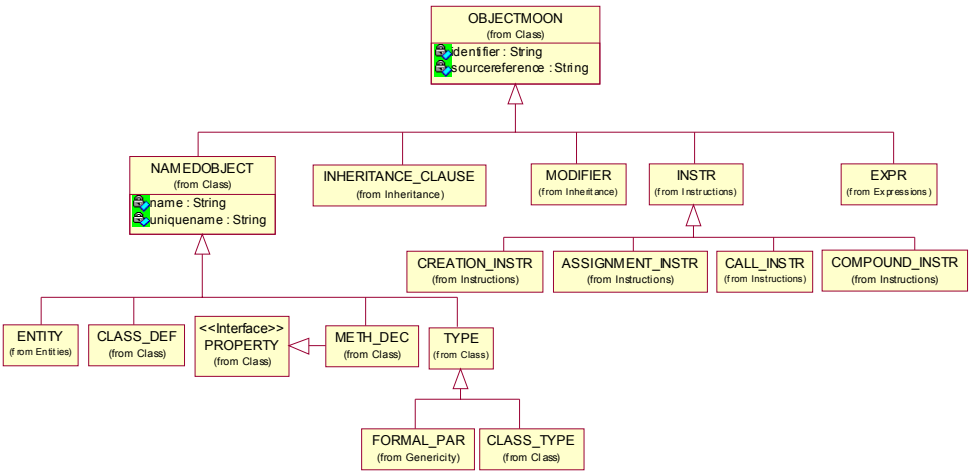
2.2 Descripción del modelo conceptual MOON

En esta sección se presenta la descripción conceptual de clases del lenguaje MOON. Se comienza exponiendo una jerarquía parcial de los distintas clases que lo conforman e indicando algunas características comunes de ellas.

El resto de secciones presentan las relaciones existentes entre las clases. Para facilitar la comprensión del modelo, se ha desglosado en diferentes diagramas de clases. La descomposición empleada se basa en las características de los lenguajes orientados a objetos estáticamente tipados con genericidad: descripción de clases, descripción e implementación de métodos, genericidad, herencia, instrucciones y expresiones. Estas secciones del documento se presentan con un diagrama de clases e instanciaciones parciales de códigos fuentes, escritos tanto en Java como en MOON, representados con diagramas de objetos de las clases definidas en MOON.

2.2.1 Jerarquía del modelo MOON

La jerarquía del modelo MOON clasifica los conceptos que han sido considerados relevantes para poder llevar a cabo refactorizaciones en lenguajes orientados a objetos estáticamente tipados con soporte de genericidad y añade unas propiedades básicas a los mismos. Esta clasificación permite manejar la evolución del modelo en el caso de tener que incorporar nuevos conceptos. (ver Diagramas de clases 1)



Diagramas de clases 1 Jerarquía parcial del modelo MOON

Todo elemento a considerar en una refactorización hereda de la clase OBJECTMOON. Los objetos de esta clase contiene un identificador único *(identifier)* representado por una cadena cuya semántica es sólo legible desde el propio el framework. El atributo *sourcereference* es una cadena de caracteres que almacena la información correspondiente al fichero fuente y el número de línea donde aparece el objeto.

En las refactorizaciones es necesario conocer las dependencias existentes entre clases para poder determinar si es posible llevarla a cabo. Estas dependencias vienen dadas por relaciones de herencia, asociaciones y relaciones de dependencia. La relaciones de herencia son representadas a través de las clases INHERITANCE_CLAUSE y MODIFIER. Las instrucciones (INSTR) y las expresiones (EXPR) dan lugar a relaciones de dependencia. Estos elementos del modelo son discutidos con mayor profundidad en las secciones posteriores 2.2.5,2.2.6.

Dentro de esta jerarquía, se distinguen un conjunto de elementos (NAMEDOBJECT) que se caracterizan por poseer un nombre único *(uniquename)* y un nombre *(name)* representados por una cadena y cuya semántica mantiene un significado especial desde fuera del propio framework. El nombre *(name)* sirve para identificar al elemento dentro de un contexto y el nombre único *(uniquename)* sirve para identificar al elemento y el contexto donde aparece.

Tomando como base las definiciones de este tipo de objetos expuestas en [Crespo 2000], a continuación se establece las reglas de composición de la cadenas para formar el nombre único *(uniquename)*.

2.2.1.1 Entidad (ENTITY)

Una entidad es un nombre (un identificador) en el texto de una clase que denota un objeto (valor o referencia) en tiempo de ejecución. Las entidades a considerar en MOON son: atributos (ATT_DEC), variables locales a un método (LOCAL_DEC), argumentos formales a un método (FORMAL_ARGUMENTS), resultado de funciones más dos entidades predefinidas self (SELF) y result (RESULT).

El Diagrama de clases 2 muestra la representación de la clasificación de las entidades añadiendo un nuevo nivel donde se consideran las entidades predefinidas (PREDEFINE_ENTITY), entidades internas (INTERNAL_ENTITY) y entidades de signatura (SIGNATURE_ENTITY).

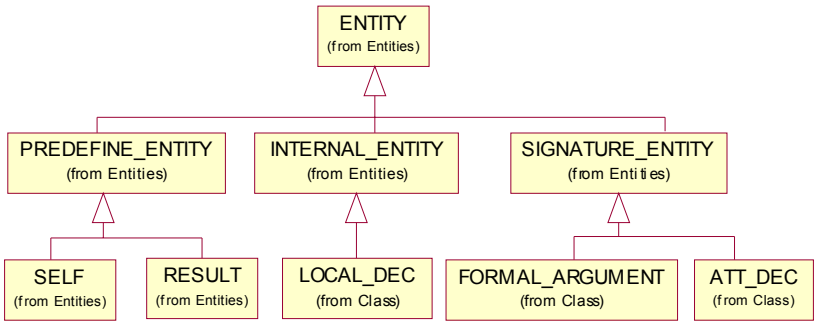


Diagrama de clases 2 Clasificación de entidades

Tanto los atributos (ATT_DEC) como los métodos (METHOD_DEC) definen el conjunto de propiedades de una clase (PROPERTY).

Dada una clase *C:*

- Las entidades correspondientes a una propiedad *p* de *C*, se identifican unívocamente *(C, _, p)*.
- La entidad predefinida *self* de *C*, (*C, _, self)*.
- Las entidades *a* correspondientes a los argumentos formales de un método *m*, *(C,m,a)*.
- Las entidades predefinidas *result* correspondientes a todas las funciones *f* implementadas en *C, (C,f,result)*.
- Las entidades locales *l* que se declaran en los métodos *m* implementados en la clase *C, (C,m,l)*.

2.2.1.2 Clases (CLASS_DEF)

El nombre de una clase basta para identificarla. En los entornos reales de programación se proporciona algún mecanismo que sirve para identificar de forma unívoca una clase añadiendo información referente al contexto de la clase. Este es el caso de los paquetes de Java las directivas include de C o de los cluster de EIFFEL. En el caso de MOON se apoyará en estos mecanismos de los propios lenguajes para identificar la clase de forma unívoca a través del nombre.

En el caso de las definiciones de las clases genéricas se debe añadir al nombre de la clase la lista de parámetros formales para identificarla. Si *C* es una clase genérica que contiene un parámetro formal *A*, el identificador de la clase será *C[A]*.

2.2.1.3 Métodos (METHOD_DEC)

MOON soporta dos tipos de métodos (METHOD_DEC), los que tienen un valor de retorno denominados funciones (FUNCTION_DEC) y los que no lo tienen, denominados rutinas (ROUTINE_DEC).

Para identificar de forma unívoca un método en MOON se utiliza el nombre de la clase *C* donde esta implementado y el nombre del propio método *m*, *(C,m)*. Hay que aclarar que en MOON no esta permitida directamente la sobrecarga de métodos, por esto no es necesario la lista de los tipos de los argumentos formales del método para identificarlo. Un mecanismo indirecto para soportar la sobrecarga se muestra en el ejemplo Código 1 donde se respeta la restricción de nombre único de un método en el ámbito de una clase.

Código Java	Código MOON

`class A{`
`  ...`
`  public int f(int a){}`
`  public int f(){ }`
`  ...`
`}`
	`class A`
	`  signatures`
	`  ...`
	`  methods`
	`  f overload int(a:  int):`
	`int;`
	`  f_overload_empty: int;`
	`  body`
	`  ...`
	`  end`

Código 1 Sobrecarga de métodos en MOON

2.2.1.4 Tipos (TYPE)

El identificador del tipo que una clase *C* implementa depende de *C*. Al soportar MOON genericidad los tipos son clasificados en parámetros formales (FORMAL_PAR) y tipos implementados por las propias definiciones de las clases (CLASS_TYPE).

Cuando la definición de una clase no es genérica la asociación entre la clase (CLASS_DEF) y el tipo (CLASS_TYPE) es única, es decir se considera a una clase como una construcción lingüística en un LOO que se usa para implementar los tipos. Por tanto el nombre único del tipo vendrá dado por el nombre de la clase.

Cuando la definición de una clase es genérica, se puede decir que es una clase determinante de un conjunto potencialmente infinito de tipos. Cada una de las instanciaciones/derivaciones genéricas realizadas se corresponde con distintos tipos (CLASS_TYPE). Dentro del conjunto de tipos obtenidos a partir de las instanciaciones genéricas se distinguen dos subconjuntos, tipos completos o completamente instanciados y los tipos no completos o no completamente instanciados:

- Los tipos completos, procedentes de instanciaciones genéricas completas (*completed = true*), son aquellos cuyo conjunto de parámetros reales no contienen ningún parámetro formal (FORMAL_PAR) es decir el conjunto de parámetros reales permanece fijo.
- Los tipos no completos, procedentes de instanciaciones genéricas no completas, (*completed = false*), cuyo conjunto de parámetros reales

contienen al menos un parámetro formal (FORMAL_PAR) es decir el conjunto de parámetros reales permanece variable dependiendo del contexto.

De la definición anterior se deduce que los tipos procedentes de instanciaciones genéricas no completas están contenidos dentro de clases genéricas, ya que utilizan un parámetro formal en la instanciación.

Para identificar los tipos de forma única procedentes de instanciaciones genéricas completas, es necesario el nombre de la clase genérica de la instanciación más la lista de los tipos de los parámetros reales que contiene la instanciación. Estos nuevos tipos tendrán un ámbito global.

Para identificar los tipos de forma única procedentes de instanciaciones genéricas no completas, se necesita el nombre de la clase genérica de la instanciación, más la lista de los tipos de los parámetros reales que contiene la instanciación, más el nombre de la clase genérica que contiene la instanciación. Estos tipos tendrán un ámbito local a la clase donde se produce la instanciación ya que tienen una dependencia sobre los parámetro formales de la clase genérica que les contiene.

Los parámetros formales sólo son visibles en el ámbito de la clase génerica donde son definidos por tanto para identificarles se utiliza el nombre de la clase genérica añadiendo el nombre del parámetro formal. Sea *C[F]* una clase genérica con parámetro formal *F*, se identifica de forma única al parámetro formal *F* por *(C,F)*.

Dada dos clases genéricas *C1[F1]* con parámetro formal *F1* y *C2[F2]* con parámetro formal *F2* y una clase no genérica *C*:

- Un tipo procedente de una instanciación genérica completa *C1[C],* se identifica unívocamente por *(C1, [C])*.
- Un tipo procedente de una instanciación genérica no completa *C1[F2]* contenida dentro de la clase *C2[F2]*, se identifica unívocamente *(C1, [F2],C2)*.

Ambas instanciaciones *C1[C]* y *C1[F2]* son nuevos tipos de datos identificados, que pueden usarse cómo parámetros reales en una nueva instanciación genérica de tipo.

- Una instanciación genérica completa *C1[C1[C]]* se identifica unívocamente por *(C1,[(C1, [C])]).*
- Una instanciación genérica no completa *C1[C1[F2]]* contenida dentro de la clase *C2[F2]*, se identifica unívocamente *(C1,[(C1, [F2],C2)],C2)*.

2.2.2 Descripción de Clases

En el Diagrama de clases 3 se representa los conceptos asociados a la definición de una clase (CLASS_DEF) y su asociación de tipo (CLASS_TYPE). La asociación entre la definición de una clase CLASS_DEF y su tipo CLASS_TYPE tiene una multiplicidad de 1, es decir se considera a una clase como una construcción lingüística en un LOO que se usa para implementar los tipos. La definición de una clase (CLASS_DEF) viene dada por el conjunto de propiedades que contiene, es decir atributos (ATT_DEC) y métodos (METHOD_DEC).

Al ser MOON un lenguaje estáticamente tipado toda entidad (ENTITY) tendrá asociado un tipo estático en tiempo de compilación.

Tanto la definición de una clase como la descripción de un método pueden ser abstractas, esta consideración se refleja en el modelo a través del atributo *deferred*.

El atributo *language* es una cadena que indica el lenguaje de programación y versión utilizada para definir la clase.

El atributo *generic* es un valor booleano que indica si la definición de una clase es o no genérica.

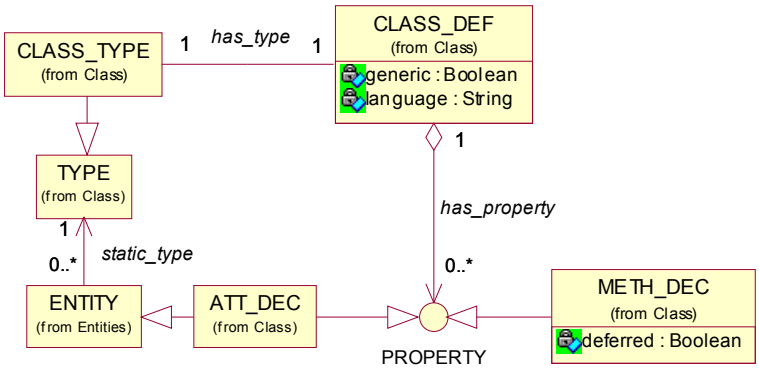


Diagrama de clases 3 Tipos y descripción de una clase.

A continuación se presentan una definición de clase no genérica Código 2 y una instanciación parcial en Diagrama de objetos 1 a partir de las clases del Diagrama de clases 3.

Código Java	Código MOON

`class A{`
`  private int x;`
`  public void f( ){`
`  g();`
`  }`
`  public void g(){`
`  x = 5;`
`  }`
`}`
	`class A`
	`  signatures`
	`  attributes`
	`  x : Int;`
	`  methods`
	`  f;`
	`  g;`
	`  body`
	`  f`
	`  do`
	`  g;`
	`  end;`
	`  g`
	`  do`
	`  x:=5;`
	`  end;`
	`  end`

Código 2 Definición de una clase no genérica.

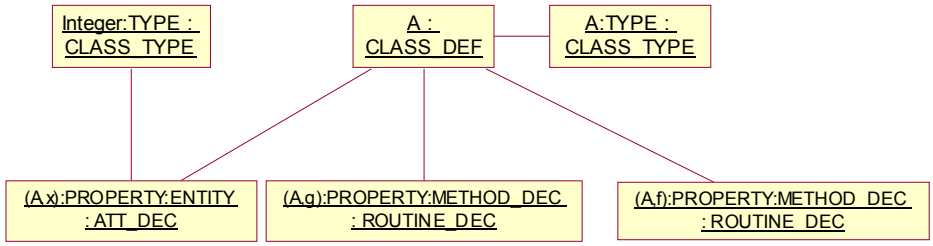


Diagrama de objetos 1 Instanciación parcial de una definición de clase no genérica

2.2.3 Descripción e implementación de métodos

MOON soporta dos tipos de métodos, los que tienen un valor de retorno denominados funciones (FUNCTION_DEC) y los que no lo tienen, denominados rutinas (ROUTINE_DEC). La descripción de los métodos esta determinada por la signatura del mismo, la lista de argumentos formales (FORMAL_ARGUMENT) y el tipo de retorno en el caso de tratarse de una función.

La implementación de un método va estar compuesta por un conjunto de variables locales (LOCAL_DEC) y un conjunto de instrucciones (INSTR).

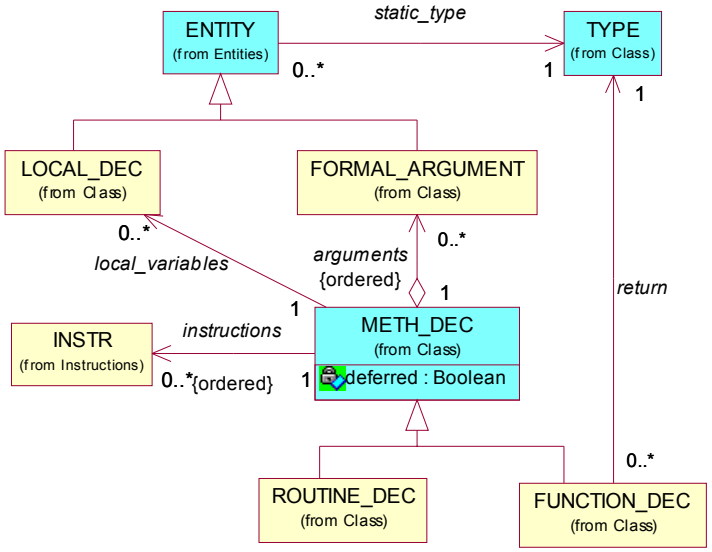


Diagrama de clases 4 Descripción de métodos

El Diagrama de objetos 2 describe la clase definida en Código 3, indicando la descripción de la función `deferred compareTo(o:Object):int`.

Código Java	Código MOON

`interface Comparable{`
`  public int compareTo(Object o);`
`}`
	`deferred class Comparable`
	`  signatures`
	`  attributes`
	`  methods`
	`  deferred compareTo(o:Object):int;`

	body end

Código 3 Definición de una clase abstracta no genérica `Comparable`

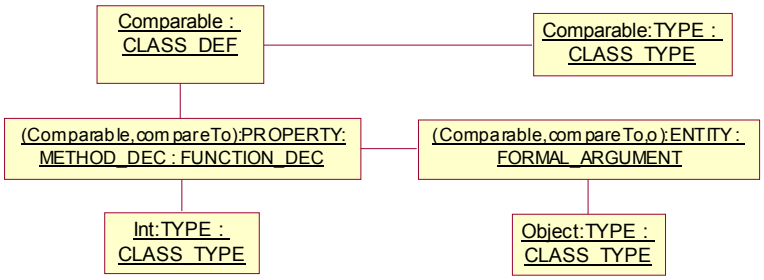


Diagrama de objetos 2 Descripción de métodos

2.2.4 Genericidad

Cada una de las instanciaciones genéricas realizadas se corresponde con distintas instancias de la clase CLASS_TYPE, de esta forma la definición de una clase genérica representa un conjunto de tipos.

La definición de una clase genérica consta de una lista de parámetros formales. El modelo MOON soporta dos variantes en lo referente a la acotación de los parámetros formales, subtipado (variante s) y cláusulas where (variante w). Ambas persiguen acotar las características de los parámetros formales para garantizar la corrección de tipos en las instanciaciones genéricas determinando un conjunto de sustituciones válidas para los parámetros formales. Si un parámetro formal no esta acotado, el tipo del parámetro real en una sustitución puede ser cualquiera.

En el Diagrama de clases 5 se representan las abstracciones identificadas en MOON y sus relaciones. Para completar el modelo se añaden dos invariantes en la clase CLASS_DEF para especificar que las relaciones *generic_instantation* y *formal_parameters* sólo son válidas si la clase es genérica, y otro sobre la clase CLASS_TYPE para especificar las condiciones de tipos completamente instanciados sobre derivaciones genéricas.

```
context CLASS_DEF
 inv:
  self.gCLASS_TYPE->notEmpty()    implies    self.generic
=True
inv:
   self.fFORMAL_PAR->notEmpty()    implies    self.generic
=True

context CLASS_TYPE
  inv:
    self.completed = False implies self.rType->exist( t
: TYPE | t.oclIsTypeOf(FORMAL_PAR))
```

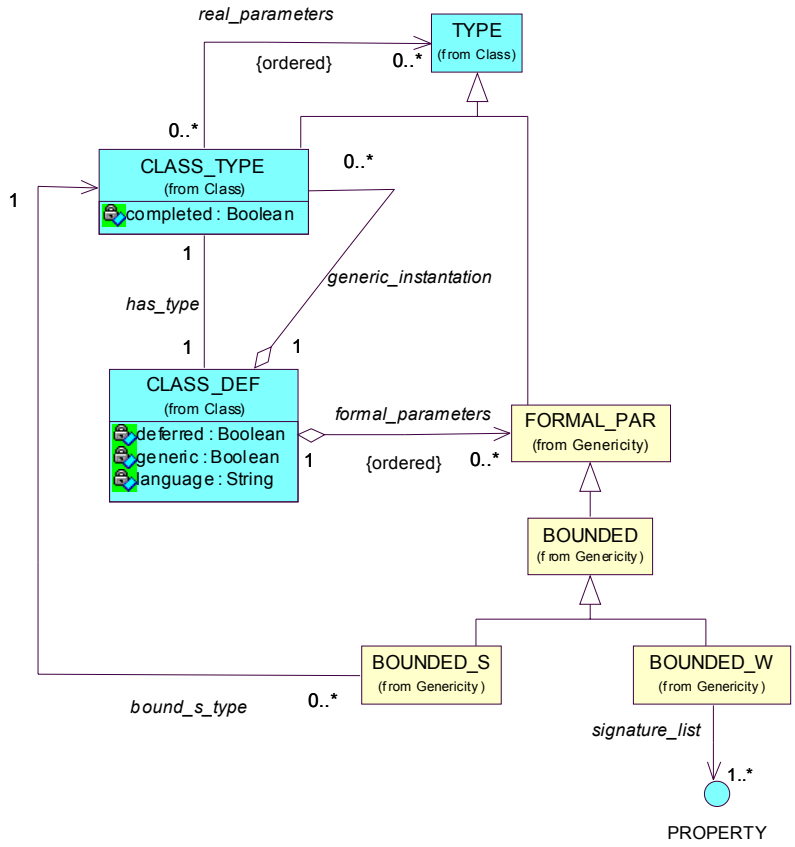


Diagrama de clases 5 Definición de clases genéricas soportando acotación de subtipado y cláusulas where

En el Código 4 se muestra la definición de tres clases abstractas dos genérica (`IteratorIF[A]`, `Collection[A]`) y una no genérica (`Comparable`). La clase genérica Collection[A] mantiene sobre su parámetro formal A una acotación de subtipado (`A -> Comparable`). En Diagrama de objetos 3 se representa dicho código a través de instancias de clases definidas en Diagrama de clases 5 y Diagrama de clases 2.

Código Java extensión GJ	Código MOON

`interface Comparable{`
`  public int compareTo(Object o);`
`}`
	`deferred class Comparable`
	`  signatures`
	`   attributes`
	`   methods`
	`    deferred compareTo(o : Object):int;`
	`  body`
	`  end`

`interface IteratorIF<A>{`
`   public A next();`
`   public boolean hasNext();`
`}`
	`deferred class IteratorIF[A]`
	`  signatures`
	`    atributes`
	`    methods`

	deferred next: A; deferred hasNext:Bool; body end

interface Collection
<A implements Comparable>{
public void add(A x);
public Iterator<A> iterator();
}
	deferred class Collection
	[A ->Comparable]
	signatures
	atributes
	methods
	deferred add(x:A);
	deferred iterator: IteratorIF[A];
	body
	end

Código 4 Definición de una clase genérica con acotación de subtipado.

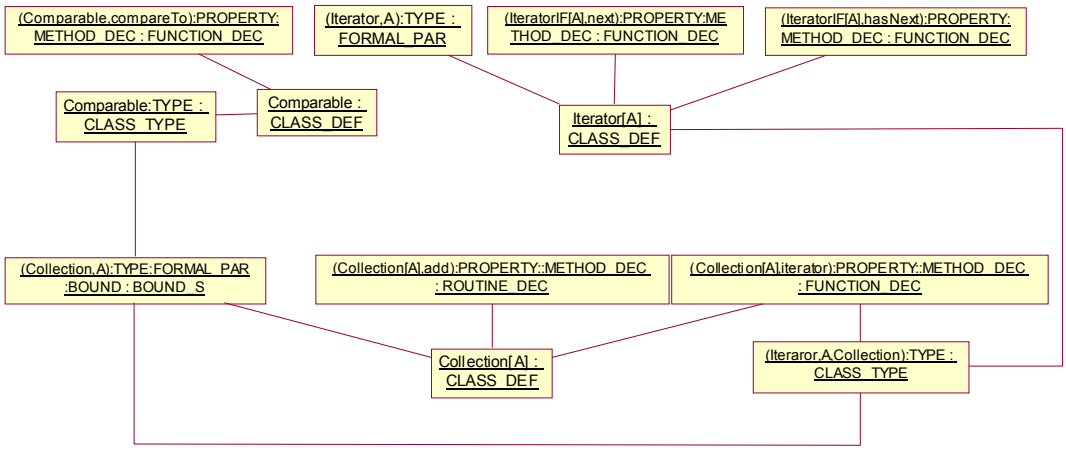


Diagrama de objetos 3 Instanciación parcial de clases genéricas con acotación de subtipado

Hay que observar en el Diagrama de objetos 3 que se ha instanciado un nuevo tipo de datos (Iterator,A,Collection) fruto de la derivación genérica no final producida por el valor de retorno de la función iterator (Collection[A], iterator).

2.2.5 Herencia

La definición de una clase en MOON puede heredar de otras clases (herencia múltiple), esta característica se representa a través de la asociación entre la definición de la clase (CLASS_DEF) y las cláusulas de herencia (INHERITANCE_CLAUSE).

Los modificadores permitidos en MOON por las reglas de herencia ⊕ RENAME, REDEFINE, MAKEEFFECTIVE, MAKEDEFERRED, dictan que una propiedad que se hereda se puede renombrar, redefinir, hacer concreta si era abstracta y hacer abstracta si era concreta respectivamente. El modificador SELECT es necesario para permitir la resolución de un conflicto de selección en ligadura tardía en presencia de herencia múltiple.

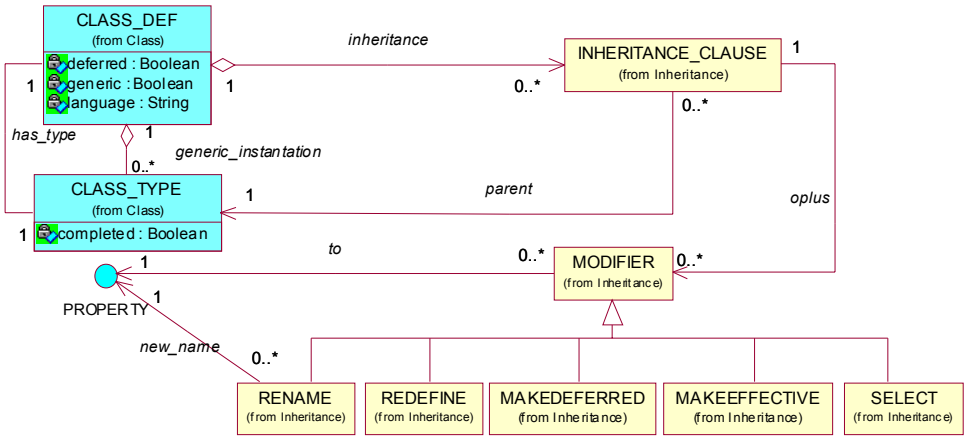


Diagrama de clases 6 Soporte de herencia

En el Código 5 se muestra la definición de una clase que hereda de una clase previamente definida en Código 2, redefiniendo la rutina f (f es una propiedad extrínseca a B). En el Diagrama de objetos 4 se modela dicho código a través de instancias de clases definidas en Diagrama de clases 6 y Diagrama de clases 3.

Código Java	Código MOON

`class B extends A{`
`  void f( ){ }`
`}`
	`class B`
	`  inherit A`
	`  redefine f;`
	`  ;`
	`  signatures`
	`  body`
	`  f`
	`  do`
	`  -- redefinición vacía`
	`  end;`
	`  end`

Código 5 Definición de una clase con herencia simple y con redefinición

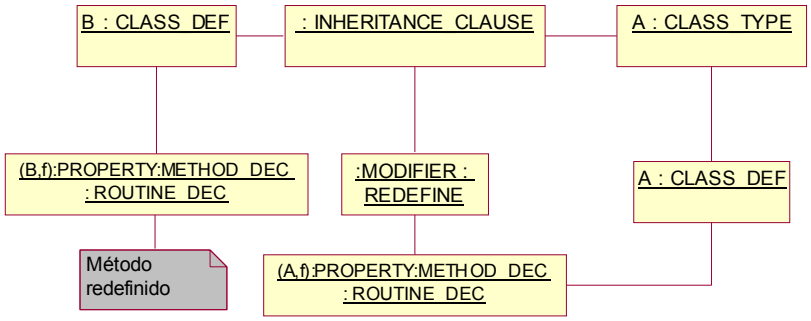


Diagrama de objetos 4 Definición de una clase con herencia simple y redefinición

En el Código 6 se muestra la definición de una clase que hereda de múltiples clases, concretando la función `compareTo`. En el Diagrama de objetos 5 se modela dicho código a través de instancias de clases definidas en Diagrama de clases 6.

Código Java	Código MOON

`class C`
`  extends A`
`  implements Comparable{`
`  public int compareTo(Object o){`
`  /* Implementación vacía */`
`  }`
`}`
	`class C`
	`  inherit A;`
	`  inherit Comparable`
	`  makeefective compareTo;`
	`  ;`
	
	`  attributes`
	`  methods`
	
	`  body`
	`  compareTo (o:Object) :int`
	`  -- implementación vacia`
	`  end;`
	`end`

Código 6 Definición de una clase con herencia múltiple y concretando una propiedad abstracta

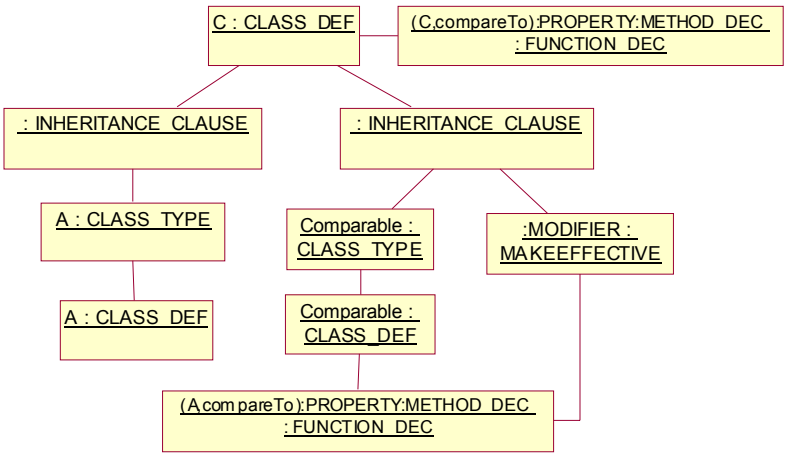


Diagrama de objetos 5 Definición de una clase con herencia múltiple y concretando una propiedad abstracta

La covarianza en especialización es una forma de polimorfismo múltiple soportada por algunos lenguajes de programación O.O. que permite redefinir métodos variando los tipos de los argumentos formales y valor de retorno. En MOON se definen dos variantes en las reglas de tipos, subtipado y conformidad. En la variante de subtipado se permiten redefiniciones siempre que se respete contravarianza en argumentos formales (FORMAL_ARGUMENT) y covarianza en el valor de retorno. En la variante por conformidad la redefinición permite covarianza en argumentos formales y en el valor de retorno. Java no permite directamente covarianza en especialización aunque es posible conseguir un efecto similar mediante la

reflexión [Gupta et al., 2000]. Por considerar esta solución propia del lenguaje Java no se ha incluido el código en la presentación de este concepto.

El Código 7 define dos jerarquía de clases en MOON donde se muestra la redefinición covariante de la rutina `f(a:A)`. El Diagrama de objetos 6 refleja las instancias de las clases correspondientes a dicho código.

Código MOON.
Jerarquía de tipos.

```
class A
  signatures
    body
    end;
end
class B
  inherit A;
  signatures
    body

    end;
end
```

Código MOON.
Herencia con redefinición covariante.

```
class C
  signatures
    methods
      f(a:A);
  body
    f(a:A)
      do

      end;
  end

class D
  inherit C
    redefine f;
  ;
  signatures
     f(b:B);
-- redefinición covariante
    body
      f(b:B)
      do
      end;
  end
```

Código 7 Herencia con redefinición covariante

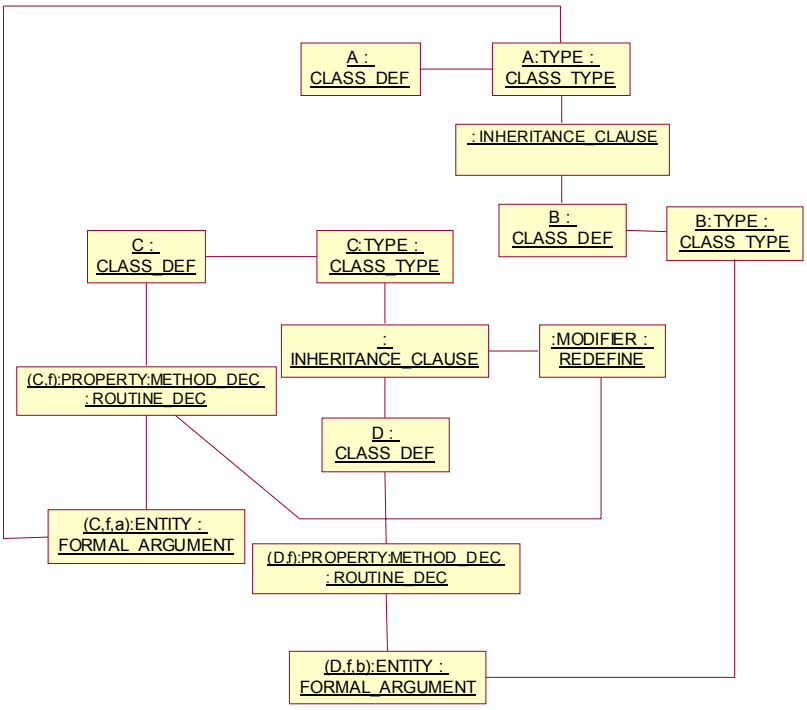


Diagrama de objetos 6 Herencia con redefinición covariante

2.2.6 Instrucciones y expresiones

Con el objetivo de poder definir y analizar las refactorizaciones se hace necesario el representar la información relacionada con la colaboración de los objetos a través del envió de mensajes incluidos en la implementación de los métodos. En MOON, los métodos correspondientes a mensajes que solicitan servicios son rutinas, mientras que los mensajes que solicitan información se corresponden con atributos y métodos definidos como funciones. Todo envió de mensaje esta asociado a una entidad. Si el mensaje se corresponde con un atributo o una función el envió es una expresión (EXPR), si se corresponde con una rutina es una instrucción (INSTR).

La longitud de un envío de mensajes e_m se define como la cantidad de llamadas encadenadas, que forman la construcción, al margen de los argumentos reales que cada llamada pueda tener, y se denota por $L(e_m)$. Siendo $e_m = e_1.e_2... .e_n$, n >= 1, la estructura de un envío de mensajes, al margen de los argumentos, $L(e_m) = n$.

Puesto que un envío de mensajes es una forma especial de expresión se puede hablar de longitud de una expresión añadiendo a la definición las expresiones que no se corresponden con envíos de mensajes (MANIFEST_CONSTANT).

En MOON se consideran las siguientes simplificaciones respecto a instrucciones y expresiones:

- Eliminación de envíos de mensaje en cascada. Cualquiera que sea la llamada en cascada $e_1.e_2... .e_n$ con n>2, al margen de los argumentos que pueda tener cada mensaje involucrado en la cascada se puede reducir considerando la siguiente secuencia de instrucciones:

$t_1 := e_1.e_2$;

$t_2 := t_1.e_3$;

...

$t_{i-1} := t_{i-2}.e_i$; $(2 < i < n)$

si $e_1.e_2 \ldots .e_n$ es una expresión, cuando i =n se tendrá $t_{n-1} := t_{n-2}.e_n$;

si $e_1.e_2 \ldots .e_n$ es una instrucción, cuando i =n se tendrá $t_{n-2}.e_n$;

Al considerar esta simplificación y la entidad predefinida en MOON SELF, el envió de mensajes se clasifica en envíos de longitud uno si la entidad asociada al envió de mensaje es SELF de forma implícita y envío de mensajes de longitud dos en caso contrario.

- Eliminación de algunas formas de expresiones. En MOON no se incluyen expresiones binarias ni unarias asumiendo que se pueden reescribir como una expresión de envió de mensajes

  a + b deberá reescribirse como a.+(b)

- Eliminación de instrucciones de control de flujo de ejecución.

Una expresión es una construcción en el texto de una clase denotando un cómputo que retorna un objeto (valor o referencia): entidades de signatura, internas y predefinidas (SIGNATURE_ENTITY, INTERNAL_ENTITY, PREDEFINE_ENTITY), envió de mensajes de solicitud de información y constantes manifiestas.

El Diagrama de clases 7 muestra el conjunto de constantes manifiestas soportadas en MOON.

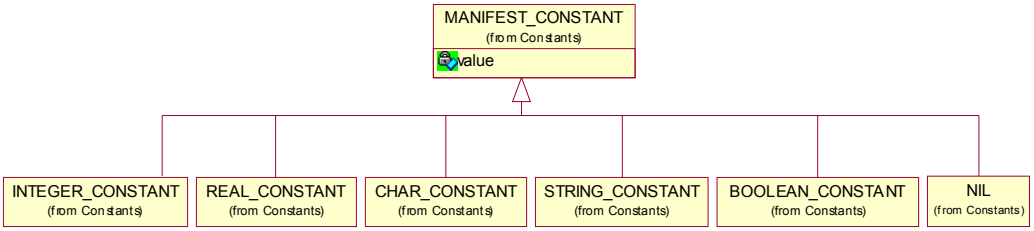


Diagrama de clases 7 Constantes manifiestas

Una expresión atómica es una entidad o una constante manifiesta. La longitud de una expresión atómica es 1. En el Diagrama de clases 8 se muestra las clases que representan la clasificación de expresiones discutida y las relaciones entre ellas.

Debido a la simplificación de eliminación de envío de mensajes en cascada expuesta anteriormente los argumentos reales que puede recibir una expresión se corresponden con expresiones atómicas respetando la ley de Demeter [Lieberherr et al.,1988] [Lieberherr y Holland,1989]. Dada una expresión `f(a.b)` se puede representar con la siguiente descomposición de sentencias `t1=a.b` y `f(t1).`

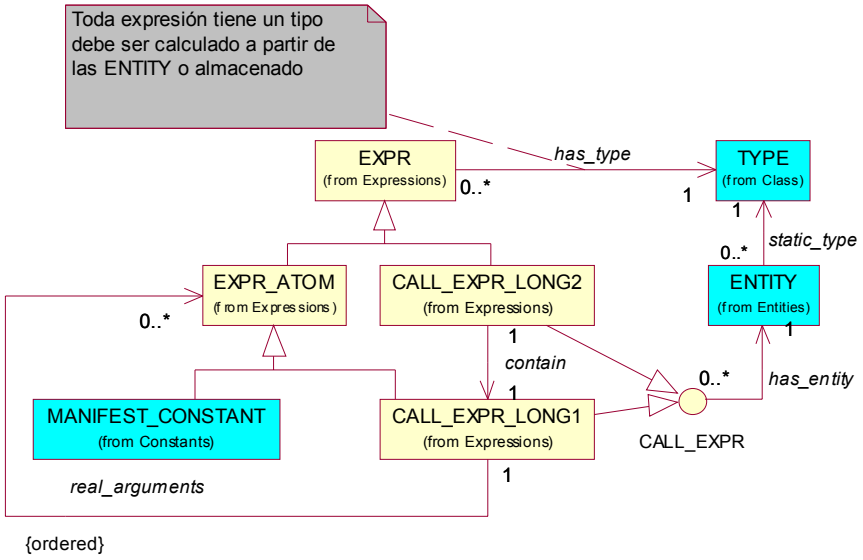


Diagrama de clases 8 Expresiones

Después de las simplificaciones vistas, las instrucciones que se consideran en MOON son: las instrucciones de creación o constructores (CREATION_INSTR), intrucciones de asignación (ASSIGNMENT_INSTR), envío de mensajes de petición de servicios (CALL_INSTR) y instrucciones compuesta o bloques de instrucciones (COMPOUND_INSTR). La representación completa de cada tipo de instrucción puede observarse en el Diagrama de clases 9.

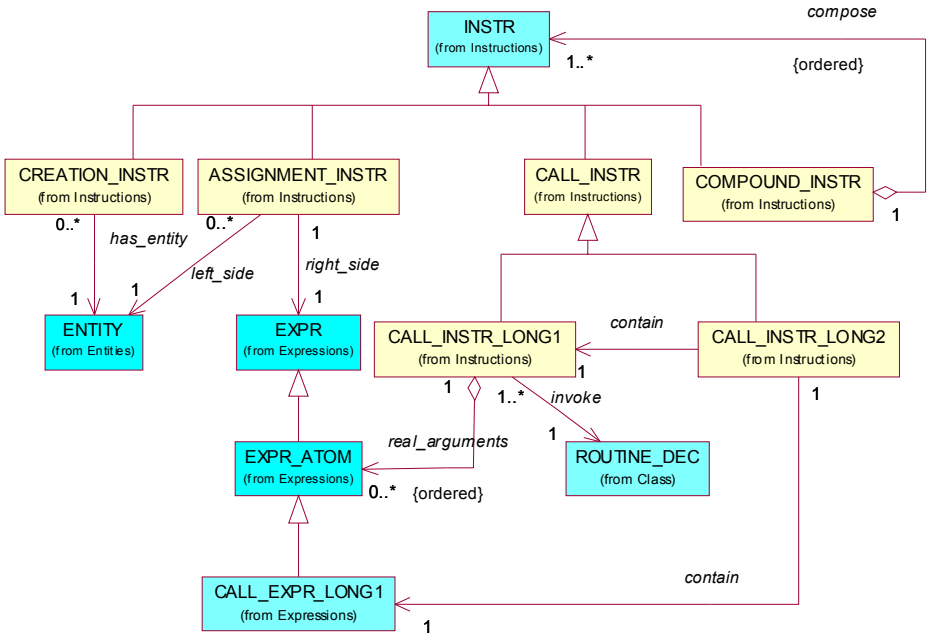


Diagrama de clases 9 Instrucciones

Tomando como base el Código 8, y suponiendo que existe un método (función o rutina) en la clase B que contiene las instrucciones:

- El Diagrama de objetos 7 representa una instrucción de asignación a una expresión de longitud 2. `x = a.fa( );`
- El Diagrama de objetos 8 representa una instrucción de asignación a una expresión de longitud 2. `x = a.a;`
- El Diagrama de objetos 9 representa una instrucción de asignación con una expresión de longitud 1. `x = fb();`

Código Java	Código MOON

`class A{`
`  int a;`
`  int fa( ){return 1;};`
`/*Funcion*/`
`  void ra( ){ };`
`/* Rutina*/`
`}`
	`class A`
	`  signatures`
	`  attributes`
	`  a:int;`
	`  methods`
	`  fa :int; -- función`
	`  ra;  -- rutina`
	`  body`
	`  fa`
	`  do`
	`  result 1;`
	`  end;`
	`  ra`
	`  do`
	`  end;`
	`  end`

`class B{`
`  A a;`
`  int b;`
`  int x;`
 	
`  int fb( ){`
`  return 1;`
`  }; /*Funcion*/`
`  void rb( ){ }; /* Rutina*/`
 	
`}`
	`class B`
	`  signatures`
	`  attributes`
	`  a:A;`
	`  b:int;`
	`  x:int;`
	`  methods`
	`  fb :Int; -- función`
	`  rb;  -- rutina`
	`  body`
	`  fb`
	`  do`
	`  result 1;`
	`  end;`
	`  rb`
	`  do`
	`  end;`
	`  end`

Código 8 Definición de dos clases no genéricas con entidades referenciadas

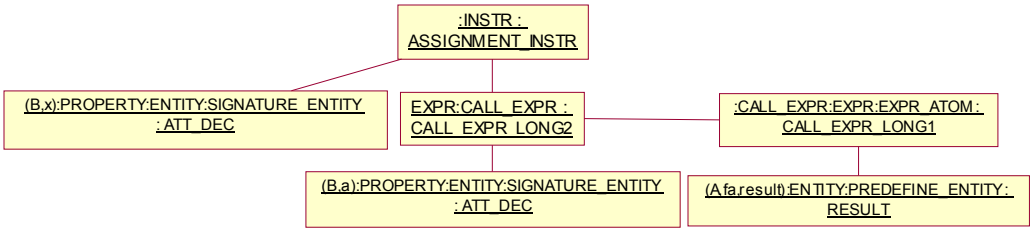


Diagrama de objetos 7 Asignación de una expresión de llamada a función longitud 2

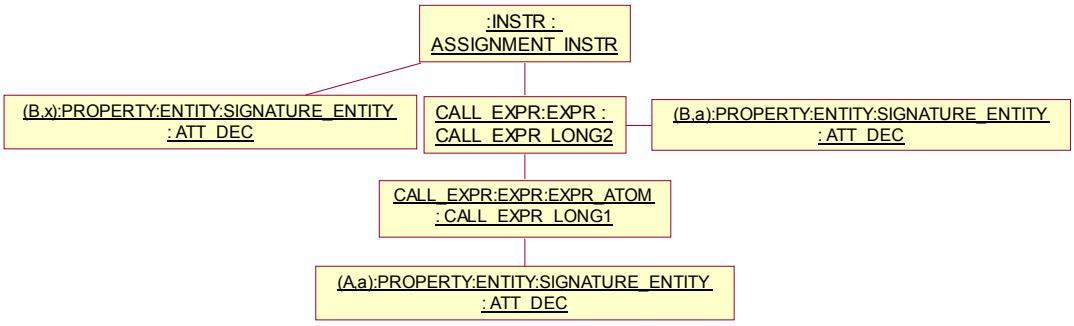


Diagrama de objetos 8 Asignación de una expresión de atributo longitud 2

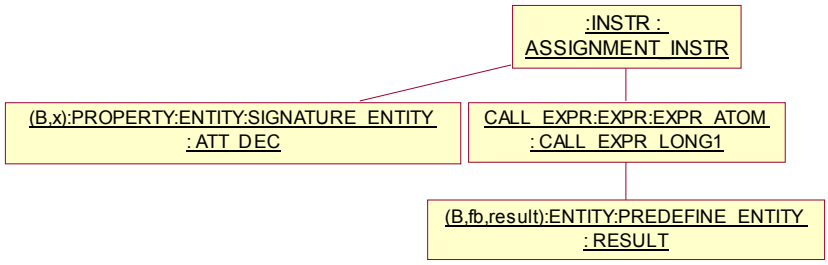


Diagrama de objetos 9 Asignación de una expresión de llamada a función longitud 1

2.3 Definición de operaciones básicas sobre las clases

En esta sección se definen los métodos selectores y modificadores tanto sobre los atributos como de las relaciones directas de las clases conceptuales identificadas en la sección anterior. Se presentan las clases clasificadas por paquetes y organizadas alfabéticamente dentro de cada paquete.

2.3.1 Paquete Class

2.3.1.1 CLASS_DEF

- `addFormalPar(formalpar:FORMAL_PAR)`

Añade el parámetro formal pasado como argumento, al final de la secuencia de parámetros formales asociados a la definición de la clase genérica.

- `addFormalPar(formalpar:FORMAL_PAR,index:Integer)`

Añade el parámetro formal pasado como argumento formal, a la secuencia de parámetros formales asociados a la definición de la clase genérica, en la posición indicada en el argumento index. Si existía otro parámetro formal en esa posición se elimina.

- `addGenericClassType(type:CLASS_TYPE)`

Añade el tipo procedente de una derivación genérica pasado como parámetro a la colección de derivaciones genéricas asociadas a la definición de la clase.

- `addInheritanceClause(inheritance:INHERITANCE_CLAUSE)`

Añade una nueva cláusula de herencia a la colección de cláusulas de herencia especificadas en la definición de la clase.

- `addProperty(prop:PROPERTY)`

Añade una nueva propiedad a la definición de la clase.

- `getClassType():CLASS_TYPE`

Obtiene el tipo asociado a la definición de la clase. Esta operación sólo esta disponible en el caso de que la clase sea no genérica.

- `getFormalPar():Sequence<FORMAL_PAR>`

Retorna la secuencia de parámetros formales asociados a la definición de la clase genérica. Este método sólo esta disponible para definiciones de clases genéricas.

- `getFormalPar(index:Integer):FORMAL_PAR`

Retorna la el parámetro formal asociado a la definición de la clase genérica situado en la posición indicada en el argumento index.

- `getGenericClassType():Collection<CLASS_TYPE>`

Retorna una colección con todas las derivaciones genéricas asociadas a la definición de clase tratada. Este método sólo esta disponible para definiciones de clases genéricas.

- `getIndexFormalPar(formalname:String):Integer`

Retorna la posición del parámetro formal asociado a la definición de la clase genérica que contiene el mismo nombre que la cadena pasada por parámetro formalname.

- `getInheritanceClause():Collection<INHERITANCE_CLAUSE>`

Retorna una colección de las cláusulas de herencia asociadas a la definición de la clase.

- `getLanguage():String`

Devuelve una cadena correspondiente al valor del atributo language, que almacena la información referente al lenguaje de programación y la versión utilizado en la definición de la clase.

- `getProperties( ):Collection<PROPERTY>`

Devuelve una colección con todas las propiedades de la clase. Sólo se incluyen las propiedades definidas en la propia clase y las que han sido modificadas a través de una relación de herencia.

- `isDeferred():Boolean`

Indica si la definición de la clase especificada es diferida.

- `isGeneric():Boolean`

Indica si la definición de la clase es genérica.

- `removeFormalPar(index:Integer)`

Elimina de la secuencia de parámetros formales asociados a la definición de la clase, el parámetro formal situada en la posición indicada con el argumento de entrada index. Desplaza la secuencia de parámetros formales situados a la derecha del parámetro formal eliminado, en una posición a la izquierda.

- `removeFormalPar(nameformalpar:String)`

Elimina de la secuencia de parámetros formales asociados a la definición de la clase, el parámetro formal que conforma con el nombre pasado como argumento de entrada en el método nameformalpar. Desplaza la secuencia de parámetros formales situados a la derecha del parámetro formal eliminado, en una posición a la izquierda.

- `removeGenericClassType(nameclasstype:String)`

Elimina de la colección de derivaciones genéricas asociados a la definición de la clase genérica, el tipo que conforma con el nombre pasado como argumento de entrada nameclasstype.

- `removeInheritanceClause(nameparenttype:String)`

Elimina de la colección de cláusulas de herencia la cláusula cuyo tipo se corresponde con el nombre del tipo pasado como argumento nameparentype.

- `setClassType(ptype:CLASS_TYPE)`

Asocia el tipo que recibe como parámetro a la definición de la clase.

- `setDeferred(pdeferred:Boolean)`

Actualiza el atributo deferred con el valor que recibe como parámetro.

- `setGeneric(pgeneric:Boolean)`

Actualiza el atributo generic con el valor que recibe como parámetro.

- `setLanguage(planguage:String)`

Actualiza el atributo language con el valor que recibe como parámetro.

2.3.1.2 CLASS_TYPE

- `addRealPar(realpar:TYPE)`

Añade el parámetro real pasado como argumento de entrada, al final de la secuencia de parámetros reales asociados al tipo procedente de una derivación genérica.

- `addRealPar(realpar:TYPE,index:Integer)`

Añade el parámetro real pasado como argumento de entrada, a la secuencia de parámetros reales asociados al tipo procedente de una derivación genérica, en la posición indicada en el parámetro index. Si existía otro parámetro en esa posición se elimina.

- `getIndexRealPar(realpar:String):Collection<Integer>`

Retorna una colección de Integer cuyos elementos representan la posición del tipo asociado a los parámetros reales que contiene el mismo nombre que la cadena pasada por parámetro realpar.

- `getClassDef():CLASS_DEF`

Retorna la definición de la clase asociada al tipo.

- `getGenericClassDef():CLASS_DEF`

Retorna la definición de la clase genérica asociada al tipo paramétrico.

- `getRealPar():Sequence<TYPE>`

Retorna una secuencia con los tipos de los parámetros reales procedentes de derivaciones genéricas.

- `isCompleted():Boolean`

Indica si el tipo esta completamente instanciado. Todos los tipos de clases no genéricas están completamente instanciados. Los tipos procedentes de clases genéricas donde alguno de sus parámetros reales no sea fijo no estarán completamente instanciados.

- `removeRealPar(index:Integer)`

Elimina de la secuencia de parámetros reales el parámetro real situado en la posición indicada con el parámetro index. Desplaza la secuencia de parámetros reales situados a la derecha del parámetro real eliminado, en una posición a la izquierda.

- `removeRealPar(namerealpar:String)`

Elimina de la secuencia de parámetros reales el parámetro real que conforma con el nombre del tipo pasado como parámetro en el método. Desplaza la secuencia de parámetros reales situados a la derecha del parámetro real eliminado, en una posición a la izquierda.

- `setClassDef(pclass:CLASS_DEF)`

Asocia una definición de clase al tipo.

- `setCompleted(pcompleted:Boolean)`

Actualiza el valor de la propiedad completed con el valor que recibe como parámetro.

2.3.1.3 FUNCTION_DEC

- `getReturnType():TYPE`

Retorna el tipo correspondiente al tipo de retorno descrito por la función representada.

2.3.1.4 METHOD_DEC

- `addFormalArg(formalarg:FORMAL_ARGUMENT)`

Añade el argumento formal pasado como argumento de entrada, al final de la secuencia de argumentos asociados a la descripción del método.

- `addFormalArg(formalarg:FORMAL_ARGUMENT,index:Integer)`

Añade el argumento formal pasado como parámetro de entrada, a la secuencia de argumentos formales asociados a la descripción del método, en la posición indicada en el parámetro index. Si existía otro argumento en esa posición se elimina.

- `addInstr(instr:INSTR)`

Añade una instrucción al final de la secuencia de instrucciones asociadas al método.

- `addInstr(intr:INSTR,index:Integer)`

Añade la instrucción a la secuencia de instrucciones asociadas al método en la posición indicada en el parámetro index. Si existía otra instrucción en esa posición se elimina.

- `addLocalDec(localdec:LOCAL_DEC)`

Añade la descripción de la variable local pasada como parámetro a la colección de variables locales asociadas al método.

- `getFormalArg():Sequence<FORMAL_ARGUMENT>`

Retorna una secuencia con los argumentos formales del método.

- `getIndexFormalArg(formalarg:String):Integer`

Retorna la posición del argumento formal que contiene el mismo nombre que la cadena pasada por parámetro formalarg.

- `getIndexInstr(pinstr:INSTR):Integer`

Retorna la posición de la instrucción que es igual a la instrucción pasada por parámetro pinstr.

- `getInstr():Sequence<INSTR>`

Retorna una secuencia con las instrucciones del método.

- `getLocalDec():Collection<LOCAL_DEC>`

Retorna una colección con las descripción de variables locales que contiene el método.

- `isDeferred():Boolean`

Indica si la definición del método especificado es diferido.

- `removeFormalArg(index:Integer)`

Elimina de la secuencia de argumentos formales asociados al método, el argumento formal situado en la posición indicada con el parámetro index. Desplaza la secuencia de argumentos formales situados a la derecha del argumento formal eliminado, en una posición a la izquierda.

- `removeFormalArg(nameformalarg:String)`

Elimina de la secuencia de argumentos formales asociados al método, el argumento real que conforma con el nombre pasado como parámetro en el método. Desplaza la secuencia de argumentos formales situados a la derecha del argumento formal eliminado, en una posición a la izquierda.

- `removeInstr(index:Integer)`

Elimina de la secuencia de instrucciones asociadas al método, la instrucción situada en la posición indicada con el parámetro index. Desplaza la secuencia de instrucciones situadas a la derecha del la instrucción eliminada, en una posición a la izquierda.

- `removeLocalDec(localdec:LOCAL_DEC)`

Elimina la descripción de la variable local pasada como parámetro de la colección de variables locales asociadas al método.

- `setDeferred(pdeferred:Boolean)`

Actualiza el atributo deferred con el valor que recibe como parámetro.

2.3.1.5 NAMEDOBJECT

- `equalsName(object:NAMEDOBJECT):Boolean`

Indica si el objeto pasado como parámetro contiene el mismo nombre que el objeto tratado, es decir si el valor del campo named de ambos coincide.

- `equalsUniqueName(object:NAMEDOBJECT):Boolean`

Indica si el objeto pasado como parámetro contiene el mismo nombre único que el objeto tratado, es decir si el valor del campo uniquenamed de ambos coincide.

- `getName( ):String`

Devuelve una cadena correspondiente al valor del atributo name, que almacena la información referente al nombre del objeto fuera del contexto donde aparece.

- `getUniqueName( ):String`

Devuelve una cadena correspondiente al valor del atributo uniquename, que almacena la información referente al nombre del objeto junto con el contexto donde aparece. Ver sección 2.2.1.

- `setName(name :String)`

Actualiza el atributo name con el valor que recibe como parámetro.

- `setUniqueName( unique:String)`

Actualiza el atributo uniquename con el valor que recibe como parámetro.

2.3.1.6 OBJECTMOON

- `equals(object:OBJECTMOON):Boolean`

Indica si el objeto pasado como parámetro es igual asimismo, es decir si el valor del campo identifier de ambos coincide.

- `getIdentifier( ):String`

Devuelve una cadena correspondiente al valor del atributo identifier, que almacena el identificador único de los objetos MOON.

- `getSourceReference( ):String`

Devuelve una cadena correspondiente al valor del atributo sourcereference, que almacena la información con path fichero del fichero fuente y el número de línea donde aparece dicho objeto.

- `setSourceReference(String reference): String`

Actualiza el atributo sourcereference con el valor que recibe como parámetro.

2.3.2 Paquete Genericity

2.3.2.1 BOUND_S

- `getBoundType():TYPE`

Retorna el tipo que acota el parámetro formal.

- `setBoundType(boundtype:TYPE)`

Asocia un tipo que acota el parámetro formal.

2.3.2.2 BOUND_W

- `addBoundProperty(boundtype:PROPERTY)`

Añade una nueva propiedad a la acotación del parámetro formal.

- `getBoundProperty():Collection<PROPERTY>`

Retorna la colección de propiedades que acota el parámetro formal.

- `removeBoundProperty(boundtype:PROPERTY)`

Elimina de la colección de propiedades que acotan el parámetro formal la propiedad pasada como parámetro.

2.3.3 Paquete Inheritance

2.3.3.1 INHERITANCE_CLAUSE

- `addModifier(modifier:MODIFIER)`

Añade el modificador pasado como parámetro a la colección de modificadores asociados a la cláusula de herencia.

- `getClassType():CLASS_TYPE`

Obtiene el tipo asociado a la cláusula de herencia.

- `getModifierMakeDeferred():Collection<MODIFIER>`

Retorna una colección con los modificadores de propiedades referentes a hacer diferidas las propiedades asociados a la cláusula de herencia.

- `getModifierMakeEffective():Collection<MODIFIER>`

Retorna una colección con los modificadores de propiedades referentes a hacer efectivas las propiedades asociados a la cláusula de herencia.

- `getModifierRedefine():Collection<MODIFIER>`

Retorna una colección con los modificadores de redefinición de propiedades asociados a la cláusula de herencia.

- `getModifierRename():Collection<MODIFIER>`

Retorna una colección con los modificadores de renombrado de propiedades asociados a la cláusula de herencia.

- `getModifierSelect():Collection<MODIFIER>`

Retorna una colección con los modificadores de selección de las propiedades en presencia de herencia múltiple asociados a la cláusula de herencia.

- `removeModifier(modifier:MODIFIER)`

Elimina de la colección de modificadores asociados a la cláusula de herencia el modificador pasado como parámetro.

- `setClassType(type:CLASS_TYPE)`

Asocia un tipo a la cláusula de herencia.

2.3.3.2 MODIFIER

- `getProperty():PROPERTY`

Retorna la propiedad modificada en la clase padre.

- `setProperty(modifiedproperty:PROPERTY)`

Asocia la propiedad modificada de la clase padre al modificador.

2.3.3.3 RENAME

- `getNewNameProperty():PROPERTY`

Retorna la propiedad renombrada de la clase hija que hace referencia a una propiedad de la clase padre.

- `setNewNameProperty(renamedproperty:PROPERTY)`

Asocia la propiedad renombrada de la clase hija al modificador de renombrado.

2.3.4 Paquete Instructions

2.3.4.1 ASSIGNMENT_INSTR

- `getLeftSide():ENTITY`

Retorna la entidad asociada al lado izquierdo de la instrucción de asignación.

- `getRightSide():EXPR`

Retorna la expresión asociada al lado derecho de la instrucción de asignación.

- `setLeftSide(pentity:ENTITY)`

Asocia la entidad pasada como parámetro a lado izquierdo de la instrucción de asignación.

- `setRightSide(pexpr:EXPR)`

Asocia la expresión pasada como parámetro a lado derecho de la instrucción de asignación.

2.3.4.2 <<interface>>CALL_EXPR

- `getEntity():ENTITY`

Retorna la entidad asociada a la expresión.

- `setEntity(pentity:ENTITY)`

Asocia la entidad pasada como parámetro a la expresión.

2.3.4.3 CALL_EXPR_LONG1

- `addRealArg(expr:EXPR_ATOM)`

Añade un nuevo argumento real pasado como parámetro al final de la secuencia de argumentos reales asociados a la expresión.

- `addRealArg(expr:EXPR_ATOM,index:Integer)`

Añade un nuevo argumento real pasado como parámetro, a la secuencia de argumentos reales asociados a la expresión, en la posición indicada con el parámetro index.

- `getRealArg():Sequence<EXPR_ATOM>`

Retorna la secuencia de argumentos reales compuesta de expresiones atómicas asociada a la invocación de una función.

- `getRealArg(index:Integer):EXPR_ATOM`

Retorna el argumento real situado en la posición indicada en el parámetro index de la secuencia argumentos reales asociados a la expresión.

- `removeRealArg(index:Integer)`

Elimina de la secuencia de argumentos reales asociados a la expresión, el argumento real situado en la posición indicada con el parámetro index. Desplaza la secuencia de argumentos reales situados a la derecha del argumento real eliminado, en una posición a la izquierda.

- `removeRealArg(expr:EXPR_ATOM)`

Elimina de la secuencia de argumentos reales asociados a la expresión, el argumento real pasado como parámetro. Desplaza la secuencia de argumentos reales situados a la derecha del argumento real eliminado, en una posición a la izquierda.

2.3.4.4 CALL_EXPR_LONG2

- `getExprLong1():CALL_EXPR_LONG1`

Retorna la expresión de longitud uno asociada a la expresión.

- `setExprLong1(pexpr: CALL_EXPR_LONG1)`

Asocia la expresión de longitud 1 pasada como parámetro a la expresión.

2.3.4.5 CALL_INSTR_LONG1

- `addRealArg(expr:EXPR_ATOM)`

Añade un nuevo argumento real pasado como parámetro al final de la secuencia de argumentos reales asociados a la instrucción.

- `addRealArg(expr:EXPR_ATOM,index:Integer)`

Añade en la secuencia de argumentos reales asociados a la instrucción, un nuevo argumento real pasado como parámetro, en la posición indicada con el parámetro index. Si existía otro argumento real en esa posición se elimina.

- `getInvokedRoutine():ROUTINE_DEC`

Retorna la rutina invocada en la instrucción de llamada.

- `getRealArg():Sequence<EXPR_ATOM>`

Retorna la secuencia de argumentos reales compuesta de expresiones atómicas pasada en la invocación de una rutina.

- `getRealArg(index:Integer):EXPR_ATOM`

Retorna el argumento real situado en la posición indicada en el parámetro index de la secuencia argumentos reales asociados a la instrucción.

- `removeRealArg(expr:EXPR_ATOM)`

Elimina de la secuencia de argumentos reales asociados a la instrucción, el argumento real pasado como parámetro. Desplaza la secuencia de

argumentos reales situados a la derecha del argumento real eliminado, en una posición a la izquierda.

- `removeRealArg(index:Integer)`

Elimina de la secuencia de argumentos reales asociados a la instrucción, el argumento real situado en la posición indicada con el parámetro index. Desplaza la secuencia de argumentos reales situados a la derecha del argumento real eliminado, en una posición a la izquierda.

- `setInvokedRoutine(proutine:ROUTINE_DEC)`

Asocia la descripción de la rutina pasada como parámetro a la invocación de la instrucción de llamada.

2.3.4.6 CALL_INSTR_LONG2

- `getExprLong1():CALL_EXPR_LONG1`

Retorna la expresión de longitud uno asociada a la instrucción.

- `getInstrLong1():CALL_INSTR_LONG1`

Retorna la instrucción de longitud uno asociada a la instrucción.

- `setExprLong1(pexpr: CALL_EXPR_LONG1)`

Asocia la expresión de longitud 1 pasada como parámetro a la instrucción.

- `setInstrLong1(pinstr: CALL_INSTR_LONG1)`

Asocia la instrucción de longitud 1 pasada como parámetro a la instrucción.

2.3.4.7 CREATION_INSTR

- `getEntity():ENTITY`

Retorna la entidad asociada a la instrucción de creación.

- `setEntity(pentity:ENTITY)`

Asocia la entidad pasada como parámetro a la instrucción de creación.

2.3.4.8 ENTITY

- `getType():TYPE`

Retorna el tipo asociado a la entidad.

- `setType(ptype:TYPE)`

Asocia a una entidad su tipo estático.

2.3.4.9 EXPR

- `getType():Type`

Retorna el tipo asociado a la expresión.

3 Extensión de MOON para dar soporte a las características concretas de Java.

En este apartado se presenta el estudio de las características especificas del lenguaje de programación Java. De esta forma se podrá capturar la información contenida en códigos fuentes escritos en Java sobre MOON. Por ser Java un lenguaje de programación orientado a objetos, las abstracciones básicas del lenguaje están contenidas en MOON, pero existen características concretas del lenguaje que no están soportadas directamente sobre el modelo conceptual, definido en la sección 2.

Las abstracciones definidas en MOON, deben dar soporte a las características concretas de los distintos lenguajes, relevantes respecto a un conjunto de refactorizaciones definidas sobre las abstracciones de MOON. En este sentido se puede considerar las características de los distintos lenguajes de programación orientados a objetos como un dominio del conocimiento común a partir del cuál se pretende definir un conjunto de operaciones primitivas definidas en las refactorizaciones. La solución que permita evolucionar el modelo conforme a una independencia del lenguaje debe dar soporte a estas características especificas además del conjunto de características comunes definidas en MOON.

Un framework es un diseño reutilizable de todo o parte de un sistema software descrito por un conjunto de clases abstractas y la forma en la que esas clases abstractas colaboran [Roberts & Johnson, 1996]. Bajo esta definición, se puede considerar el framework como un soporte que permite abordar al problema de la independencia del lenguaje. Uno de los elementos clave que aparecen en un framework son los puntos de extensión[1], que expresan las partes variables de las abstracciones del modelo, las características concretas de los lenguaje en nuestro caso. La variación se alcanza mediante puntos de extensión de caja blanca, creando clases y métodos abstractos dando soporte a una funcionalidad predefinida de caja negra. De esta forma se modela para generalizar las características comunes en varios lenguajes de programación que son localizadas en las clases del propio framework. La definición del framework, encapsula el conjunto de abstracciones con las características comunes de los lenguajes, más un conjunto de puntos de extensión para expresar las especificas.

En la Figura 3, se identifican los componentes software y sus conexiones para abordar el problema de la independencia del lenguaje bajo una solución basada en frameworks.

[1] Traducción de Hot Spots, referido también como operaciones de enganche en [Gamma et al, 2003] en el patrón de diseño método plantilla.

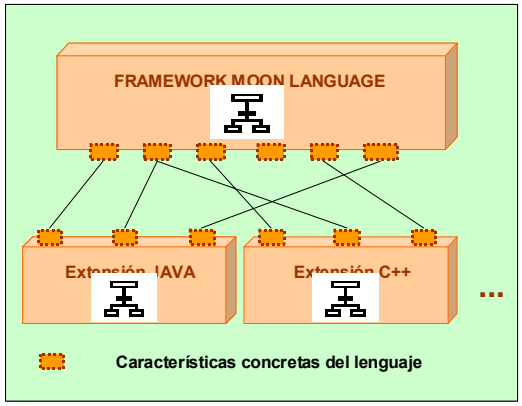


Figura 3 Extensión del framework MOON

El gran poder expresivo de la sintaxis de los distintos lenguajes de programación, hace que el diseño de un mismo programa pueda tener multitud de posibles codificaciones basadas únicamente en variaciones sintácticas del lenguaje. Para minimizar esta situación en MOON se han propuesto un conjunto de simplificaciones que permiten tratar el código de manera más homogénea: eliminación de envíos de mensaje en cascada, eliminación de algunas formas de expresión ver sección 2.2.6, eliminación de sobrecarga ver sección 2.2.1.3. Este conjunto de características basadas en una sobrecarga sintáctica del lenguaje no sirven para identificar un nuevo punto extensión sobre el framework. Para afrontar esta situación se debe definir una transformación de código de forma que se respete la semántica del código original. Estas transformaciones, denominadas simplificaciones sintácticas, adecuan la sintaxis a MOON, y deberán ser realizadas previamente a la captura de la información contenida en sus códigos fuentes.

Existe otro conjunto de características del lenguaje que será necesario ser soportadas en el framework, puesto que las refactorizaciones definidas pueden hacer uso de ellas.

3.1 Simplificaciones sintácticas sobre Java

3.1.1 Eliminación de constructores con parámetros

En MOON la construcción de los objetos siempre se hace bajo el prisma de un constructor sin parámetros, donde se define un estado inicial del objeto. Para poder soportar inicializaciones más complejas donde es necesario el suministro de unos datos iniciales, se añaden métodos auxiliares que permiten obtener un nuevo estado del objeto. Se delega de esta forma en el cliente de la clase la utilización del constructor sin parámetros más la invocación del método que realiza las inicializaciones complejas.

En el caso de Java las inicializaciones complejas se soportan directamente a través de constructores con parámetros, además también se permite una sobrecarga en los constructores.

En el Código 9 se muestra un ejemplo del tratamiento previo que debería darse para poder adecuar la esta característica no soportada por MOON.

Código Original	
Clase Java	Cliente de la clase
`public class Punto2d{`	`class ClientePunto2d{`

`  private int x,y;`
`  public Punto2d(int p_x, int p_y){`
`    x=p_x;`
`    y=p_y;`
`  }`
`}`
	`  private Punto2d p2d;`
	`  ...`
	`  /*Creación de un punto 2D*/`
	`  p2d = new Punto2d(1,2);`
	`  ...`
	`}`

Código Transformado	
Clase Java	Cliente de la clase

`public class Punto2d{`
`  private int x,y;`
`  public Punto2d(){`
`    x = 0;`
`    y = 0;`
`  }`
`  public void crearPunto2dintint(int p_x, int p_y){`
`    x=p_x;`
`    y=p_y;`
`  }`
`}`
	`class ClientePunto2d{`
	`  private Punto2d p2d;`
	`  ...`
	`  /*Creación de un punto 2D*/`
	`  p2d = new Punto2d();`
	`  p2d. crearPunto2dintint(1,2);`
	`  ...`
	`}`

Código 9 Transformación constructor sin parámetros

3.1.2 Construcción polimórfica

En MOON la invocación de una instrucción de creación esta asociada al tipo estático de la entidad (`create entidad`). Esta característica impide una asociación polimórfica directa en el proceso de construcción de un objeto. Para poder realizar asociaciones polimórficas es necesario previamente haber creado los objetos de la subclase y tener una entidad que les sustente para poder después realizar una asignación polimórfica.

El Código 10 ilustra un ejemplo de la transformación de Java para poder soportar las construcciones polimórficas.

Código Original	
Clase Java	Cliente de la clase

`public class Punto3d extends Punto2d{`
`  private int z;`
`  public Punto3d(int p_x, int p_y,int p_z){`
`    super(p_x,p_y);`
`    z=p_z;`
`  }`
`}`
	`class ClientePunto2d{`
	`  private Punto2d p2d;`
	`  ...`
	`  /*Creación de un punto 3D y asociación polimórfica*/`
	`  p2d = new Punto3d(1,2,3);`
	`  ...`
	`}`

Código Transformado	
Clase Java	Cliente de la clase

`public class Punto3d extends Punto2d{`
`  private int z;`
`  public Punto3d(){`
`    z = 0;`
`  }`
	`class ClientePunto2d{`
	`  private Punto2d p2d;`
	`  ...`
	`  /*Creación de un punto 3D y asociación polimórfica*/`

```
  public void crearPunto3Dintintint(int p_x,
       int p_y, int p_z){
    crearPunto2dintint(1,2);
    z =p_z;
  }
}
```

```
  Punto3d p3d;
  p3d = new Punto3d();
  p3d.crearPunto3Dintintin(1,2,3);
  p2d = p3d;
  ...
}
```

Código 10 Transformación construcciones polimórficas

3.1.3 Inicialización de variables de instancia

La declaración de una variable de instancia en Java consiste en un nombre de tipo, seguido del nombre del campo y opcionalmente de una cláusula de inicialización o inicializador, para dar al campo el valor inicial. Aunque los inicializadores proporcionan una gran flexibilidad en la forma de inicializar los campos, sólo son adecuados en esquemas simples de inicialización y siempre pueden ser eliminados con la inclusión de la inicialización en todos los constructores definidos para la clase. Con esta transformación el constructor encapsula todas las inicializaciones de las variables de instancia.

En MOON no existe el concepto de inicializador, por ello en el Código 11 se ilustra un ejemplo de transformación de un inicializador de una variable de instancia, encapsulando la inicialización de la misma en un constructor sin parámetros.

Clase original Java	Clase transformada Java

`public class Persona{`
`  private String nombre=”<sin_nombre>”;`
`...`
`}`
	`public class Persona{`
	`  private String nombre;`
	`  public Persona(){`
	`  nombre = “<sin_nombre>”;`
	`  }`
	`...`
	`}`

Código 11 Transformación de inicializadores de variables de instancia

3.2 Inclusión en MOON de las características específicas de Java

3.2.1 Modificadores de clase

Una declaración de una clase en Java puede estar precedida de varios modificadores de clase que otorgan a la clase unas ciertas propiedades.

- `public`. Una clase public es públicamente accesible. Cualquiera puede declarar referencias a objetos de la clase o acceder a sus miembros públicos. Sin este modificador una clase sólo es accesible desde el mismo paquete.
- `abstract`. Una clase abstract se considera incompleta y no se pueden crear instancias de dicha clase.
- `final`. Una clase final no admite subclases.

Restricción: Una clase no puede ser a la vez final y abstracta.

Para incluir estas características en el framework se propone una evolución que incorpore una clase que encapsule una clasificación de los distintos

modificadores MODIFIER que pueden aparecer en los distintos lenguajes de programación. Además se incorpora un método de consulta para que las abstracciones que puedan contener modificadores puedan acceder a ellos `getModifiers():Collection<MODIFIERS>`. La extensión JCLASS_DEF implementará los métodos para interrogar (`isPublic, isFinal e isAbstract`) por los modificadores basándose en el método `getModifiers` e interpretando la cadena asociado al modificador obtenida a partir de la invocación al método `getLanguageText.` El único modificador considerado directamente en MOON es `deferred`, que es equivalente a al modificador `abstract` de Java.

Como se muestra en Diagrama de clases 10, para realizar una instanciación del framework se añaden los distintos modificadores de Java al nuevo clasificador definido MODIFIER. Además se añade un invariante para soportar la restricción que una clase no puede ser final y abstracta a la vez.

```
context JCLASS_DEF
 inv modifier_java_class_constraint:
  self.isFinal() = True implies self.isAbstract() =
False and
   self.isFinal () = False implies self.isAbstract() =
True
```

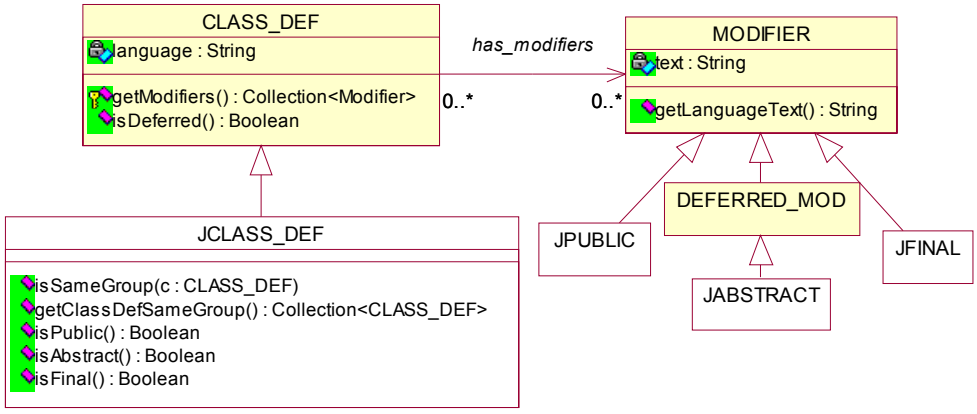


Diagrama de clases 10 Extensión modificadores de clases

3.2.2 Modificadores de variables de instancia y de métodos

Las declaraciones de los campos en Java pueden precederse con modificadores que controlan ciertas propiedades de los campos.

- Modificadores de acceso. Todos los miembros de una clase son siempre accesibles al código de la propia clase. Para controlar el acceso desde otra clase los miembros de una clases tienen cuatro posibles modificadores.
    - Private sólo son accesibles desde la propia clase.
    - Package, los miembros declarados sin modificador de acceso sólo son accesibles desde las clases del mismo paquete así cómo desde la propia clase.
    - Protected, son accesibles en las subclases de la clase, en las del mismo paquete y en la propia clase.

    - Public son accesibles en cualquier parte donde la clase sea visible.
- Static, equivale a una instancia compartida por todos los objetos de una clase. También se denominan variables de clase. Un campo estático puede ser referido dentro de su propia clase, pero cuando se accede a el externamente, se debe utilizar el nombre de la clase.
- Final, una variable final es aquella cuyo valor no puede cambiar después de ser inicializada.
- Transient, indica que un campo no es parte del estado de un objeto persistente y no necesita ser serializado con el objeto.
- Volatile, indica que un atributo es usado por threads sincronizados y el compilador no debe realizar optimizaciones sobre él.

Restricción: Un campo no puede ser a la vez final y volatile.

La cabecera de un método consta de una serie de modificadores que pueden ser:

- Modificadores de acceso que coinciden con los modificadores de campo.
- Abstract, un método abstract es aquel cuyo cuerpo aún no ha sido definido.
- Static un método estático puede acceder sólo a campos estáticos o a otros campos estáticos de la clase.
- Final, un método final no puede ser redefinido en una subclase.
- Native, son métodos implementados en lenguajes externos (por ejemplo C++), y no tienen una implementación en el código fuente Java.
- Syncronized, un método sincronizado fuerza a que la ejecución de dos hilos sea mutuamente exclusiva en el tiempo. Si un hilo invoca a un método synchronized sobre un objeto, en primer lugar se adquiere un bloqueo de ese objeto, se ejecuta el cuerpo del método y después se libera el bloqueo. Otro hilo que invoque un método synchronized sobre ese mismo objeto se bloqueará hasta que el bloqueo se libere.

Restricción: Un método abstracto no puede ser estático, final, sincronizado, nativo ni estricto.

Restricción: Todo miembro de una clase tiene un modificador de acceso.

Las abstracciones correspondientes en MOON vinculadas a estas características del lenguaje son ATT_DEC, METHOD_DEC y la interfaz PROPERTY. Tomando como base la evolución del framework presentada en la sección 3.2.1, se necesita dotar a las abstracciones que pueden ser modificadas, un método que permita tener acceso a los modificadores `getModifiers` y otro que permita añadir nuevos modificadores `addModifier`. Los modificadores introducidos en esta sección extenderán de la clase MODIFIER presentada. La extensiones de las clases JATT_DEC y JMETHOD_DEC implementará los métodos para interrogar (`isPublic, isFinal, isAbstract...`) por los modificadores basándose en el método `getModifiers` e interpretando la cadena asociado al modificador obtenida a partir de la invocación al método `getLanguageText.`

```
context JATT_DEC
 inv modifier_att:
  self.isFinal()= True implies self.isVolatile() = False
and
    self.isFinal()=  False  implies  self.isVolatile  ()=
True
 inv modifier_access:
    self.isPrivate()     xor     self.isPublic()     xor
self.isProtected()
      xor self.isPackage() = True
context JMETH_DEC
 inv modifier_meth:
  self.isDeferred()=  True  implies  self.isStatic()  =
False and
    self.isFinal()=  False  and  self.isSynchronized()
=False and
    self.isNative() = False
inv modifier_access:
    self.isPrivate()     xor     self.isPublic()     xor
self.isProtected()
      xor self.isPackage() = True
```

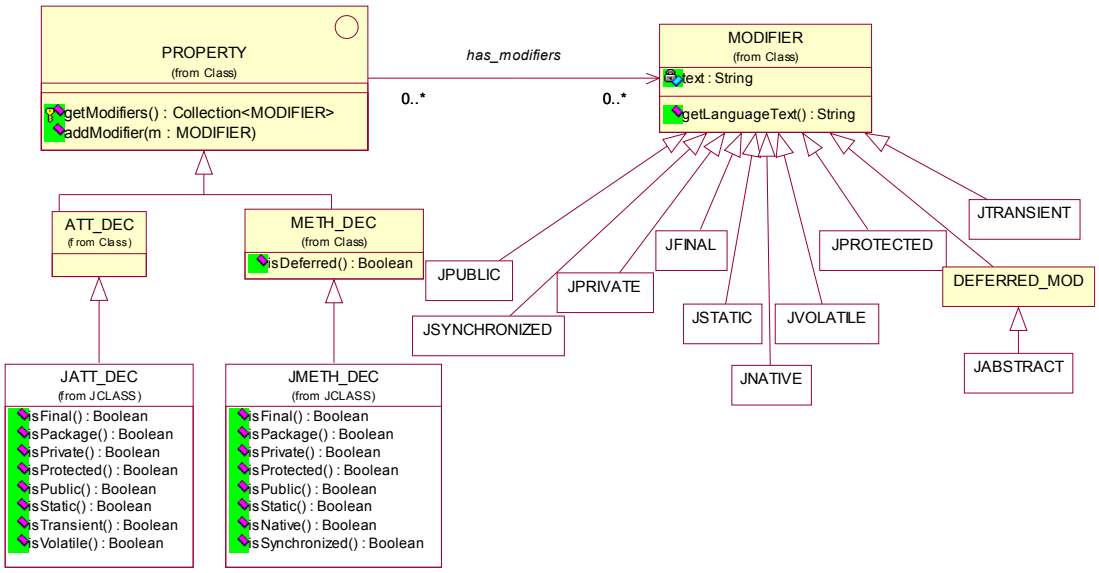


Diagrama de clases 11 Extensión modificadores de atributos y de métodos

3.2.3 Paquetes y visibilidad de paquete

Los paquetes en Java definen unidades software que se pueden distribuir independientemente y combinar con otros paquetes para formar aplicaciones. Los miembros de los paquetes son las clases, las interfaces y los subpaquetes relacionados, y pueden contener archivos de recursos que utilizan las clases del mismo paquete. Al declarar la accesibilidad de las clases e interfaces del nivel superior de un paquete se tienen dos opciones: paquete y público. Los tipos que no son públicos tienen alcance de paquete, es decir, están disponibles para cualquier otro código interior al mismo paquete, pero están ocultos desde fuera del paquete e incluso para el código de subpaquetes. Un

miembro de una clase que no se declara como public, protected o private, puede ser utilizado por cualquier código del interior del paquete.

Restricción El acceso por defecto de un identificador es paquete, excepto para los miembros de las interfaces que son públicas.

En MOON no existe un concepto similar que permita el agrupamiento de clases, por esto se hace necesario el determinar puntos de extensión que permita conocer las clases que pertenecen al mismo paquete que una clase dada. Con esta extensión se podrá tratar la visibilidad package descrita en las secciones 3.2.2, 3.2.1. Los puntos de extensión mencionados se incluyen dentro de la clase CLASS_DEF para ser definidos en la extensión de la clase para el lenguaje Java JCLASS_DEF. No se trata de definir todas las características de los paquetes, únicamente las vinculadas con las visibilidad de las clases y de sus miembros. Los paquetes son referenciados a través de un nombre único, se aconseja por convenio utilizar un nombre de dominio de internet, por ejemplo si una empresa cuyo dominio en internet es magic.com, el nombre de un paquete atrib sería com.magic.atrib. Con esta particularidad se puede representar el agrupamiento de las clases con una clase JPACKAGE como una extensión sobre la clases MOONNAMED.

El Diagrama de clases 12 ilustra la extensión propuesta.

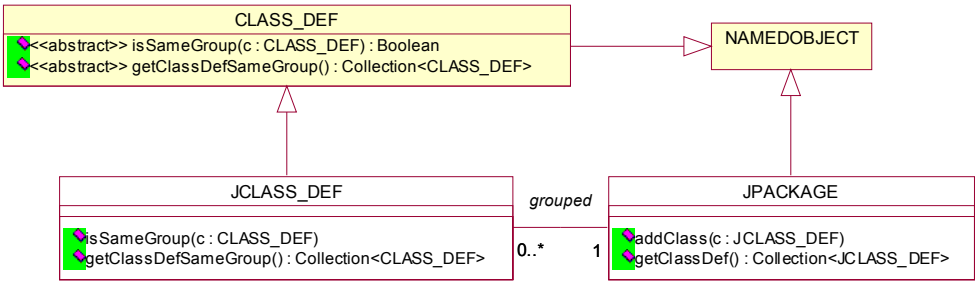


Diagrama de clases 12 Extensión paquetes y visibilidad de paquetes

3.2.4 Tipos primitivos y de referencia

Los tipos en Java se dividen en tipos primitivos (`boolean, char, byte, short, int, long, float, double`) y tipos de referencia (tipos de clases, de interfaces y arrays).

Las interfaces en Java producen nombres de tipos de la misma forma que las clases, por lo que pueden declarar variables de esos tipos. Una interfaz puede declara tres tipos de miembros: constantes (campos), métodos y clases e interfaces anidadas. Todos los miembros de una interfaz son implícitamente públicos, pero por convención se omite el modificador public.

Restricción: Las constantes o campos de una interfaz son implícitamente public static y final, por convención se omiten estos modificadores en las declaraciones de estos campos. Además estos campos deben tener inicializadores.

Restricción: Los métodos que se declaran en una interfaz son implícitamente abstractos y públicos y por convención se omiten los modificadores en los mismos.

Al igual que sucede con las clases una declaración de interfaz puede ir precedida por modificadores:

- public: una interfaz public es accesible públicamente. Sin este modificador una interfaz sería accesible sólo desde su propio paquete.
- abstract: todas las interfaces son implícitamente abstract porque todos sus miembros son abstract, por convención el modificador abstract siempre se omite.

Para soportar en MOON el concepto de interfaz presentado se propone una evolución del framework en el que se incorpore un clasificador dinámico, CONSTRUCTION_MODE, que permita agrupar las posibles variaciones para producir tipos en las distintas construcciones sintácticas de los lenguajes. En MOON la clase es la única construcción sintáctica con la que se permite generar tipos. Las interfaces de Java son construcciones que sirven para generar tipos por tanto pueden considerarse como tipo de CONSTRUCTION_MODE. La extensión concreta de Java JCLASS_TYPE incorpora métodos que permite interrogar si un tipo procede de una interface, Diagrama de clases 13.

Cada tipo de datos primitivo tiene un tipo de clase asociado (clases de envoltura) donde se definen constantes y métodos útiles. MOON no soporta directamente este concepto de tipos de datos primitivos, pero puede tratarse como una especialización de los tipos procedentes de definiciones de clases vacías (CLASS_TYPE). Para incorporar esta característica en el framework, se incorpora un nuevo elemento a la clasificación de construcciones lingüísticas del lenguaje para generar tipos JPRIMITIVE_CONS_MODE, además se añade un método en la clase JCLASS_TYPE, que permita interrogar si un tipo de clase es primitivo, Diagrama de clases 13.

Hay que considerar que los tipos primitivos están previamente definidos en el propio lenguaje de programación, impidiendo la definición de nuevos tipos primitivos a los programadores. Por esto, la información relacionada tanto con tipos primitivos como con las clases básicas del lenguaje, debe ser adquirida antes de capturar la información de cualquier clase definida por un usuario ya que puede tener dependencias sobre ellas. Mientras que la captura de información de las clases de usuario se aplicará un proceso de parseo sobre los códigos fuentes, este conjunto de clases básicas y tipos primitivos pueden ser adquiridos empleando otros procesos de carga de datos.

Cada tipo tiene literales, que son la forma en que se escriben las constantes. Los literales que existentes en Java son: literales de referencia (`null`), booleanos, de carácter, de enteros, en coma flotante, de cadena de texto, y de clase.

En Java para cada tipo existe un objeto `Class` asociado. Los literales de clase permiten nombrar al objeto `Class` de un tipo directamente, añadiendo ".class" al nombre de un tipo. Por ejemplo, el literal `java.util.Iterator.class` referencia el objeto `Class` para la interfaz Iterator. El literal `boolean.class` referencia un objeto `Class` que representa el tipo primitivo boolean.

En el Diagrama de clases 7 Constantes manifiestas de la sección 2.2.6, se mostraron las constantes soportadas en MOON y se ve la correspondencia con las constantes de Java. La única constante no soportada por MOON es la correspondiente a literales de clase, por tanto deberá ser añadida como un nuevo tipo de las constantes existentes JCLASS_CONSTANT. Al igual que en Java, en la conceptualización de MOON para cada tipo existe un objeto

TYPE asociado. Los literales de clase permiten nombrar al objeto TYPE directamente, añadiendo ".class" al nombre único (`uniquename`) de un tipo.

Los tipos de envoltura (wrapper) asociados a los tipos primitivos de Java tienen su correspondencia con los tipos básicos de MOON. Para considerar todos los tipos básicos hay que añadir los tipos Byte, Short, Long y Double.

```
context JCLASS_TYPE def:
  let isInterfaz : Boolean = self.hCONSTRUCTION_MODE ->
     select(c: CONSTRUCTION_MODE|
       m.oclIsTypeOf(JINTERFACE_CONST_MODE))->notEmpty()

context JCLASS_DEF
 inv modifier_interfaz:
  self.hCLASS_TYPE.isInterfaz     =     True     implies
self.isAbstract() = True
 inv modifier_interfaz_att:
  self.hCLASS_TYPE.isInterfaz     =     True     implies
self.hJATT_DEC->forAll
  (att:JATT_DEC|att.isFinal() = True and att.isPackage()
= False and
   att.isPrivate() = False and att.isProtected()= False
and
   att.isPublic() = True and att.isStatic() = True and
   att.isTransient()  =  False  and  att.isVolatile()  =
False)
inv modifier_interfaz_meth:
  self.hCLASS_TYPE.isInterfaz     =     True     implies
self.hJMETH_DEC->forAll
  (meth:JMETH_DEC|meth.isFinal()=False                and
meth.isPackage()=False and
   meth.isPrivate()=False  and  meth.isProtected()=False
and
   meth.isPublic()=True and meth.isStatic()=False and
   meth.isSynchronized()=False and meth.isNative()=False
and
   meth.isDeferred()=True)
```

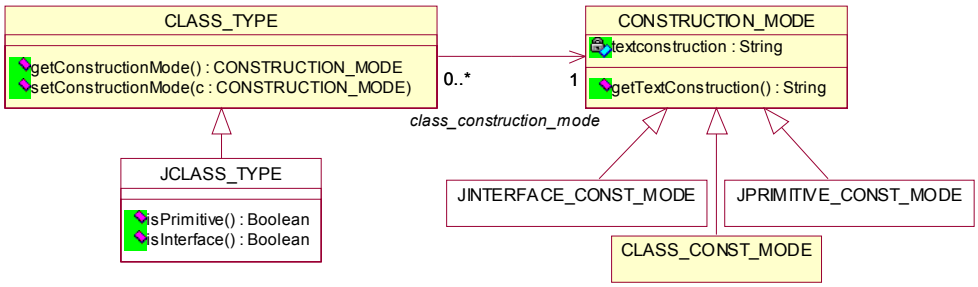


Diagrama de clases 13 Extensión tipos primitivos e interfaces

Los arrays son versiones implícitas de `Object`, que heredan todos sus métodos, esta relación permite polimorfismo en los arrays, es decir se puede asignar un array a una variable de tipo `Object` y reasignarla de nuevo. Los arrays se crean como cualquier otro objeto. La principal limitación de los

arrays es que no se pueden extender para añadir nuevos métodos, en este sentido los arrays se pueden ver como clases finales.

Los arrays utilizan una notación especial para representar sus nombres. Esta notación consiste en un código que representa el tipo de componente del array, precedido del carácter [. Los tipos de los componentes se codifican como se indica en la siguiente tabla.

B	byte
C	char
D	double
F	float
I	int
J	long
Lnombredeclase;	clase o interfaz
S	short
Z	booleano

Tabla 3-1 Notación de tipos de componentes de los arrays

Por ejemplo un array de int se nombra como [I, y un array de Object se nombra como [Ljava.lang.Object;. Un array multidemensional es un array en el que el tipo de sus componentes es a su vez un array. Por ejemplo, una declaración como int[][] se nombra como [[I, es decir un array cuyos componentes se nombran como [I.

Para soportar los arrays en MOON, se pueden considerar como clases genéricas por tanto se corresponden con una instancia de una definición de clase genérica previamente establecida por el lenguaje. Hay que diferenciar esta instancia de la clase Array definida en el paquete java.util.

3.2.5 Excepciones

Las excepciones proporcionan una forma clara de comprobar posibles errores sin oscurecer el código, proporcionando también un mecanismo para señalar directamente la existencia de errores.

Las excepciones en Java son objetos que extienden de la clase `Throwable` o de una de sus subclases (`Exception`, `RuntimeException` o `Error`). Existen dos tipos de excepciones principalmente: excepciones comprobadas, son aquellas que están incluidas en la declaración de un método, excepciones no comprobadas que se corresponden con aquellos errores estándar que se pueden producir en tiempo de ejecución en cualquier fragmento de código. Las excepciones comprobadas que un método puede lanzar se declaran con una cláusula `throws`. En esa cláusula se declara una lista de tipos de excepción separada por punto y comas, de esta forma se permite a un método lanzar varios tipos de excepciones comprobadas.

Redefinición de métodos y lanzamiento de excepciones:

- No se permite que los métodos de redefinición o de implementación (procedentes de herencia de métodos abstractos), declaren más excepciones comprobadas en la cláusula `throws` que las que declara el método heredado.

- Se permite lanzar subtipos de las excepciones declaradas, ya que podrían ser capturas en bloque `catch` correspondiente a su supertipo.
- Si el método de redefinición o de implementación no lanza una excepción, no necesita declararlo.

En MOON no se ha considerado el tratamiento de excepciones. Para poder dar soporte a esta característica, en MOON se incorpora un nuevo punto de extensión `getException`, en la clase METHOD_DEC, que permite conocer el conjunto de excepciones que puede lanzar un método cuando es invocado (ver Diagrama de clases 14). Cada excepción es tratada directamente como un objeto de la clase CLASS_DEF y por tanto tendrá asociada su correspondiente CLASS_TYPE.

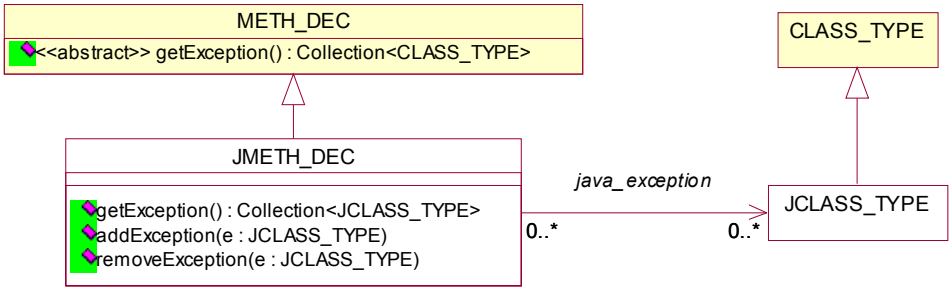


Diagrama de clases 14 Extensión excepciones.

Respecto al lanzamiento de instrucciones se incorpora una nueva instrucción THROW_INSTR, que esta asociada a una entidad cuyo tipo es una excepción.

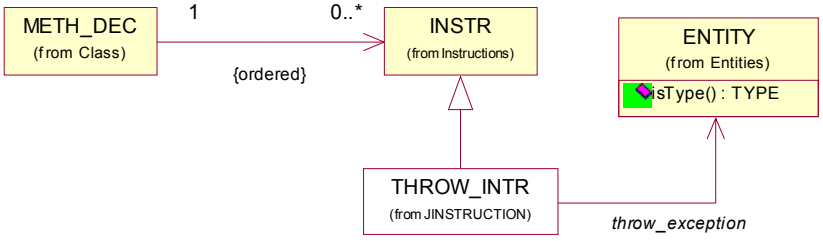


Diagrama de clases 15 Extensión lanzamiento de excepciones

4 Extensión de MOON para dar soporte a las características concretas de GJ (Generic Java)

GJ es una de las propuestas para definir tipos genéricos en Java a través de una extensión del lenguaje [Bracha et al., 1998], [Bracha et al., 1998a], [Bracha et al., 1998b]. Esta propuesta se distingue por enfrentarse al problema del código legado es decir que ¿ qué sucede con el código que explota la genericidad basada en el lenguaje?.

En las secciones 2.2.1.4 y 2.2.4 se presento las abstracciones del framework del lenguaje modelo únicamente en lo referente a genericidad. En este apartado se presentan las características concretas de GJ en lo referente a genericidad y el análisis de la extensión del framework ante estas nuevas características.

4.1 Simplificaciones sintácticas sobre GJ

4.1.1 Acotación múltiple por subtipado

En MOON los parámetros formales sólo pueden estar acotados por un único tipo. Esta característica difiere respecto a GJ que permite una múltiple acotación sobre los parámetros formales. Para poder soportar esta peculiaridad se propone una transformación de código en la cual se añade una nueva clase que implementa los tipos que forman parte de la acotación. De esta forma se puede expresar la acotación utilizando el nuevo tipo creado. En Código 12 se muestra un ejemplo de la transformación propuesta.

Clase original GJ	Clase transformada GJ

`class OrderedList`
`  <T implements Comparable,Sumable>{`
`  ...`
`}`
	`interface CompSum`
	`  extends Comparable,Sumable{}`
	
	`class OrderedList`
	`  <T implements CompSum>{`
	`  ...`
	`}`

Código 12 Transformación acotación múltiple por subtipado

4.2 Inclusión en MOON de las características específicas de GJ

4.2.1 Acotación F

La acotación F es una variante de la acotación por subtipado cuando se trabaja con tipos mutuamente recursivos en el conjunto de parámetros formales de la clase genérica que les contiene.

Un tipo recursivo es aquel cuya especificación esta definida en términos del propio tipo.

Las reglas sintácticas del lenguaje modelo MOON soportan directamente la acotación F, por abordar la acotación de subtipado y los tipos procedentes de instanciaciones genéricas no completas. Sin embargo a la hora de dar soporte sobre el framework, se debe reflejar la semántica de la coexistencia de ambos tipos de acotaciones, subtipado y F.

Para dar soporte a esta nueva característica se añade dos nuevos puntos de extensión sobre la clase BOUNDED_S :

- `isBoundedF():Boolean`, que sirve para interrogar si la acotación por subtipado es una acotación F.
- `FORMAL_PARAM getRecursiveFormalParam()` que permite obtener el parámetro formal recursivo que forma parte de la acotación.

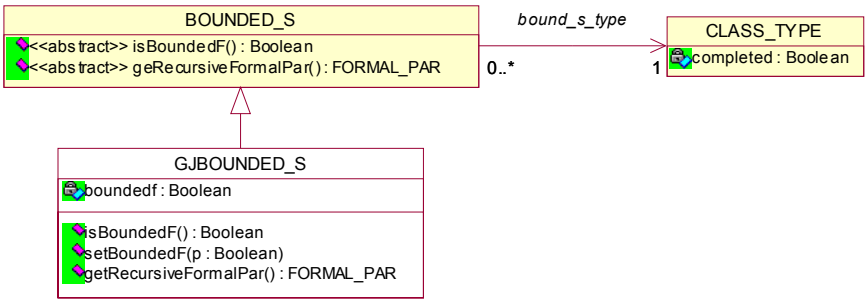


Diagrama de clases 16 Extensión acotación F

Además se añade una restricción. *"Si el parámetro formal esta acotado por la variante F entonces el CLASS_TYPE obtenido a través de la relación bound_s_type no esta completamente instanciado (completed = false)".*

```
context GJBOUNDED_S
 inv bounded_F:
  self.isBoundedF()=   True   implies   self.bCLASSTYPE-
>completed = False
```

En el ejemplo de Código 13 se muestra la definición de una clase genérica con acotación F sobre sus parámetros formales escrito tanto en GJ como en MOON, y en la Diagrama de objetos 10 se presenta una instanciación de dicho ejemplo con la extensión propuesta.

Código Java extensión GJ	Código MOON

`interface ConvertibleTo<A>{`
`  A convert();`
`}`
	`deferred class ConvertibleTo[A]`
	`  signatures`
	`  attributes`
	`  methods`
	`  deferred convert:A;`
	`  body`
	`end`

`class ReprChange`
`  <A implements ConvertibleTo<B>,`
`  B implements ConvertibleTo<A>>{`
`  A a;`
`  void set(B x){`
`  a = x.convert( );`
`  }`
 	
`  B get( ){`
`  return a.convert( );`
`  }`
`}`
	`class ReprChange[A ->ConvertibleTo[B],`
	`  B ->ConvertibleTo[A]]`
	`  signatures`
	`  atributes`
	`  A a;`
	`  methods`
	`  setB (x:B)`
	`  getB:B;`
	`  body`
	`  setB (x:B)`
	`  do`
	`  a := x.convert( );`
	`  end`
	`  getB : B`
	`  do`
	`  result := a.convert();`
	`  end`

	end

Código 13 Definición de una clase genérica con acotación F

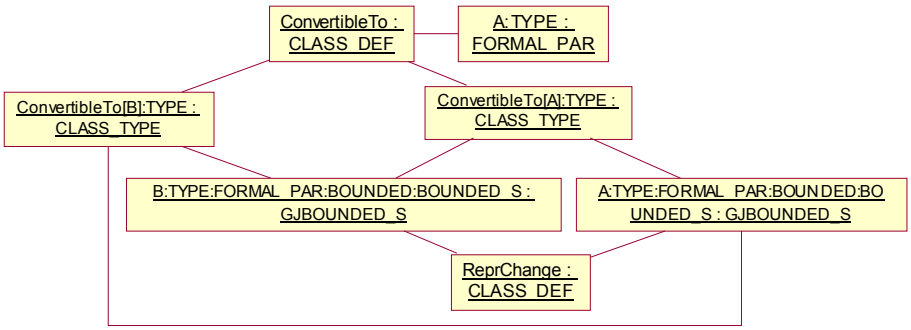


Diagrama de objetos 10 Instanciación de acotación F en el framework

4.2.2 Tipos raw y erasure

Los programas GJ son traducidos a bytecodes de JVM. Este proceso de traducción elimina todos los parámetros formales de la clase genérica mapeando todos los tipos de las variables a sus correspondientes acotaciones e insertando cast donde sea necesario. En este proceso se obtiene el tipo que implementa la genericidad de forma convencional (con el tipo universal) denominado *erasure*. Es decir, el tipo *erasure* de una clase genérica C[T1,...,Tn] es C. En el Código 14 se puede observar un ejemplo de este proceso de traducción.

Tipo parámetrico	Tipo Erasure

`class Cell<A>{`
`  A value;`
`  Cell(A v){`
`  value=v;`
`  }`
 	
`  A get(){`
`  return value;`
`  }`
 	
`  void set(A v){`
`  value=v;`
`  }`
`}`
	`class Cell{`
	`  Object value;`
	`  Cell(Object obj){`
	`  value = obj;`
	`  }`
	
	`  Object get(){`
	`  return value;`
	`  }`
	
	`  void set(Object obj){`
	`  value = obj;`
	`  }`
	`}`

Código 14 Correspondencia tipo paramétricos y tipos Erasure

Para facilitar la integración con el código legado es posible usar como tipo el *erasure* de una clase genérica. Un tipo como este es denominado tipo *raw*. Las variables cuyo tipo estático sea *raw* pueden ser asignadas con tipos procedentes de instanciaciones paramétricas (`Vector = Vector <String>`). La asignación contraria (`Vector<String> = Vector`) esta permitida en GJ a pesar de no ser segura desde la semántica de genericidad (`Vector` podría tener diferentes tipos de elementos). La comprobación de

tipos se produce en tiempo de compilación, la seguridad respecto a estos nuevos tipos es detectada por el compilador, y este emite un `unchecked warning` cuando no la puede garantizar. En el Código 15 se muestra un ejemplo de emisión de este tipo de avisos.

Tipo parámetrico

```
Cell cell = new Cell<String>("abc");
cell.get();
cell.set("cdf");
unchecked call to method set(A) as a member of the raw type Cell
cell.set(new Integer(5));
unchecked call to method set(A) as a member of the raw type Cell
Cell<String> cell1 = cell;
unchecked assignment: Cell to Cell<java.lang.String>
```

Código 15 Chequeo de tipos RAW en GJ

Los RAW tipos pueden ser considerados cómo el único tipo procedente de la definición de clases genéricas. En el modelo expuesto existen dos asociaciones entre CLASS_DEF y CLASS_TYPE, una para representar las instanciaciones genéricas del tipo *generic_instantation*, y otra para representar el tipo asociado a la definición de la clase *has_type*. Para soportar esta nueva característica hay que añadir un nuevo tipo que extienda de CLASS_TYPE denominado GJ_RAWTYPE. De esta forma se mantiene la asociación *has_type* relaciona la definición de la clase genérica con su tipo RAW.

5 Conclusiones y líneas de trabajo futuras

Como conclusión fundamental de este trabajo es que el modelo conceptual propuesto permite representar las abstracciones definidas en el lenguaje MOON. Consecuentemente se posibilita la captura de información de un sistema software para poder llevar a cabo un conjunto de refactorizaciones.

En lo referente a independencia del lenguaje, se han reflejado un conjunto de puntos de extensión para almacenar las características concretas del lenguaje Java y su extensión GJ. Estas características pueden ser necesarias en la definición de una refactorización a través de precondiciones, acciones y postcondiciones que se deben satisfacer para llevar a cabo una refactorización.

Como línea de trabajo futuro se propone el estudio de las características particulares del lenguaje de programación Java que no han sido tratadas:

- Inicializadores de arrays
- Inicializadores estáticos
- Inicializadores de interfaz: Si una interfaz requiere datos compartidos y modificables, se puede lograr este efecto utilizando una constante con nombre que se refiera a un objeto que almacene los datos.
- Acceso miembros estáticos a través del nombre de la clase.
- Clases Internas y anónimas

La evolución de MOON en lo referente a la búsqueda de nuevos puntos de extensión esta determinada por el estudio de más lenguajes de programación

orientados a objetos, en este sentido se pueden considerar analizar nuevos lenguajes .Net Common Language Runtime.

En lo referente a refactorizaciones , es necesario modelar en el framework un soporte donde se puedan almacenar la definición de las mismas.

Otras líneas de trabajo futura, enfocada fundamentalmente en el diseño de herramientas que asistan el proceso de refactorizaciones de código, vienen dadas por:

- Generación de código del lenguaje original una vez realizada la refactorización. Este problema es debido a que MOON no refleja todas las abstracciones que pueden aparecer en código de los lenguajes concretos.
- Establecimiento de un esquema de almacenamiento persistente para el repositorio de clases candidatas a refactorizar.

Bibliografía

[Aksit et al, 1999] Aksit Mehmet, Tekinerdogan Bedir, Marcelloni Francesco, Bergmans Lodewijk. " Deriving Object-Oriented Frameworks from Domain Knowledge" In conference ECCOP 1999 pages367-378

[Beck 1999] Beck K." Extreme Programming explained: embrace change." Ed. Addison-Wesley 1999.

[Bracha et al., 1998] Bracha,G., Odersky, M., Stoutamire D., Wadler, P (1998) . "GJ Specification"

[Bracha et al., 1998a] Bracha,G., Odersky, M., Stoutamire D., Wadler, P (1998) . "GJ: Extending the Java programming language with type parameters".

[Bracha et al., 1998b] Bracha,G., Odersky, M., Stoutamire D., Wadler, P (1998). "Making the Future Safe for the Past: Adding Genericity to the Java Programming Language.". To appear in Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications, October 1998

[Crespo 2000] Crespo , Yania (2000). " Incremento del potencial de reutilización del Software mediante refactorización". PhD thesis, Departamento de Informática, Universidad de Valladolid

[Crespo 2001] Crespo, Y., Cardeñoso V., Marqués J.M. "Un lenguaje Modelo para la definición y análisis de refactorizaciones" . En actas PROLE'01, Almagro, España. Noviembre 2001.

[Demeyer et al., 1999] S. Demeyer, S. Tichelaar, and P. Steyaert. "FAMIX 2.0 the FAMOOS information exchange model." technical report, University of Berne, Aug. 1999.

[Fowler et al., 1999] Fowler M., Beck K., Brant J., Opdyke W., Roberts D. "Refactoring: Improving the Design of Existing Code" Ed. Addison-Wesley 1999.

[Gamma et al, 2003] Gamma E, Helm R., Johnson R, Vlissides J. "Patrones de diseño. Elementos de Software orientado a objetos reutilizables". Ed Addison Wesley 2003.

[Gosling et al., 2000] Gosling, J., Joy, B., Steele, G., Bracha G. "The Java ™ Language Specification.Second Edition.". Addison-Wesley June 2000

[Gupta et al., 2000] Gupta S., Hartkopf J., Ramaswamy S. "Covariant specialization in Java". Publicado en JOOP (The Journal of Object-Oriented Programming), Mayo 2000, Volumen 13, número 2 por SIGS Publications.

[Judson et al, 2003] Judson S.R.,Carver D. L., France R."A Metamodeling Approach to Model Refactoring" OOPSLA 2003. Octubre

[Lanza & Ducasse, 2002] Lanza Michele and Ducasse Stephane. "Beyond Language Independent Object Oriented Metrics: Model Independent Metrics" 2002, 6 th ECOOP Workshop on Quantitative Approaches in Object - Oriented Software Engineering.

[Lieberherr y Holland,1989] Lieberherr K. y Holland I. "Assuring good style for object-oriented programming". IEEE Software, 6(5):38-48.

[Lieberherr et al.,1988] Lieberherr K., Holland I., Riel A. "Object-oriented programming: An objective sense of style". In conference Proceedings of OOPSLA'88, pages 323-334.

[Mens et al, 2002] Mens, Tom, Serge Demeyer, and Dirk Janssens. "A Graph Rewriting for Object-oriented Software Refactoring". FWO-WOG, January 2002.

[Mens & Lanza 2002] Mens, Tom, Lanza Michele. "A Graph-Based Metamodel for Object-Oriented Software Metrics". Published in Electronic Notes in Theoretical Computer Science, Volume 72, Number 2, Elsevier Science, 2002

[OMG, 2003] Object Management Group (OMG). "Unified Model Language Specification. UML version 1.5". Marzo 2003. http:\\www.omg.org.

[Roberts & Johnson, 1996] Roberts Don, Johnson Ralph. "Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks". In Patterns Languages of Programming Design, Addison-Wesley Publishing Co 1996, pages 471-486.