Tesis de licenciatura

Reificación de Relaciones en lenguajes orientados a objetos

Andrés Taján y Pablo A. Revert

Diciembre de 2009

Departamento de Computación

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

{atajan, pablo.revert@gmail.com}

Director

Dan Rozenfarb

"When two objects, qualities, classes, or attributes, viewed together by the mind, are seen under some connection, that connection is called a relation"

- Augustus De Morgan
Matemático y lógico británico
Formuló las leyes de Morgan

Resumen

Generalmente, cuando apreciamos que dos entidades mantienen un *vínculo*, decimos que se encuentran relacionadas. De esta forma podemos comprender el significado de "Juan *es padre de* Pedro". Si más tarde se nos presenta el vínculo "Mario *es padre de* Jorge", vemos que éstas poseen cierta semejanza. Llamamos *Relación* a esta *idea de semejanza*. Una *Relación* describe la "forma" que tendrán los *vínculos* que de ella se desprendan.

Los lenguajes y técnicas de modelado orientado a objetos (OO) -como UML- han popularizado el uso de Clases y Relaciones para describir y diseñar software. Sin embargo, los lenguajes de programación carecen de elementos sintácticos y semánticos para expresar Relaciones directamente. Esto lleva al programador a implementarlas en forma ad-hoc y facilita que la abstracción se pierda en el código.

En esta tésis estudiamos las consecuencias de reificar la abstracción "Relación" en los lenguajes de programación OO. Así, mientras desarrollamos un modelo conceptual, prototipamos un framework en Smalltalk Squeak.

Comprobamos que el ambiente resultante ofrece importantes beneficios al pensar y elaborar programas. Un estilo de programación más declarativo, con transiciones más naturales entre un diseño e implementación; nuevas posibilidades al definir atributos y colaboraciones entre objetos (gracias a las a la reificación de *relación*, *rol* y *vínculo*); una sencilla integración con herramientas de de nivel meta (ej. persistencia, interfaz de usuario, distribución); soporte a sharing anticipado y no anticipado; objetos con protocolos uniformes, inferidos de la mera definición de la la relación; son algunas de ellos.

Finalmente, lo prometedor de los resultados obtenidos y la naturalidad y fluidez con que el modelo evolucionó en forma paradigmática hacia otras áreas o dominios de problema (ej. Programación orientada a roles, Traits, Metaprogramación) invita a continuar explorando las posibilidades y límites de esta línea de investigación.

Abstract

When two objects or entities under seen under some connection, we commonly say that they are related. In this way we can understand what does “John *is father of* Peter” means. Later, when someone tells us that “Karl *is father of* George”, we can quickly note a resemblance between both facts. That *resemblance idea,* is what we call *Relationship.* A *Relationship* describes the “shape” that her *links or connections* will have.

Object-oriented (OO) modelling languages and techniques –like UML-, have made popular the use of Classes and Relationships to describe and to design software. However, OO programming languages lack syntactic and semantic elements to express these relationships directly. This leads the programmer to do ad-hoc implementations and makes it easier for the abstraction to get lost in the code.

In this thesis we study the results of reifying the abstraction of "Relationship" in OO programming languages. Thus, as we develop a conceptual model, we make a framework prototype in Squeak Smalltalk.

We realized that the new environment offers significant benefits upon thinking and developing programs. A programming style more declarative, with natural transitions between a design and its implementation; a new better chances upon defining attributes and collaborations between objects (through relationship, role and links reifications); an easy integration with meta-tools (e.g., persistence, user interface, distribution); support for anticipate as well as unanticipated sharing; and more complete and standard protocols (as a consequence of relationships declarations); are some of them.

Finally, the promising of results obtained and the naturalness and fluency with which the model has evolved in paradigmatic way towards other areas or problem domains (e.g. role-oriented programming, Traits and Meta-programming) invites to continue exploring the chances and limits of this research.

Agradecimientos

El presente informe no solo marca el fin de nuestra tesis, sino también el de nuestra carrera de Llicenciatura en Ciencias de la Computación. Es por ello que deseamos agradecer tanto a quienes nos apoyaron y colaboraron durante este trabajo, como a quienes estuvieron cerca nuestro, acompañándonos en el transcurso de nuestra carrera.

A quienes colaboraron en este trabajo

A Dan, quién desde el momeno en que nos propuso el tema y aceptó dirigirnos, confió en nosotros, guiándonos y dándonos libertad para evolucionar y respetando en todo momento nuestros tiempos.

A Daniel Altman y Hernán Tylim por ofrecernos y permitirnos utilizar su framework de Conjuntos Semánticos SSet durante la etapa de prototipación de nuestro estudio.

A Nacho Lo Russo, Ana Merlino, y Mariana Lo Russo, por ayudarnos a organizar y mejorar la calidad estructural y gramatical de este informe.

A todos aquellos que luego de preguntarnos "¿y de que se trata la tesis?" supieron tener la paciencia de escucharnos y tratar de entendernos, permitíendonos madurar las ideas que hasta el momento sólo estaban en nuestra cabeza, y que hasta supieron dispararnos otras, como cuando Feli, tía de Pablo, nos hizo ver la diferencia entre comportamiento accidental y escencial.

A nuestra casa de estudio

A la Facultad de Ciencias Exactas y Naturales por enseñarnos a pensar de otra forma, por abrirnos la cabeza. Por enseñarnos a ver lo importante, el conocimiento, y dejar de lado todo el resto. A la Universidad de Buenos Aires, y a la República Argentina, por permitirnos tener acceso a una educación laica, plural, pública, gratuita y de primer nivel, la cual creemos es base piramidal de la democracia, de la igualdad social y del país que queremos construir.

Agradecimientos personales de Andrés

A Pablo, por la paciencia y confianza con que supo acompañar mis puntos de vista, aún cuando éstos no estuvieran alineados con sus creencias o convicciones.

A Sole, que me alentó a reanudar el proyecto tesis en un momento en que ya hacía tiempo no se encontraba entre mis prioridades, proyecto que luego supo apoyar y acompar con paciencia y tolerancia, a sabiendas que éste postergaba otros proyectos importantes de nuestra pareja.

A Andrés, Martín, Luciano (el Curti) y Pablo, compañeros incansables de grupo en la facu -hoy convertidos en amigos de fierro-, de quienes aprendí muchísimo durante los años de la carrera, y que sin su amistad la facu no me hubiese resultado tan agradable. A sus papás y a la Peti, que siempre me recibieron y atendieron de 10, haciéndome sentir "en casa", aún en situaciones donde las largas jornadas de estudio entorpecieren planes, actividades o simplemente la dinámica familiar.

A Toti y el Chulo, dos de mis grandes amigos, que siempre han estado y ayudado desde distíntos ángulos. Ambos super presentes desde mi llegada a Buenos Aires y arranque del CBC.

A mis amigos de la secundaria -Valen, Adrián, Mauro, Pablo y las chicas-, con quienes transité grandes e importantes momentos de mi vida y formación.

A mi hermano Nacho, por acompañarme y apoyarme siempre, como en los los momentos de estudio donde para no aburrirse mientras estudiábamos, nos cebaba mate y cocinaba.

A mis viejos, Ana Lía y Negro, por bancarme en mi decisión de estudiar esta carrera. Por haber vivido cada uno de mis exámenes como si fuera propio y por su respeto ante mis decisiones de priorizar otros aspectos de mi vida por sobre la finalización de mis estudios, tan valorado por ellos.

A los chicos de la DGE, Vani, Adriano, Julián, Edu, Nat, Majo, Hernán, que siempre han estado presentes aunque más no sea con esa pregunta sentida "Tuqui, ¿cómo vas con la tesis?". A mi grupo de tenis –Juli, Augusto, Lau, Gastón y Lou- que aunque efímero en cuanto al tenis, supo acomañar un 2005 de cambio en mi vida –cambio que se renueva hasta hoy.

A Julián Dunayevich, Dan Rozenfarb y Ana Merlino de quienes he aprendido muchísimo, y que gracias a a la confianza y libertad me brindaron, me han permitido madurar y cruzar naturalmente el puente (sin dejar de ver la otra orilla) entre la formación teórica y académica de la facu y, la realidad y pragmatismo de la dinámica laboral-profesional.

A Pert Consultores, por permitirme disponer nada menos que de 8 horas semanales durante el último año y medio para completar este trabajo.

Me resta nombrar a muchos -amigos, familiares y compañeros de trabajo- en quienes siempre he encontrado apoyo y una palabra de aliento no solo para este trabajo sino también para otros aspectos de mi vida.

A todos, muchas gracias.

Agradecimientos personales de Pablo

A quienes me acompañaron en la previa a esta tesis, a mis compañeros de facultad. A ellos que nunca les importó si asistía o no a clase, que nunca se fijaron en cuanto tiempo les daba sino que aceptaban lo que les podia dar y me ayudaban con todo lo que me hiciera falta.

Al Curti, Luciano Burotti, por esas madrugadas de mate y estudio, por la actitud que siempre le puso y que nos inspiraba a todos. A Emiliano Real por esas largas horas de estudio y trabajo compartido, por esas útlimas brazadas que aunque parezcan pocas son las que más cuestan. A Emilio Platzer, que allá en los inicios de la carrera me hizo conocer "la exactas" que yo amo, aquella de la que hablaba mi vieja.

A mi compañero de tesis, por el Dune y tantas otras excusas que nos sirvieron para no dormir durante días enteros, las cuales fueron sólo el comienzo de varios otros proyectos. A Nachito, quien supo no solo soportarnos durante toda la carrera, sino que me ha hecho sentir siempre mas que bienvenido, como un hermano mas en su casa.

A Pocho, Jorge Poccioni, que hizo cosas inimaginables para que yo me recibiera. A mis compañeros de trabajo, que siempre me aguantaron mi ausencias por estudio, mis noches sin dormir y todas las otras consecuencias laborales de mis épocas de estudio extremas. En especial a Maximiliano Carratello, a Fernando Perla, a Hernán Mancasola, a Jose Dogherty y a Mariano Cuello, quienes siempre me cubrieron y apoyaron, y por quienes me sentí siempre muy respaldado. A Pablo Vispo, que cada vez que le pedí una mano se encargó de darme el espacio necesario para que pudiera estudiar. A Mauro Mambretti que me alentó siempre a que terminara la carrera, y dio prioridad a ello por sobre el trabajo.

A Alfredo, que a lo largo de todos estos años ha sido una persona muy especial en mi vida, y que me ha acompañado, alentado y enseñado en cada acto de ella.

A mi Viejo, que a medida que pasa el tiempo lo siento mas presente que nunca, y del cual casi sin darme cuenta, sigo aprendiendo día a día. A mi vieja que me ha iniciado en algo tan bello como es la computación, me ha dado la mayoría de los valores que hoy tengo y de la cual sigo aprendo día a día tantas cosas que ni ella se imagina. A mi hermano, que me ha alentado siempre, me ha sabido acompañar en todo momento y tolerar en esas épocas de estudio y trabajo furiosos. A él, que con su ejemplo me ha enseñado que nunca hay que bajar los brazos, que todo es posible. A Bebe, por tantas velas prendidas.

A mi familia que me hizo quien soy, y me dio las armas necesarias para enfrentar esta vida. A mi familia, la de sangre y la política, por siempre confiar en mí y darme el apoyo tan necesario en esta vida.

A la petisa, mi esposa, a quien amo con locura. A ella, quien me aporta el balance que mi vida necesita, y quien siendo generalmente la mas perjudicada de todas mis ausencias, me sabe acompañar y alentar en cada proyecto que emprendo.

A mi hijo , que aunque aún no lo sepa, es quien me da la fuerza para emprender todo aquello que por emprender me queda.

A todos, muchas gracias.

Contenidos

Capítulo 1

Introducción

Generalmente, cuando apreciamos que dos entidades mantienen un *vínculo*, decimos que se encuentran relacionadas. De esta forma podemos comprender el significado de "Juan es padre de Pedro" o de "Argentina está en America del Sur". Si más tarde se nos presenta el vínculo "Mario es padre de Jorge", vemos que éste posee cierta semejanza con el de Juan y Pedro. Llamamos *Relación* a esta *idea de semejanza*. Una *Relación* describe la "forma" que tendrán los *vínculos* que de ella se desprendan.

La idea de *Relación* ha sido ampliamente utilizada por la Ciencia de la Computación. Diagramas de Entidad-Relacion (ER), Bases de Datos Relacionales y el Lenguaje Unificado de Modelado (UML) son algunos de los más recientes ejemplos. En el caso de los lenguajes de Programación Orientados a Objetos (POO), diversos autores [Goldberg and Robson 1983][Alpert et al. 1998][Rumbaugh 1987] coinciden en que el mismo Smalltalk hace uso de la idea de *Relación* al representar la *Instancia*, *Herencia*, *Referencia* y *Observación*.

En particular, en la POO las relaciones resultan útiles al permitir abstraer en forma natural los vínculos e interacciones entre objetos. Sin embargo, los lenguajes de POO no contienen sintaxis o semántica para expresar las relaciones directamente. Si bien cualquier programa puede implementar relaciones particulares en forma ad-hoc, la abstracción puede perderse en el mecanismo de implementación.

Al notar esta ausencia, algunos autores [Rumbaugh 1987][Kolp 1997][Noble and Grundy 1995][Bierman and Wren 2005] propusieron *reificar*[1] la *Relación* en los lenguajes de POO con el objetivo de llevar esa "idea de *Relación"* a un nivel superior, en el que pueda ser formulada y manipulada explícitamente.

Rumbaugh [Rumbaugh 1987] fue pionero al describir el problema, sobre el que trabajó en un lenguaje al que llamó DSM [Shah et al. 1989]. Más tarde, Kolp [Kolp 1997] definó un meta-modelo de relaciones en objetos y presentó una posible arquitectura de solución sobre cómo implementarlo en un lenguaje con reflexividad como CLOS. En 2005, Bierman y Wren [Bierman and Wren 2005] trabajaron en una especificación formal para introducir Relaciones en un lenguaje fuertemente tipado. En este sentido desarrollaron un lenguaje prototípico y reducido basado en un subconjunto funcional de Java.

Estos trabajos coinciden en la importancia de complementar los lenguajes de POO con Relaciones y en gran mediada sobre como continuar sus investigaciones: especialización entre relaciones, propiedades matemáticas (ej. reflexividad, transitividad), relaciones derivadas o deducidas, y el potencial de construir nuevas herramientas a partir de estas, son algunos puntos recurrentes.

Tomando como punto de partida estos estudios, nuestro trabajo se centró en explorar, profundizar y analizar las posibilidades que brinda el incoporar Relaciones de primera clase en un ambiente de programación puramente orientado a objetos.

---

[1] ***Reificación***. Etimológicamente, el término *Reificación* deriva del Latín *res* (cosa) + *facere* (hacer). *Reificación* puede ser "traducido" como hacer-cosa, el hecho de convertir algo abstracto en una cosa u objeto concreto.

Para explorar el dominio procedimos a la prototipación de un Framework de Relaciones en Smalltalk Squeak, con el objetivo de hacer madurar en forma ágil e iterativa un modelo de Relaciones expresado puramente en objetos. La única condición que nos planteamos fue dar prioridad a lo conceptual por sobre los aspectos de performance. En todo momento nos permitimos desarrollar aquellas herramientas que consideramos necesario para mostrar, validar o complementar cierto aspecto del modelo. Fue así como nuestra filosofía de trabajo nos permitió hacer evolucionar libremente nuestra investigación para llegar naturalmente a los resultados obtenidos.

1.1 Glosario y acrónimos

Reicifación	Según la enciclopdia Encarta [Encarta 1996] el término empleado en filosofía que significa, etimológicamente, 'convertir algo en cosa', 'cosificar'. En cierto modo, puede entenderse como un proceso de transformación de representaciones mentales en cosas, o de sujetos en cosas. En los procesos de reificación, las nociones abstractas son concebidas como objetos. Pero también los sujetos y las relaciones dinámicas pueden quedar convertidas en elementos estáticos, que pueden ser sometidos a un dominio como si se tratara de cosas o elementos estáticos.
Aplicación	Es un software creado para resolver un problema particular de un Dominio determinado
Dominio	Ver Dominio de Problema
Dominio de Problema	Recorte o fracción de la realidad que estamos interesados en modelar
Framework	Según Rozenfarb [Rozenfarb, 2001] un Framework es un conjunto de objetos cooperantes conformando un diseño que encarna la teoría de un dominio. Este diseño debe ser además reusable, o en otras palabras, un Framework debe estar preparado para facilitar su aprovechamiento en nuevas aplicaciones que utilicen dicho dominio.
LMOO	Acrónimo, referente a los Lenguajes de Modelado Orientados a Objetos.
LPOO	Acrónimo, referente a los Lenguajes de Programación Orientados a Objetos.
ML	Acrónimo, referente a Method Lookup.
Ontología	(Del gr. □v, □ντος, el ser, y -logía). Según la RAE es la parte de la metafísica que trata del ser en general y de sus propiedades trascendentales. Representa el estudio filosófico de la naturaleza del ser, su existencia y realidad. La ontología trata con preguntas que conciernen qué entidades existen o se puede decir que existan, y cómo tales entidades pueden ser agrupadas, relacionadas dentro de una jerarquía o subdivididas de acuerdo a similitudes y diferencias. Para la Ciencia de la Computación, una ontología es una representación formal de un conjunto de conceptos dentro de un dominio y las relaciones entre esos conceptos.

OO	Acrónimo, referente a lo Orientado a Objetos
POO	Acrónimo, referente la Programación Orientada a Objetos.
PPOO	Acrónimo, referente al Paradigma de Programación Orientado a Objetos.
Smalltalk	Smalltalk es un ambiente de programación orientado a objetos. Hay muchas implementaciones de Smalltalk, para distintas plataformas, comerciales, open source. Smalltalk es un ejemplo puro del paradigma de orientación a objetos. Es decir, todo lo que hay que en el ambiente son objetos y mensajes. El primer Smalltalk fue creado por un equipo liderado por Alan Kay e integrado por Dan Ingalls, Ted Kaehler y Adele Goldberg
Squeak	Squeak es una versión de Smalltalk moderna y open-source, a partir de la cual se encolumnó una gran comunidad de desarrolladores muy activa actualmente, que continúa mejorándolo y aportando ideas y desarrollos, lo cual hace de Squeak un ambiente en continua evolución. Squeak fue creado en 1995 en el marco de un proyecto que tenía por objetivo experimentar en un software educativo y construir el Dynabook. El proyecto estaba conformado por Alan Kay, Dan Ingalls, Ted Kaehler, John Maloney y Scott Wallace.
Software	Conjunto de objetos que colaboran enviándose mensajes

1.2 Convenciones y Notación

1.2.1 Tipografías

Durante este trabajo se utilizan las siguientes convenciones tipográficas.

Tipo	Tipografía
Clase del Modelo de Relaciones	**RRelation**
Clase fuera del Modelo de Relaciones	`Person`
Selector / Método / Mensaje	`#fatherOf:`
Instancia de Relación	*FatherChild*
Instancia de Clase	*juan*
Extracto / Ejemplo de Código	

```
| rel |
rel := RRelation new
    name: 'FatherChild'
    yourself.
```

Nota / Observación	**Nota.** Esto es una nota
Referencia Bibliográfica	[Booch et al. 1999]

1.2.2 Notación gráfica de Relaciones

Para la notación gráfica de relaciones se utilizó una representación basada en la definida por UML [Booch et al. 1999], con diferentes personalizaciones.

A continuación se presentan las más comunes y frecuentes.

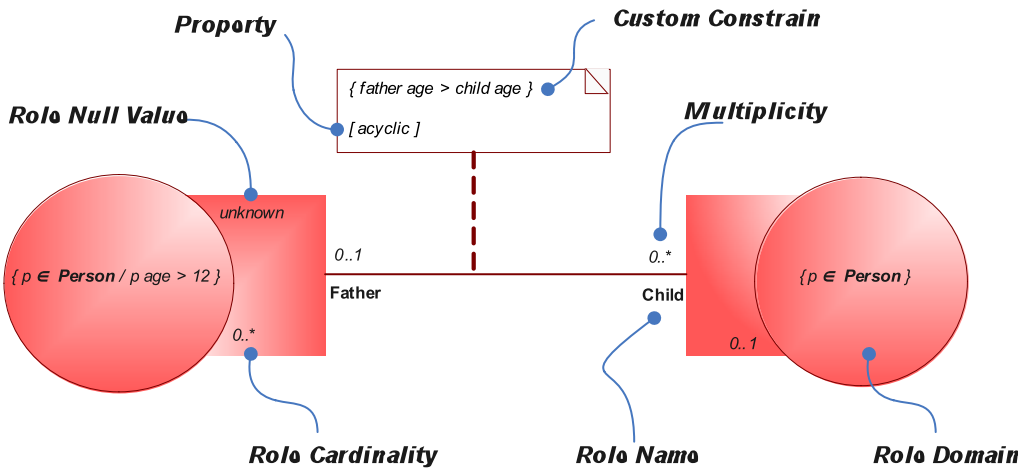


Figura 1. Notación gráfica de Relaciones

En el transcurso del informe, la presentación de nuevas características irá acompañada de su correspondiente representación o definición gráfica.

1.2.3 Notación Smalltalk

Para la notación del código Smalltalk ver [Goldberg et al., 1989].

1.2.4 Uso del Inglés

A los efectos de favorecer el polimorfismo en el modelo resultante, se utilizó inglés tanto para el nombre de las clases, de las relaciones y de los objetos. También se lo usó en la definición de los protocolos .

1.3 Organización del informe

En lo que resta de este primer capítulo presentaremos un resumen sobre el estado del arte. En él mostramos cómo los LPOO representan actualmente las relaciones. También analizamos el impacto en los lenguajes de modelado OO (LMOO), y cómo estos ya contemplan desde hace tiempo la abstracción Relación como una construcción de primera clase. Finalmente introducimos los principales elementos de los trabajos realizados sobre esta área de estudio.

En en próximo capítulo describimos el Modelo de Relaciones y exponemos el prototipo de Framework desarrollado. En todo momento intentamos expresar las decisiones que nos llevaron a dichos resultados. Luego mostramos una serie de Herramientas que ayudan a complementar tanto el Framework como a terminar de brindar los elementos para considerar madura la reificación del concepto Relación. En el último apartado presentamos un conjunto de consecuencias que se desprenden de esta etapa del trabajo

El capítulo 3 introduce lo que denominamos Modelo de Comportamiento basado en relaciones. La concepción de este modelo resultó producto de distintos indicios e ideas que nos surgieron durante la maduración del Modelo de Relaciones. Por ello, y en función de nuestra filosofía de trabajo optamos por extender nuestra área de investigación y así analizar sus posibilidades. Al igual que en el capítulo anterior, complementamos el trabajo con distintas herramientas y sobre el final mencionamos algunas de sus principales consecuencias.

El capítulo 4 presenta los resultados de las pruebas de concepto que hemos llevado adelante. La primera de estas pruebas está destinada a validar el potencial del modelo de Relaciones para integrarse a nuevas Herramientas. La siguiente prueba de concepto ayuda a comprender la utilidad práctica de modelar cierto dominio de problema en un ambiente de POO con Relaciones. Por último se expone como afecta el Modelo de Comportamiento soluciones conocidas como el Pattern de State.

En el capítulo 6 se discute tanto el Modelo de Relaciones como el de Comportamiento con trabajos, tecnologías y áreas de estudio en común. Se comparan trabajos sobre Reificación de Relaciones, investigaciones sobre programación orientada a roles, Traits y frameworks de meta-herramientas.

Finalmente, en el último capítulo se presenta la conclusión y se detalla un el conjunto de trabajos futuros para continuar con esta investigación.

1.4 Estado del Arte

Nuestro estudio se enfocó sobre el dominio de las Relaciones en el paradigma de programación orientado a objetos. Por ello consideramos necesario describir la manera en que los actuales lenguajes permiten representar relaciones y descibir las falencias de estos mecanismos. También vimos apropiado describir los lenguajes de modelado que actualmente son utilizados para diseñar software sin dejar de mencionar las algunas herramientas que los utilizan para generar código. Finalmente analizamos las características principales de trabajos sobre el tema.

1.4.1 Lenguajes de programación

Los lenguajes de POO actuales[2] no ofrecen en forma nativa construcciones semánticas de primera clase que permitan representar y manipular Relaciones en forma explícita.

En otras palabras, los lenguajes orientados a objetos basados en clases como Smalltalk proveen elementos que permiten representar directamente las Clases, sus atributos y métodos, pero a excepción de *InstanceOf*, *Inheritance y Reference*, no contienen sintaxis o semántica para expresar Relaciones propias del dominio de problema a modelar [Rumbaugh 1987][Kolp 1997].

Por supuesto que cualquier programador puede implementar manualmente relaciones particulares en forma ad-hoc, pero la abstracción puede perderse en el mecanismo de implementación [Rumbaugh 1987][Kolp 1997].

Históricamente los programadores han utilizado dos mecanismos para modelar una Relación en un ambiente de POO basado en clases. En algunas ocaciones utilizan las variables de instancia, mientras que en otras recaen en la necesidad o conveniencia de crear una nueva clase.

1.4.1.1 Relaciones a través de variables de instancia

El mecanismo más común es aquel en que los programadores implementen las relaciones con variables de instancia de las clases participantes. Los vínculos o asociaciones de esta Relación se representan como referencias de un objeto a otro [Rumbaugh 1987] o como colecciones que contienen referencias a los objetos relacionados [Kolp 1997].

Por ejemplo, modelar la relación *WorkIn* requiere definir la variable de instancia `employees` en la clase `Company` y agregar a la clase `Person` la variable de instancia `company`. De esta forma, los vínculos `(juan as Employee, Sun as Employer)` y `(pedro as Employee, Sun as Employer)` encontrarán una representación como la que se muestra en la siguiente figura.

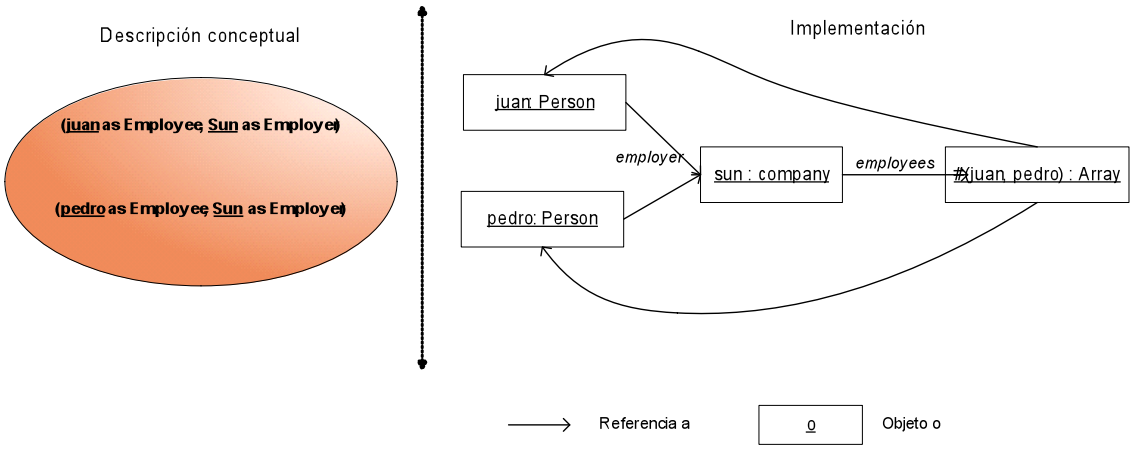


Figura 2. WorkIn implementada con variables de instancia

Uno de los problemas de este enfoque es que como los punteros y las referencias son de carácter unidireccional, este mecanismo falla naturalmente al representar bidireccionales. Es por esto que los programadores se ven en la necesidad de agregar variables de instancia en todas las clases participantes

[2] Smalltalk Visual Works, Smalltalk Squeak, Dolphin Smalltalk, C++, Java, .Net, Python, Self, Strongtalk, CLOS, Ruby.

de la relación. Esto impide representar explícitamente la restricción de que los objetos vinculados deben en todo momento estar referenciándose mutuamente.[Rumbaugh 1987] [Kolp 1997]

Otro de los problemas que presenta es la distribución de la semántica de la relación. La semántica de la relación es definida a través de la implementación de los métodos en las clases `Person` y `Company`. Debido a esto el prograd or deberá tomar decisones en ambos lados para mantener el invariante de la Relación y deberá definir las colaboraciones entre un `Person` y una `Company` cuando se agregan o remueven vínculos.

Un elemento no menor es el trabajo de programador para representar la relación. En nuestro ejemplo deberá programar al menos los métodos *`employees`*, *`employees:`*, *`addEmployee:`*, *`addEmployees:`*, *`removeEmployee:`*, *`removeEmployees:`*, *`isEmployeeOf:`* entre otros en el clase `Company`, y los métodos de la clase `Employee` *`company`*, *`company:`*, *`isWorking:`*, etc.

Si más tarde este mismo programador debe implementar la relación *FatherChild*, deberá volver a tomar deciones e implementaciones similares a las que llevó adelante en *WorkIn*. Esto tiende hacia la falta de uniformidad entre implementaciones de distintas Relaciones, ya sea por falta de autodisciplina del programador o porque distintas implementaciones de Relaciones son realizadas por diferentes programadores.

1.4.1.2 Relaciones con clases ad-hoc

Los programadores utilizan clases ad-hoc para representar relaciones generalmente cuando se enfrentan a alguna de las siguientes situaciones:

- les resulta complejo utilizar variables de instancia, o
- les resulta más natural pensar la Relación como un objeto del dominio

Frecuentemente la primera situación lleva al programdador a la segunda. Tomemos como ejemplo la relación *HusbandWife*. Al imaginarnos un vínculo entre esposos puede interesarnos agregar a él la fecha de casamiento y el lugar en que contrajeron matrimonio. Comienza a surgir la idea o abstracción Casamiento. Luego el programador decide crear una nueva clase `Marriage` con las siguientes variables de instancia: *`husband`*, *`wife`*, *`date`*, y *`place`*. Las instancias de `Marriage` representan los vínculos de aquello que originalmente concevimos como la relación *HusbandWife*.

Bajo estas circunstancias, es probable que un programador deba resolver los siguientes problemas

- ¿Quién será el objeto contenedor de todos los casamientos?
- ¿Cómo una persona consultará dicho conocimiento?
- ¿Cómo modelar el rol de Husband y cómo el de Wife (serán clases, se implementará dicho protocolo en la clase Persona, …)
- Programar los métodos correspondientes al protocolo de Husband y Wife en forma simétrica, para permitir agregar o quitar vínculos y consultar cosas tales como ¿estás casado? ¿con quién?

Luego el programador deberá tomar decisiones y definir estos mecanismos en forma ad-hoc una y otra vez ante cada nueva Relación que se le presente.

En consecuencia, tanto con esta estrategia como con la de utilizar variables de instancia, el programador es forzado a especificar detalles irrelevantes a la lógica de aplicación, resultando difícil separar la abstracción de su implementación [Rumbaugh 1987] [Kolp 1997]. Por ejemplo, deberá determinar la estructura de datos con la cual representar el universo de empleados de una empresa (será un Set, o una OrderedCollection, o un Dictionary para indexar por nombre). También deberá

preocuparse por el nombre de los métodos, los tiempos de búsqueda, de inserción, etc. Muchas veces optará por utilizar un diccionario para representar una relación semántica, por ejemplo una biblioteca que cataloga sus libros por cierto identificador (desconocido para el libro) y otras veces utilizará la misma estructura para representar un modelo con ciertos requisitos de performance, que nada tienen que ver con la semántica de la relación a modelar.

1.4.2 Lenguajes de Modelado

Las técnicas de modelado Orientadas a Objetos fueron desarrolladas para modelar la perspectiva del usuario sobre el sistema de un modo semánticamente significativo que siga los pasos de la conceptualización humana [Suscheck and Sandén 2003]. Las métodos de desarrollo de software OO y los lenguajes de programación han permitido una nueva forma de pensar sobre los problemas con modelos complementarios organizados alrededor de conceptos del mundo real[Kolp 1997].

Siguiendo esta filosofía, al modelar un sistema hay que identificar no solo las entidades del dominio, sino también cómo se relacionan estas entidades entre sí [Booch et al. 1999]. En este sentido, las relaciones han sido particularmente útiles en el diseño de grandes sistemas conteniendo muchas clases que interactúan, ya que permiten abstraer en forma natural las interacciones entre las instancias de dichas clases [Rumbaugh 1987].

No sorprende entonces que una de las técnicas más utilizadas y aceptadas en la actualidad por la comunidad de objetos para documentar y expresar el diseño de un sistema sea UML [Booch et al. 1999], y en particular su Diagramas de Clase. Los Diagramas de Clase dan a las Clases y las Relaciones un mismo nivel semántico.

El poder expresivo de los Diagramas de Clase para reflejar la vista estática y estructural de un modelo permitió que más tarde surjan herramientas con mecanismos para realizar una ingeniería directa (generar código) hacia determinado lenguaje de programación. Sin embargo, las decisiones sobre cómo representar una relación, aunque en forma automatizada, siguen las mismas estrategias que utilizaría un programador.

1.4.3 Lenguajes y prototipos con Relaciones

Ya hemos mencionado durante la introducción de este informe que Rumbaugh [Rumbaugh 1987] fue pionero en este campo de estudio. Y fue el quién junto a Shah, Hamel y Borsari [Shah et al. 1989] construyeron el lenguaje DSM que introducía soporte para Relaciones. También hemos dicho que Manuel Kolp [Kolp 1997] definó un meta-modelo de relaciones en objetos y presentó una posible arquitectura de solución sobre cómo implementarlo en un lenguaje con reflexividad como CLOS. Lamentablemente no tenemos constancia de que haya sido implementado. Por último en su trabajo [Bierman and Wren 2005], Bierman y Wren formalizaron un lenguaje reducido (sobre un lenguaje prototípico basado en un subconjunto funcional de Java) que soporta Relaciones. En las próximas subsecciones describimos algunas de sus principales características.

1.4.3.1 DSM

DSM[Shah et al. 1989], o Data Structure Manager, combina -en el contexto de lenguaje C- elementos de la programación orientada a objetos con conceptos semánticos del modelado de datos. Además, soporta Relaciones de asociación y agregación.

En DSM la declaración de una Relación se realiza en análogamente a la de una clase.

```
DEFINE works-for RELATION
employer :company 1 - * employees: person

DEFINE belongs-to RELATION
club *-’ members: person
```

Existen 2 tipos de relaciones que pueden ser definidas: las binarias y las cualificadas. Las relaciones cualificadas son un caso especial entre las binarias y las ternarias. Son útiles para calificar un vínculo dentro de una relación. Por ejemplo, si quisieramos acceder a un vínculo según el id de un empleado en una compañía.

Con respecto al manejo de cardinalidad, DSM permite combinaciones 1..1, *..1, 1..*, *..*.

Los vínculos de una relación son internamente implementados utilizando tablas de hash. Estos pueden ser agregados y removidos dinámicamente en tiempo de ejecución. Para tal fin la Relación ofrece operaciones (métodos) para acceder y modificar su estado. También ofrece mecanismos para consultar y testear el contenido de la Relación.

Gracias a los nombres de rol, DSM permite crear métodos de acceso a la Relación en las clases de los objetos participantes, generando así una idea de atributo o variable de instancia virtual. En nuestro ejemplo los métodos que se generarían serían del estilo `company::add-employees` y `person::get-employer`.

1.4.3.2 Propuesta de Kolp

Llamamos modelo de Kolp a la arquitectura propuesta por Kolp [Kolp 1997] para implementar relaciones en lenguajes orientados a objetos con características de Reflexión como el lenguaje CLOS.

Si bien la arquitectura no es sencilla de comprender tal como se encuentra expresada [Kolp 1997], hemos decidido presentarla ya que brinda una idea general sombre cómo utilizar las características de Reflexión para reificar el concepto Relación.

Las características de un sistema con Relfexión pueden resumirse en

- Arquitectura de dos niveles: el *nivel base* provee primitivas para crear y manipular objetos al *nivel de la programación de aplicación.* El meta nivel representa el modelo del *nivel base*.
- Conexión entre niveles: Ambos niveles se encuentran conectados de forma tal que un cambio en el *meta nivel* genere el efecto correspondiente en el *nivel base*.
- La arquitectura se puede generalizar a más de dos niveles.

En el modelo Kolp, la abstracción Relación se introduce a través de la definición de metaclases y superclases como se muestra en la figura a continuación.

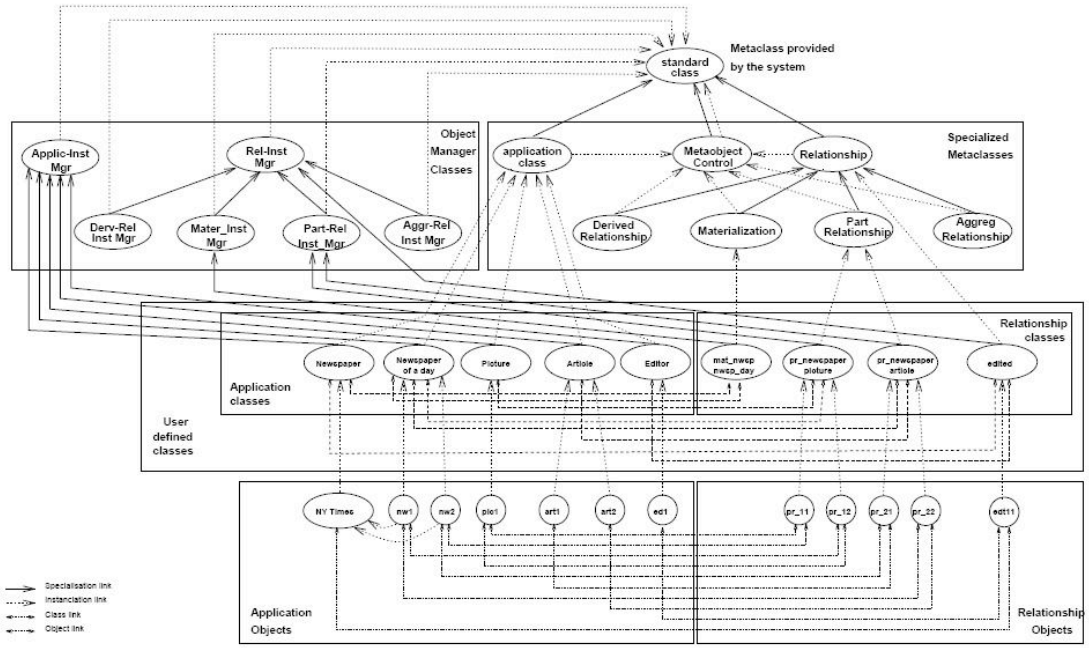


Figura 3. Propuesta de solución de Manuel Kolp

El protocolo del metaobjeto `Relationship` consiste en dos bloques principales: *metaclases especializadas* y *clases de administración de objetos*. Las *clases de administración de objetos* influyen directamente a los *objetos de aplicación.*

La metaclase `MetaobjectControl` habilita métodos para que dinámicamente manipulen las *clases de relación* (Ej.: `Edited`) entre dos o más *clases de aplicación* (Ej.: `Newspaper`, `Editor`). También es responsable de controlar y resolver conflictos producto de la misma definición de la relación.

Las metaclases `Relationship`, `AgregationRelationship`, `PartRelationship`, `DerivedRelationship` y `Materialization` brindan especializaciones de *clases de relación* con constructores y destructores de objetos que respectivamente capturan la semántica de creación y borrado de cada una de ellas.

1.4.3.3 RelJ

El enfoque del trabajo de Bierman y Wren (2005) fue más formal que los anteriores, pero considerablemente más reducido en cuanto a poder expresivo.

Ellos describieron cómo un lenguaje fuertemente tipado como Java puede ser extendido para soportar relaciones de primera clase. La herramienta con la que trabajaron fue un pequeño núcleo de Java, que llamaron RelJ.

En primer lugar, RelJ ofrece determinada sintaxis que permite declarar relaciones.

```
relationship Attends
     from Student to Course {
     int mark;
}
```

A diferencia de DSM, las instancias de una relación (vínculos) son objetos de primera clase. RelJ soporta a su vez especialización entre Relaciones.

```
relationship ReluctantlyAttends
    extends Attends
    from LazyStudent to Course {
    int missedLectures;
}
```

Sin embargo la semántica de esta especialización difiere de aquella entre clases. La Herencia de Relaciones en RelJ se basa en un forma restrictiva de delegación, en la que tienen que garantizarse los siguientes invariantes.

- *Invariante 1*. Sea la relación r_2 que especializa a r_1. Para cada instancia *i* de r_2 entre o_1 y o_2, también hay una instancia de r_1 entre o_1 y o_2 a quien *i* delega por los campos declarados en r_1.
- *Invariante 2*. Para cada relación *r* y un par de objetos o_1 y o_2, hay a lo sumo una instancia de *r* entre o_1 y o_2.

RelJ también ofrece la posibilidad de agregar o remover instancias a una relación a través de *statement expressions*.

```
...
// Remove Alice attending programming
alice.Attends -= programming;
```

Al ser un lenguaje prototípico, el formalismo de RelJ limita el sistema de tipos de Java a a los nombres de las clases y al tipo primitivo `boolean`. RelJ tampoco no soporta estructuras como conjunto de conjuntos.

Por último, ofrece las siguientes operaciones para manipular las relaciones.

- Dado el objeto *o* y la relación *r*, encontrar los objetos relacionados a *o* a través de la relación *r.*
- Dado el objeto *o* y la relación *r*, encontrar las instancias de *r* en que *o* participa.
- Dado una instancia de r, acceder a sus campos a través de los pseudo-campos from y t

Capítulo 2

Modelo de Relaciones

En este capítulo presentamos el modelo de Relaciones al que hemos arribado luego de nuestra investigación. Este modelo ha madurado luego de sucesivas iteraciones y refactorizaciones sobre el prototipo de Framework de Relaciones en Smalltalk Squeak.

El proceso de elaboración priorizó los aspectos conceptuales por sobre cuestiones como la performance.

2.1 Motivación

Fue en durante un trabajo práctico de la materia Diseño Avanzado Orientado a Objetos[3] donde por primera vez tomamos conciencia de posibilidad de reificar el concepto Relación en el ambiente de POO.

En su enunciado se mencionaban las propiedades matemáticas de relaciones (transitividad, reflexividad, etc.), indicaba que cada relación poseía su propia semántica, y que la declaración de un vínculo podría realizarse en forma explícita (Ej. Juan es amigo de Pedro), por comprensión (Ej. Todos los mozos de Villa Bosh son amigos entre sí) o por deducción (ej. Deducir "Juan es amigo de Mario" porque "Pedro es amigo Mario" y la relación Amistad es transitiva).

Por otra parte también se hacía referencia a que los objetos en condiciones de participar de una relación podía ser especificados por extensión (Ej. Juan, Pedro y Mario) o por comprensión (ej. Las personas mayores a 10 años). El enunciado exponía además algunas de las características de las relaciones presentes en UML, como multiplicidad y tipos de agregación. También dejaba entrever la posibilidad de investigar sobre operaciones matemáticas como la unión y intersección entre relaciones.

El grado de interés que tomó en nosotros el tema fue directamente proporcional al grado de avance en aquel trabajo práctico. Más tarde decidimos realizar nuestra Tesis de Licenciatura sobre este tema.

Aquel trabajo nos permitió comenzar a vislumbrar algunas ventajas y posibilidades que generaba el incorporar Relaciones en un LPOO. Cuando comenzamos esta investigación planteamos esos y algunos otros aspectos como elementos motivadores. A continuación presentamos algunas de estas ventajas y posibilidades.

[3] Materia Diseño Avanzado Orientado a Objetos, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, Argentina.

2.1.1 Transición natural entre diseño e implementación

El programador ya no debe codificar implementaciones ad-hoc de relaciones, sino que haciendo uso de las herramientas del ambiente, solo se debe preocupar por especificar y configurar las mismas acorde a la documentación del modelo. Esto resulta en una transición del diseño a la implementación más natural, suave y directa.

2.1.2 Nuevos Atributos

Al existir la entidades Relación y Vínculo, distintos atributos y métodos pueden ser agregados a éstas, brindando nuevos elementos para modelar con mayor fidelidad el dominio de problema.

2.1.3 Nuevas Colaboraciones

Al existir la entidad Relación, es posible enviar mensajes a la Relación y definir mecanismos que permitan consultar su estado a través de operaciones para agregar o borrar vínculos; consultar por la pertenencia a cierto vínculo; seleccionar un subconjunto de sus vínculos; o iterar sobre ellos.

La existencia de la entidad Vínculo puede aportar también ventajas en este sentido.

2.1.4 Nuevas Herramientas

Gracias a la posibilidad de aplicar operaciones directamente sobre la Relación, y la capacidad de éstas de disponer y concentrar la información del dominio en forma ordenada, uniforme y completa; muchas expresiones pueden sean escritas concisamente [Rumbaugh 1987] [Kolp 1997]. De esta forma, nuevas herramientas y frameworks podrán montar su solución sobre las Relaciones que describen el modelo, extendiendolas con atributos exclusivos del domino de problema que pretenden resolver.

2.1.5 Sharing No Anticipado

Ya no sería necesario conocer (anticipar) durante la fase conceptual de diseño todas las relaciones en las que un objeto puede potencialmente participar. Estas podrían ser agregadas más tarde, y en forma dinámica [Rumbaugh 1987][Kolp 1997]. De esta forma, los objetos que se encuentren en el dominio de las nuevas relaciones heredarán o adquirirán dinámicamente el comportamiento del o los roles en que participen.

2.1.6 Protocolos de rol, dinámicos y uniformes

Mucho del comportamiento de un objeto puede ser derivado de la información de la relación en la que está en condiciones de participar. Así, una persona al tener la capacidad de participar en la relación *FatherChild* como Padre, sabrá responder al mensaje `addHijo:` ó `hijos` sin necesidad de que el programador deba implementarlo.

Esto trae como inmediata consecuencia protocolos completos y uniformes en todos los objetos del sistema, evitando comportamientos incompletos o incoherentes entre si como los que se presentan en Dolphin Smalltalk 5.x

Patrón a seguir	Behavior	Presenter	View
all<SubComponents>	allSubclasses	-	allSubViews
all<SubComponents>Do:	allSubclassesDo:	allSubPresentersDo:	allSubViewsDo:
<parentComponent>	superclass	parentPresenter	parentView
<parentComponent>:	superclass:	parentPresenter:	parentView:
add<SubComponent>:	addSubclass:	add:	addSubView:
Remove<SubComponent>:	removeSubclass:	remove:	removeSubView:

Tabla 1. Protocolos no uniformas.

En el cuadro anterior podemos observar cómo el nombre de los métodos de `Behavior` no respetan el mismo patrón que `Presenter` y `View`. Incluso, dentro del mismo framework ModelViewPresenter de Dolphin vemos como `Presenter` y `View` no siguen un mismo patrón (ej: patrón add<SubComponente>).

2.2 Definición Formal de Relación

Previo a entrar en la etapa de descripción del modelo y framework resultante de nuestro trabajo, consideramos conveniente exponer una primera formalización sobre qué es una Relación en un ambiente orientado a objetos y basado en clases.

2.2.1 Relación

Una Relación ***R*** se define sobre los conjuntos $X_1, \ldots, X_k$. A los conjuntos $X_1, \ldots, X_k$ se los llama dominios de la Relación ***R***.

Cada dominio X_i contiene objetos que satisfacen cierta condición. Decimos que cada dominio X_i representa el conjunto de objetos en condiciones de participar en la en la Relación ***R*** cumpliendo el Rol_i.

La variable k indica el número de "roles" intervinientes en la relación. Es un número natural, llamado aridad, dimensión o grado de la relación. Una relación con k roles es llamada k-aria, con excepción de k = 2 o k = 3 donde comúnmente se refiere a ellas como binarias y ternarias respectivamente.

2.2.2 Vínculo

La Relación ***R*** describe una "forma o tipo" de "conexión o correspondencia" a través de la cual pueden vincularse entre sí los objetos de los diferentes dominios .

Definimos un Vínculo ***v*** de ***R*** como la conexión o correspondencia entre o_1, …, o_k donde el objeto o_i pertenece al dominio X_i u o_i es nulo. La semántica de este vinculo estará determinada por ***R***.

Es importante diferenciar y no confundir al vínculo ***v*** con la tupla (o_1, …, o_k) ya que si bien la definición de ***v*** es representada en términos de la tupla (o_1, …, o_k), ***v*** responde a un concepto más amplio que esta.

2.2.3 Propiedades

La Relación ***R*** puede tener o cumplir ciertas propiedades que permitan conocer o inferir ciertas reglas de su comportamiento.

El caso más natural lo encontramos en las relaciones de grado 2, con las propiedades transitividad, reflexividad, simetría, etc. También es posible pensar en otras propiedades como por ejemplo "La relación es acíclica, o no contiene ciclos".

2.2.4 Conocimiento de una Relación

El conjunto de vínculos ***v*** de ***R***, ***V****(R)*, conforman el conocimiento de ***R***. Si bien en su definición matemática una relación es un subconjunto del Producto Cartesiano de X_1, …,X_k, en nuestra definición esto no se cumple ya que una tupla puede contener elementos nulos.

2.2.5 Invariante

Con el objetivo de que el conocimiento de ***R*** sea consistente, ***R*** debe mantener un Invariante ***I*** entre los vínculos de ***V****(R)*. Este invariante está conformado por un conjunto de restricciones y un conjunto de reglas.

En un momento *t*, la Relación ***R*** tiene un estado ***E****(R, t)* = $X_1(t)$, …, $X_k(t)$,$V(R, t)$ donde $X_i(t)$ representa el estado del conjunto X_i en *t* y ***V****(R, t)* el conjunto de vínculos en dicho momento *t*.

2.3 Descripción del Modelo

2.3.1 Creación de relaciones

Al ser nuestro objetivo el de reificar el concepto Relación, nuestro primer paso fue una crear una nueva Clase que la represente. Así surgió la **`RRelation`**, la cual a través de sus instancias describe las relaciones existentes en el ambiente.

En segundo lugar debíamos contener y administrar el conjunto de todas las relaciones existentes. Fue entonces como surgió el concepto de Administrador de Relaciones con su consecuente **`RRelationManager`**, responsable de registrar y desregistrar relaciones en el ambiente, y ofrecer un punto de acceso único para operar sobre estas.

Tomemos como ejemplo la Relación *FatherChild entre personas, definida tal como se muestra en el gráfico*[4] *a continuación*

[4] Para mayor información sobre la notación gráfica utilizada ver subsección "Notación Gráfica" dentro de "Como leer este documento"

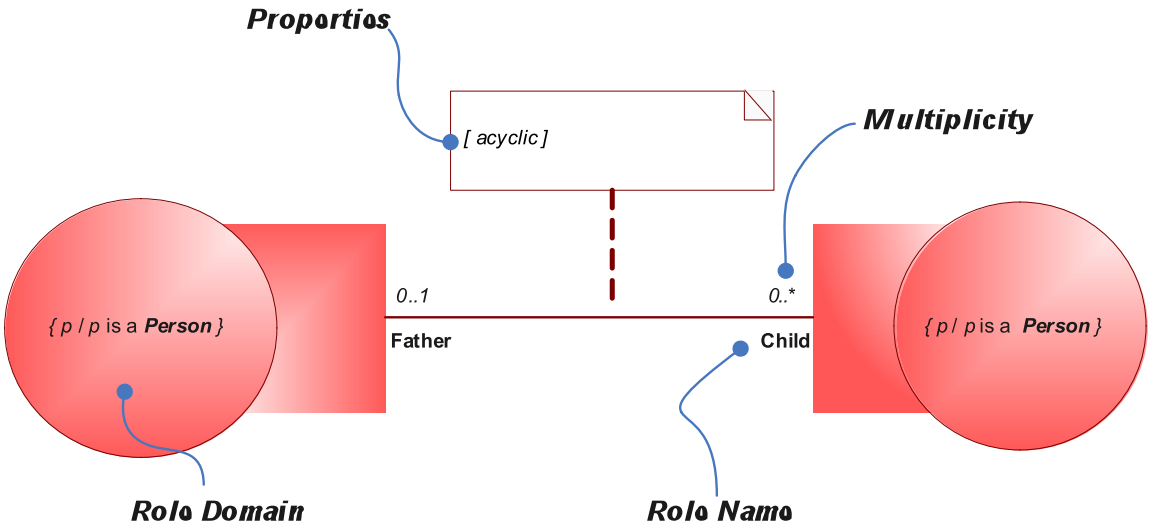


Figura 4. FatherChild. Diagrama de Relación

Programáticamente[5] podemos instanciar de forma ágil y rápida *FatherChild* tal como se muestra a continuación.

```
fatherChildRelation := (RRelation new)
    name: 'FatherChild';

    withRoleNamed: 'Parent';
    withRoleNamed: 'Child';

    all: Person asDomain canParticipateAs: 'Parent';
    all: Person asDomain canParticipateAs: 'Child';

    a: 'Child'
           mustParticipateAtLeast: One time atMost: One time
           withOthersParticipatingAs: 'Parent';

    beAcyclic;
    yourself.
```

Como resultado de la ejecución de éste código contamos con el objeto `fatherChildRelation`, el cual representa *FatherChild*, tal como expresamos en el diagrama.

Para graficar su constitución y composición y así comenzar a describir su modelo, presentamos su diagrama de objetos.

[5] Más adelante mostraremos herramientas gráficas para crear y editar relaciones.

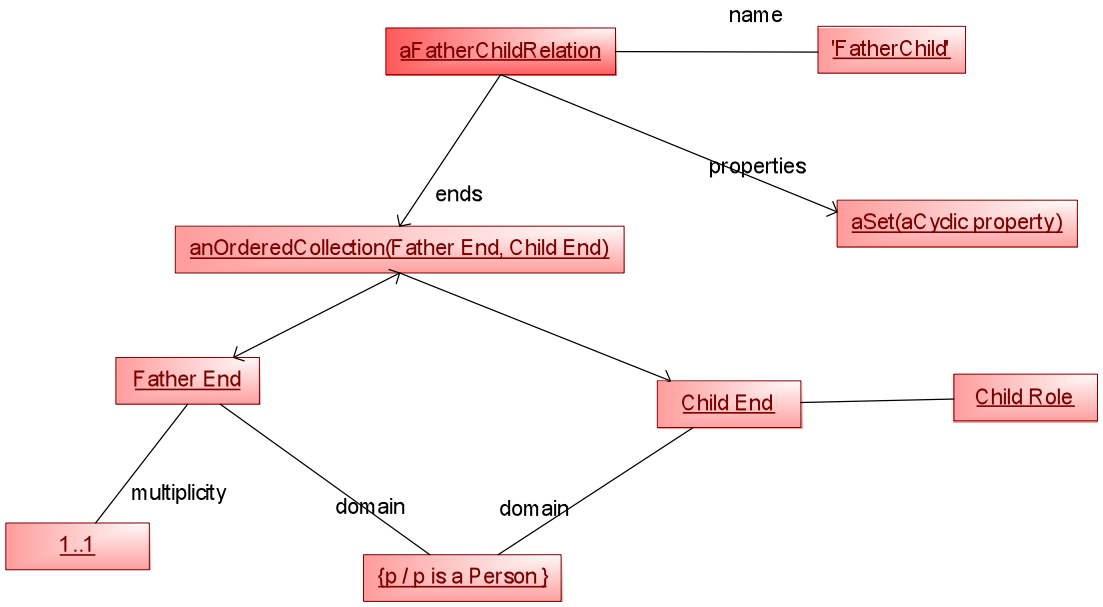


Figura 5. FatherChild. Diagrama de objetos

Una vez definida es necesario notificar la existencia de la Relación al ambiente. Para ello debemos registrarla en el **`RRelationManager`**.

```
"Registramos la relación en el ambiente"
RRelationManager instance
    registerRelation: fatherChildRelation.
```

Durante este proceso, el administrador de Relaciones tiene como responsabilidades asegurarse que la Relación pueda ser accesible, que cuando lo sea su conocimiento sea consistente y que los objetos participantes hayan "tomado conciencia o aprendido" de la misma.

En las siguientes subsecciones exponemos cada uno de estos pasos.

2.3.1.1 Descripción de la Relación

Describir adecuadamente una Relación es de vital importancia para obtener los resultados esperados y explotar al máximo las capacidades del modelo.

Esto requiere expresar su semántica, definiendo adecuadamente los nombres de los roles; el dominio de potenciales objetos participantes en cada rol; las propiedades (transitiva, simétrica, sin ciclos, etc); condiciones a satisfacer; cardinalidad y multiplicidad; entre otras.

La semántica definida será pues quien guíe y/o rija las políticas sobre el conocimiento de la relación y el comportamiento de los objetos participantes.

A continuación mencionaremos los elementos esenciales para de su descripción.

> **Nota** Existen otras características y capacidades que pueden enunciarse durante el proceso de creación. Sin embargo, hemos decidido no mencionarlas aquí, con el fin de simplificar esta sección. Serán presentadas en secciones posteriores.

Roles

Los Roles y los modelos basados en roles son abstracciones y mecanismos de descomposición. Mientras que las Clases estipulan las capacidades de objetos como individuos, la noción de un Rol se enfoca en la ubicación y responsabilidades de dichos objetos u elementos dentro de un sistema o subsistema[Kendall 1999].

Desde la perspectiva de un usuario del Framework, los roles son definidos por su nombre y por el dominio de objetos en condiciones de actuar como tal.

La elección del nombre de un Rol es muy importante y requiere que se utilice el nombre específico del dominio de problema en cuestión en lugar de nombres abstractos que no representen la semántica que se desea modelar (por ejemplo 'Father' en lugar de 'Parent').

El modelo contempla la posibilidad de especializar relaciones en forma similar a como se hace con la *Herencia entre Clases*.

> **Nota** Aunque dentro del modelo está contemplado, en el marco de nuestra tesis, dicha funcionalidad fue recortada del alcance del Framework.

Dominio de un Rol

Es razonable suponer en *FatherChild* que toda persona puede ser `Child` pero que sólo puede participar como `Father` cuando sea de sexo masculino y tenga más de 12 años.

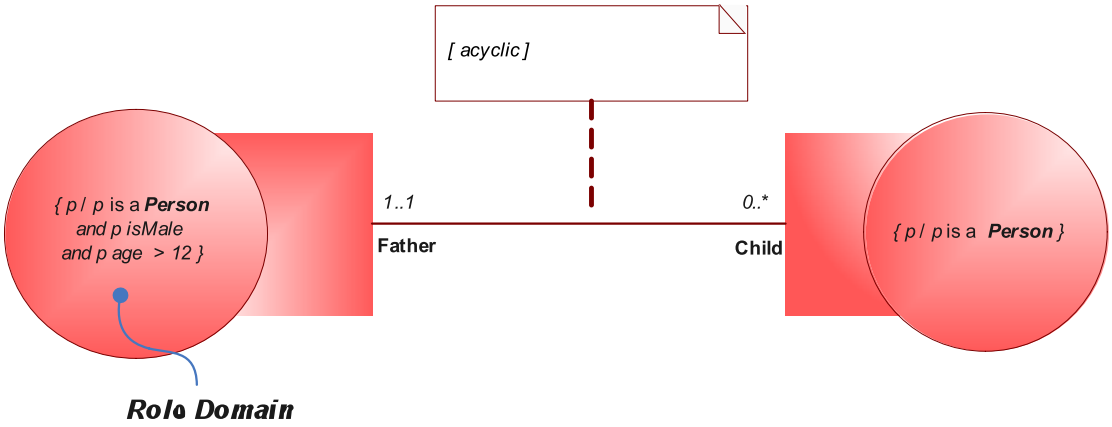


Figura 6. FatherChild. Diagrama de Relación 2

Como veremos más adelante, nuestro modelo ofrece la posibilidad de definir Restricciones sobre una relación, de las características de OCL [OCL][OMG 2003]. Con éstas podríamos resolver esta problemática. Sin embargo consideramos que es más natural modelarla con dominios.

Aquí es donde entra en juego lo que denominamos "Dominio de un Rol". El dominio de un Rol `A` representa a todos los objetos o elementos en condiciones de participar de la relación, es decir, representa a aquellos objetos que actuando como `A` pueden vincularse con otros que cumplan el o los otros roles definidos.

> **Nota** Este es uno de los aspectos en que nuestro modelo se diferencia del propuesto por UML[Booch et al. 1999]. UML enfoca la solución del problema utilizando otros elementos: las Clases o Interfaces como descripción de universo de potenciales participantes, y restricciones en OCL para discriminar cuales se encuentra en condiciones de participar.

El modelo acepta que cualquier `Set` cumpla las veces de un Dominio. Sin embargo, durante el transcurso de nuestro trabajo utilizamos `SSet`[Altman and Tylim 2007].

SSets es un Framework que permite definir conjuntos semánticos. Esto ofrece un gran número de ventajas respecto de los conjuntos como mera enumeración de elementos, entre las que se distinguen

- describir conjuntos por compresión, soportando conjuntos infinitos o demasiado grandes para ser enumerados
- embeber la semántica dentro del conjunto
- auto actualización, sin requerir clientes externos responsables de mantener actualizado el conjunto de elementos.

De esta forma, contando con SSet podemos definir el Dominio de `Father` como sigue

```
fahterDomain := SSet
    discriminatorString: '{[ person age > 12 ]}'
    inUniverse: Person asDomain
```

Análogamente, podríamos definir el tipo de dato Char[10] y Varchar[10] como sigue:

```
char10Domain := SSet
    discriminatorString: '{[ string size = 10 ]}'
    inUniverse: String asDomain
char10Domain preferredPrintString: 'Char[10]'.

varchar10Domain := SSet
    discriminatorString: '{[ string size <= 10 ]}'
    inUniverse: String asDomain
varchar10Domain preferredPrintString: 'Varchar[10]'.
```

De esta manera, consideramos que el uso de Dominios no sólo es natural sino que resulta sencillo y fácil de implementar.

Cardinalidad y Multiplicidad

Los términos Cardinalidad y Multiplicidad son y han sido utilizados en los lenguajes de modelado por mucho tiempo. Sin embargo, programadores, diseñadores y analistas muchas veces los confunden.

En esta sección pretendemos introducir la interpretación semántica a la cual hemos arribado en el marco de nuestro trabajo. Para ello volvamos a considerar la relación *FatherChild* y planteémonos las siguientes preguntas:

a) ¿Cuántas veces Juan puede ser Padre?
b) ¿Cuántas veces Juan puede ser Hijo?
c) ¿Cuántos hijos puede tener Juan como Padre?
d) ¿Cuántos padres puede tener Juan como Hijo?

Las dos primeras preguntas hacen mención a la cantidad de veces que un objeto puede participar en la relación cumpliendo determinado Rol. Utilizaremos el concepto de Cardinalidad para responderlas.

Las últimas, c) y d), indagan partiendo de que dado un objeto que cumpla cierto Rol, con cuántos objetos que actúen como el otro Rol podrá relacionarse. En este caso utilizaremos el concepto de Multiplicidad.

Nota Cuando las relaciones son binarias, las respuestas para a las preguntas *a)* y *c)* serán iguales. Lo mismo ocurrirá con las respuestas a *b)* y *d)* .
Esto, sumado a que comúnmente la mayoría de las relaciones modeladas son binarias podría ser el principal causante de la confusión sobre la semántica de estos conceptos.

Cuando las relaciones son binarias es posible deducir lógicamente, "en forma cruzada", la multiplicidad de la cardinalidad y viceversa. El Framework utiliza este principio para simplificar el uso del modelo.

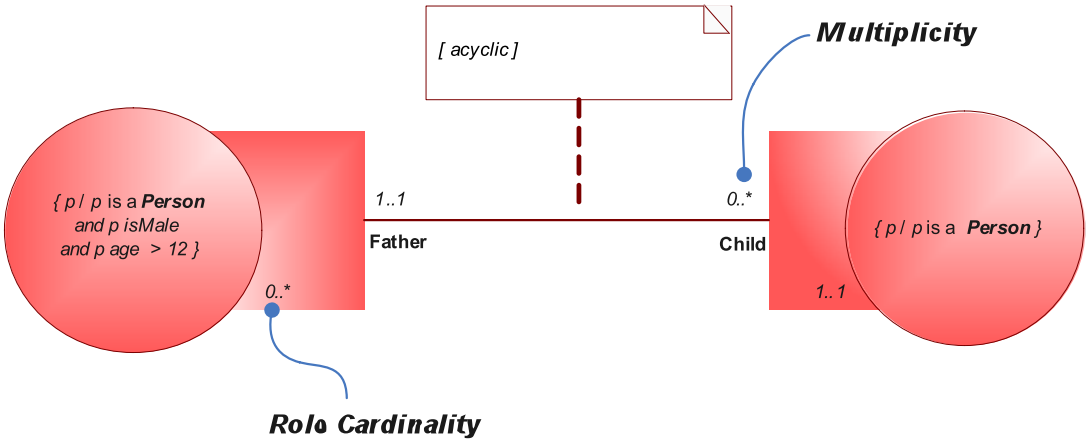


Figura 7. FatherChild. Diagrama de Relación 3

Sin embargo al profundizar la descripción de relaciones n-arias, es importante considerar y definir correctamente ambos conceptos a los efectos de evitar que el modelo se encuentre con ambigüedades y/o falta de información para inferir el correcto tratamiento de las mismas.

Cardinalidad

El concepto de cardinalidad se utiliza frecuentemente en los modelos de Entidad-Relación (ER) y es adoptado por los autores Elmasri-Navathe[Elmasri and Navathe 2000], Batini-Ceri, Dey-Storey-Barron y por métodos como Merise o Yourdon.

Comúnmente se entiende como *Cardinalidad de un Rol en una Relación* a la cantidad mínima y máxima de veces que un objeto del dominio de un rol deberá –o podrá- participar. En la literatura se representa como un intervalo o como un par de números (min, max) asociado al extremo del rol.

Multiplicidad

Luego de la aparición de UML[UML], el concepto de Multiplicidad se hizo popular y fue adoptado por los autores tales como Chen, Teorey, Howe, Ullman-Widom, Jones-Song, Loizou-Levene, McAllister, y Mannila-Raiha.

UML define Multiplicidad en una Relación[6] como "la multiplicidad de un rol de la asociación". Así, cuando se indica una multiplicidad en un extremo de una asociación, se está especificando que para cada objeto de la clase en el extremo opuesto debe haber tantos objetos en este extremo.

Aquí observamos que esta definición es incompleta para relaciones que no sean binarias. Por ello extendimos dicha definición para uniformar el criterio hacia relaciones cualquier grado.

Definimos *Multiplicidad de un Rol en una Relación en función de los restantes roles* diciendo: Sea una Relación de grado *n*, el Rol *i,* una tupla de participantes (P *1*, .., P *i-1*, P *i+1*, .., P *n*) cumpliendo los roles (Rol *1*, .., Rol *i-1*, Rol *i+1*, .., Rol *n*); la multiplicidad del Rol i representa la cantidad mínima y máxima de

[6] UML también utiliza este concepto en el contexto de clases y en el contexto de variables de instancia de clases.

veces en que los participantes (P *1*, .., P *i-1*, P *i+1*, .., P *n*) podrán y/o deberán vincularse con participantes del Rol i.

Propiedades

Algunos dominios de problema presentan características comunes que pueden ser definidas con solo mencionar una o varias de las propiedades que encontramos en las relaciones matemáticas: reflexividad, simetría, antisimetría, transitividad, etc.

En otras situaciones, como la que observamos en el ejemplo de la Relación *FatheChild* notamos que no es posible que un Padre se encuentre a si mismo como uno de sus ancestros, o dicho de manera más abstracta y matemática, que el grafo conformado por sus vínculos contenga ciclos.

Al describir una Relación, se pueden detallar cuáles serán las Propiedades que rigen o expresan su semántica. En nuestro modelo reflejamos estas propiedades como instancias de la clase **`RProperty`**.

2.3.1.2 Inicialización del Conocimiento

Tomemos ahora del ejemplo el proceso de registración de la relación *Implication entre conceptos, reflexiva, transitiva*, uno esperaría que luego de la registración, por ser la relación de carácter reflexiva, todo concepto se implique a si mismo. En otras palabras que el conocimiento de *Implication* satisfaga su invariante.

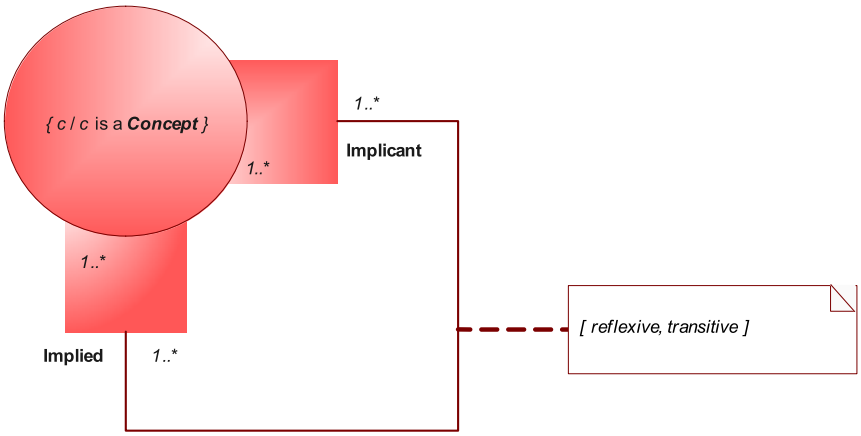


Figura 8. Implication. Diagrama de Relación

Para cumplir con esta expectativa, previo a la publicación el Administrador solicita o indica a ésta que calcule el estado inicial de su Conocimiento, para comportarse tal como se indica en los siguientes diagramas de secuencia

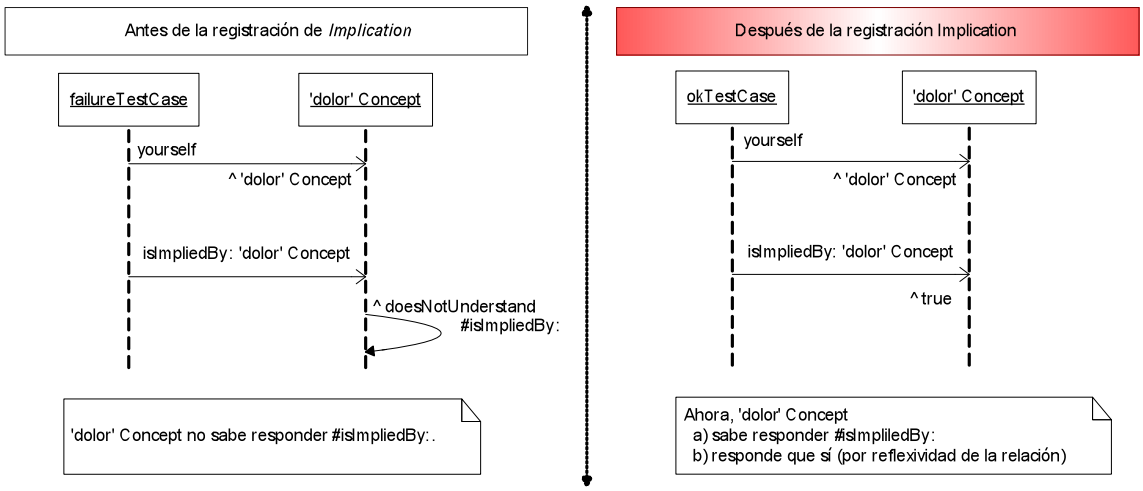


Figura 9. Implication. Esquema del proceso de registración (antes y después)

En la figura anterior podemos apreciar dos aspectos de interés. En primer lugar que `'dolor' Concept` "aprendió" a responder el mensaje `isImpliedBy:` y por el otro, que su respuesta satisface la propiedad de reflexividad, y por lo tanto nuestras expectativas.

2.3.1.3 Inferencia de Comportamiento

Como se desprende de la sección anterior, uno de los alcances que dimos a nuestro trabajo fue que los objetos participantes de una relación adquieran comportamiento en función de esta.

Aunque no es la intención de este apartado explicar qué comportamiento adquirirá un objeto por participar de una relación (lo cual se verá más adelante) consideramos conveniente ejemplificar este concepto. La idea subyacente es que un objeto no sólo se comporte tal como expresa su Clase, sino también siguiendo la semántica de las relaciones en que participa.

Por ejemplo, antes de registrar la relación *FatherChild*, la instancia de `Person` *juan* nada sabrá sobre padres e hijos. Sin embargo, luego de dicha registración, el mismo *juan* se encontrará en condiciones de comprender y responder a mensajes como `children`, `isChildOf:`, `allChildrenDo:`, entre muchos otros.

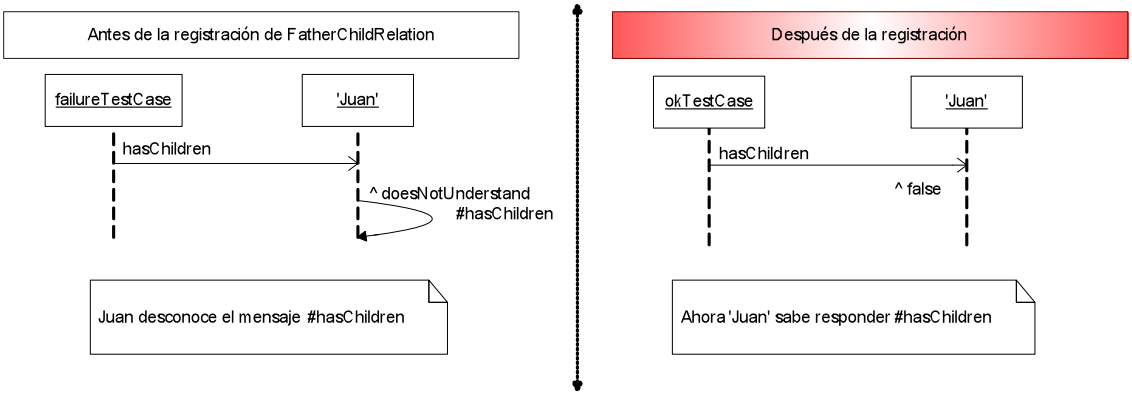


Figura 10. FatherChild. Esquema del proceso de registración (antes y después)

Como elección de diseño optamos por determinar cuál será el comportamiento que una determinada relación brindará a sus objetos participantes en esta etapa del proceso de creación de la Relación.

2.3.1.4 Publicación de la Relación

Llevar a las Relaciones al mismo nivel semántico que las Clases no sólo requiere la creación de la Relación, también es necesaria su publicación al ambiente a fin de anunciar su existencia y ofrecer medios de acceso a ella.

Por ello, el Administrador de Relaciones registra el nombre de la Relación como nombre global lo asocia a la Relación.

```
fatherChildRelation := "..."

RRelationManager instance registerRelation: fatherChildRelation.

(fatherChidlRelation = FatherChild)
    ifTrue: [ 'Registrada globalmente con el nombre FatherChild' ]
    ifFalse: [ 'No fue registrada aglobalmente' ]
```

Si el proceso de registración se realiza normalmente, la evaluación del código anterior retorna el String *'Registrada globalmente con el nombre FatherChild'*

2.3.2 Adquisición del conocimiento

Hasta aquí, hemos analizado algunas de las capacidades del modelo para describir una Relación: Propiedades, Cardinalidad, Multiplicidad, Roles, Dominio, Vínculo, Roles.

También hemos trasmitido la necesidad de definir o calcular el estado inicial Conocimiento de la Relación con el fin de que al ser publicada satisfaga su invariante.

Tal como lo expresamos en la definición formal de Relación, su Conocimiento representa el conjunto de sus Vínculos. Nuestro modelo ofrece una abstracción para modelarlo, la cual denominamos **RKnowledge**. Cada relación tendrá entonces su propio instancia de **RKnowledge**, la cual reflejará conceptualmente[7] el conjunto de los Vínculos entre "objetos con la capacidad de participar en la Relación" en un momento dado de tiempo.

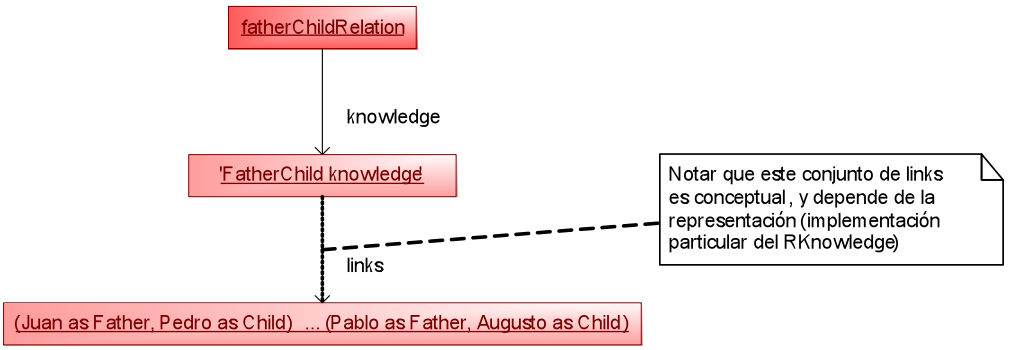


Figura 11. FatherChild's knowledge. Diagrama de Objetos

Resumiendo lo anterior, el Conocimiento de una Relación es el responsable de mantener y/o proveer la información necesaria para responder preguntas como ¿quién es el padre de Pedro?, ¿cuáles son los hijos de Juan? o ¿Juan es hijo de Pedro?, entre muchas otras.

[7] Decimos "conceptualmente" porque un RKnowledge puede estar representado por diferentes implementaciones, como diccionarios, grafos, conjuntos, bases de datos, etc.

En las próximas subsecciones detallaremos las diferentes circunstancias, situaciones y modos que Modelo y Framework ofrecen para adquirir información sobre nuevos vínculos de una relación. Luego, presentaremos los mecanismos utilizados para mantener la coherencia e integridad del Conocimiento.

2.3.2.1 Por obligatoriedad de participación

Planteemos ahora la siguiente definición de la Relación *Categorization*.

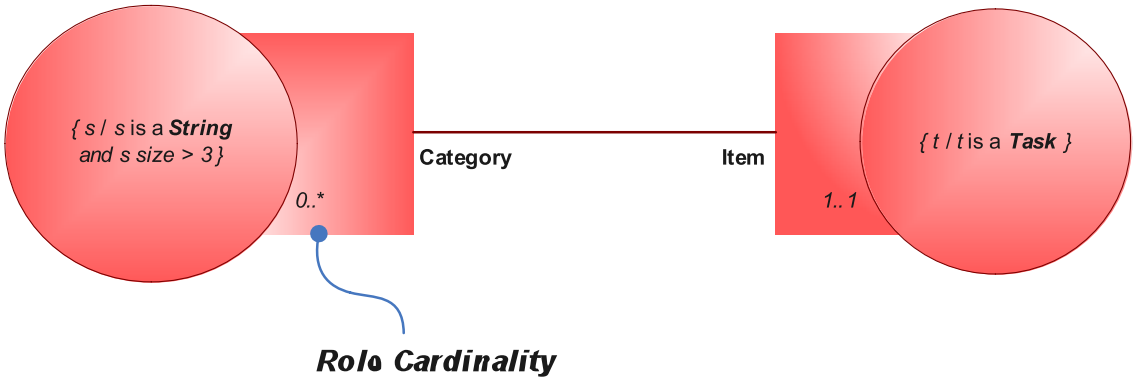


Figura 12. Categorization. Diagrama de Relación

La Cardinalidad del `Item` nos indica todo que *todo* elemento del dominio `Item` debe participar como mínimo una vez en la relación. Dada esta condición, cada *`item`* deberá por lo tanto estar vinculado a una *`categoría`*. Como este invariante debe cumplirse "siempre", incluso inmediatamente después del proceso de registración de la Relación, nos encontramos con el siguiente dilema, ¿a que *`categoría`* vinculamos cada *`item`*? La posibilidad de elegir una al azar no parece ser el modo más apropiado.

Como solución a esta situación introdujimos al modelo el concepto de Participante Predeterminado o Default de un Rol, permitiendo durante su definición, indicar cual de los elementos del dominio será el Participante Default.

De esta forma, por ejemplo, podemos extender la definición de la relación para definir al String *'Unclassified'* como el participante predeterminado del rol `Category`.

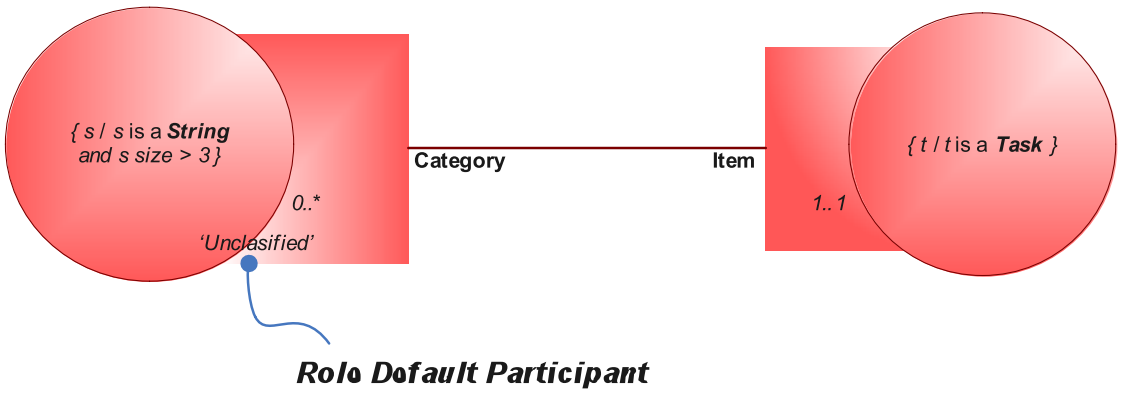


Figura 13. Categorization. Diagrama de Relación 2

En el ejemplo a continuación observamos el estado del conocimiento de la relación *Categorization* antes y después del proceso de inicialización.

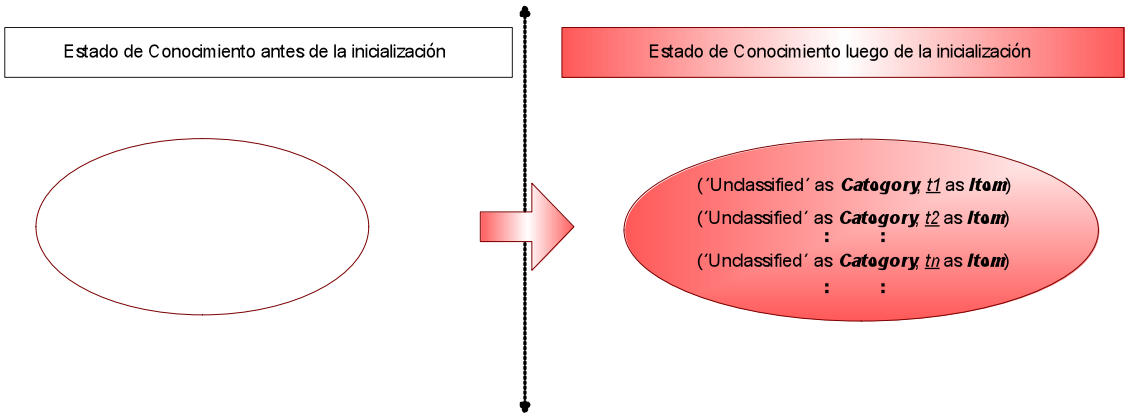


Figura 14. Categorization. Esquema del proceso de inicialización (antes y después)

Podemos apreciar la responsabilidad del Framework para inferir a partir de la cardinalidad si deben establecerse vínculos predeterminados y, en caso afirmativo, crearlos y agregarlos al conocimiento.

Nota Esta responsabilidad también recae sobre el modelo ante cambios en el dominio de uno de los roles.

2.3.2.2 Por declaración explícita de un hecho o vínculo

La forma más frecuente de modificar los vínculos entre objetos, ya sea en los lenguajes OO como en nuestro modelo, es por declaración explícita de los mismos al establecer y disolver vínculos en la medida que transcurre el tiempo.

Partiendo del supuesto que nuestro ambiente cuenta ya con la relación *FatherChild* imaginemos que un programador desea enunciar el siguiente hecho:

- "Juan es padre de Pedro"

Aquí se establecerá un vínculo entre Juan y Pedro, donde Juan actúa como Padre y Pedro actúa como Hijo. Este vínculo será agregado al conocimiento de *FatherChild*

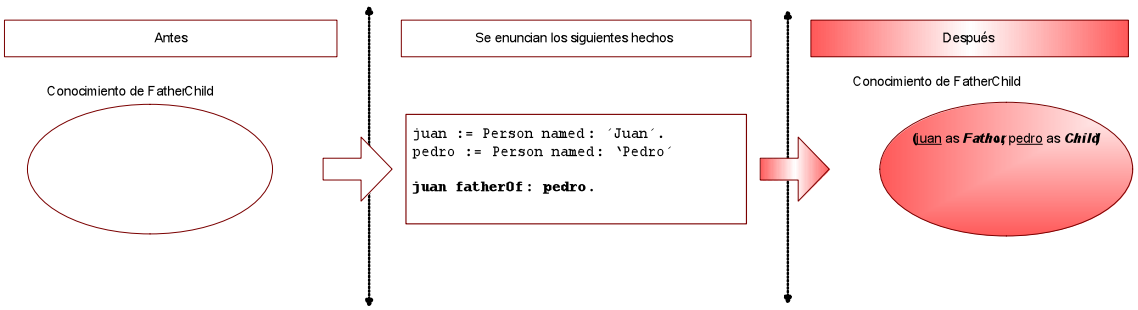


Figura 15. Juan es padre de Pedro. Esquema de adquisición de conocimiento

2.3.2.3 Por deducción o inferencia

Como ya hemos anticipado, el primer eslabón dentro la cadena de adquisición de conocimiento se materializa en el proceso de Creación de la Relación. Luego de este proceso, en relaciones declaradas como reflexivas, cada elemento del dominio[8] se encontrará inmediatamente vinculado a sí mismo.

[8] Notar que en una relación declarada como reflexiva se asume: a) es binaria y b) ambos roles tienen el mismo dominio.

También ocurrirá algo parecido cuando se agreguen elementos al dominio, donde éstos deberán ser también vinculados a sí mismos.

Partiendo del supuesto que nuestro ambiente cuenta ya con la relación *Implication, -reflexiva y transitiva-* imaginemos que nuestro programador desea enunciar los siguientes hechos:

- "El concepto *A* implica los siguientes conceptos: *B* y *C*"
- "El concepto *C* implica al concepto *D*"

Para hacerlo procede declarando nuevas instancias de conceptos. Estas nuevas instancias pasan a formar parte del dominio de la relación y por lo tanto son alcanzadas por las reglas orientadas a satisfacer el invariante de la misma.

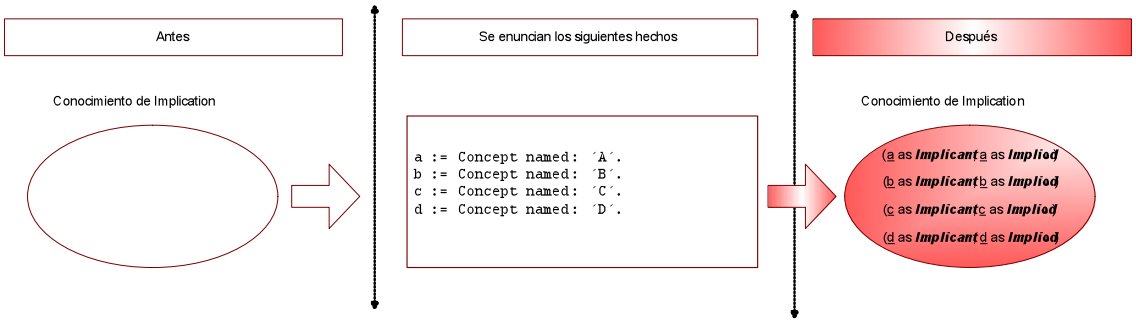


Figura 16. A, B, C y D'. Esquema de adquisición por deducción al agregar elementos al dominio

Algo diferente ocurrirá cuando se declare en forma explícita los vínculos entre A, B, C y D. Aquí se procederá a la constitución y agregado de los mismos al conocimiento, para luego aplicar la regla derivada de la propiedad transitiva, a partir de la cual se infieren nuevos vínculos.

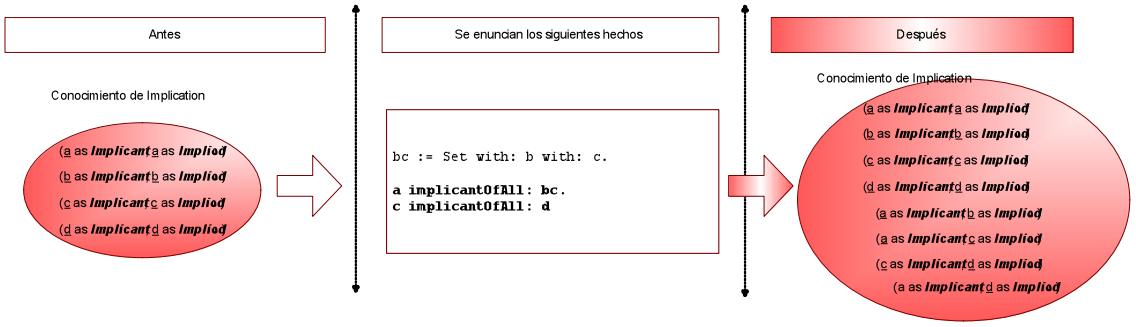


Figura 17. Esquema de adquisición explicita y luego por transitividad

Para lograr este objetivo, en primer lugar nuestro modelo permite describir las relaciones indicando que satisfacen ciertas propiedades.

```
implicationRelation := (RRelation new)
    name: 'Implication';

    withRoleNamed: 'Implicant';
    withRoleNamed: 'Implied';

    all: Concept asDomain canParticipateAs: 'Implication';
```

```
all: Concept asDomain canParticipateAs: 'Implied';

a: 'Implication'
       mustParticipateAtLeast: One time
       withOthersParticipatingAs: 'Implied';
a: 'Implied'
       mustParticipateAtLeast: One time
       withOthersParticipatingAs: 'Implication';

beReflexive;
beTransitive;
yourself.
```

A partir de este momento, es responsabilidad del **RKnowledge** hacer cumplir dichas propiedades. Cómo veremos más adelante, el modo o estrategia que éste utilice para lograrlo, dependerá de la implementación y configuración particular del **RKnowledge** de la relación.

Por ejemplo, si la relación utiliza una implementación de conocimiento explícito -como el **RExplicitKnowledge**-, éste deberá determinar al momento de su inicialización que implementación de reglas y restricciones usará para que lo asistan a mantener el invariante de la relación. Dentro del alcance de nuestro trabajo las únicas reglas implementadas son **RReflexiveRule**, **RSymetricRule** y **RTransitiveRule**.

2.3.2.4 Basado en otras Relaciones

Planteemos ahora una situación totalmente diferente. El usuario de nuestro Framework desea definir la relación *GrandfatherGrandchild, entre personas*. Pero nos dice que no quiere redundar en definir explícitamente los vínculos de esta relación, ya que existe una regla general que permite deducirla de la relación *FatherChild*. ¿Qué respuesta ofrece el Framework para satisfacer este tipo de solicitud?

Nuestra solución ofrece dos alternativas para satisfacer la necesidad de este cliente.

Con la primera, da al diseñador/programador la capacidad de definir un objeto que a) observe la relación *FatherChild*, b) detecte los cambios en su Conocimiento y c) cree o borre los vínculos necesarios en la relación *GrandfatherGrandchild*. Llamamos a este objeto Productor de Conocimiento y lo abstraemos como un **RKnowledgeProducer**.

En el marco de este trabajo, vemos como Productor de Conocimiento a todo objeto en condiciones de definir y agregar nuevos vínculos al Conocimiento de una relación. Así, las siguientes entidades pueden ser conceptualmente productores de Productor de Conocimiento:

- Un usuario del Framework que define un hecho en forma explícita
- Una entidad que, en base a la cardinalidad, declara que existe cierto vínculo por default entre 2 o más participantes
- Las Reglas de reflexividad, transitividad y simetría.
- Un objeto especialmente personalizado para las necesidades de un dominio de problema o de un usuario en particular.

La segunda alternativa es diseñar un **RKnowledge** específico a las necesidades de nuestro cliente. Luego, la relación *GrandfatherGrandchild* será configurada con este conocimiento, capaz de mantenerse actualizado por sí mismo. En cuanto a su implementación, el programador tiene dos posibilidades. En una de ellas puede crear y configurar una instancia de la clase **RPluggableKnowledge**. La otra es crear una subclase de **RKnowledge** y darle la inteligencia necesaria.

2.3.2.5 Delegado a un Conocimiento personalizado

La segunda alternativa nos propone entonces delegar a dicho **`RKnowledge`** personalizado gran parte de la responsabilidad de adquirir, mantener y representar los vínculos de una relación.

Su competidora -la alternativa de **`RKnowledgeProducer`**- aplica principalmente en aquellas relaciones que mantengan y administren sus vínculos en forma explicita (ej: a través de un **`RExplicitKnowledge`**) y administre su propio conjunto de vínculos. Si bien esto es teóricamente posible, en ocasiones puede resultar poco práctico y/o inconveniente.

Por poner un ejemplo, pensemos en integrar nuestro modelo de Relaciones con el ambiente tradicional de Squeak. Ya mencionamos que existen muchas relaciones embebidas en su código, interesantes de formalizar y reificar vía nuestro framework. En dicho caso, al detectar una de estas relaciones dispersas en el código, uno podría a) definir la relación, b) crear un **`RKnowledge`** personalizado que sirva de adaptador entre el framework y la implementación Squeak y, c) configurar la relación con dicho **`RKnowledge`**.

Luego, aunque el cambio resultará transparente, dispondremos ahora de las ventajas del modelo y las herramientas que hayamos definido en nuestro framework.

2.3.3 Coherencia e integridad del Conocimiento

En la sección anterior hemos mencionado las diferentes formas en que el modelo adquiere conocimiento. Parte de dicha adquisición se realiza a los efectos de satisfacer las propiedades y de esta forma cumplir con el invariante de la relación.

Pero el invariante no sólo está pensado para adquirir conocimiento a cualquier costo, sino también para restringirlo. Ya al introducir la utilidad e importancia de Dominios dejamos deslizar cierta característica para expresar tanto la definición como el comportamiento futuro de una Relación: las Restricciones.

Declarar y definir con precisión una Relación, nos permite contar con información relevante para inferir reglas que rijan su comportamiento y ciclo de vida. Nuestro Framework las utiliza para inferir determinadas Restricciones (**`RContraint`**), permitiendo a su vez al programador definir las suyas, acorde al problema a resolver.

Como consecuencia obtenemos Restricciones inferidas y restricciones explícitas.

2.3.3.1 Restricciones inferidas

De la información provista durante la definición, pueden inferirse varios tipos de restricciones. A continuación describiremos los soportados tanto por el modelo como por el Framework.

Inferidas del dominio

En su momento, mientras analizábamos la Relación *FatherChild* nos preguntábamos si ¿toda persona puede ser padre y/o hijo? Llegamos a la conclusión que toda persona puede ser Hijo pero no así Padre (solo aquellas mayores de 12 años y de sexo masculino pueden serlo).

Comprendimos entonces que era necesario describir esto con conjuntos que representen Dominios de los potenciales participantes para cada uno de los Roles.

Ahora nos enfrentamos a otra situación. Cómo manejar situaciones donde un Productor de conocimiento nos indique ‘el número 5 es el padre de Juan’ o ‘Ana Julia es el Padre de Juan’ o, siendo Cristian un bebé recién nacido, ‘Cristian es el padre de Juan’

Surge aquí la abstracción de Restricción de Dominio (**`RDomainConstraint`**). Debido a esta, todo vínculo para el cuál alguno de los participantes involucrados no se encuentren dentro del dominio del rol en el cual están participando serán rechazados.

Inferidas de las propiedades de la relación

Una de las formas de Adquisición de conocimiento es la de deducir información a partir de un estado actual de conocimiento y acorde a sus propiedades (Transitividad, Simetría, etc).

En forma análoga, existe una Restricción para propiedades como la Antisimétrica y la Irreflexiva, representadas por **`RAsymetricConstraint`** y **`RAntiReflexiveConstraint`**.

En el primer caso Framework verificará que dado el contexto de conocimiento actual y el vínculo que se está agregando no se genere una inconsistencia respecto de la propiedad antisimétrica. En el segundo, simplemente que el vínculo a agregar no sea de carácter reflexivo (el mismo participante para ambos roles).

El Framework también permite declarar una relación con una propiedad que exprese “Sin ciclos”. En este caso, la Restricción inferida no permitirá agregar un nuevo vínculo si al hacerlo, el grafo resultante de sus vínculos contiene ciclos. Con esto, el modelo permite evitar que en una Relación como *FatherChild* una persona sea ancestro de sí mismo.

Por participante requerido nulo (inferidas de la multiplicidad)

De la multiplicidad podemos determinar qué participantes son requeridos en un Vínculo. A partir de esto, esta restricción rechazará todos los vínculos en los cuales uno de los participantes requeridos sea nulo.

A continuación definimos la relación *Observation*.

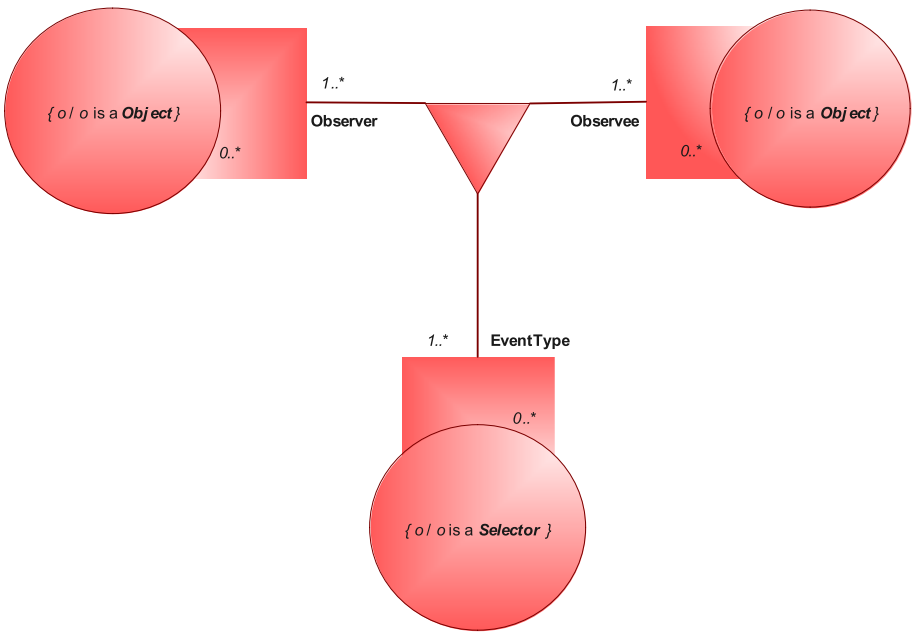


Figura 18. Observation. Diagrama de Relación (ternaria)

De la multiplicidad podemos inferir

- Que un **`Observee`** es requerido cada vez que un **`Observer`** lo observa en un **`EventType`**.

- Que cuando un **Observer** observa a un **Observee**, debe hacerlo a través de al menos un **EventType**

De las definiciones anteriores se desprende que en un Vínculo de esta relación, los participantes de **Observer**, **Observee** y **EventType** no podrán ser elementos nulos, dado que su multiplicidad indica que son requeridos para sus coparticipantes.

Luego, cuando un Productor defina un nuevo vínculo, será responsabilidad del Framework rechazar aquellos vínculos donde no se cumpla con alguna de las premisas anteriores, resultando que en ningún vínculo, el participante de **Observer**, **Observee** y **EventType** podrá ser nulo.

El Framework define entonces la Restricción "Dado un vínculo, si el rol es requerido, el participante asociado es distinto de nulo", donde un rol será requerido en los vínculos si su multiplicidad es del tipo N..M con N>0. Esta se encuentra representada por **RRequiredNotNullConstraint**

Inferidas de las claves

Existen otras situaciones donde los roles son requeridos, esto es cuando forman parte de alguno de los identificadores o claves de un vínculo. Una clave permite identificar un vínculo (ese y sólo ese) dentro del conocimiento de una relación.

Por ello en *FatherChild*, el rol **Child** es clave dado que podrá existir un solo vínculo por cada Child. No así con Father donde podrá haber varios vínculos para el mismo padre (uno por cada uno de sus hijos).

En el caso de la relación *Implication* tanto el rol **Implied** como **Implicant** son necesarios para conformar la clave que identifique el vínculo. En este caso la identificación del vínculo se daría de la siguiente manera "*Dame el vínculo en el cual el concepto A sea Implicante y el B sea el implicado*".

> **Nota** El concepto de *Clave* utilizado en nuestro trabajo es similar al de *Clave Única*, *Clave Primaria* y *Clave Secundaria* utilizado en las Bases de datos Relacionales. Sin embargo, a diferencia de las bases de datos relacionales donde el diseñador debe indicar cuáles son los campos de una tabla que conforman una de las claves, el modelo resultante de nuestro trabajo cuenta con la capacidad de inferirlas a partir de la definición de cardinalidad y multiplicidad.

Modelamos este aspecto del dominio de problema con la clase **RKeyNotNullConstrain**

2.3.3.2 Restricciones explicitas o personalizadas

Más allá de las restricciones que se derivan de la mera especificación de dominios, cardinalidad, multiplicidades de los roles y propiedades de la relación, existen otras propias del dominio de problema en cuestión.

Para ilustrar esta situación, volvamos a la Relación *FatherChild*. Al definir los dominios especificamos que para ser padre una persona debía tener más de 12 años. Sin embargo nada dijimos sobre la diferencia de edad entre un padre y sus hijos.

Suponiendo que debe existir una diferencia de más de 12 años redefinimos la relación tal como se muestra en el siguiente diagrama.

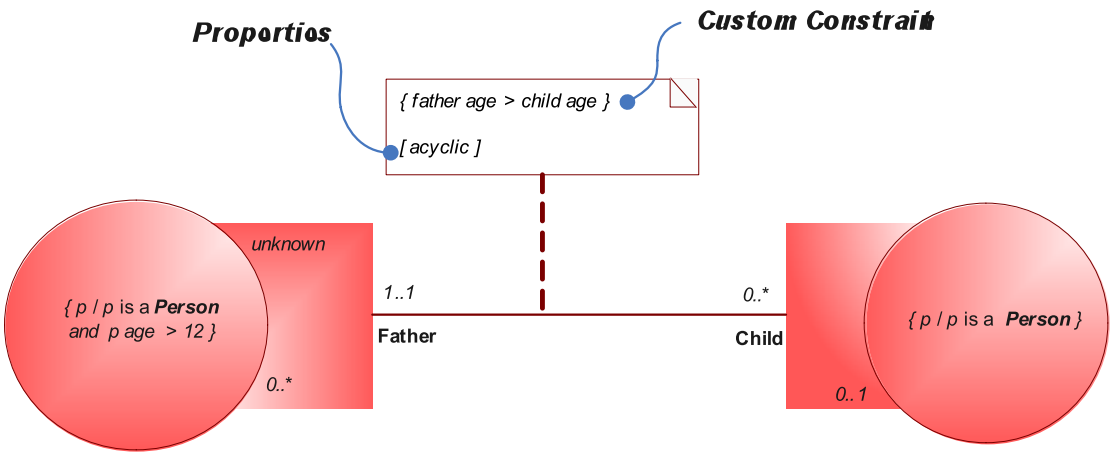


Figura 19. FatherChild. Diagrama de Relación 4.

¿Cómo modelar este nuevo tipo de restricción? Nuestra estrategia apuntó a dos frentes.

Por un lado seguir una línea símil OCL[9]. Para ello utilizamos el framework LogicExpressions [Altman and Tylim 2007] el cual brinda la posibilidad de representar expresiones de lógica de primer orden como objetos, permitiendo evaluar dichas expresiones y operar con ellas. De esta forma, podemos expresar esta restricción "father age > child age" con tan solo redefinir programáticamente la relación

```
fatherChildRelation := (RRelation new)
    name: 'FatherChild';

    withRoleNamed: 'Parent';
    withRoleNamed: 'Child';

    all: Person asDomain canParticipateAs: 'Parent';
    all: Person asDomain canParticipateAs: 'Child';
    a: 'Child'
           mustParticipateAtLeast: One time atMost: One time
           withOthersParticipatingAs: 'Parent';

    beAcyclic;
    ensure: 'father age > child age';

    yourself.
```

2.3.3.3 Resolución de Conflictos

Al modelar el conocimiento como un conjunto de vínculos es necesario resolver mediante alguna estrategia en particular el dominio de problemas de qué hacer ante conflictos.

En los lenguajes OO, al estar las relaciones embebidas en variables de instancia, todo se limita a modificar el valor de la misma en forma explicita.

Sin embargo, ahora tenemos la posibilidad de enunciar un hecho "Juan es padre de Pedro". Al hacerlo puede pasar que Pedro sea hijo de Alberto. ¿Cómo resolver este conflicto? Existen al menos las siguientes alternativas

[9] Aunque limitado a lógica de primer orden durante el alcance de nuestro trabajo.

- Se elimina del conocimiento “Pedro es hijo de Alberto” y se agrega “Pedro es hijo de Juan”
- Se genera una excepción indicando “Pedro ya tiene un padre”

La situación tiende a complicarse cuando algunos vínculos fueron enunciados por el usuario del framework y otros por las reglas. Todo el conocimiento (explícito o deducido) tiene la misma relevancia? ¿Toda fuente de conocimiento tiene el mismo grado de confianza? La antigüedad del conocimiento, ¿provee alguna idea sobre cómo resolver estos conflictos?

> **Nota** Este escenario tiene importantes similitudes con la problemática que aborda la Teoría del Cambio (Theory of Change)

En nuestro trabajo no abordamos en profundidad este problema. Optamos por una estrategia simple.

Cuando se presenta conflicto, se tienen las siguientes consideraciones

- Conocimiento Explícito sobreescribe Conocimiento Deducido
- Conocimiento Nuevo sobreescribe Conocimiento Viejo
- Todas las fuentes de conocimiento tienen el mismo grado de confianza.
- Cuando se enuncia un No vínculo, si el mismo no existe, se ignora silenciosamente.

2.3.4 Vínculos de una Relación

Junto a la reificación del concepto Relación, nació también reificar el concepto de Vínculo como instancias de la clase **`RLInk`**. Estos representan el enlace existente entre 2 o más objetos (dependiendo del grado de la relación).

> **Nota** En UML, los vínculos entre objetos son comúnmente llamados instancias de asociación.

Se abrió aquí una serie de posibilidades hasta ahora difícil de lograr en los lenguajes de programación orientados a objetos. La toma de conciencia de la entidad Vínculo nos permite modelar fácilmente quién fue la fuente de información, cuándo se constituyó, dónde, qué grado de confianza podemos darle al mismo, etc. También sirven de contenedor de otros posibles atributos y clases de asociación de una relación.

Pero en su forma más pura, los vínculos se establecen entre 2 o más objetos, cada uno de ellos cumpliendo un rol específico de los declarados en la relación, quedando la semántica del mismo determinada por la Relación.

En ocasiones es complejo responder con exactitud qué representan los vínculos debido a que su interpretación depende de la relación y del dominio de problema en cuestión. Pero abstractamente podemos decir que un vínculo representa un *enlace* entre ciertos participantes, cada uno de los cuales cumple un rol diferente de la relación. Por ejemplo, el hecho que Juan sea padre de Pedro, se representa con el vínculo `('Juan' as Father, 'Pedro' as Child)` en la relación *FatherChild*,

Sin embargo, un vínculo podría representar un objeto de primer orden en el dominio de problema a modelar. Por ejemplo, al modelar la relación *MarriedWith entre hombres y mujeres*, ¿Que representa el enlace entre Juan y María en el marco de esta Relación?

Figura 20. HusbandWife. Diagrama de Relación.

En primer lugar que son esposos. Pero también uno podría pensar en que representa su unión ante la sociedad, su Casamiento. Entonces, ¿por qué no modelar el vínculo como una instancia de Casamiento? Es decir, que existan vínculos que sean en realidad objeto del dominio y no una entidad abstracta del framework de Relaciones.

En las próxima sección abordamos éstas y otras cuestiones relacionadas con qué representa un vínculo y como describirlo.

2.3.5 Propiedades de Vinculación

En ciertas situaciones los vínculos de una relación requieren ser aggiornados, o calificados con propiedades. Si consideramos la Relación *HusbandWife entre Esposos y Esposas* es posible que deseemos extender sus vínculos con propiedades como fecha, lugar y testigos de casamiento.

> **Nota** El término propiedades se refiere aquí a las propiedades o atributos que puede tener un vínculo dentro de dicha relación y no a las Propiedades de una Relación (reflexiva, transitiva, etc)

El modelo de relaciones ofrece dos mecanismos para lograr esta característica. Por un lado, permite definir Propiedades de un Vínculo, mientras que por el otro brinda la posibilidad de explicitar directamente la Clase o tipo de Vínculo (a partir del cual podrán derivarse o inferirse estas propiedades).

En las siguientes dos subsecciones presentamos estas dos alternativas.

2.3.5.1 Propiedades de un Vínculo

Dada una relación, el Framework permite especificar cuales serán las propiedades de sus vínculos. Para ello utiliza la abstracción **`RLinkProperty`**, la cual describe un aspecto, atributo o propiedad del Vínculo.

Un **`RLinkProperty`** ayuda a describir el nombre de la propiedad (o rol), el dominio de objetos en condiciones de participar, la cardinalidad y la multiplicidad entre otras.

Por ejemplo, en la Relación *HusbandWife*, podemos declarar las propiedades *fecha de casamiento*, *testigo por parte del esposo*, y *testigo por parte de la esposa*. Gráficamente lo representamos tal como se presenta el diagrama de relaciones a continuación.

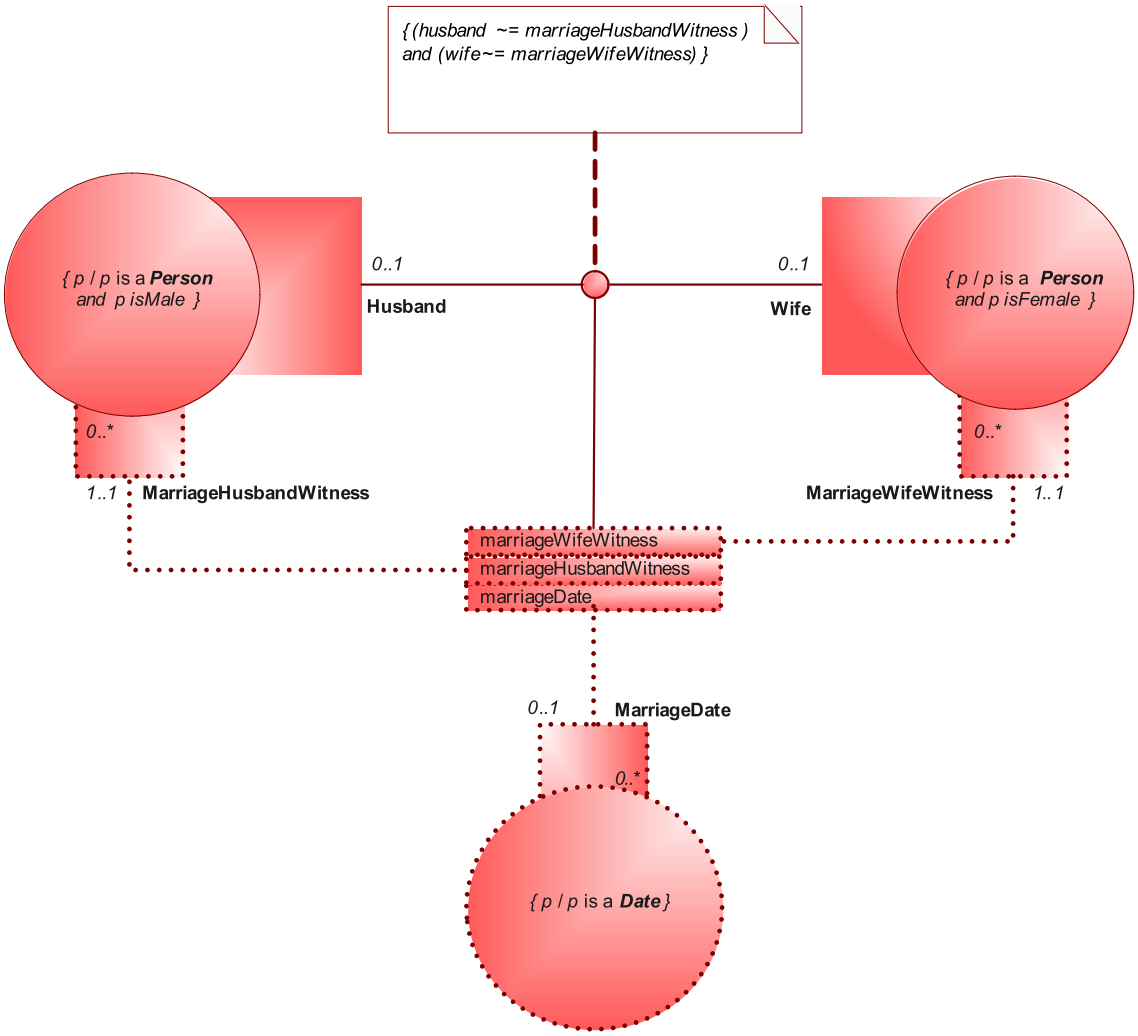


Figura 21. HusbandWife. Diagrma de Relación 2

En forma programática podemos expresarlo de la siguiente manera.

```
husbandWifeRelation := …
…
husbandWifeRelation
     addLinkProperty:
            (RLinkProperty
                   named: 'HusbandWitness'
                   domain: (Person asDomainWhere: 'person isMale').
                   cardinality: (RInterval zeroOrMany)
                   multiplicity: (RInterval one).
     addLinkProperty:
            (RLinkProperty
                   named: 'WifeWitness'
                   domain: (Person asDomainWhere: 'person isFemale').
                   cardinality: (RInterval zeroOrMany)
                   multiplicity: (RInterval one).
     addLinkProperty:
            (RLinkProperty
                   named: 'MarriageDate'
                   domain: Date asDomain
                   cardinality: RInterval zeroOrOne
```

```
                    multiplicity: RInterval zeroOrOne);
    ensure: '(husband ~= husbandWitness) and (wife ~= wifeWitness)';
    yourself.
```

Al declarar la relación con este detalle sobre las propiedades de sus links, consideramos apropiado proveer al Framework de nuevas capacidades. Entre éstas podemos mencionar la de brindar mecanismos de acceso, determinar valores predeterminados y nulos y validar participación en el dominio entre otras.

Nota Como explicamos en la sección de Trabajos Futuros, es importante analizar la conveniencia o no de modelar este tipo de Propiedades de de Link utilizando relaciones entre el Link y sus propiedades.

A continuación observamos un ejemplo funcional sobre cómo una vez definida la relación *HusbandWife* es posible modificar los atributos de sus vínculos.

```
juanHusbandOfAnaLink := juan husbandOf: ana.
juanHusbandOfAnaLink
    marriageDate: Date Today;
    marriageHusbandWitness: pedro;
    marriageWifeWitness: maria;
    yourself.
```

2.3.5.2 Clase o Tipo del vínculo

En ocasiones el Vínculo entre 2 o más objetos puede representar una entidad del dominio de problema a modelar. En este tipo de situaciones es probable que se desee o sea necesario especificar la entidad que represente sus vínculos.

Por ejemplo, que las tres propiedades definidas en el ejemplo anterior de *HusbandWife* se refieran a datos del casamiento de dos personas permite suponer que el Vínculo representa en realidad dicha unión, es decir el Casamiento entre las mismas.

Aquí, el analista pude considerar apropiado redefinir la relación de *HusbandWife* como *MarriedWith*, brindándole un mayor nivel semántico. En primer lugar al cambiarle el nombre, pero en segundo al especificar que el tipo de vínculo es un **`Marriage`**.

Nota UML utiliza el concepto "Clase de Asociación" para modelar lo que en nuestro trabajo llamamos "Tipo de Vínculo".

Con este antecedente, consideramos conveniente que el modelo cuente con la capacidad de definir el tipo de vínculo, o la Clase que cumplirá las veces de asociación entre los objetos relacionados, el cual gráficamente representamos de la siguiente manera

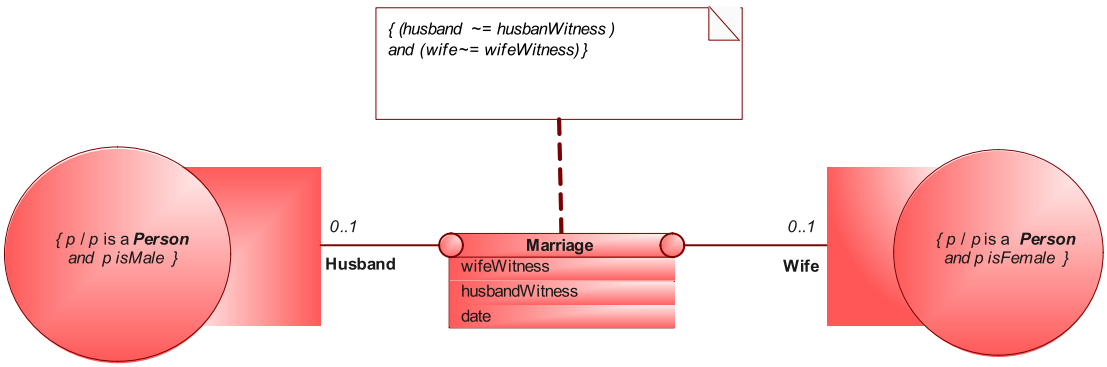


Figura 22. MarriedWith. Diagrama de Relación

Y en forma programática como se muestra a continuación.

```
marriedWithRelation := …
…
marriedWithRelation
     linkType: Marriage
     ensure: '(husband ~= husbandWitness) and (wife ~= wifeWitness)';
     yourself.
```

El único requisito de este tipo de vínculo es que sepa responder al mensaje `#properties` con un conjunto de **`RLinkProperty`**, describiendo sus propiedades.

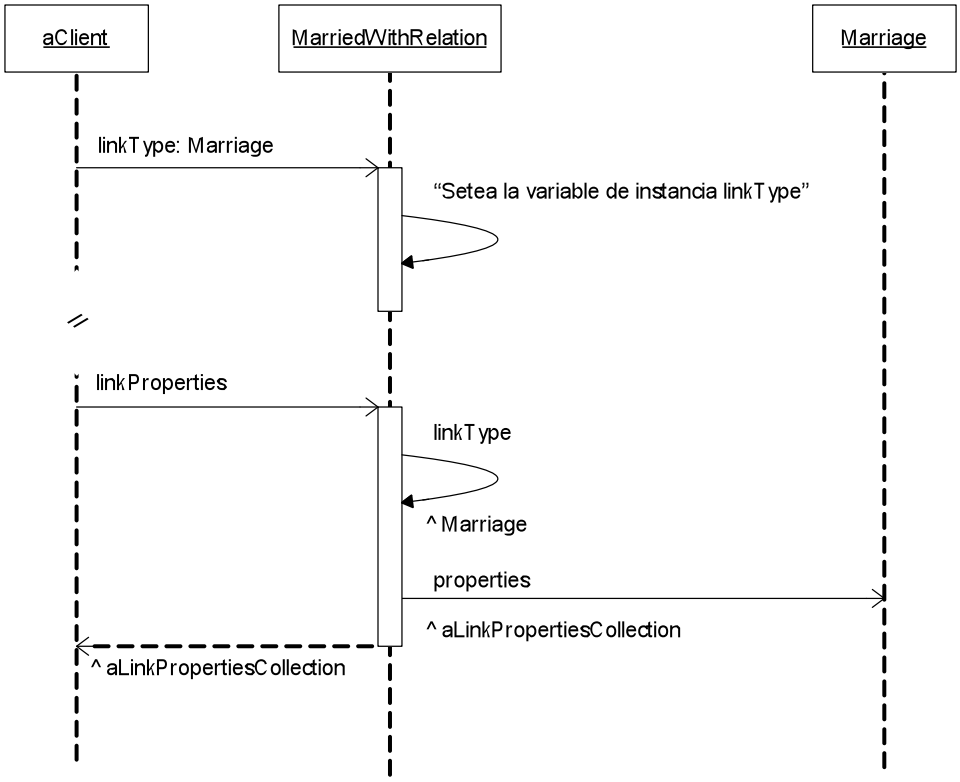


Figura 23. Use Marriage's Link. Diagrma de Secuencia

En el diagrama anterior se observa cómo la relación delega el mensaje `#linkProperties` a **`Marriage`**.

Con este prerrequisito satisfecho, el modelo debe comportarse en la misma forma en que lo hubiera hecho si sólo se hubiesen especificado las propiedades, pero creando instancias del tipo de vínculo (ej. **`Marriage`**) cada vez que se establece un nuevo vínculo.

Por esto último, y con el objetivo de independizar la interfaz de tipo de vínculo (**`Marriage`**) de la implementación de **`RLink`** en el framework, decidimos utilizar el Pattern Adaptor [Gamma et al. 1995]. Creamos una nueva clase, **`RLinkTypeAdaptor`**, que cumple el rol de Adaptador del tipo de vínculo. Luego, al momento de crear un nuevo vínculo, el Framework instancia un **`RLinkTypeAdaptor`** basado en su tipo.

Luego, el comportamiento de la relación hacia el mundo no cambia, como podemos observar en el ejemplo a continuación.

```
juanMarriedWithAnaLink := juan husbandOf: ana.
juanMarriedWithLink
     date: Date Today;
     husbandWitness: pedro;
     wifeWitness: maria;
     yourself.
```

Para acceder a la instancia de **`Marriage`** correspondiente al vínculo que acaba de crearse vasta enviar el mensaje `#value`.

```
juanHusbandOfAnaLink value
```

> **Nota.** La única diferencia visible será el valor del vínculo (`#value`). Si sólo se definen propiedades, el valor de un vínculo resultará ser el mismo vínculo, es decir una instancia de **`RLink`**. En caso de que se haya especificado un tipo de vinculo, el valor de un vínculo será instancia de dicho tipo (por ejemplo, instancia de **`Marriage`**)

2.3.5.3 Identificadores y Calificadores

Ciertas ocasiones hacen que las propiedades de un vínculo sean funcionales a uno de los extremos y no al resto. Por ejemplo cuando

- Una biblioteca categoriza sus libros por tema, o dicho de otra forma, la biblioteca *califica* cada libro con uno o más temas
- Una biblioteca asigna un código único (ej. un ID Alfanumérico) a cada libro para luego *indentificarlo* en forma univoca.

Frecuentemente en la programación Orientada a Objetos se utilizan Diccionarios o Tablas de Hash como variables de instancia de los objetos que desean catalogar o identificar.

Con Relaciones, podríamos definir 2 relaciones Ternarias: *LibraryTopicBook* y *LibraryIdBook*. Sin embargo, consideramos que este podría no ser un enfoque 100% adecuado en estos casos por ser una decisión *unilateral* de la Biblioteca calificar e identificar sus libros de cierto modo. Incluso, en ambos ejemplos, los libros desconocen cómo están calificados o cómo los han identificado.

Del análisis anterior surge que ciertas propiedades de un vínculo pueden ser de uso exclusivo o privado de uno de sus roles. También se desprende que las propiedades pueden cumplir las veces de *Identificador* o *Calificador*. Entre éstas existe una diferencia esencial: mientras un *Identificador* permite

que un participante identifique uno y solo uno sus vínculos, un *Calificador* ofrece a éste la posibilidad de etiquetar 1 o más de sus vínculos según variados criterios.

Con estas necesidades en mente optamos por extender el modelo de propiedades a los efectos de permitir

- Discernir si una propiedad es del Vinculo, o exclusiva de alguno de sus roles.
- Informar a la relación que propiedad es utilizada como Identificador por parte de un Rol
- Informar a la relación que propiedad es utilizada como Calificador por parte de un Rol

En forma gráfica podemos expresarlo tal como se muestra a continuación.

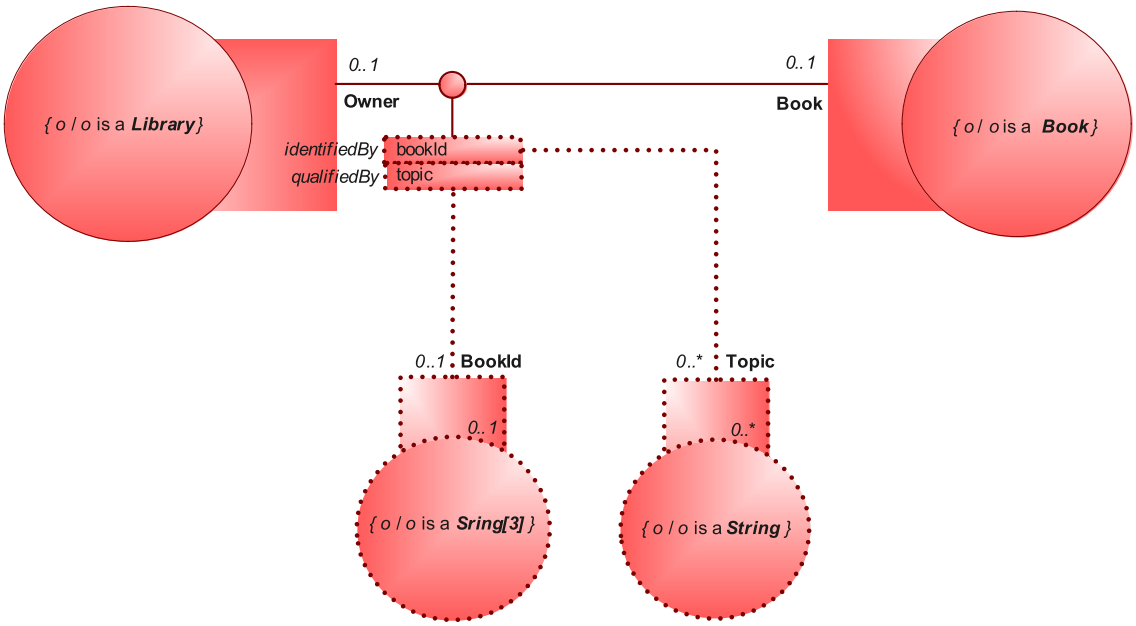


Figura 24. LibraryBook. Diagrama de Relación

De manera programática también es posible exponerlo como se muestra a continuación.

```
libraryBooksRelation := (RRelation new)
    name: 'LibraryBooks';

    withRoleNamed: 'Library';
    withRoleNamed: 'Book';

    all: String asDomain
           canParticipateAs: 'Library';
    all: String asDomain
           canParticipateAs: 'Book';

    a: 'Book'
           mustBeRelatedAtMost: One time
           withOthersParticipatingAs: 'Library';

    withLinkPropertyNamed: 'BookId';
    withLinkPropertyNamed: 'Topic';

    all: (String asDomainWhere: 's length = 3' )
           canParticipateAs: 'BookId';
```

```
    all: String asDomain
           canParticipateAs: 'Topic';

    a: 'Library' identifyLinksWith: 'BookId';
    a: 'Library' qualifyLinksWith: 'Topic';

    yourself.
```

En el siguiente ejemplo observamos cómo registrar libros en una biblioteca..

```
fceynLibrary := 'Biblioteca de la FCEyN de la Universidad de Buenos Aires'.

fceynLibrary
    addBook: 'Design Patterns. Elements of Reusable Object-Oriented Software'
           bookId: '843'
           topics: (Set
                          with: 'Computer Science'
                          with: 'Object-Oriented Programming');
    addBook: 'UML. El Lenguaje Unificado de Modelado'
           bookId: '844'
           topics: (Set
                          with: 'Computer Science'
                          with: 'Software Engineering').
```

Ya es possible entonces consultar por estos libros. La relación, soporta diferentes mensajes de consulta para acceder a esta información. En el ejemplo siguiente podemos observar alguno de ellos.

```
"Consulto por el libro con identificador 843. Al inspeccionar, se obtendrá el
libro 'Design Patterns. Elements of Reusable Object-Oriented Software'"

fceynLibrary bookIdenfiedBy: '843'.


"Consulto por los libros catalogados con tema 'Computer Science'.  Se obtendrá un
conjunto de libros con 'Design Patterns. Elements of Reusable Object-Oriented
Software' y 'UML. El Lenguaje Unificado de Modelado' como elementos"

fceynLibrary booksWithTopic: 'Computer Science'.
```

2.3.6 Representación de Conocimiento

Hasta aquí hemos avanzado en la descripción del modelo, desde cómo especificar una relación hasta cómo el Framework lleva adelante el proceso de adquisición y mantención de la consistencia y coherencia del conocimiento.

En esta sección nos preocuparemos por explicar cómo consideramos que el Conocimiento puede ser representando.

Como ya hemos mencionado, consideramos Conocimiento de una relación a la información representada por el conjunto de sus Vínculos –en un momento dado-. En nuestro modelo, un **`RKnowledge`** asociado a una relación refleja conceptualmente el conjunto de Vínculos existente entre "objetos con la capacidad de participar en la Relación".

Pero aunque conceptualmente el Conocimiento de una relación es un conjunto de vínculos, durante nuestro trabajo se nos han ocurrido o presentado diferentes formas, mecanismos y/o estrategias de su representación.

2.3.6.1 Estrategias de implementación del Conocimiento

La representación o implementación de un conocimiento depende de varios aspectos y dimensiones, a veces ortogonales y otras veces contradictorios. Estas características dependen tanto del dominio de problema de Relaciones, como del dominio de problema a modelar; aspectos de performance y aspectos semánticos; entre otros.

A continuación exponemos algunos de los aspectos involucrados en la representación del conocimiento de una relación.

Aspecto	Descripción
Criterio de orden entre los vínculos	Los vínculos dentro de un conocimiento pueden estar ordenados secuencialmente, ordenados bajo algún criterio lógico o no ordenados. Un conocimiento no ordenado es aquél en el cual no hay criterio de orden entre los objetos relacionados con cierto participante A. Por ejemplo, los hijos de *juan*. Por el contrario, decimos que un conocimiento es ordenado cuando los objetos relacionados con un participante A, mantienen un criterio de orden entre ellos. Por ejemplo los integrantes de una cola de atención en PagoYa. UML utiliza la etiqueta {ordered} para reflejar este aspecto
Localización	Decimos que la localización del conocimiento es interna cuando su representación se encuentra autocontenida en estructuras internas (ej: un conjunto, un grafo, etc). Por otro lado, cuando el conocimiento se encuentra almacenado en objetos o fuentes de datos externas, decimos que el conocimiento es externo. Ejemplos de esto puede ser representar el conocimiento sobre una base de datos relacional, en un motor de prolog, o en otro modelo dentro del mismo Smalltalk.
Cache	Cualquiera sea la elección de representación utilizada, es posible desear disponer de mecanismos de cache para optimizar el tiempo de respuesta de las operaciones sobre el conocimiento.
Mecanismos de deducción	A la hora de llevar adelante procesos de deducción e inferencia, también es posible establecer diferentes estrategias. Por un lado utilizando mecanismos propios, como las reglas de inferencia por Reflexividad, Transitividad, etc. Por otro lado utilizando otros subsistemas como una máquina de Prolog, que resuelva en forma lógica y natural esta problemática Otra alternativa es modelando relaciones basadas en otras relaciones. Por ejemplo *GrandfatherGrandChild* basado en *FatherChild*
Estrategias sobre cuándo deducir	Cuando las relaciones poseen propiedades matemáticas como Transitividad y Simetería es necesario deducir el conocimiento (ej: aplicar clausura transitiva) en base al existente para mantener el invariante. Entendemos que existen al menos dos estrategias para llevar adelante el proceso deductivo. La primera consistiría en deducir tan pronto se adquiere nuevo conocimiento. La siguiente esperaría y actuaría bajo demanda (eager vs. lazy).
Adquisición	El conocimiento de una relación puede ser adquirido mediante diferentes mecanismos. El primero supone la intervención de un usuario que dispare cierta secuencia de

	colaboraciones. La siguiente asume que existe un productor de conocimiento autónomo, que observa el ambiente (pattern Observer) y luego de ciertos eventos define un conjunto de hechos (ej: un productor de conocimiento para *Ancestors* basado en *FatherChild*.). Una tercer alternativa de adquisición es la calculada. El conocimiento (o su representación) saben como ir a buscar el conocimiento definido otra parte. Otros formas de adquisición podrían desprenderse al combinar las anteriores.
Ordenes algorítmicos	Aquí lo que esta en juego el orden del algoritmo de inserción / borrado o consulta de un conjunto de links. (ej. O(1), O(n),O(log n))
Estructura de representación	Aunque muchas veces ligado a la optimización del orden de complejidad, la estructura de representación puede variar de Conocimiento a Conocimiento de dos relaciones distintas. Alternativas son Conjuntos de Links, Grafos, Diccionarios, Árboles Binarios, u otras ad-hoc. Como ya hemos mencionado, es posible representar el conocimiento en fuentes externas. En este caso podríamos penser en Bases de datos relacionales, motores de prolog, archivos, u otros modelos dentro de la misma imagen Smalltalk.
Lectura y Escritura	El conocimiento de una relación podría ser de Sólo Lectura, o Lectura y Escritura.

Tabla 2. Aspectos y dimiensiones de un conocimiento.

Dada la amplitud de este dominio de problema, hemos decidido crear la abstracción **`RKnowledgeRepresentation`** a los efectos de encapsular esta problemática , independizando nuestro Framework de la misma.

A su vez, hemos dejado fuera del alcance de nuestra tesis el realizar análisis mas profundos sobre este tema –tanto conceptual como en términos concretos e implementativos.

Entendemos que existen diferentes Representaciones de Conocimiento, las que debido a sus pros y contras compiten para dar solución a distintos dominios de problema.

2.3.7 Atributos de Extensión

El modelo de relaciones permite a sus usuarios describir o especificar en forma sencilla el comportamiento de un programa para representar cierto dominio de problema. Sin embargo nuestros usuarios se encontrarán también frente a otros dominios de problemas; ortogonales al que necesitarán resolver. Uno de estos dominios es, por ejemplo, el desarrollo de la interfaz de usuario (UI) del sistema.

Como veremos durante la prueba de concepto “Editor de objetos de negocio”, es posible modelar una solución a este dominio utilizando el Framework de Relaciones para acceder a toda la información representada sobre el dominio de problema.

Relaciones brinda así un importante servicio a todo este tipo de Herramientas, al presentar la información sobre el dominio en forma ordenada, estructurada y uniforme, facilitando y simplificando el desarrollo de éstas.

Sin embargo la información que hasta aquí el modelo de relaciones puede ofrecerle a estas nuevos frameworks o herramientas no es suficiente. Por ejemplo un buen Framework de UI requerirá no solo conocer el modelo sino también como representarlo visual y gráficamente. ¿Qué editor de fechas utilizamos?; ¿utilizamos un Combo Box o un List Box para representar las categorías de un item? ¿qué

control para editar/mostrar un color utilizamos? ¿usamos el mismo control para visualizar que para editar?, etc.

Si queremos que nuestro Framework de UI sea flexible, debemos permitir que sus usuarios declaren cómo representar cada aspecto. En el caso de fechas, estos pueden desear utilizar un control para mostrarla y otro control para seleccionarla, tal como mostramos en el diagrama a continuación

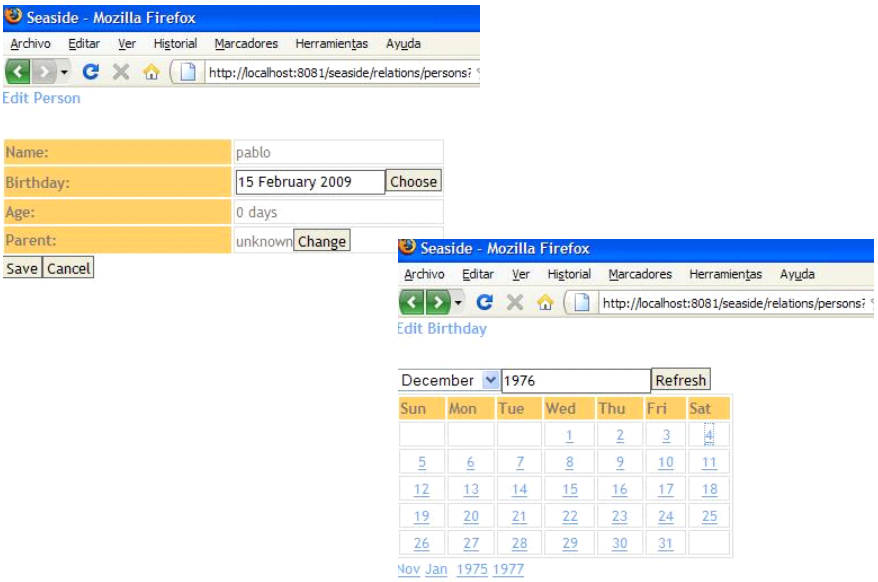


Figura 25. Editor Magritte. Ejemplo

Ahora bien, ya hemos dicho que la *Descripción del Modelo* puede proveerla el mismo Framework de Relaciones. El desafío es ofrecer desde el Framework de Relaciones, la flexibilidad suficiente para que las nuevas herramientas puedan integrarse a éste, aggiornándolo o atribuyéndole distintas propiedades, sin tener que duplicar información del dominio para representar su propio dominio.

La apuesta es brindar la capacidad de extensión de un dominio de problema expresado puramente en relaciones para permitir su integración e intercambio con otros dominios ortogonales como lo son la persistencia y la interfaz de usuario.

Dentro del alcance de nuestro trabajo optamos por relegar esta característica del modelo de relaciones. Sin embargo, creemos firmemente que mucha de la información requerida por esta clase de herramientas podría ser expresada si el Modelo y Framework proveyeran la facilidad de Atributos o Propiedades de Extensión personalizadas, ya sea de la relación, del rol, del dominio del rol, del tipo del rol, etc.

2.3.8 Operaciones sobre una Relación

Reificar el concepto Relación nos llevó a definir la clase **`RRelation`** con todas las características y protocolos necesarios para describir su estructura y su semántica. A su vez, optamos por modelar su conocimiento con un **`RKnowledge`**, el cual permite representar la información sobre sus vínculos de acuerdo a las necesidades puntuales de la Relación, el Dominio y la performance.

En esta sección nos proponemos describir brevemente los protocolos que dimos a **`RRelation`** a los efectos de permitir modificar su conocimiento, consultarlo e iterar sobre sus vínculos.

Así, definimos los siguientes 3 protocolos en **`RRelation`**

- *Adding Participation*
- *Testing Participation*
- *Removing Participation*

La conjunción de estos protocolos brinda a los clientes de una Relación la posibilidad de agregar y remover vínculos, consultar por la pertenencia de cierto objeto, seleccionar un subconjunto de sus vínculos según determinado criterio, iterar sobre los vínculos aplicando determinada acción, entre otras.

A continuación presentamos con mayor detalle estos protocolos.

2.3.8.1 Protocolo adding-participation

Este protocolo permite por ejemplo que el conocimiento sobre hecho "Juan es padre de Pedro" pueda ser volcado a la Relación *FatherChild*. Para esto podría utilizarse el siguiente método de la clase **`RRelation`**

```
add: aObject as: aRoleIdentifier
      relatedTo: bObject as: bRoleIdentifier
```

Así por ejemplo, el método `fatherOf:` sobre una persona puede implementarse tal como se muestra a continuación:

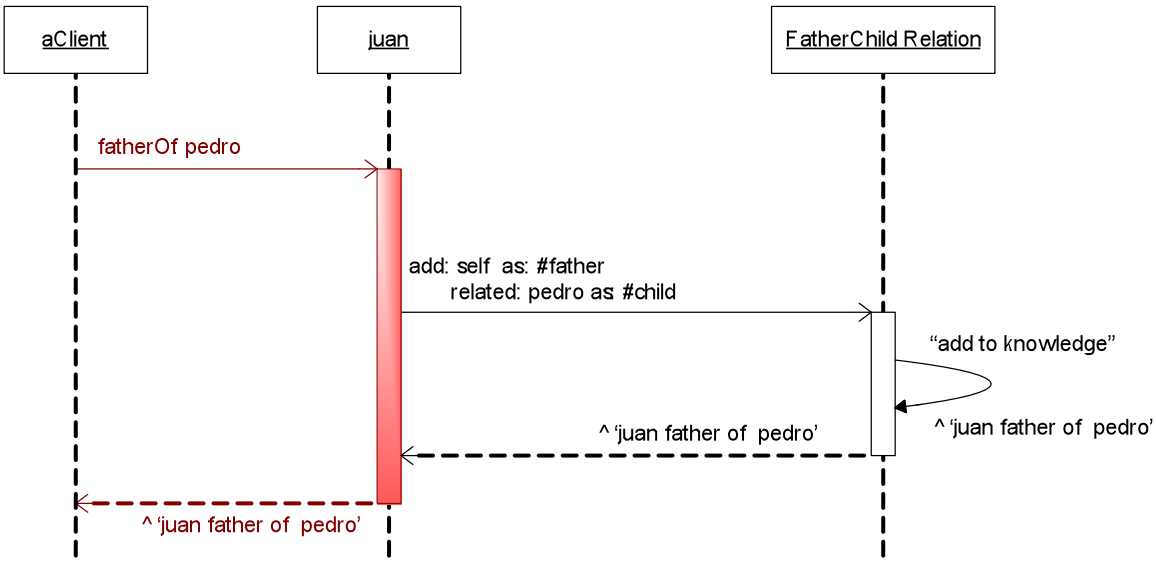


Figura 26. Protocolo de RRelation. Diagrama de Secuencia.

En muchas ocasiones podemos desear también expresar o describir quienes son Todos los hijos de Juan. Por ejemplo, un cliente de la Relación podría desear especificar "Pedro y José son (todos) los hijos de Juan". El protocolo ofrece aquí la siguiente interfaz

```
add: aObject as: aRoleIdentifier
      relatedToAll: bCollection as: bRoleIdentifier
```

indicando que cada uno de los `bCollection` es un `bRoleIdentifier`. Luego, siguiendo la línea del ejemplo anterior, podemos expresar el método `fatherOfAll:` tal como se expresa a continuación.

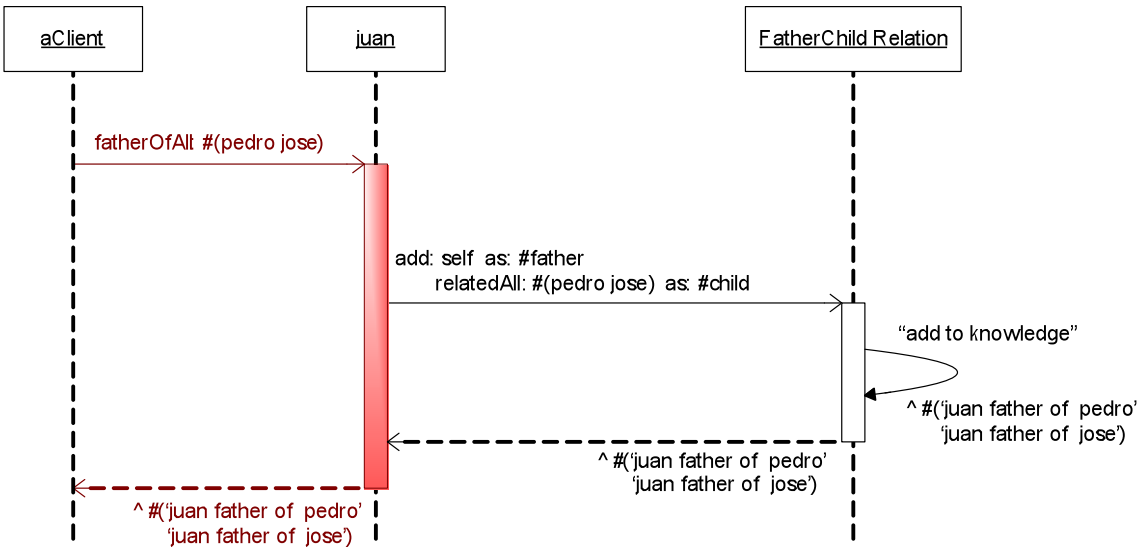


Figura 27. Protocolo de RRelation. Diagrama de Secuencia 2

> **Nota** Esta modalidad de interfaces sólo son provistas para relaciones de grado hasta 3. Para agregar nuevos vínculos a una relación de aridad mayora a 3 se deben utilizar métodos genéricos.

Más adelante, en la sección "Operaciones genéricas", se mencionan interfaces genéricas, no dependientes de la aridad o grado de la relación para agregar vínculos .

2.3.8.2 Protocolo removing-participation

En forma similar, este protocolo se encarga de definir el conjunto de métodos que permiten remover vínculos del conocimiento de una relación, por ejemplo

```
remove:as:
remove:as:related:as:
remove:as:relatedAll:as:
```

> **Nota** Esta modalidad de interfaces sólo son provistas para relaciones de grado hasta 3. Para agregar nuevos vínculos a una relación de aridad mayora a 3 se deben utilizar métodos genéricos.

2.3.8.3 Protocolo testing-participation

El tercer protocolo responde a la necesidad o interés de consultar el estado del conocimiento.

Por ejemplo, en primer lugar para saber si un objeto esta actuando bajo cierto rol, para responder a preguntas del estilo ¿Es Juan un Padre?. Aquí **`RRelation`** provee el método `isParticipating:as:`

A partir de este, podemos modelar el método `isFather` sobre una persona en función de este, como mostramos en el diagrama a continuación.

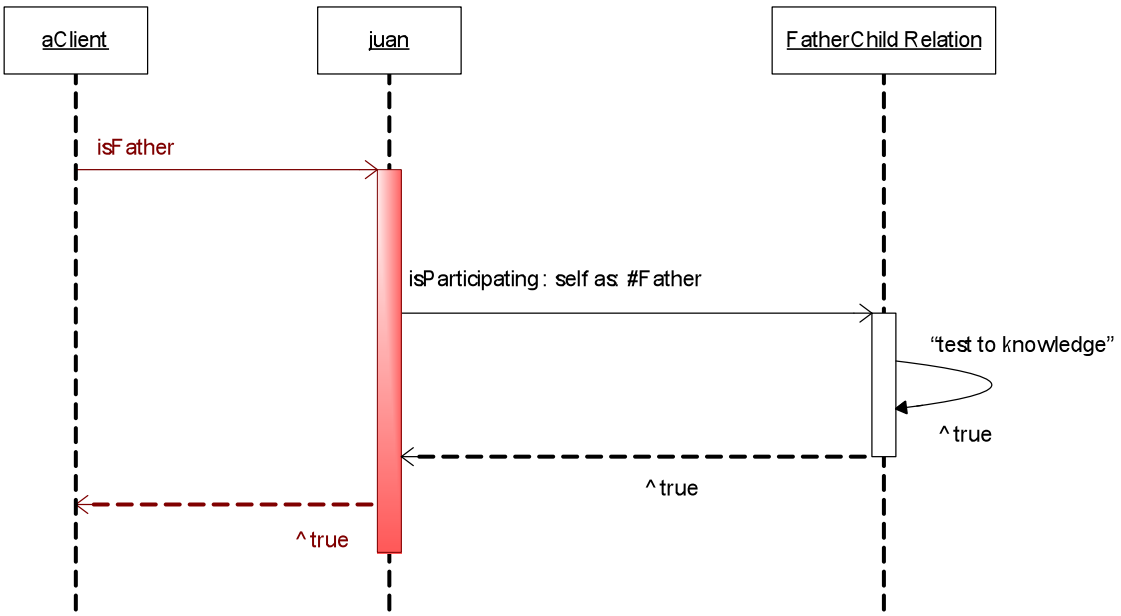


Figura 28. Protocolo de RRelation. Diagrama de Secuencia 3

Otra pregunta frecuente que deseamos responder son las del tipo ¿Es Juan el padre de Pedro?. Nuevamente aquí este protocolo nos ofrece herramientas para responderla, proveyéndonos de la interfaz `is:as:related:as:`

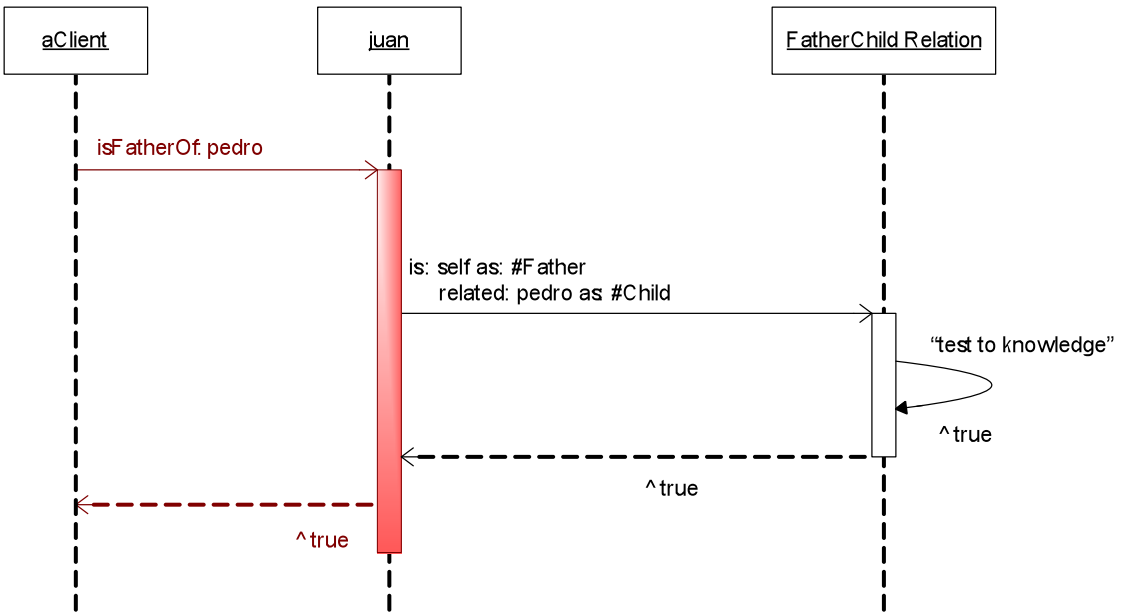


Figura 29. Protocolo de RRelation. Diagrama de Secuencia 4

En base a la expresividad de este protocolo, podemos consultar a la relación para responder a preguntas del estilo ¿Juan es el padre Pedro y de José? con el método `is:as:relatedAll:as:`

También resulta de particular interés obtener y seleccionar los objetos relacionados con cierto otro objeto al, por ejemplo, desear responder cosas como ¿quién es el padre de Pedro? o ¿cuáles son los hijos de Juan?. Con este fin, proveímos al protocolo métodos como los siguientes

```
participatingAs:related:as:
allParticipatingAs:related:as:
```

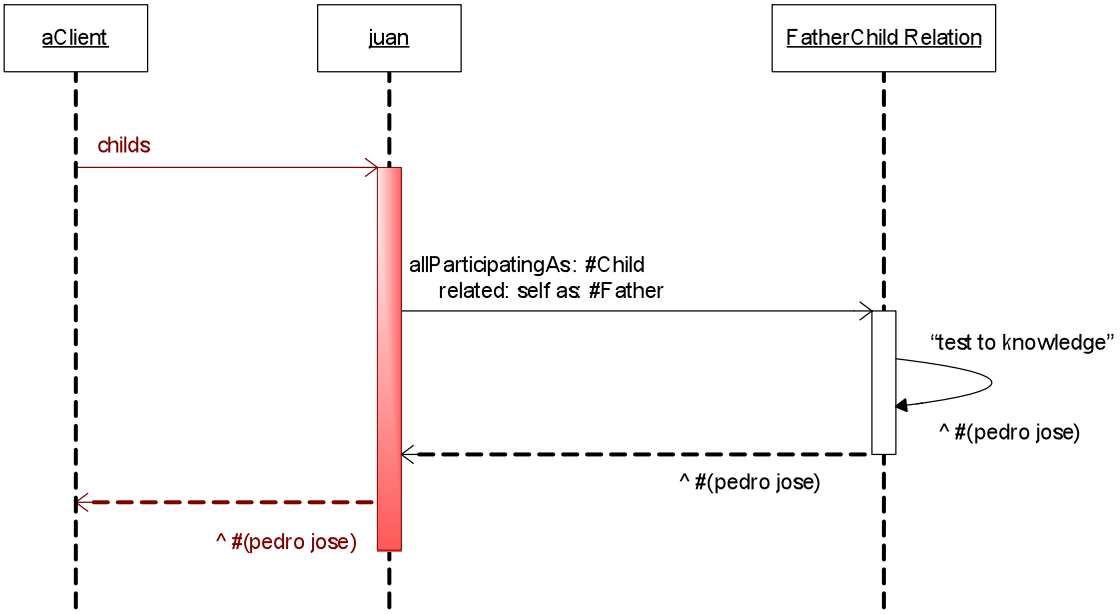


Figura 30. Protocolo de RRelation. Diagrama de Secuencia 5

Nota Este tipo de interfaces solo son provistas para relaciones de grado hasta 3. Para remover nuevos vínculos a una relación de aridad mayora a 3 se deben utilizar métodos genéricos.

2.3.8.4 Protocolo de operaciones genéricas

En las secciones anteriores mostramos como interactuar con **`RRelation`** para agregar, remover y consultar su conocimiento en relaciones de hasta grado 3.

En este punto fue donde consideramos que continuar desarrollando adaptaciones al protocolo en función del grado de la relación no era el camino apropiado. Decidimos entonces abstraer los casos anteriores proveyendo un nuevo protocolo, genérico, que permitiese realizar estas operaciones sin importar el grado de la relación en cuestión.

Detallamos en este protocolo en el anexo "Detalle de Operaciones sobre Relación"

2.3.8.5 Otras operaciones y/o complementos

Además de los protocolos definidos, existen otras alternativas o formas de operar sobre relaciones que brindan mayor riqueza al modelo y la solución en general.

A pesar de que hemos decidido dejarlos fuera del alcance de esta tesis, en las próximas secciones presentamos un breve resumen de estas posibilidades.

Enunciados por comprensión o por extensión

Supongamos que deseamos expresar el siguiente hecho "Juan es amigo de todos sus compañeros de trabajo". Aquí nos encontramos con un enunciado por comprensión, y un modelo que parece ofrecer sólo la capacidad de expresar hechos por extensión. ¿Cómo adaptar nuestro modelo para soportar esta nueva característica?

La preguntas a resolver serían a) cómo almacenar esta información y b), como expresar la misma (¿en función de links, o en función de hechos?).

En primer lugar, tendríamos que utilizar SSet para expresar los conjuntos por Comprensión: "Todos los compañeros de trabajo de Juan". Luego, utilizar una de las interfaces definidas y transferirle la responsabilidad al **`RKnowledge`** de la relación.

Cuando el **`RKnowledge`** detecte que es un conjunto por Comprensión podrá actuar dependiendo de si el conjunto es numerable o no numerable. En caso de ser numerable, podría definir un **`RKnowedgeProducer`** *on-the-fly* para que atienda a los cambios de este conjunto, y modifique este conocimiento cuando esto ocurra. Por el contrario, si el conjunto es no numerable, podría utilizar una **`RKnowledgeRepresentation`** especial que soporte información en forma de Hechos tal como el que ha sido enunciado y no pretenda transformarlo en links (ya que esto no será posible porque el conjunto por comprensión es no numerable y no se puede iterar sobre éste).

2.3.9 Notificación de Eventos

Independientemente del mecanismos de Representación del conocimiento de una relación, todo vínculo de una relación se encuentra conceptualmente contenido en dicho conocimiento.

Por este motivo, hemos decidido brindar a la Relación la capacidad de notificar los cambios en este conocimiento a los objetos interesados.

Así, **`RRelation`** implementa un mecanismo por el cual las Relaciones notifican a los interesados cuando cambian los vínculos que conforman su conocimiento. El mecanismo se compone de dos partes:

- Suscripción a la relación de los objetos a los cuales se debe notificar
- Detección de los cambios en el conocimiento de la relación

2.3.9.1 Suscripción de objetos a los cuales notificar

Squeak trae incorporada una implementación del pattern Observer [Gamma et al. 1995], mecanismo mediante el cual un objeto puede observar a otro y ser notificado cuando ocurre un cierto evento. El concepto es que el observador se suscribe a un tipo de evento del objeto observado y cuando ocurre un evento de este tipo, el observador recibe un mensaje.

En nuestro modelo utilizamos este mecanismo para que otros objetos puedan enterarse cuando hay un cambio en el conocimiento de la relación. Dependiendo del cambio producido y detectado, pueden dispararse diferentes eventos, a saber

- `#knowledgeChanged`
- `#linkAdded`
- `#linkRemoved`
- `#linkChanged`

Para que un cliente se suscriba a estos eventos basta con enviar a la relación un mensaje solicitando dicha suscripción. Los mensajes son los siguientes:

- `#whenKnowledgeChangedSend:to:`
- `#whenLinkAddedSend:to:`
- `#whenLinkRemovedSend:to:`
- `#whenLinkChangedSend:to:`

Cuando un objeto suscripto a un evento deja de estar interesado en el mismo, no tiene más que enviar un nuevo mensaje a la relación, indicando su falta de interés en el mismo. Así, análogamente tenemos los siguientes mensajes

- `#removeKnowledgeChangedNotificationsTo:`
- `#removeLinkAddedNotificationsTo:`
- `#removeLinkRemovedNotificationsTo:`
- `#removeLinkChangedNotificationsTo:`

Cuando se desea que una Relación deje de enviar o emitir notificaciones en general, pueden utilizarse los siguientes mensajes:

- `#removeKnowledgeChangedNotifications`
- `#removeLinkAddedNotifications`
- `#removeLinkRemovedNotifications`
- `#removeLinkChangedNotifications`

A modo de ejemplo, pensemos en la relación *Categorization*. Supongamos que disponemos de una herramienta gráfica –`CategoryRelationPresenter`- que pretende ser Observer de los cambios en el conocimiento de la relación. Una posible implementación de esta integración podría resultar en un código similar al que mostramos a continuación

```
CategoryRelationPresenter >> registerToEvents
    self relation
            whenLinkAddedSend: #onLinkAdded: to: self;
            whenLinkRemovedSend: #onLinkRemoved: to: self;
            yourself.

CategoryRelationPresenter >> unregisterFromEvents
    self relation
            removeLinkAddedNotificationsTo: self;
            removeLinkRemovedNotificationsTo: self;
            yourself.

CategoryRelationPresenter >> onLinkAdded: aLink
    self
            addCategoryIfNotExists: aLink category;
            addTask: aLink task to: aLink category.

CategoryRelationPresenter >> onLinkRemoved: aLink
    self removeCategory: aLink category.
```

2.3.9.2 Detección de cambios en el conocimiento de la relación

Ya dijimos que cuando una relación detecta un cambio en sus elementos, va a notificarlo a los observadores. Ahora bien, ¿cómo detecta los cambios?

Esto es dependiente del **`RKnowledge`** utilizado. Un **`RKnowlege`** Explícito podrá determinar y notificar estos cambios luego que se haya agregado o eliminado un vínculo y se haya finalizado su posterior deducción.

Por otra parte, un `RKnowledge` por comprensión podrá utilizar un mecanismo similar al que utiliza el Framework de SSet[Altman-Tylim], es decir, disponer de *Change Detectors* especializados. Los *Change Detectors* son básicamente objetos encargados de detectar los cambios y avisar al `RKnowledge`. De

esta manera, puede decidirse e implementarse en cada caso la mejor estrategia para la detección de cambios.

2.3.10 Relaciones n-arias.

Hasta aquí nos hemos ocupado de describir Relaciones en general, sin involucrarnos demasiado en la aridad o grado de las mismas. Sin embargo, nuestro modelo es capaz de manejar relaciones *n*-arias, siendo *n* cualquier natural mayor o igual a 2.

> **Nota** El modelo de relaciones desarrollado contempla la posibilidad de definir relaciones n-arias como las de los ejemplos que siguen en esta sección. Sin embargo existen algunos algunos aspectos que aún deben ser analizados y explorados con mayor profundidad en futuros trabajos. Uno de ellos es la la interpretación semántica de multiplicidad y su relación con participantes requeridos de un vínculo.

Las relaciones n-arias son consideradas por la bibliografía actual como relaciones poco frecuentes en el modelado de diferentes dominios de problema. Aunque durante el inicio de nuestra tesis no encontramos objeción a tal postulado, avanzada la misma encontramos algunas situaciones donde la utilización de relaciones n-arias podría resultar de utilidad o al menos ser una alternativa de implementación.

> **Nota** Algunos autores consideran las relaciones binarias con identificadores cómo si fueran ternarias con características especiales [Rumbaugh 1987].

Ya hemos presentado un ejemplo donde se utiliza la relación *Observation* –ternaria- para modelar el problema de un `Observer` observando un `EventType` en un `Observee`.

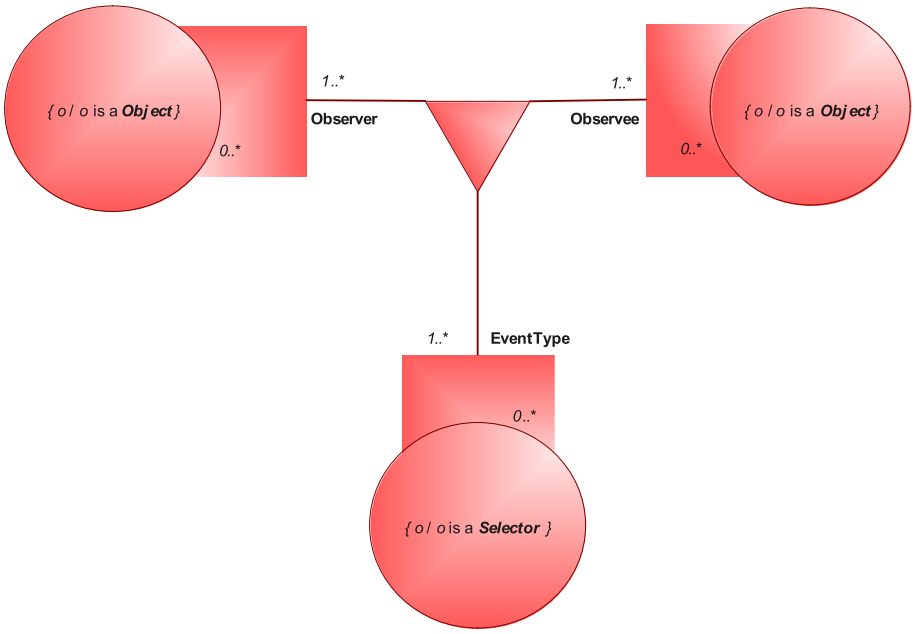


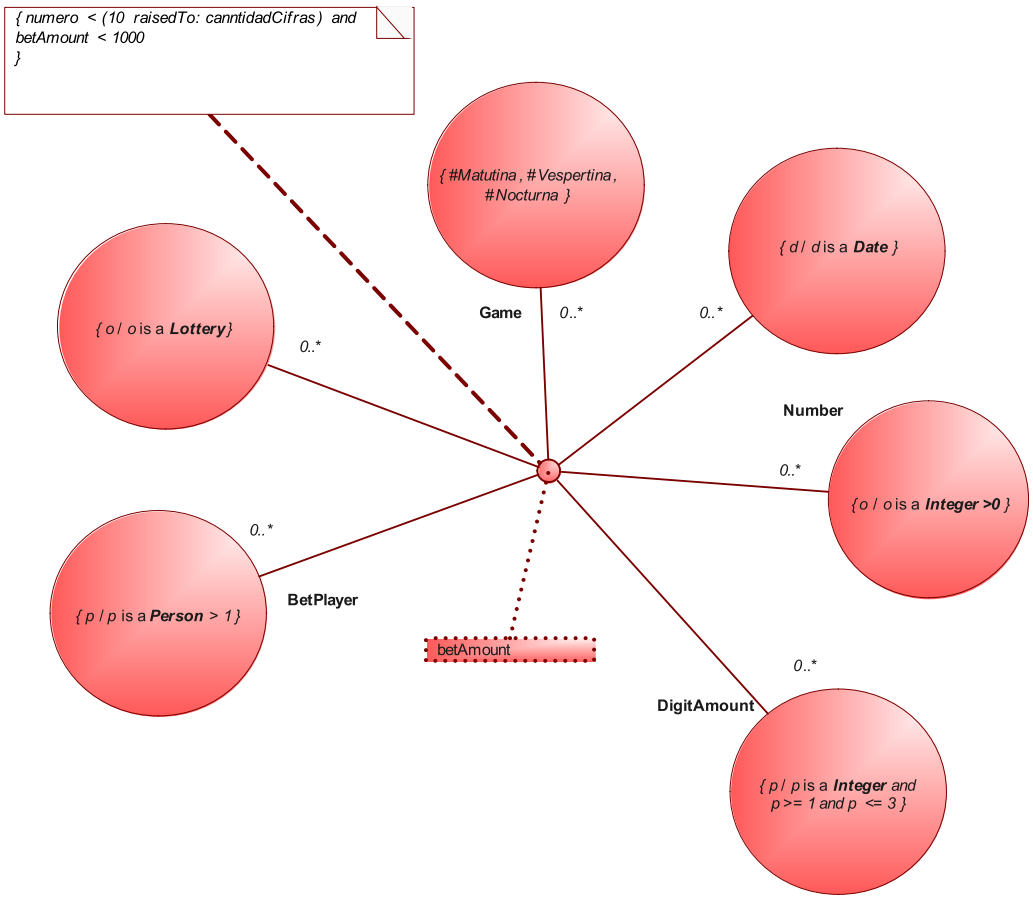
Figura 31. Observation. Diagrama de Relación (ternaria)

Otro ejemplo, seguramente más controversial y cuestionable, sea modelar una apuesta de de lotería con relaciones.

En el enfoque clásico del paradigma de objetos, modelar una apuesta de lotería seguramente resulte en la construcción de la clase `LotteryBet`. Partiendo de este supuesto

- `una LotteryBet` modelará una apuesta, con su fecha de juego, el tipo de juego (ej. Vespertina), la lotería (ej. Loteria Nacional), el apostador, la cantidad de cifras a las que apuesta y el número apostado. También incluirá el monto apostado;
- al abordar el problema, el programador tendrá que codificar las reglas que indiquen si una apuesta es válida (ej. que el número apostado sea de la cantidad de digitos a los que se apuesta, que el monto apostado sea menor a 1000 pesos[10], o que la lotería sea una de las definidas);
- se deberá resolver cómo almacenar las apuestas, definiendo en un objeto responsable de contenerlas. Este contenedor deberá a su vez proveer métodos de acceso para consultar y acceder a las apuestas bajo diferentes criterios;
- siguiendo esta solución, cuando sea necesario editar gráficamente una apuesta será necesario conocer los distintos candidatos (todos los juegos, todas las posibles loterías, y las posibles cifras, así como el calendario de días disponibles). Éstos deberán ser provistos por el modelo subyacente.

La existencia de un modelo de relaciones permite pensar el problema desde otro ángulo: uno donde exista una relación *LotteryBet* que describa vínculos que representen apuestas de lotería. En la figura siguiente se presenta una descripción gráfica de la relación.



[10] Suponiendo que existiese dicho límite.

Figura 32. Apuesta de loteria. Diagrama de Relación

Es evidente que modelar éste problema con relaciones no hace el dominio de problema más simple, pero el soporte que brinda el framework podría ayudar a manipular su complejidad accidental[11]

- en forma declarativa se describiría la relación *LotteryBet* con sus roles: fecha de juego, el tipo de juego, la lotería, el apostador, la cantidad de cifras a las que apuesta y el número apostado.
- se representaría el monto apostado como una propiedad del vínculo;
- el programador definiría las validaciones de negocio sobre las apuestas, sin ocuparse de aquellas inferidas de la cardinalidad, multiplicidad y dominios de los roles;
- la misma relación y su representación de conocimiento se encargarían de contener y servir de repositorio para el universo de apuestas;
- al contar con los dominios de cada rol, el modelo podría ofrecer a editores gráficos la información sobre los distintos candidatos en condiciones de participar.

Al inicio del trabajo esta solución nos hubiese parecido poco natural o forzada. Al familiarizarnos con relaciones estas sensaciones fueron desapareciendo y hoy creemos que puede ser una solución alternativa viable. No hemos profundizado ni madurado el problema lo suficiente como para arribar a conclusiones sobre las ventajas, desventajas y límites de uno y otro enfoque, pero creemos que sería una línea interesante de desarrollar en futuros trabajos.

2.4 Herramientas

Recorriendo el camino hacia la completa reificación del concepto Relación en los ambientes de objetos basados en clases, notamos que la definición una clase **`RRelation`** no es suficiente para dejar a las relaciones a un mismo nivel semántico que -por ejemplo- las clases, y la relaciones *InstanceOf* y *SublcassOf*.

El principal argumento para sostener esta afirmación es que la mayoría de los lenguajes dan soporte a estas construcciones tanto sintáctica como semánticamente. Necesitamos entonces construir un soporte sintáctico, semántico[Rumbaugh 1987] y un conjunto de herramientas sobre las relaciones en forma similar al que soportan las clases y las relaciones *InstanceOf* y *SublcassOf*.

En este sentido, hemos desarrollado y trabajado sobre diversas herramientas que facilitan el trabajo del programador para manipular relaciones.

2.4.1 Browser de Relaciones

En forma similar al `ClassBrowser` de Squeak, hemos creado el **`RRelationBrowser`** con el fin de poder manipular, crear y editar relaciones.

En la figura siguiente observamos un **`RRelationBrowser`**.

[11] Entendemos por complejidad accidental aquella que agrega el software al implementar cierto dominio. La complejidiad del dominio la llamamos escencial.

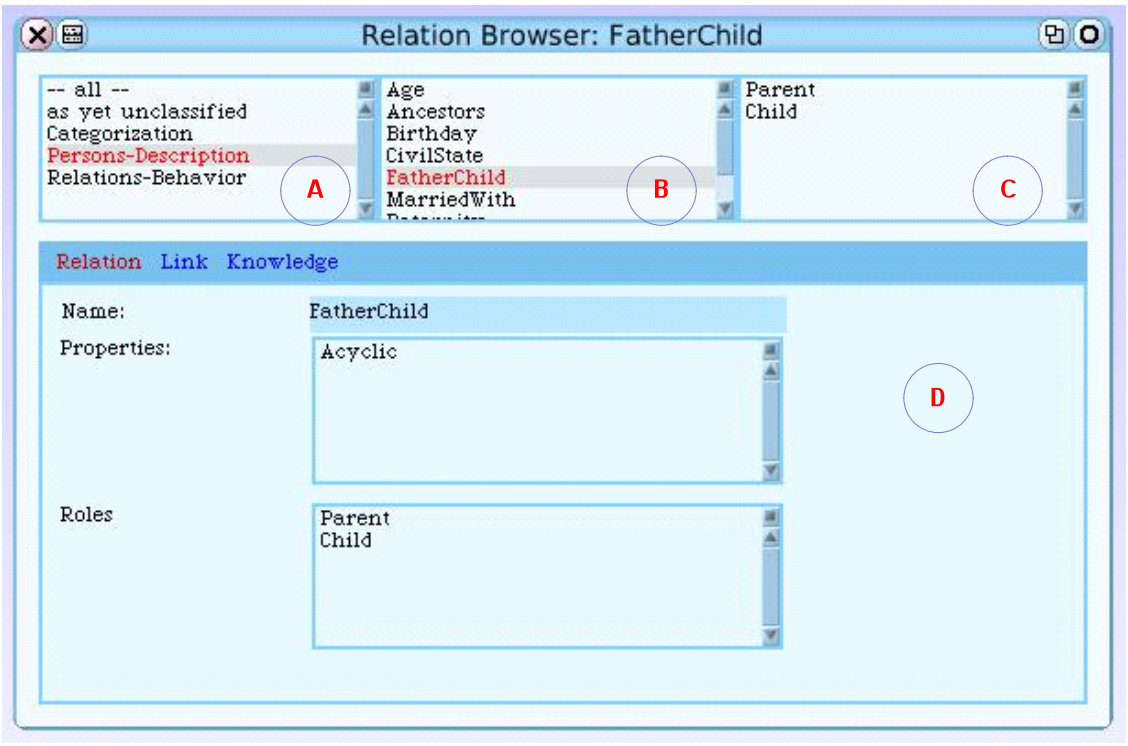


Figura 33. RRelationBrowser. Browser de Relaciones. FatherChild

El panel A presenta el dominio de problema en que aplica una relación. De esta manera podemos clasificar las relaciones y operar con ellas en diferentes y ortogonales dominios de problema. Por ejemplo, la relación *FatherChild* se encuentran dentro en la categoría `Persons-Description`.

El panel B muestra un simple listado de las relaciones actualmente registradas en el **`RRelationManager`** del ambiente para el dominio seleccionado en A. Seleccionando una de ellas se despliegan sus roles en el panel C, e instancia un editor de la relación en D. Llamamos a este editor **`RRelationInspector`**.

Como se muestra en la siguiente figura, al elegir uno de sus roles, la vista del inspector D cambia para presentar el editor del rol seleccionado.

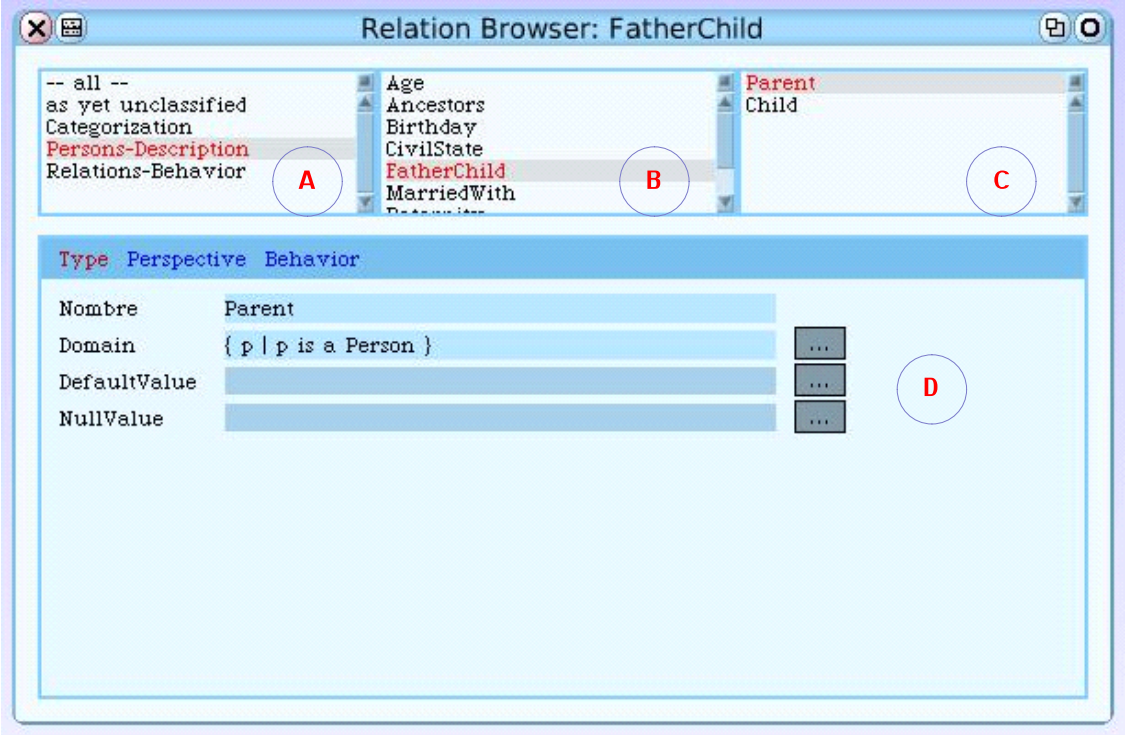


Figura 34. RRelationBrowser. Browser de Relaciones. FatherChild (Father role)

Los inspectores permiten presentar y editar la descripción de una relación y las características de sus roles.

> **Nota.** En el marco de este trabajo dejamos fuera del prototipo funcional la posibilidad de refactorizar relaciones ya registradas. Por tal motivo el **`RRelationBrowser`** permite visualizar una relación existente y crear nuevas relaciones.

En la siguientes secciones presentamos al inspector de relaciones y al de roles.

2.4.2 Inspector de Relación

Este inspector tiene por objeto trabajar y operar sobre la definición de una Relación, dejando fuera de su alcance inspeccionar el estado de conocimiento (conjunto de vínculos).

Como observamos en la Figura 33, el inspector organiza la presentación de una relación en tres lengüetas: *Relation*, *Link* y *Knowledge*. Además, permite que el explorador de relaciones le indique que especialice su vista en cierto rol seleccionado por el usuario.

Al hacer clic sobre *Relation* se puede acceder y manipular la información esencial de la relación, como el nombre, el conjunto de roles y sus propiedades.

La solapa *Link* conceptualiza la especificación sobre cómo estará conformado un vínculo de esta relación, permitiendo definir cuál será su tipo, cuáles sus propiedades y qué restricción de negocio aplica sobre ellos.

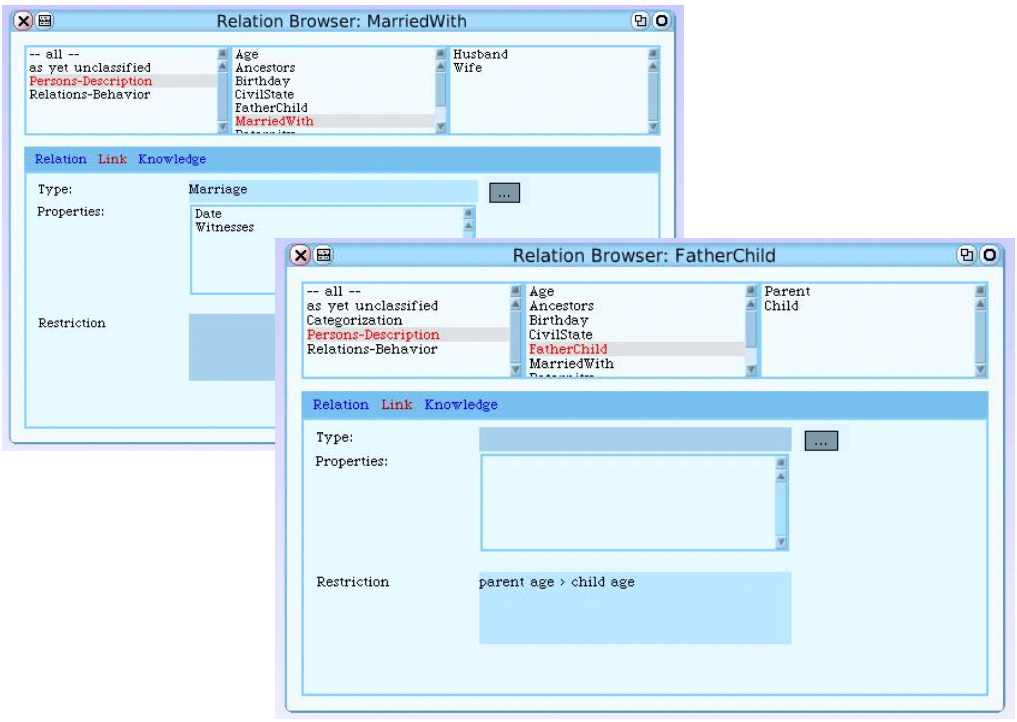


Figura 35. RRelationInspector. Solapa 'Link'

La solapa *Knolwedge* nos ofrece la posibilidad de analizar cómo es o especificar cómo deseamos que sea la representación del conocimiento de la relación bajo inspección. Dada la complejidad de esta información hemos optado delegar esta responsabilidad a un objeto especializado en ello: **`RKnowledgeSpecificationInspector`**.

Por último, y tal como se aprecia en la Figura 34, cuando se le solicita que muestre la información relativa a cierto Rol invoca a un inspector especializado (**`RRolInspector`**)

2.4.2.1 Inspector de Rol

Como hemos dicho, el inspector de relación se apoya en el **`RRolInspector`** para presentar la información de cierto Rol en la relación.

En el ejemplo siguiente podemos observar al **`RRolInspector`** inspeccionando el rol `Category` de la relación *Categorization*. El inspector presenta nuevamente tres nuevas solapas: *Type*, *Perspective*, *Behavior*.

La solapa *Type*, muestra la información de Tipo que describe un rol, como ser su Nombre, el Dominio de valores que puede tomar, el Valor predeterminado y en caso de corresponder, el Valor nulo de dicho tipo.

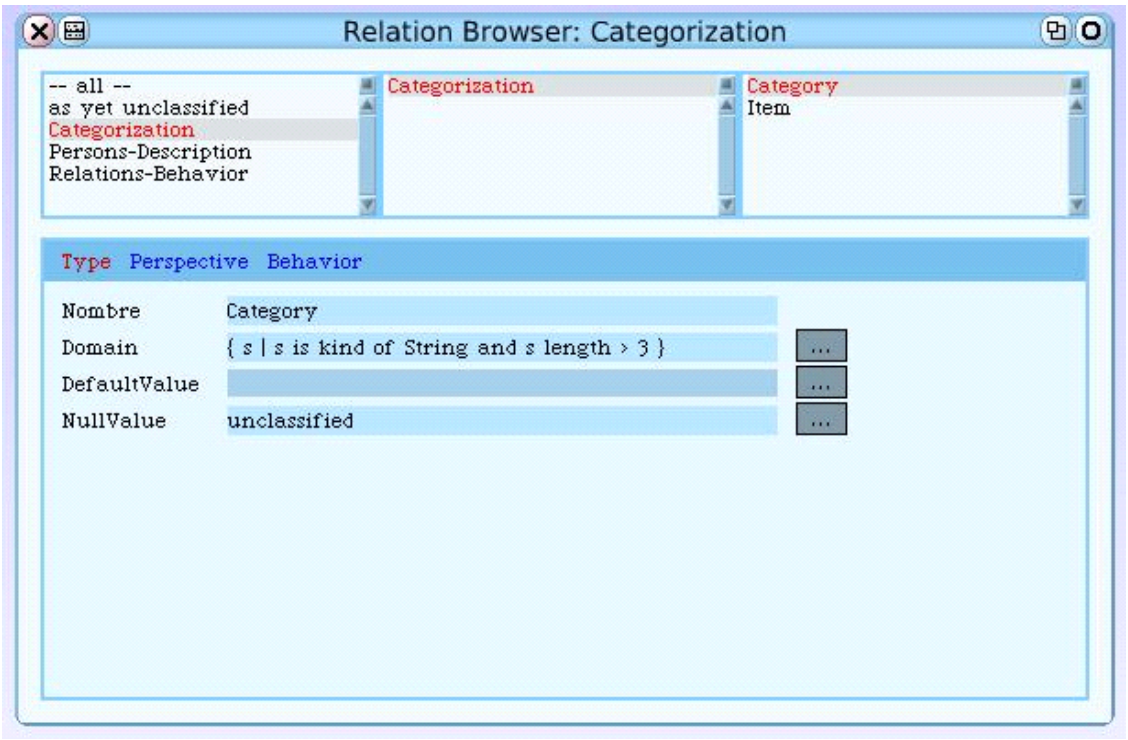


Figura 36. RRolInspector. Solapa 'Type'

La próxima solapa, *Perspective*, es responsable de presentar la perspectiva de este rol en la relación. Así, como puede apreciarse en la próxima figura, brinda la lógica para expresar las especificationes de cardinadad un valor en dicha relación, multiplidad de su rol con respecto a los restantes roles de la relación, cualidades sobre si "es navegable desde los restantes roles".

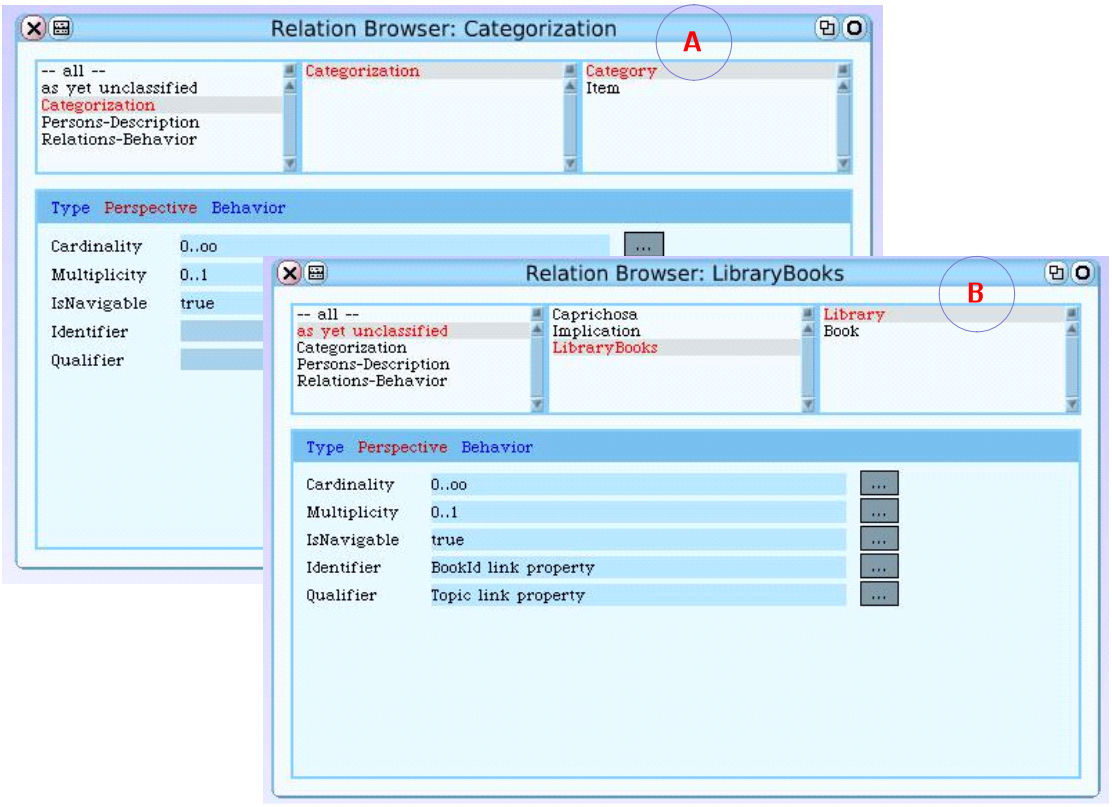


Figura 37. RRolInspector. Solapa 'Perspective'

El ejemplo B de la figura anterior nos permite comprender con mayor facilidad la semántica de los campos Identifier y Qualifier. Estos indican qué propiedades de vinculación (definidas al describir el Link) hacen la veces de identificadores y cuáles las de calificadores del vínculo -desde la perspectiva de un participante de este rol-.

La lengüeta llamada *Behavior* presenta información correspondiente al comportamiento que tendrán los objetos participantes de este rol y será desarrollada en la sección Herramientas del próximo capítulo.

2.4.2.2 Inspector de la Especificación de un conocimiento

Como ya hemos expresado, la representación o implementación de un conocimiento depende de varios aspectos y dimensiones, a veces ortogonales y otras veces contradictorios. Estas características dependen tanto del dominio de problema de Relaciones, como del dominio de problema a modelar; aspectos de performance y aspectos semánticos; entre otras.

El espíritu de un **`RKnowledgeSpecificationInspector`** es ofrecer una herramienta que permita describir estas características de forma clara, precisa y consistente.

A su vez, hemos dejado fuera del alcance de nuestra tesis realizar análisis más profundos sobre este tema –tanto conceptual como en términos concretos e implementativas

> **Nota.** Como ya hemos expresado, el dominio de problema de la representación de conocimiento ha quedado fuera del alcance de nuestro trabajo. Por tal motivo esta herramienta se encuentra en estado prototipico, y no debe ser considerada un producto acabado del modelo. Es más, muchas de sus vistas no son 100% funcionales. Nuestro objetivo ha sido hacer un cierre gráfico del modelo y exponer nuestras ideas de solución para futuros trabajos.

Su vista de presentación vuelve a encontrarse organizada por lenguetas que representan los aspectos y dimensiones.

La solapa *Location* -ver Figura 38 (a)- presenta las alternativas sobre dónde representar el conocimiento.

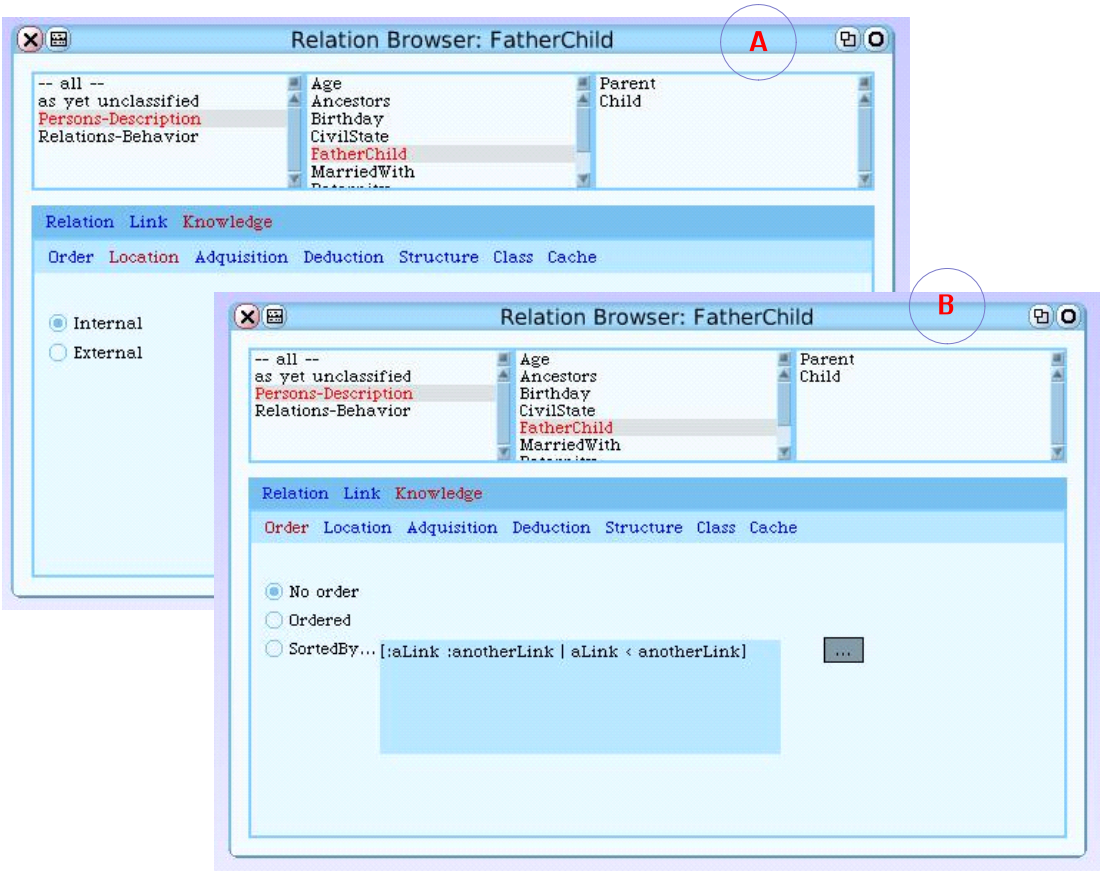


Figura 38. RKnowledgeSpecificationInspector. Solapas 'Order' y 'Location'

La solapa *Order* -ver Figura 38 (b)- tiene por objetivo exponer el orden en que deben ser organizados los vínculos en los que participa cierto objeto en una relación.

En la figura siguiente podemos observar las alternativas de adquisición de conocimiento. En particular, el ejemplo nos muestra cuales son los productores candidatos a ser productores de conocimiento de *Ancestors*.

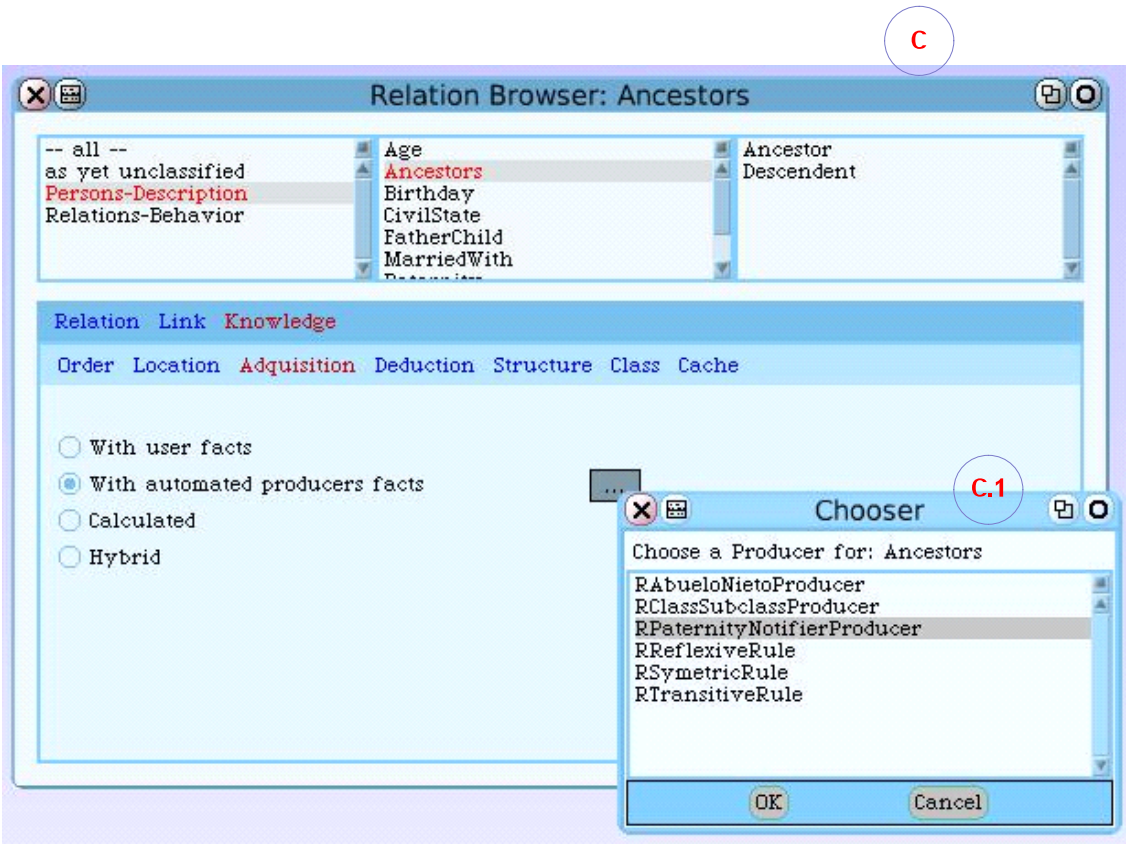


Figura 39. RKnowledgeSpecificationInspector. Solapa 'Acquisition'

Otra de las decisiones a tomar mientras se describe cómo se comportará el conocimiento de una relación es la estrategia de deducción a realizar (si es que corresponde). La solapa *Deduction* (ver siguiente figura) nos permite indicar la opción deseada.

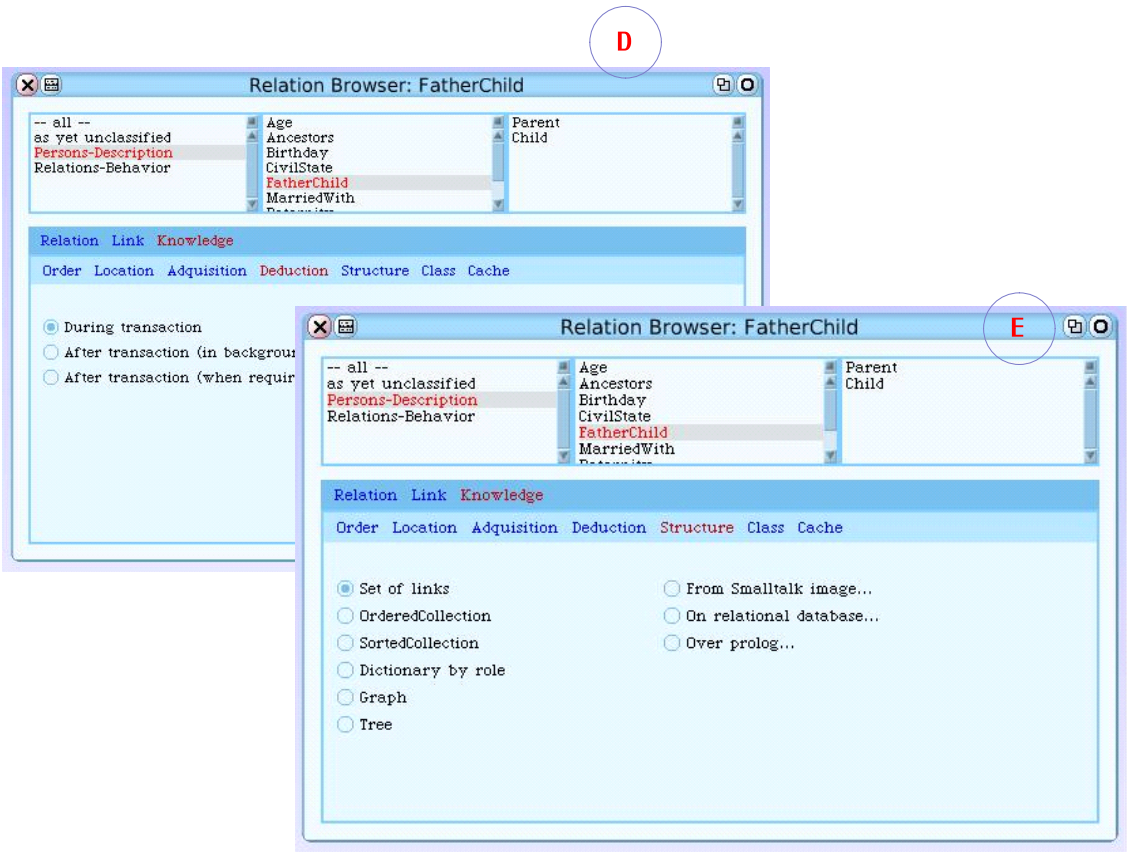


Figura 40. RKnowledgeSpecificationInspector. Solapas 'Deduction y 'Structure'

Con respecto a las estrategias y estructuras de implementación existe una gran variedad de posibilidades. La solapa *Structure* (ver figura anterior) es la responsable de presentar estas alternativas.

También es posible optar por una implementación particular de Conocimiento. Para ello existe la solapa Class, la cual lista todas los tipos de **`RKnolwedge`** que aplican a la relación en cuestión.

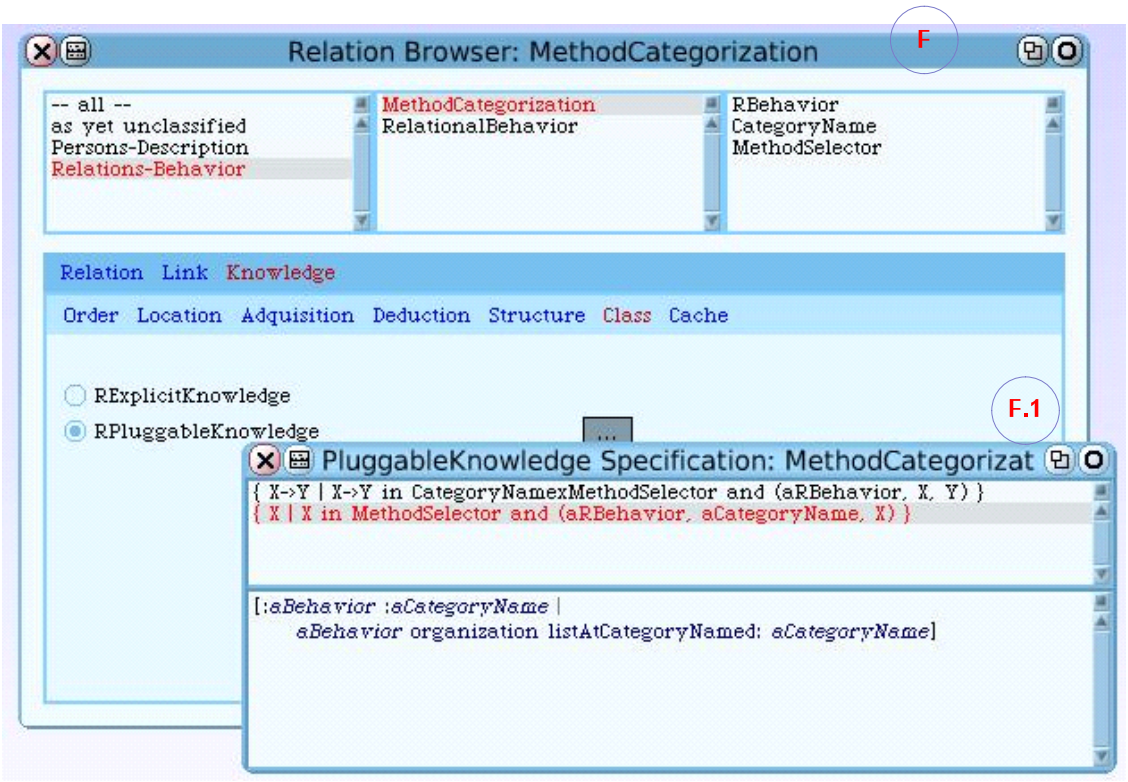


Figura 41. RKnowledgeSpecificationInspector. Solapa 'Class'

En la figura anterior podemos observar que al seleccionar la clase RPluggableKnowledge (figura F) el editor nos permite enchufar comportamiento siguiendo una semántica clara y uniforme (F.1).

Para finalizar, es posible detallar el criterio de Cache que deseamos tenga nuestro conocimiento.

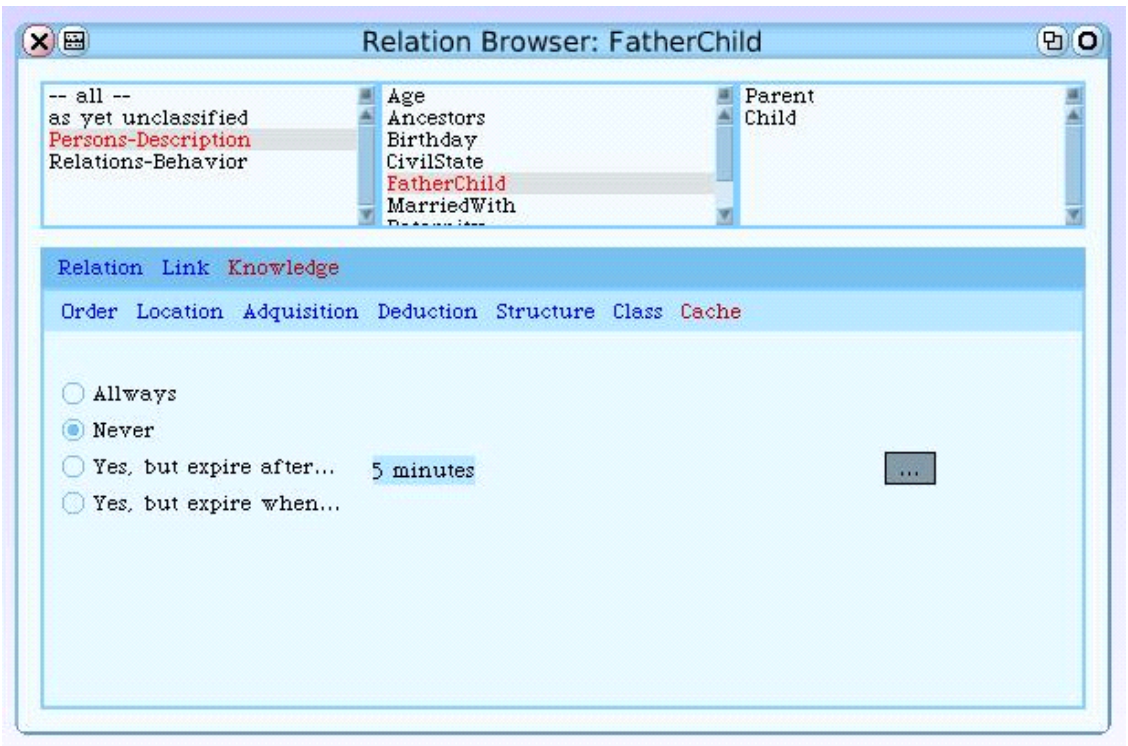


Figura 42. RKnowledgeSpecificationInspector. Solapa 'Cache'

2.4.3 Inspector de un objeto

Al igual que los exploradores, los inspectores son una importantísima herramienta que permiten a usuarios del ambiente interactuar y dialogar con los objetos directamente.

Si un objeto adquiere nuevas capacidades, cualquiera que fuesen, éstas deberían ser presentadas a través de su inspector.

Al permitir a un objeto participar en una relación, estamos aumentando las capacidades de dicho objeto. En consecuencia el inspector clásico no es suficiente, ya que sólo es capaz de presentar las variables de instancia de un objeto, cuando también resultaría de sumo interés mostrar las relaciones en las cuales el objeto estuviera en condiciones de participar.

En consecuencia hemos desarrollado un inspector que toma en cuenta también esta nueva capacidad. En la figura siguiente presentamos un ejemplo.

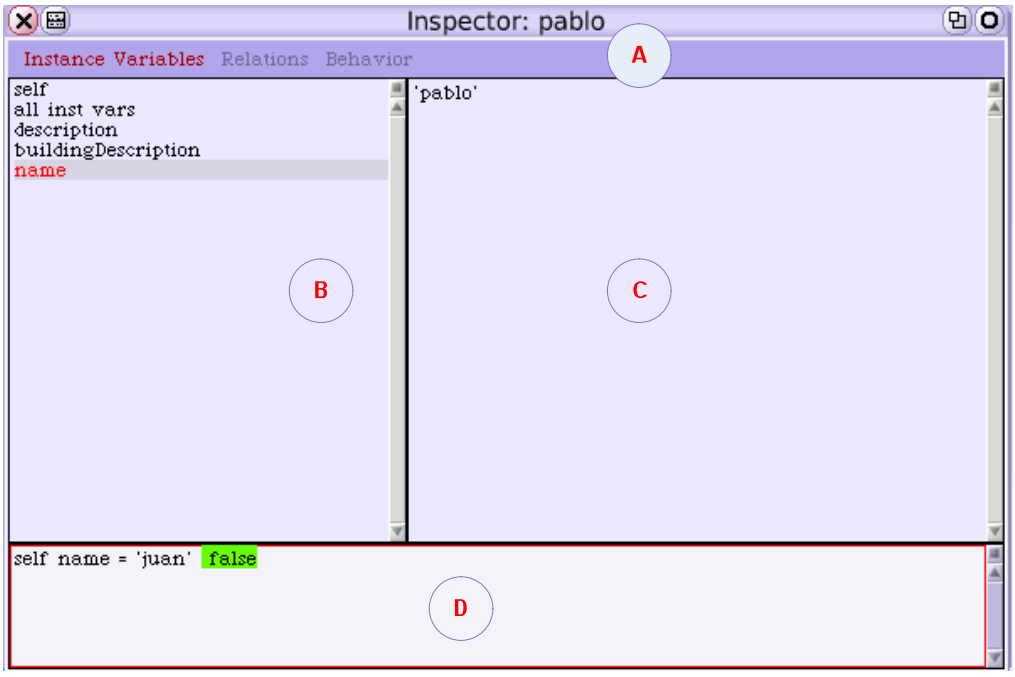


Figura 43. RObjectInspector. Inspector de un objeto

En las etiquetas de la franja A, podemos observar: "Instance Variables", "Relations" y "Behavior".

Al seleccionar "Instance Variables" se despliega el inspector clásico de Squeak, presentando las variables de instancia en el panel B. Al seleccionar una de las variables se presenta en el panel C el objeto a que referencia.

Por último, si hiciéramos clic en "Behavior" se desplegaría el Inspector de Comportamiento. Este inspector permite observar los diferentes comportamientos (conjunto de métodos) que puede tomar un objeto a lo largo de su vida. En el próximo capítulo desarrollaremos la idea y fundamentos de esta característica.

2.4.4 Meta herramientas

"I would rather write programs to help me write programs than write programs."

— Dick Sites

Al disponer de un modelo que nos permite acceder en forma ordenada y uniforme al universo de relaciones por un lado, a las relaciones de un objeto por otro, y en particular a los vínculos de dichos objetos, tenemos la libertad de comenzar a pensar en diferentes herramientas y en nuevos frameworks que se basen en el nuestro de relaciones.

Es común observar implementaciones ad-hoc de frameworks de persistencia que requieren, cuanto menos, una meta-descripción básica de las relaciones participantes a los efectos de estar en condiciones de persistir con la suficiente inteligencia.

Otros frameworks que también requieren de esta clase de información son los de presentación o vista del modelo. En particular, tomaremos este caso como una de las pruebas de concepto de nuestra trabajo.

2.5 Consecuencias

2.5.1 Capacidad auto descriptiva

Nuestra hipótesis es que disponer de un ambiente con el Framework de Relaciones mejora sustancialmente el proceso de desarrollo, principalmente porque su utilización lleva al programador a formalizar las relaciones, entidades y roles subyacentes, logrando un adecuada separación de responsabilidades.

Otra consecuencia de este proceso se observa en una en las características el modelo resultante, donde la información necesaria para la definición de una relación se preserva ordenada, organiza y estructurada en el mismo modelo.

Esto marca una diferencia con los actuales lenguajes orientados a objetos. En éstos, las relaciones son representadas entre las variables de instancia y los métodos de las clases involucradas, haciendo que la estructura en su conjunto no sea visible o fácilmente interpretable[Rumbaugh 1987].

Debido a ello, aunque fuera del alcance de este trabajo, resulta sencillo desarrollar un Inspector gráfico del modelo ofrezca una vista similar a un diagrama de clases de UML.

De esta manera, obtendríamos una vista en tiempo real de nuestro modelo, dejando atrás parte de aquellos problemas sobre diferencias entre documentación y código.

2.5.2 Menor Gap semántico

El Gap Semántico representa la diferencia entre dos descripciones de un objeto bajo distintas representaciones lingüísticas, por ejemplo lenguaje natural y lenguaje de programación.

En computación, este concepto es particularmente relevante cada vez que actividades humanas, observaciones y tareas se transfieren y representan computacionalmente.

Escribir un programa requiere traducir el conocimiento específico del usuario o experto del dominio en reglas formales que serán procesadas por una computadora. Consecuentemente, cualquier mapeo de

una aplicación del mundo real en una aplicación computacional requiere, por un lado, cierto background de conocimiento técnico del usuario, y, por el otro, un mejor conocimiento del dominio de problema en cuestión por parte del programador. Es aquí donde se manifiesta el gap.

El propósito final ante un problema de estas características, es disponer de un software que permita al usuario representar su conocimiento de la manera más directa y sencilla posible, sin necesidad de involucrarse en los detalles de su implementación.

En este sentido, es de esperar que los lenguajes de programación continúen incrementando su nivel de abstracción (antes Assembler, luego C, más tarde Prolog, Smalltalk y Java), que surjan nuevos frameworks que ayuden a definir, organizar e integrar el conocimiento de alto nivel, y que éstos puedan complementarse entre sí. Luego, naturalmente surgirán más y mejores interfaces que acerquen al usuario al modelo computacional.

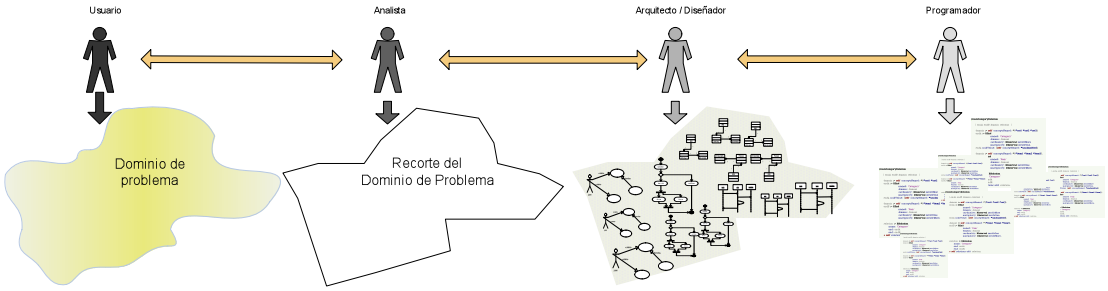


Figura 44. Esquema de proceso de desarrollo

Al incorporar relaciones en un LPOO, es posible desarrollar herramientas que brinden al diseñador la posibilidad de trabajar directamente sobre el ambiente, describiendo clases, objetos y las relaciones entre estos.

A nuestro entender, esta posibilidad es un indicio de que el ambiente posee un nivel de expresividad superior. Si consideramos al diseñador como usuario, podemos pensar sería capaz de representar su conocimiento de manera más sencilla y directa que antes, sin involucrarse en detalles de implementación. Se estaría así ante una disminución parcial el gap semántico.

2.5.3 Meta herramientas

El hecho de contar con un modelo auto-descriptivo que expone en forma ordenada y uniforme tanto semántica como su conocimiento nos da la posibilidad de pensar, predicar y operar en términos de este.

Los exploradores e inspectores provistos con el framework son un ejemplo. Ellos logran independizarse del dominio de problema, reutilizándose en cada implementación.

Algo similar ocurre con herramientas gráficas, como editores de objetos (ver Prueba de Concepto) y mecanismos de persistencia que puedan ser desarrollados en función de las capacidades del framework de relaciones.

En términos generales, se vislumbra un gran potencial al pensar en nuevas soluciones a problemas conocidos que puedan ser desarrolladas y reutilizadas.

Por ejemplo, tomemos el problema de copiar un objeto, donde onde debe tomarse la decisión: ¿qué variables de instancia debo copiar?. Como la respuesta se encuentra en el tipo de vínculo que este objeto posee con sus variables de instancia (agregación fuerte, débil o composición), un programador que no cuenta con un ambiente con relaciones tendrá analizar e implementar esto en cada nueva clase que defina el método `copy`, lo que además de engorroso, resulta propenso a error u omisión.

Por el contrario, contar con relaciones le permitiría modelar una solución que resuelva este problema en forma general, basada en la información y tipo e vínculos de los objetos a copiar.

2.5.4 Responsabilidades del programador

Así como relaciones ayuda a solucionar la recurrencia en la toma de la decisión sobre cómo copiar un objeto, existe otro conjunto de responsabilidades que una y otra vez podría evitar el programador.

En un mundo sin relaciones el programador se enfrenta a la necesidad de decidir cómo representar cada una de las relaciones. Así tendrá que definir si utiliza variables de instancia, o si la relación es lo suficientemente importante como para encapsularla en un nuevo objeto. Cuando opte por hacerlo con variables de instancia tendrá que considerar la estructura de datos apropieda (ej. Set, OrderedCollection). A esto se le suma el optar por dónde y cuándo validar las restricciones y cómo reportar o notificar potenciales errores. También es necesario definir los eventos, cómo y cuándo notificarlos.

Como es de imaginar a esta altura del trabajo, el framework de Relaciones abstrae al programador de varias de las decisiones de implementación de una relación.

2.5.5 Trabajo del programador

Con menos responsabilidades, y un framework representando el modelo de relaciones, el programador se encontrará frente a un proceso de desarrollo más declarativo, que requiera la configuración de relaciones y entidades a través herramientas provistas por el ambiente. Luego, sí será necesario que se preocupe por una correcta representación del comportamiento interactivo entre los objetos del dominio.

Con respecto a la definición y codificación de los protocolos de las clases del dominio, tal como se verá en el capítulo siguiente, el framework resolverá mucho del trabajo que se lleva adelante habitualmente.

De esta forma, se relajará la necesidad de implementar relaciones en forma ad-hoc, evitando no solo el esfuerzo de codificación, sino también el de llevar adelante estratégias de testing sobre este aspecto de la implementación.

Capítulo 3

Modelo de Comportamiento

En la sección anterior describimos el modelo y Framework de Relaciones, el cual sumado a las herramientas desarrolladas -que analizaremos más adelante en la sección Herramientas-, nos han permitido llevar las relaciones a ser una construcción semántica del mismo nivel que las clases, con la consecuente reificación de las mismas. Así, hemos desarrollado un ambiente que permiten que Clases y Relaciones convivan entre sí permitiendo modelar en forma más directa el modelo estructural subyacente a cierto dominio de problema.

Sin embargo, durante el desarrollo del mismo notamos en forma recurrente la necesidad de complementar el mismo con un modelo de comportamiento basado en estas mismas relaciones.

En esta sección exponemos los indicios y motivación que nos llevaron a desarrollar el modelo y Framework de comportamiento basado en relaciones. También describiremos las características principales y resultados de su aplicación.

3.1 Concepción y Motivación

Los mecanismos de sharing de comportamiento y conocimiento entre objetos se encuentran dentro de las características más útiles, pero también fuertemente debatidas, de los lenguajes orientados a objetos[Stein et al. 1988]. Por tal motivo, no nos extrañó que al modificar la forma en que los objetos se relacionan entre sí se nos presenten nuevas ideas o alternativas respecto a cómo implementar estos mecanismos.

Como ya hemos mencionado, que por agregar Relaciones al ambiente comienza a existir nueva información y nuevos conceptos que los objetos participantes deben aprender, adquirir e interpretar. Por ejemplo, antes de registrar la relación *FatherChild*, una instancia de `Person` nada sabrá sobre los conceptos padre e hijo. Sin embargo, luego de dicha registración, esta instancia deberá comprender y responder a mensajes como `children`, `isChildOf:`, `allChildrenDo:`, entre muchos otros. Gracias a la información contenida en la descripción de una relación (nombre de roles, cardinalidad, multiplicidad, valores predeterminados, etc), este tipo de comportamiento podría ser automáticamente generado por el mismo modelo, sin necesidad de delegar en el programador esta responsabilidad rutinaria y de poco valor agregado.

Otro de los indicios que motivaron nuestra decisión de trabajar sobre un modelo de comportamiento fue el de encontrar ejemplos o situaciones donde cierto protocolo no es propio de la clase de un objeto sino del rol que cumple en este momento. Por ejemplo el Pattern Observer [Alpert et al. 1998] asigna al objeto en observación (Observee) la responsabilidad de notificar sus cambios utilizando el método `changed:` y espera que los dependientes (Observers) se vean notificados mediante el envío del mensaje `update:`

En Smalltalk-80, tanto el protocolo de un `Observer` como el de un `Observee` se encuentra implementado en la clase `Object`. Entendemos que al disponer de Relaciones esta implementación suena poco apropiada. Uno esperaría que los conceptos `Observee` y `Observer` sean una entidad concreta -como un Rol- y que dispongan de su propio protocolo y consecuente comportamiento, para que sólo los objetos que actúan bajo estos roles adquieran tanto su protocolo como su comportamiento.

En forma independiente a la maduración de los aspectos anteriores, y ya desde el inicio de nuestro trabajo, identificamos *Instancia* y *Herencia* como potenciales relaciones a refactorizar y manipular a través de nuestro ambiente. Si bien descartamos esto último por considerarlo fuera del núcleo de nuestra tesis, ello no nos impidió mantener presente la idea de que estas relaciones se encuentran en un mismo nivel semántico –aunque de diferentes dominios de problema- que cualquier otra relación así como *FatherChild*, *Observation*, *Implication*, *PlayAs*, etc.

Más adelante, conversando sobre la forma de modelar e implementar el comportamiento de un Rol comprendimos que el Method Lookup de Smalltalk no es más que un algoritmo cuya materia prima son las relaciones de *Instancia* y *Herencia*. Inmediatamente visualizamos el potencial de un nuevo Method Lookup que no sólo se limite a estas relaciones, sino también se adapte y opere sobre cualquier otra relación definida en el ambiente.

Finalmente, la conjunción y simbiosis de estos aspectos, ideas y elementos fueron quienes nos motivaron e inducieron a la construcción de un modelo de comportamiento basado en Relaciones.

3.2 Descripción del Modelo

3.2.1 Tipos de Comportamiento

Desde su concepción, maduramos el concepto Comportamiento basándonos en la abstracción `Behavior` de Smalltalk. En Smalltalk, las instancias de esta clase tienen como responsabilidad describir el comportamiento de otros objetos[12]. `Behavior` provee el mínimo estado necesario para compilar métodos, crear y correr instancias. Si bien la mayoría de los objetos son creados como instancias de su subclase, `Class`, ésta es un buen punto de partida para proveer comportamiento específico por instancia [Squeak].

Si bien nuestra visión de comportamiento no requiere que éste tenga o contemple la capacidad de crear y correr instancias, por lo demás, la clase Behavior describe adecuadamente nuestras ideas y expectativas. Por esto, podemos asumir que al hablar o especificar un Comportamiento, estamos pensando en una instancia o subinstancia[13] de la clase `Behavior`.

[12] Dentro de esta idea incluimos tanto el Conocimiento (variables de instancia o propiedades) y lo que otros autores llaman comportamiento (metodos).

[13] Dado c, instancia de la clase C, se dice que c es subinstancia de A sii C, en su cadena de superclases hasta Object, encuentra a A.

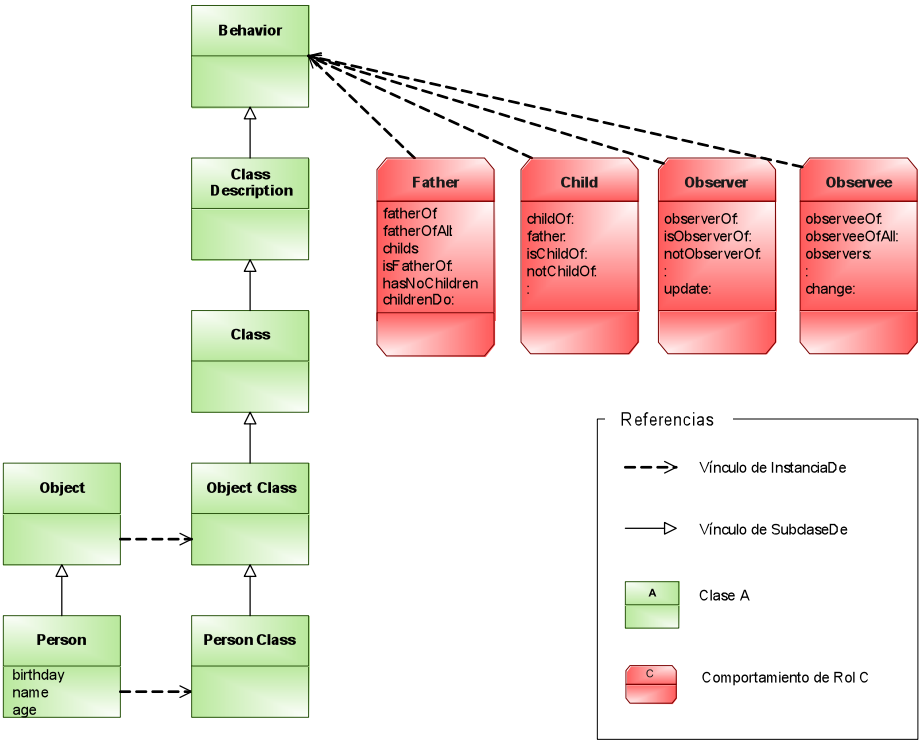


Figura 45. Comportamientos. Diagrama de Comportamientos.

En función de la definición anterior, de la figura se desprende que `Object`, `Person`, `Father`, `Child`, `Observer` y `Observee` son comportamientos.

Podemos observar que hay al menos dos formas o tipos de comportamiento: aquel que se especifica como una clase y aquellos que representan el comportamiento de un rol de una relación.

3.2.1.1 Comportamiento de Clase

Por comportamiento de Clase entendemos a aquél que el programador declara al definir y programar las Clases. Las instancias de esta clase adquieren este comportamiento.

3.2.1.2 Comportamiento de Rol

Por el otro lado, el comportamiento de un Rol (de una Relación) es aquel que conjuga, por un lado el conocimiento propio de la estructura y definición de la relación desde la perspectiva de dicho rol y, por el otro, el comportamiento específico de dicho rol en el dominio de problema en cuestión.

El comportamiento en función de la estructura y definición de la relación se desprende o infiere de aspectos tales como

- Nombre (en singular y plural) del Rol
- Nombre (en singular y plural) de los otros Roles
- Cardinalidad
- Multiplicidad para con cada uno de los otros Roles
- Visibilidad de los otros roles
- Navegavilidad hacia los otros Roles
- Relación de Lectura / Lectura y Escritura

Dado que esta información se encuentra disponible en la relación, este comportamiento puede ser deducido en forma autónoma, sin contar con la intervención de un programador.

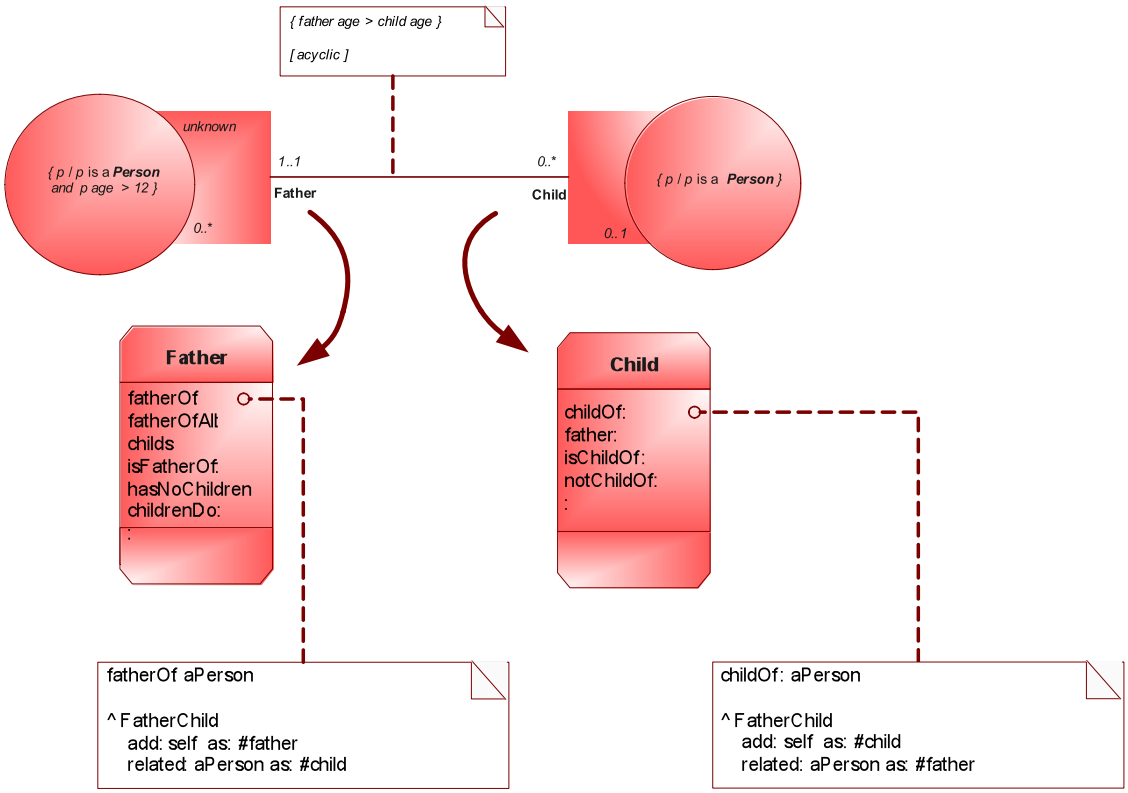


Figura 46. FatherChild. Esquema de proceso de inferencia de comportamiento de Rol

Sin embargo, de la definición y declaración de una relación nada puede saberse sobre el comportamiento específico que analistas, diseñadores o programadores deseen asignarle a los roles participantes. Un ejemplo de este caso es el de la relación *Obervation*. Aquí, si queremos cumplir con el Pattern Observer necesitamos especializar el comportamiento tanto del Rol `Observer` como el de `Observee`.

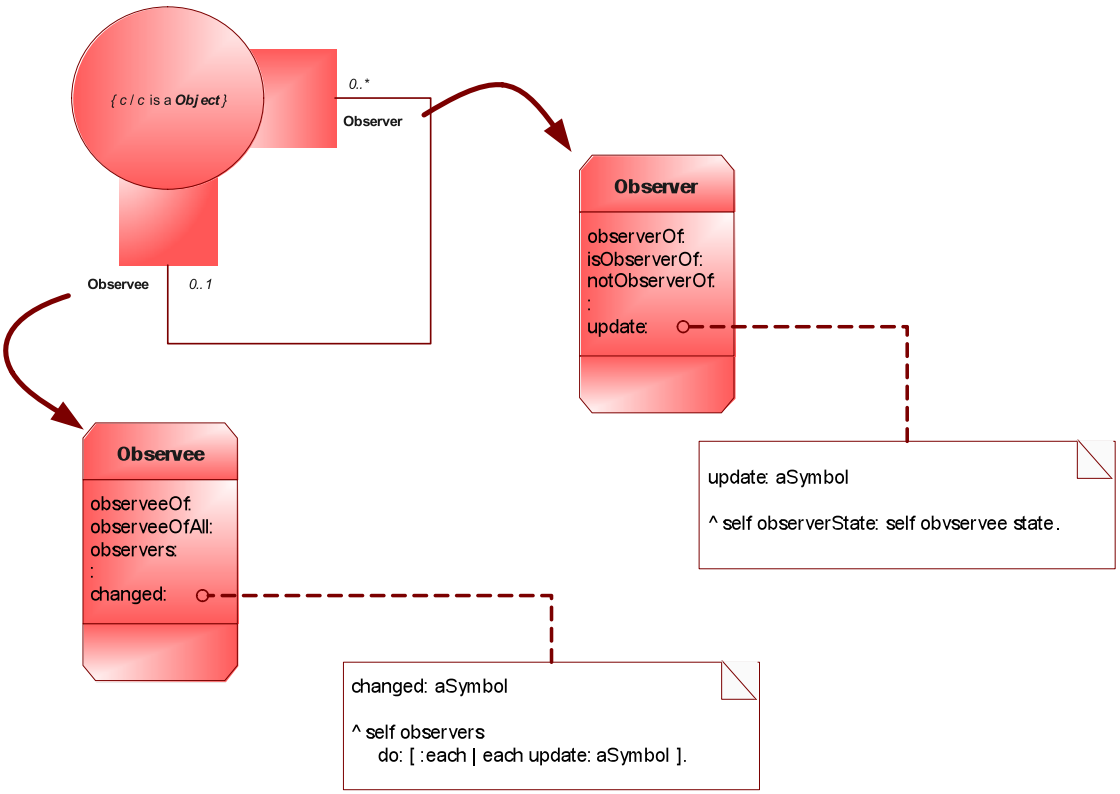


Figura 47. Observation. Esquema de proceso de inferencia de comportamiento de Rol

En la figura anterior vemos cómo a través de los métodos `changed:` y `update:`, se complementa el comportamiento deducido de la relación de aquel específico del dominio de problema en cuestión.

3.2.1.3 Otros comportamientos

Los comportamientos de Clase y de Rol fueron los que en primer lugar evidenciamos en nuestro trabajo. Pero al comenzar a profundizar estas ideas, se nos presentaron otras situaciones donde haciendo uso de los mismos principios podríamos modelar problemas conocidos en forma diferente.

Por ejemplo, el objetivo del pattern State [Alpert et al. 1998] es permitir que un objeto altere su comportamiento cuando su estado interno cambie. Este pattern modela como una jerarquía de Clases los posibles estados de un objeto, o, mejor dicho, los comportamientos que deberá tener un objeto cuando se encuentre en dicho estado.

Nuestro idea es que los comportamientos puedan ser representados por instancias la clase `Behavior`, y ser vinculados con los objetos que así se comporten a través de relaciones destinadas a tal fin.

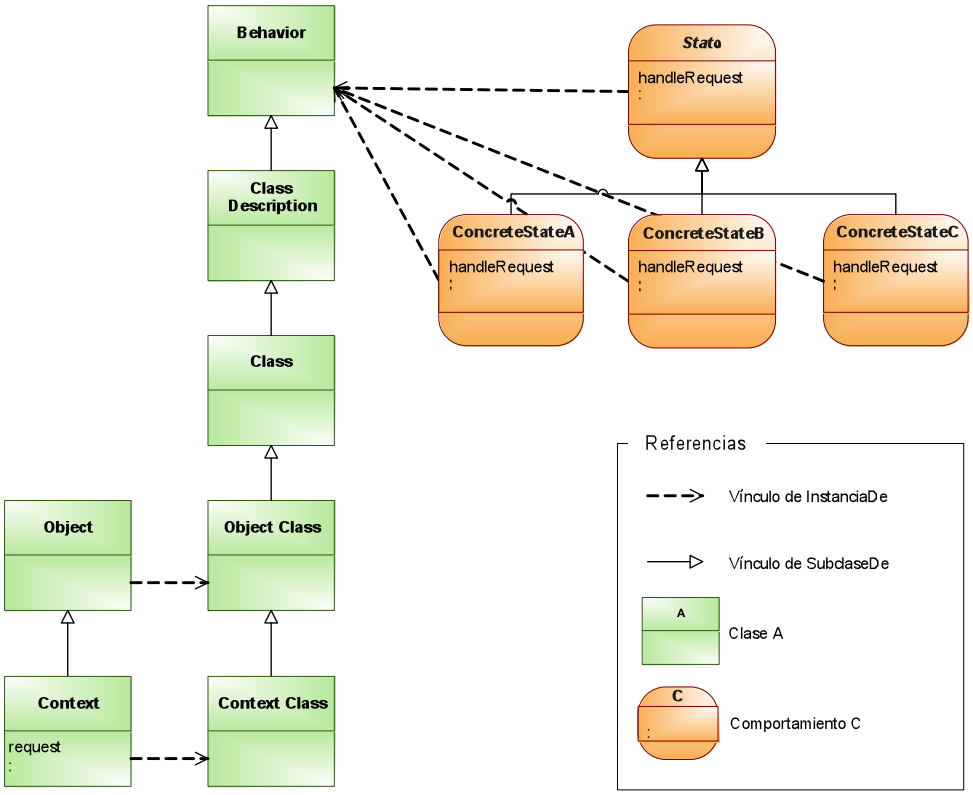


Figura 48. Comportamiento State. Diagrama de Comportamiento

Siguiendo estos mismos principios se podrían enfocar problemas similares como por ejemplo el propuesto en el pattern Strategy [Alpert et al. 1998].

3.2.2 Relaciones de Comportamiento

Una vez que identificamos e implementamos distintos comportamientos, era necesario relacionarlos con los objetos que así debían comportarse. Fue aquí donde comprendimos la importancia de las Relaciones en el contexto del comportamiento de los objetos. Luego, durante el progreso de nuestra investigación, vimos que no fuimos pioneros al respecto.

Adele Goldberg y David Robson [Goldberg and Robson 1983] entendieron que, y ya desde Smalltalk-80, Smalltalk posee y mantiene, entre otras[14], la relación de *InstanceOf* y *SubclassOf*. Así mismo, [Bobrow and Stefik 1986][Stein et al. 1988][Bardou and Dony 1996] notaron que todos los mecanismos de Sharing (o Inheritance), aún con sus diferencias, se basan en relaciones y comparten los siguientes puntos en común

- Están basados en una o varias relaciones[15]
- Utilizan estas relaciones para lograr Inheritance
- Necesitan una semántica clara y precisa de las relaciones sobre las que operan.
- Tienen como fin lograr algún tipo de sharing entre objetos

[14] Los otros tipos de relaciones que identifican son la Herencia o Subclasificación, la Referencia y Observación

[15] Por ejemplo, en los lenguajes basados en clases las relaciones *InstanceOf* y *SubclassOf*, y en Self la relación *InheritFrom*. [UNG87]

En un ambiente basado en clases, la relación *InstanceOf* es una relación de comportamiento esencial e inmutable. Es de comportamiento porque permite mediante la navegación de sus vínculos comprender el comportamiento de los objetos participantes. Es esencial porque define la esencia de los objetos que vincula al asociarles el comportamiento de su clase. Y es inmutable justamente porque al definir la esencia de un objeto, sus vínculos no podrán ser removidos mientras dure el ciclo de vida de los objetos vinculados.

Pero no todo comportamiento es esencial. Un objeto puede tener comportamientos complementarios a los de su clase. Existen diversas situaciones que afectan o alteran el comportamiento de un objeto en forma accidental[16].

Generalmente, estas situaciones pueden ser representadas en el contexto de una relación. Llamaremos a dicho tipo de relaciones, Relaciones de Comportamiento Accidental. De comportamiento porque al igual que la relación de *InstanceOf*, permiten navegar desde un objeto hasta sus comportamientos. Accidental porque a diferencia de la relación *InstanceOf*, un objeto puede mutar de comportamiento durante su ciclo de vida.

En las siguientes subsecciones analizamos diferentes relaciones de comportamiento.

3.2.2.1 La Relación InstanceOf

La bibliografía y estado del arte entienden a la relación *InstanceOf* como la relación "Es Un" [Johnson and Foote 1988], entre Objetos y Clases.

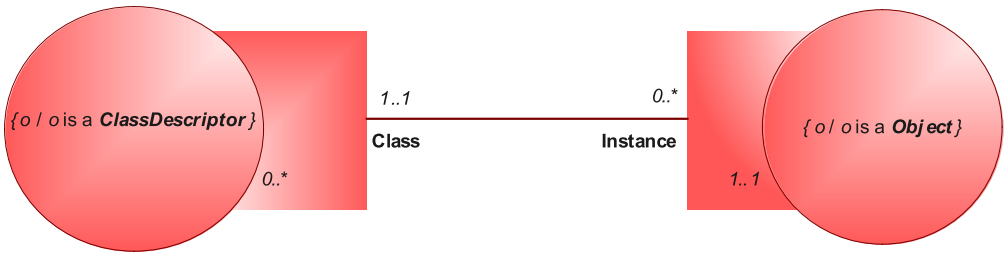


Figura 49. InstanceOf. Diagrama de Relación

En una clase (por ejemplo `Person`) se definen las variables de instancia (o propiedades) que constituirán los atributos de sus instancias. Por otro lado, también se expresan un conjunto de métodos que definen el "comportamiento" inherente de dichas instancias.

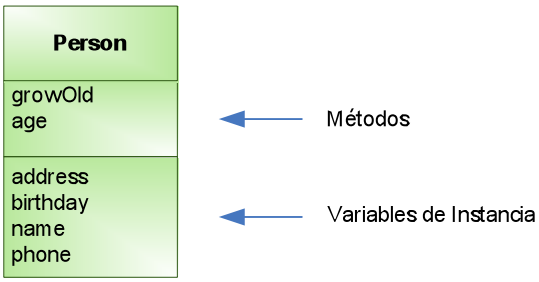


Figura 50. Person. Diagrama de Clase

[16] Distintos trabajos han abordado este tema. Entre ellos podemos mencionar Subjectivity, Split Object y Traits. En el capítulo 5 analizamos las diferencias y similitudes con alguno de ellos.

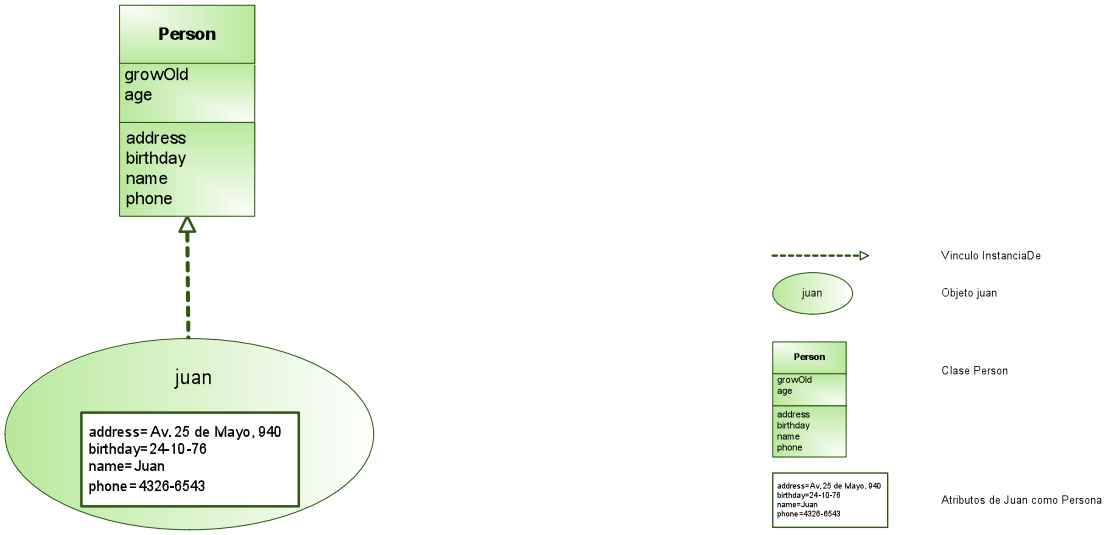
Analicemos conceptualmente el proceso que se da lugar cuando se ejecuta el siguiente extracto de programa

```
juan := Person new
            name: 'Juan';
            yourself.
```

En primer lugar podemos imaginar que se crea un objeto al que llamaremos momentáneamente `o`, el cual posee identidad: es él y no otro. En segundo lugar, imaginamos que se establece un vínculo entre `o` y la clase que le dio la vida; y que por tanto define su esencia: es `o` "una persona", y no un árbol, o un meteorito. La semántica de este vínculo es definida por la relación *InstanceOf*, indicando "este objeto `o` es una instancia de la clase `Person`, y por lo tanto es una Persona".

Siguiendo el ejemplo, nos vemos tentados a decir que se le da a `o` el nombre 'Juan'. Lo que en realidad asumimos es que *`juan`* por ser considerado una persona ya es capaz de conocer cuales eran sus propiedades esenciales (*`address`*, *`birthday`*, *`name`*, *`phone`*). Pero, ¿cómo fue que aprendió esto? ¿Quién le enseño? ¿Cómo es que tienen la capacidad de retener su nombre?

Comúnmente uno puede pensar que estas propiedades se encuentran de alguna manera "dentro" del objeto *`juan`*, como algo propio de él, y que su comportamiento lo obtiene de su clase (a través Method Lookup).



Sin embargo, a la luz de este trabajo, podría verse de manera diferente. Diremos que las propiedades no son *de juan*, sino de juan en tanto es persona, es decir, son propiedades de la relación con el concepto de persona.

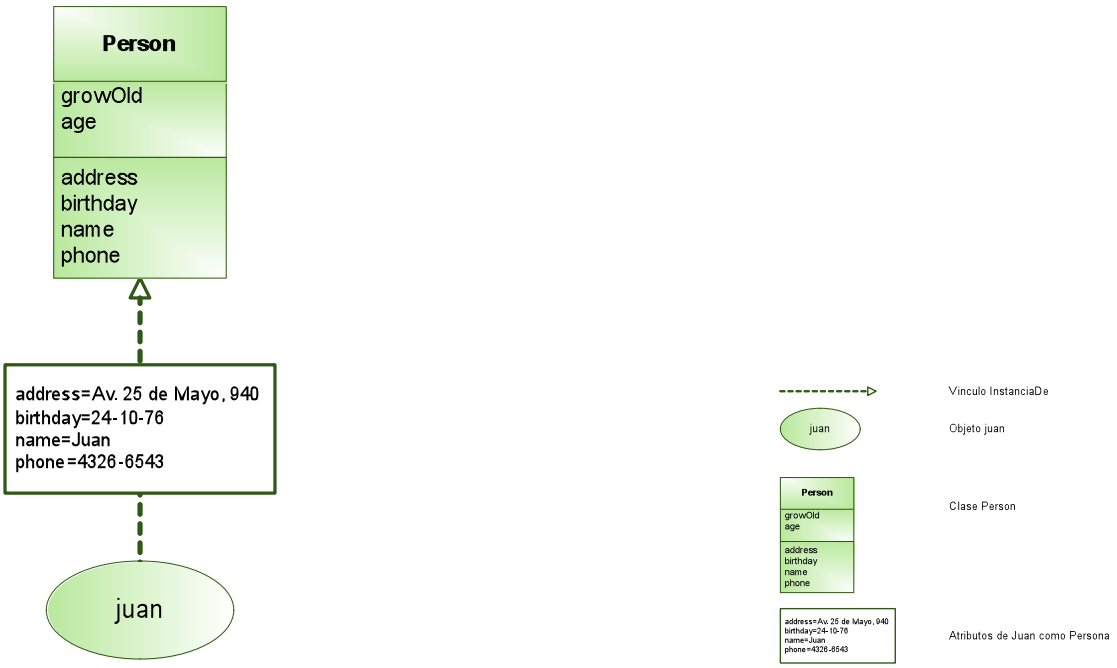


Figura 51. Juan es una Persona. Diagrama de Objetos

Si bien este razonamiento podría parecer a priori caprichoso, debemos recordar que semánticamente esta relación es una relación de carácter inmutable, es decir que una vez establecido dicho vínculo, este no se eliminará hasta tanto `juan` deje de existir. Por tal motivo, estos atributos acompañarán a `juan` durante toda su vida.

Por último, cuando a `juan` le preguntamos por edad, identificándose como persona buscará el método `age` de la clase `Person`. En Smalltalk, este procedimiento es llevado adelante por el Method Lookup, el cual navegando a través del vínculo `(juan as Instance, Person as Class)` de la relación *InstanceOf* accederá y ejecutará el método `age` de la clase `Person` sobre `juan`[17].

3.2.2.2 La Relación "Podría participar como"

Decimos que un objeto *podría participar como un rol* en una relación cuando tiene el potencial de participar del dominio de dicho rol, aún cuando no lo haga en este momento.

Luego, si un objeto tiene el potencial de participar bajo cierto rol, entonces debe comprender y entender el protocolo del mismo. Para ello, siguiendo nuestra línea argumental debe existir una relación de comportamiento que permita vincular dichos objetos con el comportamiento del rol. Así definimos la relación *CouldParticipateAs* tal como presentamos a continuación.

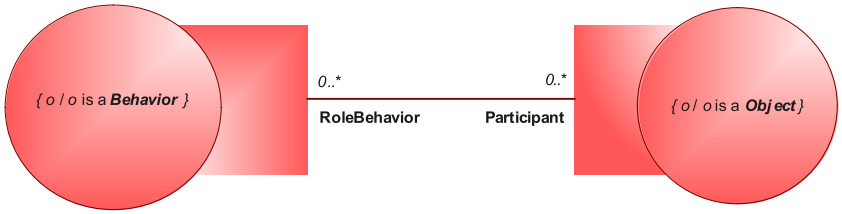


Figura 52. CouldParticipateAs. Diagrama de Relación.

[17] Por simplicidad, evitamos hablar acá de mecanismos de sharing y reutilización (herencia en este caso), la cual describiremos más adelante .

Notemos que a diferencia de la Relación *InstanceOf* donde una instancia tiene un único comportamiento, en la definición de *CouldParticipateAs*, un participante puede tener ninguno, uno o muchos comportamientos en un momento dado, según los roles en los que tenga el potencial de participar.

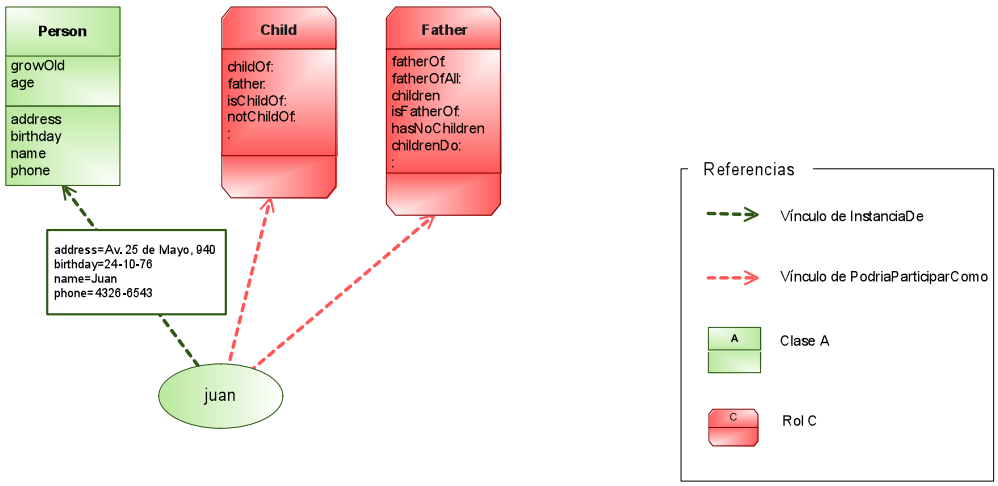


Figura 53. Juan. Vínculos de Comportamiento

3.2.2.3 Otras relaciones de Comportamiento

Mientras que la relación *InstanceOf* sirve entonces para vincular objetos con su comportamiento esencial (`Class`), *CoudParticipateAs* permite vincular objetos con su comportamiento accidental de Rol (`RoleBehavior`). Pero como ya hemos dicho en la sección Comportamiento, existen o pueden existir otros tipos y formas de comportamiento accidental como por ejemplo el comportamiento por Estado (`State`).

Surgen aquí otras relaciones más específicas. Por ejemplo, siguiendo el caso motivador del pattern State para una conexión TCP podríamos pensar en la relación *TCPConnectionStateOf*.

Figura 54. TCPConnectionStateOf. Diagrama de Relación

En este caso, el comportamiento de una `TCPConnection` depende del estado en que se encuentra. Pero como el estado en que se encuentra esta determinado por el vínculo entre dicha `TCPConnection` y un `TCPConnectionState`, decimos que es gracias a esta relación que adquiere dicho comportamiento. Luego vulgarmente decimos que *TCPConnectionStateOf* es una relación de comportamiento.

En conclusión, desde nuestro punto de vista consideramos que *R* es una relación de Comportamiento si dado *a* *R* *b*, la semántica de *R* indica que *a* se comporta (o puede hacerlo) como expresa *b*.

3.2.3 Relaciones auxiliares al Comportamiento

En un ambiente basado en clases, existe otro tipo de relación que ayuda a representar y modelar adecuadamente el comportamiento de los objetos del dominio de problema.

A diferencia de las Relaciones de Comportamiento, éstas no tiene como objetivo vincular objetos con su comportamiento. Su fin principal es el de relacionar los comportamientos entre sí para organizarlos según diferentes criterios. El caso más significativo es la *Herencia*.

3.2.3.1 La Relación de SubclassOf

La relación de *SubclassOf* tiene y ha tenido diferentes nombres relativos al punto de vista con que han sido observadas. Así podemos referirnos a ella como Relación de *Herencia*, o Relación de *Generalization*, *Specialization* y de hasta *SubtypeOf*

En los ambientes basados en clases, esta relación puede cumplir diferentes funciones, como por ejemplo promover el reuso de código, organizar y clasificar clases, definir protocolos estándar y hacer extensiones a clases existentes [Johnson and Foote 1988]. Para ello, es común encontrar software donde diseñadores y/o programadores hayan sobrecargado tanto el concepto de Clase como el de la Relación de *SubclassOf*, dando a éstas demasiadas responsabilidades [Black and Sharli 2004].

Sin embargo, y más allá de los usos que se les dé a esta relación, existe un consenso generalizado en que semánticamente represente la especialización entre Clases [Johnson and Foote 1988].

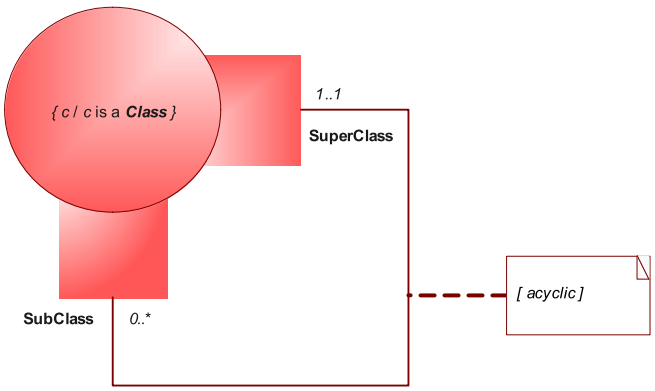


Figura 55. Inheritance. Diagrama de Relación

En nuestro trabajo adherimos a esta interpretación, flexibilizando el hecho de que la especialización sea sólo entre clases para ser posible también entre comportamientos.

3.2.3.2 La Relación de Composición de Traits

La aparición de Traits [Traits] en los ambientes basados en clases encontró sus orígenes en evitar la sobrecarga semántica de la relación *SubclassOf*, devolviendo a estas jerarquías la posibilidad de ser utilizadas para fines de clasificación [Black and Sharli 2004] (especialización) conceptual.

En sus trabajos [Scharli et al. 2003a][Scharli et al. 2003b] Nathanael Scharli, Oscar Nierstrasz, Stéphane Ducasse y Andrew P. Black logran este objetivo mediante la construcción de paquetes de comportamiento y su posterior composición -según diversos criterios-, permitiendo composición entre Traits y, entre Traits y Clases.

Si bien ellos hablan de “composición” como una operación, en el marco de nuestro trabajo podemos permitirnos considerar –al menos conceptualmente- que existe la relación *Composition*. Cuando un

Trait A quiere usar los servicios provistos por un Trait B, se lleva adelante un procedimiento de composición de A por B. En este proceso pueden producirse conflictos, motivo por el cual éstos deben ser resueltos explícitamente. Una vez finalizado el proceso, A ha sido compuesto por B, o lo que es lo mismo, se ha establecido un vínculo de composición entre A y B. En este contexto o vínculo se encuentran las decisiones del operador respecto a cómo han sido resueltos estos conflictos, los sinónimos utilizados, y los bindings realizados.

En conclusión, así como *SubclassOf* permite especializar comportamientos, la *Composition* permitiría conjugarlos y combinarlos.

> **Nota** Dentro del alcance de nuestro trabajo ha quedado fuera la utilización de Relaciones Auxiliares al comportamiento.

3.2.4 Method Lookup basado en Relaciones de Comportamiento

Los elementos que hasta aquí hemos descripto nos permiten concluir que el comportamiento de un objeto no sólo depende de las relaciones *InstanceOf* y *SubclassOf*, sino otras de carácter accidental y dinámico.

Esta idea resulta viable en tanto y en cuanto podamos redefinir el algoritmo sobre el que trabaja el Method Lookup (ML).

Para ello primero refrescamos como opera el ML tradicional y luego planteamos una alternativa que contemple relaciones.

3.2.4.1 Method Lookup tradicional (con Herencia Simple)

Tradicionalmente, el Method Lookup ha funcionado de la siguiente manera en los lenguajes basados en Clases con relación de *SubclassOf* (herencia simple).

Descripción del Algoritmo

Sea el objeto *o*, instancia de la clase C. Cuando o recibe el mensaje *m*, cuyo selector es s se procede como se muestra a continuación:

1. El ML navega desde *o* hacia C a través del vínculo *InstanceOf*
2. El ML busca en C un método cuyo selector sea igual a s.
3. Si s existe en C, ejecuta el método en el contexto de *o* y retorna el resultado.
4. Si s no existe en C, ML navega desde C hacia C' a través del vínculo *SubclassOf*.
5. Si C' no es *nil*, ML vuelve al paso 2 buscando en C'.
6. Si C' es *nil*, ML no encontró a s en el comportamiento de *o*. Luego invoca el método `doesNotUnderstand:` *m* en el contexto de *o*[18].

Binding de pseudo-variables self y super.

Excepcionalmente, cuando el mensaje a invocar está referenciado en un método en ejecución s a través de la pseudo-variable *super*, el ML opera de la siguiente manera

a. El ML determina la clase C donde se encuentra definido el método s.
b. El ML continua ejecutando el paso 4.

[18] Este paso se presenta en el ML de Smalltalk.

Manejo y resolución de conflictos

Las condiciones de vinculación y manejo que las pseudo-variables self y super hacen que ante potenciales conflictos exista un criterio unívoco sobre cómo resolverlo (es decir, cómo determinar el método a ser invocado).

Un objeto sólo puede tener un comportamiento –el de su clase-, y una clase sólo puede tener una superclase.

Cuando una Clase y alguna de sus superclases poseen un método para el selector del mensaje a responder, esta ambigüedad es resuelta o bien por el paso del algoritmo o por las pseudo-variables, entre los cuales explicita unívocamente el orden en que debe buscarse el método.

3.2.4.2 Method Lookup basado en Relaciones, simple

En esta sección expondremos el criterio de ML utilizado durante nuestro trabajo. Es importante dejar en claro que éste es de carácter experimental y que tiene por principal objetivo mostrar la viabilidad de este enfoque.

Idea del Algoritmo

La idea central del algoritmo es utilizar no sólo la relación *InstanceOf* sino todas las relaciones de comportamiento en las que un objeto participe. El algoritmo se explica a continuación.

Tomamos la primera relación -supongamos *InstanceOf*-. Dado que es una relación 1 a 1, navegamos desde el objeto receptor hacia su comportamiento a través del vinculo *isInstanceOf*. Una vez que obtenemos el comportamiento (en este caso la clase del objeto receptor), procedemos en forma similar al ML tradicional.

En caso de no encontrar el método correspondiente en la cadena de *SubclassOf*, tomamos la próxima relación de comportamiento, supongamos *CouldParticipateAs*. A diferencia de *InstanceOf*, ésta es una relación N a N. Por tal motivo podrían existir potencialmente muchos vínculos *couldParticipateAs* en donde participe el receptor. Aquí iteramos sobre cada uno de estos vínculos, procediendo por cada uno de ellos de la misma manera en que lo hicimos para con el vínculo *isInstanceOf*.

Si aún seguimos sin encontrar el método, pasamos a la próxima relación (ej. *StateOf*) y así sucesivamente hasta que encontremos el método a ejecutar o no existan más relaciones de comportamiento a analizar. Si no encontramos el método invocamos el `doesNotUnderstand`.

Debemos aclarar que aunque la idea del algoritmo es de relativa sencillez, cualquier implementación, por simple que sea, requiere analizar y definir los siguientes aspectos

- Orden de prioridad entre las Relaciones de Comportamiento
- Dada una Relación de Comportamiento (1 a N o N a N), orden de prioridad entre los vínculos del objeto receptor.

En función de esta inherente complejidad optamos por definir una nueva, suprema y calculada relación de comportamiento, que agregue las relaciones de su tipo, abstrayendo todo tipo de criterio y decisión de orden y prioridad entre ellas. Así nació la relación *BehaveAs*, o “Se comporta como”.

Descripción del Algoritmo.

Cuando o recibe el mensaje m cuyo selector es s, se procede como se muestra a continuación:

1. El ML pregunta a o por sus comportamientos (a través de la relación *BehaveAs*)
2. La relación *BehaveAs* retorna una colección ordenada de comportamientos de o ($c_1 . . c_n$)
3. Por cada c_i

3.1. Si `s` existe en c_i, ejecuta el método en el contexto de o y retorna el resultado.

3.2. Si `s` no existe en c_i, ML navega desde c_i hacia c_i' a través del vínculo *SubclassOf*.

3.3. Si c_i' no es `nil`, ML vuelve al paso 2.1 con c_i'.

3.4. Si c_i' es `nil`, ML no encontró a `s` en la rama de comportamientos de c_i. (continua con c_{i+1})

4. Si no hay más comportamientos por analizar, ML no encontró a `s` entre el comportamiento de o. Luego invoca el método `doesNotUnderstand: m` en el contexto de o.

Nota Dado que no hemos refactorizado el ML en la VM de Squeak sino que utilizamos el doesNotUnderstand, esta implementación de ML dará prioridad siempre al comportamiento de clase por sobre cualquier otro comportamiento que tenga un objeto.

Binding de pseudo-variables self y super.

Por simplicidad este algoritmo sólo soporta invocaciones a un mensaje referenciado a través de la pseudo-variable `super` cuando el comportamiento es el de clase.

Manejo y resolución de Conflictos

Las condiciones de vinculación y manejo de las pseudo-variables `self` y `super` ofrecen una semántica clara y precisa para la desambiguación de los conflictos que pueden presentarse.

Se pueden presentar 2 tipos de conflicto

- Un mismo método se encuentra en un comportamiento c y alguno de sus "super comportamientos" (determinados por sus vínculos *SublcassOf*).
- Un mismo método se encuentra en un comportamiento c_i y en c_{i+n}

En el primer caso, el conflicto se resuelve con los mismos principios en que se hace con el algoritmo tradicional.

En el segundo caso, el conflicto se resuelve dando prioridad al comportamiento de c_i por sobre el de c_{i+n}. Esta prioridad la determina el criterio de orden que asignará la relación *BehaviorAs* a los comportamientos de un objeto.

En la bibliografía existen trabajos y criterios más complejos y sofisticados de manejo y resolución de conflictos. Sin embargo, dado el carácter experimental de nuestro modelo de comportamiento, optamos por mantener simple la solución, sin complejizar la idea fundamental del algoritmo.

Nota Existen trabajos relacionados con Split Objects [Bardou and Dony 1996] donde se propone una estrategia que asegura una semántica uniforme de envío de mensajes a lo que ellos llaman *viewpoints*.
La estrategia utilizada en el lenguaje SELF [Chambers et al. 1991] para abordar el problema planteó que el *parent* de un objeto sea priorizado. Los *parents* con distintos niveles de prioridad son ordenados en función de ésta. Los *parents* en un mismo nivel de prioridad no son ordenados y, el intento de acceso a un slot ambiguo genera un error.

Aunque fuera del alcance de nuestro trabajo, un estudio superfial de las estratégias existentes nos permiten comprender una complejidad propia de este tipo de soluciones. Chambers, Ungar, Chang y Hölzle coinciden en [Chambers et al. 1991] que, si bien las técnicas de herencia múltiple y mixins son

útiles y han sido aplicadas en el propio SELF, presentan una serie de desventajas y que los efectos de ordenar los *parents* puede sorprender tanto a programadores novatos como expertos.

Tal vez, la existencia de relaciones semánticas de comportaminto –y no una única relación abstracta como *InheritFrom* en SELF- ofrezcan a estas técnicas y mecanismos una nueva perspectiva, capaz de ayudar a disminuir el impacto de alguno los problemas mencionados en la literatura.

La relación *BehaveAs*

Como ya mencionamos, la relación *BehaveAs* es una relación calculada en base del resto de las relaciones de comportamiento (funciona como una especie de agregación de todas éstas). Desde la perspectiva del ML podría comparársela con la relación *InheritFrom* en SELF.

> **Nota** Conceptualmente, la relación *BehaveAs* no es estrictamente necesaria para el ML propuesto. Sin embargo su existencia nos permite simplificar su algorítmo, haciendo más fácil su comprensión e implementación.

Supongamos que en cierto momento de tiempo `juan` presenta el siguiente estado de vínculos (relativo a sus relaciones de comportamiento).

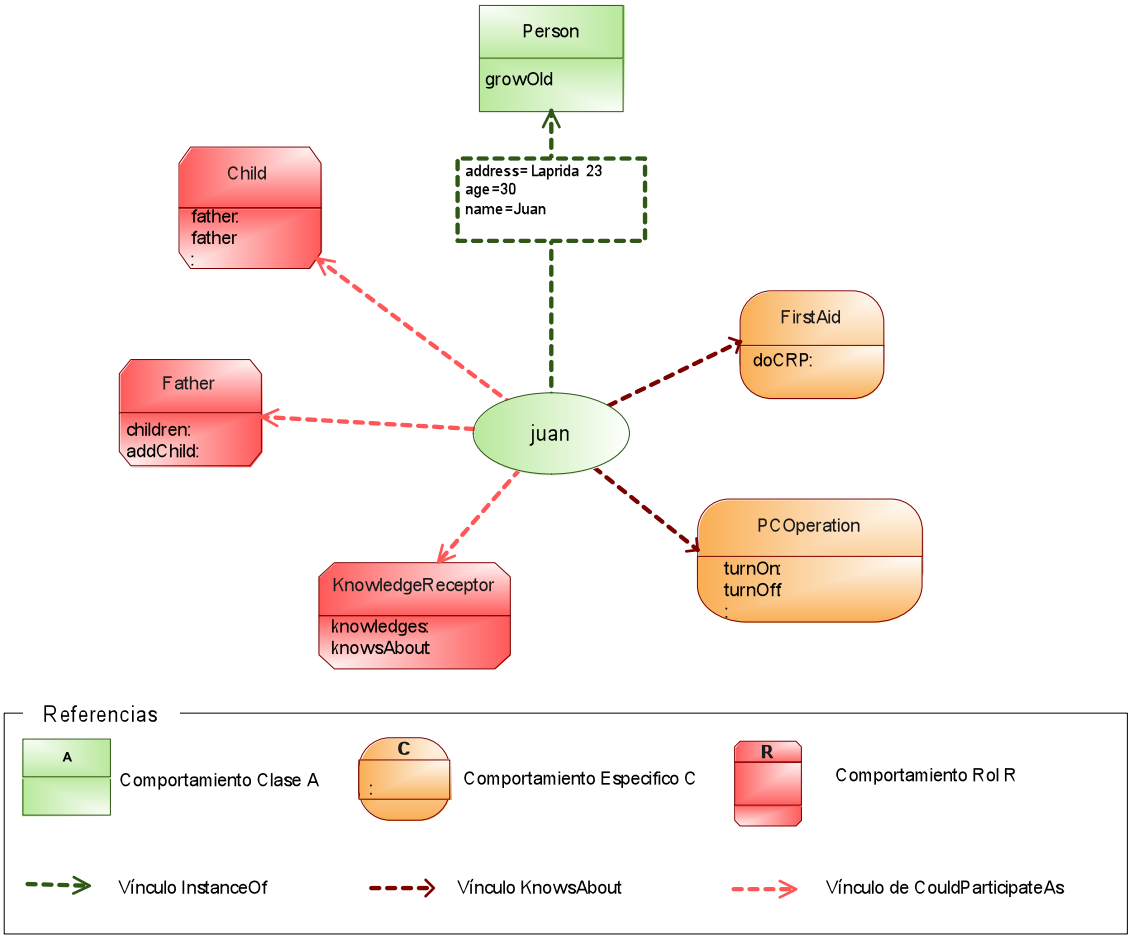


Figura 56. Juan. Vínculos de Comportamiento 2

Vemos en la figura que `juan` sabe responder al comportamiento de `Child`, al de `Father` y que no sólo posee la capacidad de adquirir nuevos comportamientos (`KnowledgeReceptor`) sino que ya ha adquirido los de operación de PC (`PCOperation`) y primeros auxilios (`FirstAid`).

La relación *BehaveAs* es responsable de organizar lo que acabamos expresar y colaborar así con el ML. Es por ello que es necesario agregar la noción de prioridades entre comportamientos, y evitar así posibles conflictos. El criterio definido es el siguiente:

1. El vínculo de comportamiento de Clase (relación *InstanceOf*)[19]
2. El vínculo de comportamiento más reciente [20] (el comportamiento más recientemente adquirido).

Definimos entonces una relación estándar, y a continuación le configuramos un Conocimiento ad-hoc, el cual será responsable de calcular la relación (o parte de ella cuando corresponda). Además, especificamos un criterio de orden entre sus vínculos.

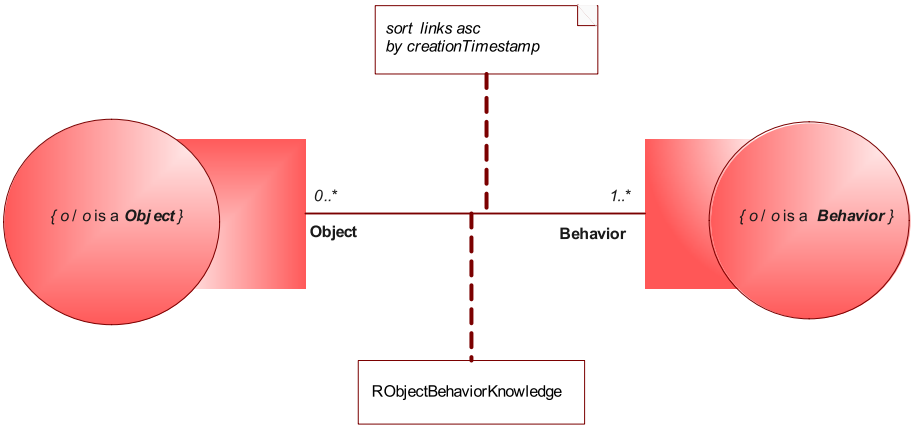


Figura 57. BehaveAs. Diagrama de Relación

Así, `juan` comienza a participar en los siguientes vínculos de la relación *BehaveAs*.

[19] Esto es así porque no hemos reificado el method lookup a nivel Virtual Machine de Squeak.

[20] Como se mencionó anteriormente, se definió este criterio debido al carácter experimental de nuestro trabajo, orientado a mantener una solución simple y clara del algoritmo.

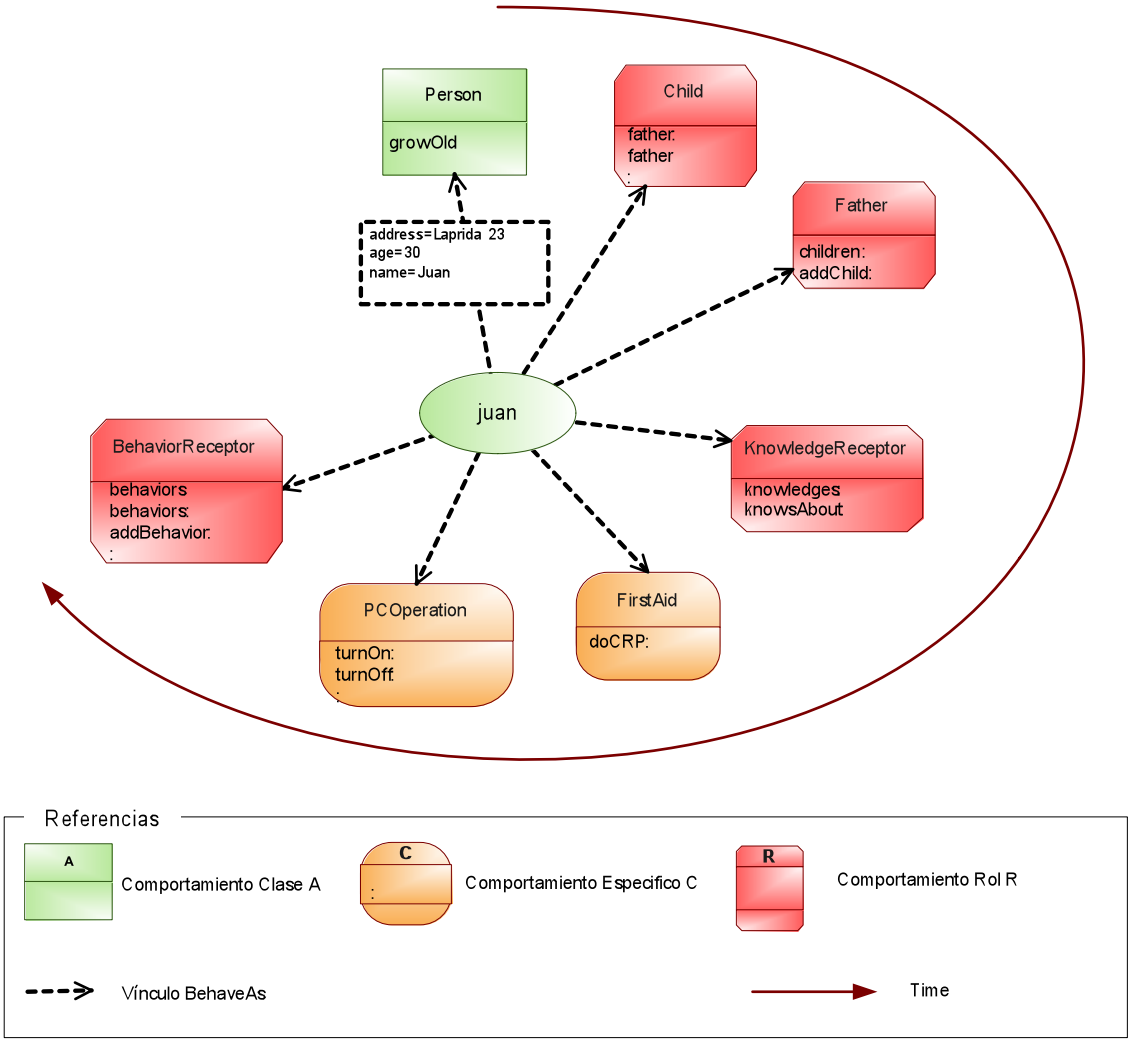


Figura 58. Juan. Vínculos de BehaveAs

En la figura anterior, la orientación de la flecha responde a la secuencia temporal (hipotética) en que *juan* fue adquiriendo los comportamientos. Notar que al registrarse la relación *BehaveAs*, *juan* adquiere el comportamiento de Rol BehaviorReceptor.

> **Nota**. En el gráfico sólo se observan los vínculos calculados por la relación BehaveAs.

Con este contexto, cuando el ML consulte a *juan* por sus comportamientos (paso 1) enviándole el mensaje `behaviors`, se activará el método correspondiente en el Rol `BehaviorReceiver`[21], el cual solicitará a la relación *BehaveAs* los comportamientos asociados. Ésta los responderá (paso 2) en el siguiente órden: `BehaviorReceptor`, `PCOperation`, `FirstAid`, `KnowledgeReceptor`, `Child`, `Father`.

[21] Es necesario que este método esté cableado, ya que si no se entraría en un loop infinito.

3.3 Herramientas

3.3.1 Generador de Comportamiento de Rol

Como ya lo anticipamos, el comportamiento que un objeto adquiere por participar de una relación bajo cierto Rol puede ser deducido en forma autónoma por una herramienta sin necesidad de contar con la intervención de un programador.

Para llevar adelante esta tarea se requiere de información contenida en la misma definición de la relación. A saber

- Nombre (en singular y plural) del Rol
- Nombre (en singular y plural) de los otros Roles
- Cardinalidad
- Multiplicidad para con cada uno de los otros Roles
- Visibilidad de los otros roles
- Navegavilidad hacia los otros Roles
- Relación de Lectura / Lectura y Escritura

Con esta información el **`RRoleBehaviorBuilder`** -objeto responsable de la creación de este tipo de comportamiento- puede definir un nuevo comportamiento con cierto protocolo e implementación.

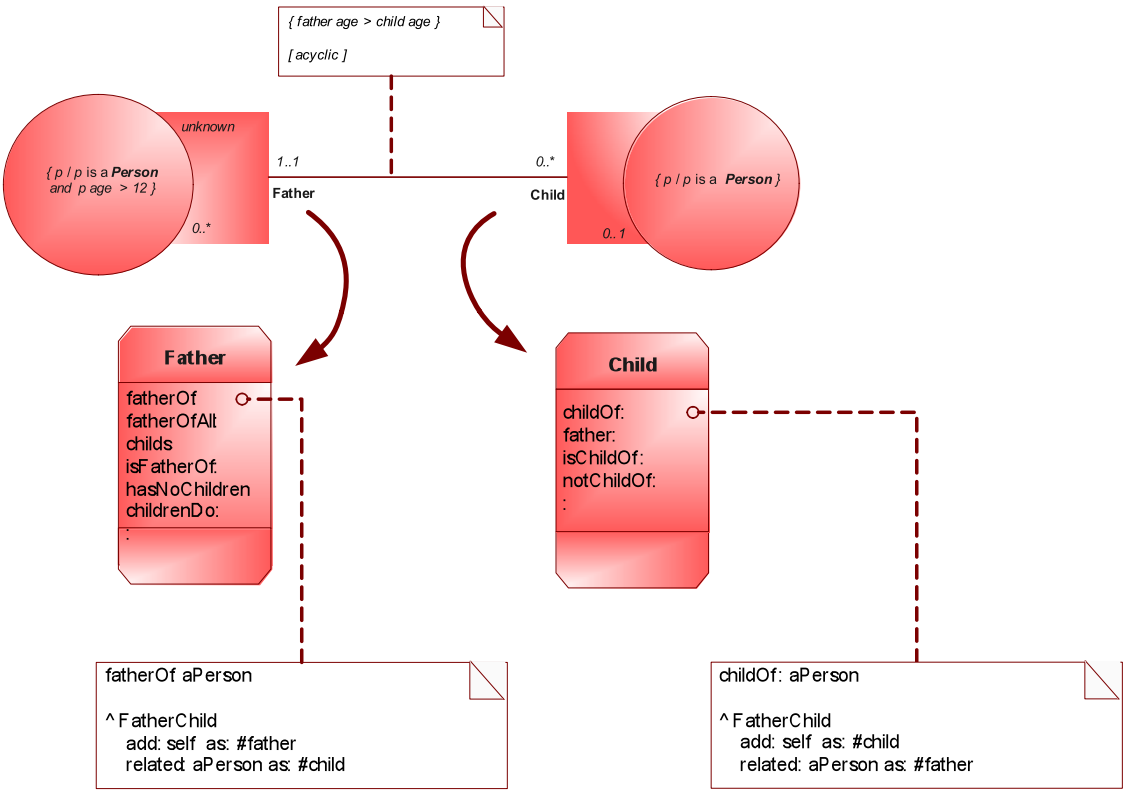


Figura 59. FatherChild. Esquema de proceso de inferencia de comportamiento de rol 2.

Pero para lograr flexibilidad necesitamos contar con un mecanismo que nos permita indicarle al **RRoleBehaviorBuilder** qué consideramos Comportamiento de Rol. Esto es de relativa complejidad ya que necesitamos describir abstracta y formalmente aquello que entendemos como Comportamiento de Rol.

A los efectos de lograr este objetivo, sumamos a nuestro modelo dos abstracciones. Llamamos a la primera **RTemplateBehavior**, y le dimos la responsabilidad de representar un subconjunto del comportamiento de Rol abstracto. La segunda, que llamamos **RTemplateMethod**, modela un método abstracto de un **RTemplateBehavior**.

En las siguientes secciones profundizamos estos conceptos.

3.3.1.1 Plantilla de Método

Existe una gran cantidad de métodos cuya implementación posee una estructura análoga.

Para ilustrarlo pensemos en las relaciones *FatherChild* y *WholePart*. Luego comparemos una posible implementación del método `addChild:` y `addPart:` en los Comportamientos de Rol de `Father` y `Whole`, respectivamente.

```
Father >> addChild: aChild
  ↑FaherChild
            add: self
            as: #father
            related: aChild
            as: #child
```

```
Whole >> addPart: aPart
  ↑ WholePart
            add: self
            as: #whole
            related: aPart
            as: #part
```

Como podemos observar, aunque no idénticos, ambos métodos son análogos entre sí. Luego, si abstraemos la estructura subyacente nos encontramos con algo que podemos expresar de la siguiente manera

```
add!RoleB!: a!RoleB!
  ↑!Relation!
            add: self
            as: !roleAId!
            related: a!RoleB!
            as: !roleBId!
```

donde

- *!Relation!* representa la referencia a la relación en cuestión
- *!RoleB!* hace referencia al nombre del rol B
- *!roleAId!* representa el identificador del rol A
- *!roleBId!* representa el identificador del rol B

Nota RoleA y RoleB dependen de la perspectiva del Rol cuyo comportamiento deseamos expresar. Así, cuando queremos representar el comportamiento de `Father`, el Role A será `Father` y el B `Child`. Esto se invertirá si la perspectiva es en función de `Child`.

En base a esta abstracción, podemos considerar que `Father>>addChild:` y `Whole>>addPart:` son equivalentes respecto de la clase de equivalencia definida por el método abstracto `add!RoleB!:`

El patrón expresado por el método abstracto `add!RoleB!:` es lo que en nuestro trabajo venimos a llamar **`RTemplateMethod`**.

Siguiendo este ejemplo, imaginemos que deseamos iterar sobre los hijos de una persona. Se nos ocurren las siguientes implementaciones

```
aPerson children do: [ :each | Transcript show: each name ]
```

y

```
aPerson childrenDo: [ :each | Transcript show: each name ]
```

Si bien ambas implementaciones son equivalentes (producirán los mismos resultados), la segunda alternativa es claramente más cercana a las convenciones del mundo de objetos, cumpliendo con la Ley de Demeter, que tiende a softwares más mantenibles y adaptables.

Nota La ley de Demeter es reconocida en la comunidad de objetos [Rozenfarb, 2001] como una de las mejores herramientas para favorecer el encapsulamiento, y por ende el buen diseño. La misma sostiene que uno no debería obtener una parte interna de un objeto para enviarle un mensaje, sino que debe enviarlo al objeto, el cual puede implementar la operación reenviando el mensaje a su parte. El resultado de utilizar la ley de Demeter es que el método que envía el mensaje depende solamente de la interfase de sus colaboradores, no de su estructura.

En los ambientes basados en clases, una de las desventajas de utilizar Demeter es que en ocasiones requiere a los programadores escribir un gran numero de pequeños métodos “wrapper” [DemeterLaw] como es el caso de `childrenDo:`. Por ello resulta muy difícil volcar, por ejemplo, el protocolo de **`Collection`** para operar sobre los hijos de una persona, y aún más continuar repitiéndolo.

Para graficar lo anterior, planteamos para cada método en el protocolo de enumeración de Collection cómo se vería una implementación con y sin la aplicación de la ley de Demeter.

A. Protocolo de enumeración en collection	B. Protocolo de enumeración para una persona sobre sus hijos **(sin demeter law)**	C. Protocolo de enumeración para una persona sobre sus hijos **(con demeter law)**
`allSatisfy:`	`children allSatisfy:`	`childrenAllSatisfy:`
`anySatisfy:`	`children anySatisfy:`	`childrenAnySatisfy:`
`associationsDo:`	`children associationsDo:`	`childrenAssociationsDo:`
`collect:`	`children collect:`	`childrenCollect:`
`collect:thenSelect:`	`children collect:thenSelect:`	`childrenCollect:thenSelect:`
`count:`	`children count:`	`childrenCount:`

detect:	children detect:	childrenDetect:
detect:ifNone:	children detect:ifNone:	childrenDetect:ifNone:
detectMax:	children detectMax:	childrenDetectMax:
detectMin:	children detectMin:	childrenDetectMin:
detectSum:	children detectSum:	childrenDetectSum:
difference:	children difference:	childrenDifference:
do:	children do:	childrenDo:
do:separatedBy:	children do:separatedBy:	childrenDo:separatedBy:
do:without:	children do:without:	childrenDo:without:
groupBy:having:	children groupBy:having:	childrenGroupBy:having:
inject:into:	children inject:into:	childrenInject:into:
intersection:	children intersection:	childrenIntersection:

Tabla 3. Demeter Law sobre el protocolo de Collection.

Si quisiéramos utilizar Demeter tanto en los comportamientos de `Father` como de `Whole`, un programador debería reescribir todos estos métodos una segunda vez. Y una tercera si quisiéramos hacerlo para `Category`, y una cuarta para `Observer`. Es evidente que esto no resulta práctico.

Pero el problema no termina aquí. También ocurre que debido a dicha necesidad de replicación de código, las protocolos comienzan a quedar incompletos e incoherentes entre sí. Por ejemplo, en Squeak, aunque `isEmpty` esta definido en 21 clases, solo en 2 de ellas también se define `notEmpty`, y solo una define `ifEmpty:` and `ifNotEmpty:`. [Black and Sharli 2004]

Los **`RTemplateMethod`** ayudan sortear estos obstáculos, permitiendo definir abstractamente y por única vez la implementación de cada uno de estos métodos.

A modo de ejemplo, resulta muy sencillo definir un **`RTemplateMethod`** para el caso “chidrenDo:”

```
!roleBs!Do: aBlock
  self !roleBs! do: aBlock
```

Luego, al aplicarlo al comportamiento de los roles `Father` y `Whole` resultará en las siguientes implementaciones

```
childrenDo: aBlock
  self children do: aBlock
```

y

```
partsDo: aBlock
  self parts do: aBlock
```

3.3.1.2 Plantilla de Comportamiento

En la sección anterior analizamos el potencial de expresar métodos del protocolo de Rol en términos abstractos con instancias de **`RTemplateMethod`**.

El otro componente que utilizamos durante nuestra solución para representar comportamiento abstracto fue el **`RTemplateBehavior`**. Un **`TemplateBehavior`** representa un subconjunto del comportamiento de Rol. Para ello se componen de un conjunto de **`RTemplateMethod`**.

Sin embargo, más allá de la capacidad de organización de un conjunto de **`RTemplateMethod`**, la principal responsabilidad de un **`RTemplateBehavior`** es indicar cuándo los métodos podrán aplicar al rol en cuestión, y cuándo no. Por ejemplo, para la relación *FatherChild*, no tiene sentido dar a un `Child` protocolo de `addFather:` (un `Child` solo puede tener un `Father`).

3.3.1.3 Proceso de generación del Comportamiento de Rol

Durante el proceso de registración de la relación se lleva adelante la creación del Comportamiento de cada uno de sus Roles.

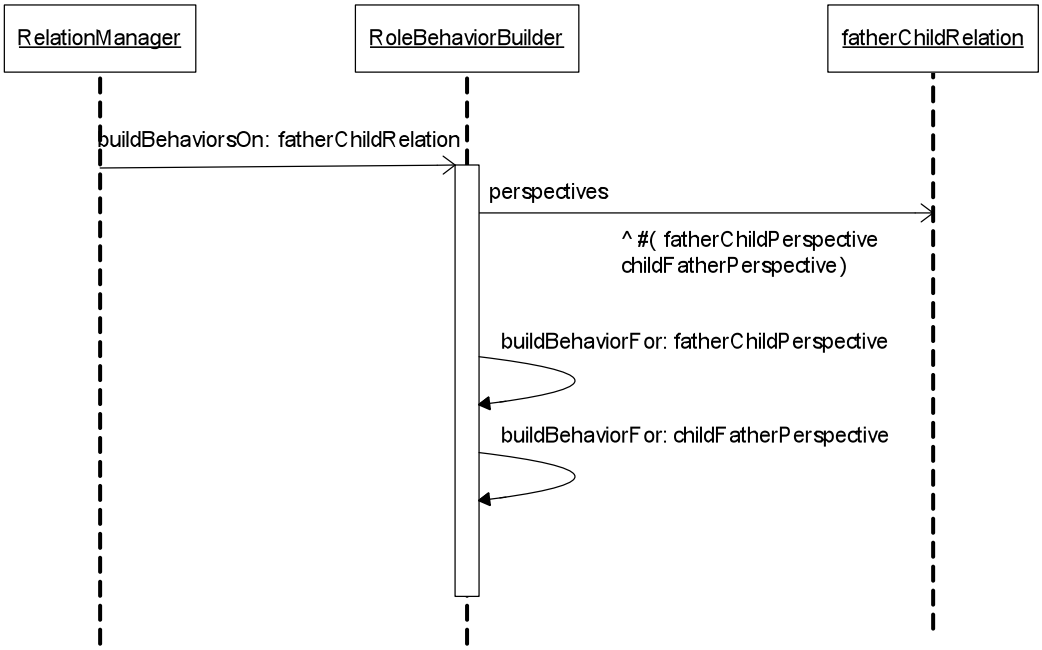
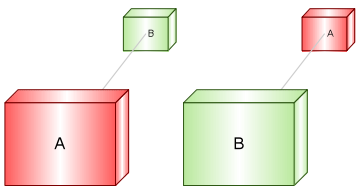


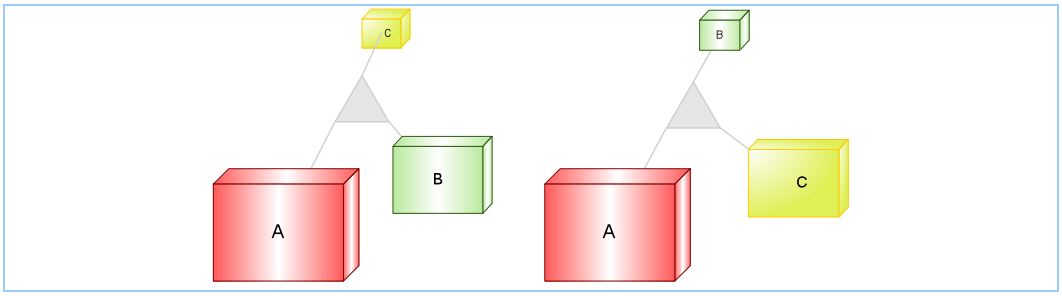
Figura 60. RoleBehaviorBuilder. Diagrama de Secuencia

El **`RRoleBehaviorBuilder`** predeterminado es responsable de generar el comportamiento de cada uno de los roles de la Relación. En el ejemplo vemos como por cada una de las perspectivas de una relación *FatherChild* se construye o genera comportamiento.

Nota. Una perspectiva (**`RPerspective`**), es una visión de la relación desde el punto de vista de uno de sus roles. Por ejemplo, en relaciones binarias solo se presentan 2 perspectivas: "Rol A-Rol B", y "Rol B-Role A"



En la figura anterior observamos como cada rol posee su propia y única perspectiva: observa al otro rol. Sin embargo, en una relación ternaria, un mismo rol puede observar la relación desde más de una perspectiva: estas perspectivas dependen del orden en que observe los otros roles.



Retomando el proceso de creación de Comportamiento, un **`RRoleBehaviorBuilder`** aplica sobre cada una de estas perspectivas los **`RTemplateBehavior`** definidos.

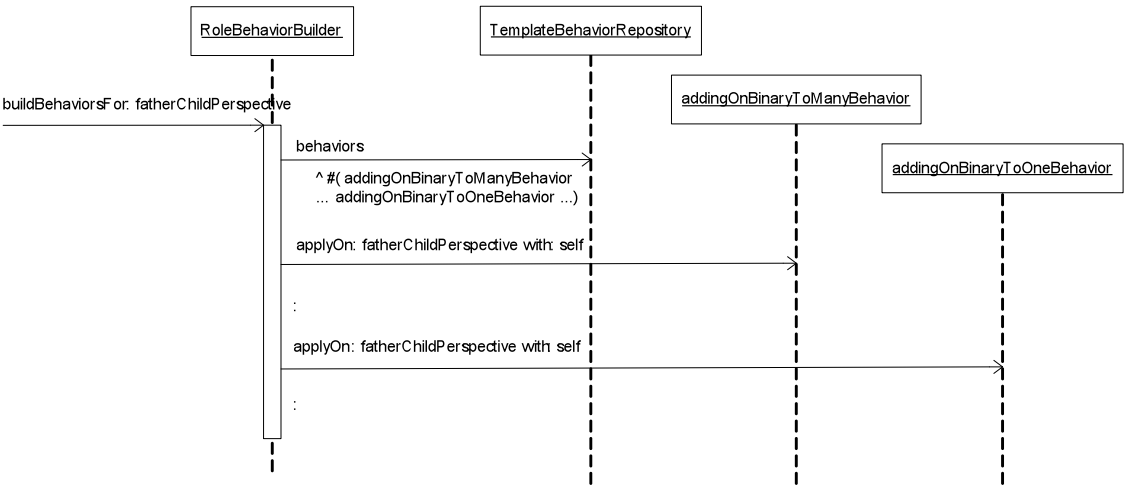


Figura 61. RoleBehaviorBuilder. Diagrama de Secuencia 2.

Si el **`RTemplateBehavior`** valida su precondición para dicha perspectiva, se procede a la aplicación de cada uno de los **`RTemplateMethod`**, de lo que resulta un método concreto a incorporar en el Comportamiento del Rol A de dicha perspectiva.

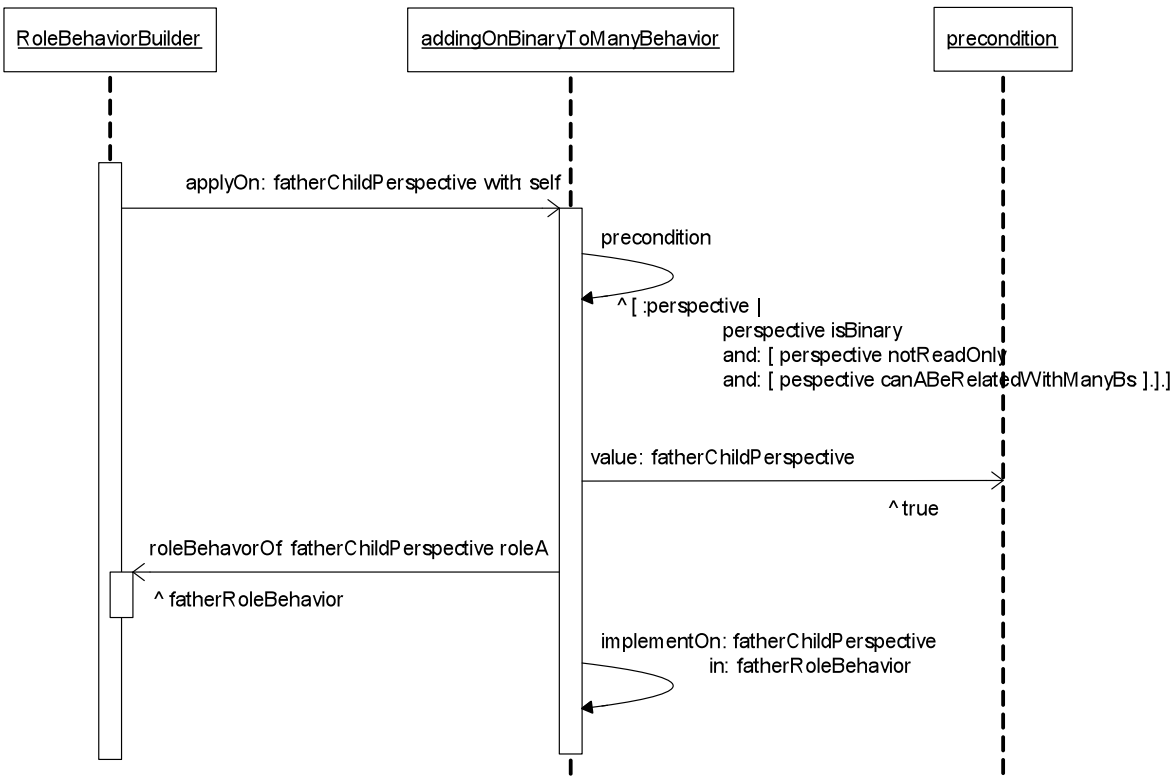


Figura 62. RoleBehaviorBuilder. Diagrama de Secuencia 3

Es durante la ejecución del método `implementOn:in:` donde se lleva adelante el proceso de creación de cada uno de los métodos, acorde a lo definido en cada **`RTemplateMethod`** del **`RTemplateBehavior`**.

> **Nota.** Si bien en nuestro trabajo optamos por implementar los **`RTemplateMethod`** sobre los `CompiledMethod` de Smalltalk, es oportuno mencionar que analizamos la posibilidad de brindar una solución más automática einteligente donde no se requiera la mera generación repetitiva de código. Aunque por cuestiones de alcance dejamos fuera esta posibilidad, consideramos interesante seguir investigando sobre el tema en posibles trabajos futuros.

3.3.2 Browser de Plantillas de comportamiento

Como hemos mencionado, es de sumo interés que usuarios o programadores avanzados con conocimiento en el Framework puedan agregar, modificar o remover patrones de comportamiento. Es por ello que necesitábamos un Browser que les permitiese interactuar con el modelo. Así surgió el **`RTemplateBehaviorBrowser`**.

Dentro de sus características esenciales, esta herramienta debe permitir

- Agregar/Remover un **`RTemplateBehavior`**
- Editar un **`RTemplateBehavior`** agregando, editando o removiendo un **`RTemplateMethod`**
- Editar la precondición de un **`RTemplateBehavior`**

En la figura siguiente se muestra el **`RTemplateBehaviorBrowser`**

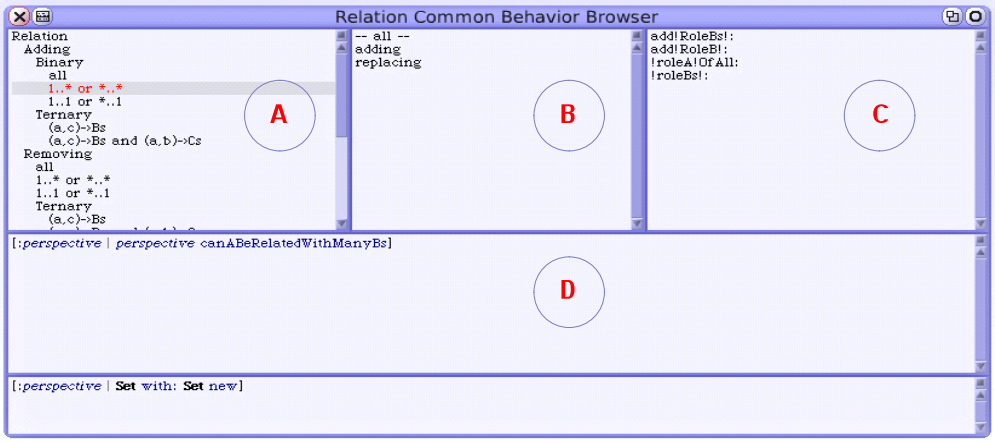
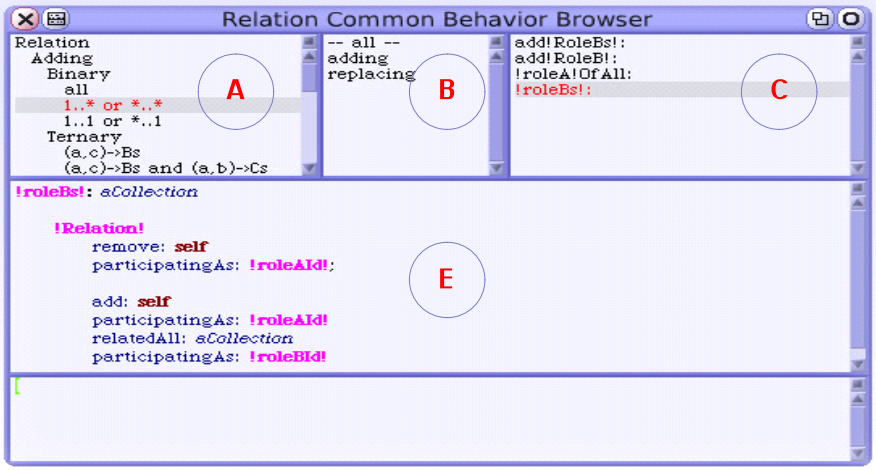


Figura 63. RTemplateBehaviorBrowser. Browser de patrones de comportamiento

En el panel A observamos los **`RTemplateBehavior`** definidos. Estos se encuentran ordenados jerárquicamente en función de la expresión de su precondición para aplicar sobre una perspectiva.

Seleccionado un **`RTemplateBehavior`**, en el panel B se despliegan las categorías en que se encuentran organizados los **`RTemplateMethod`** (similar a como en un `ClassBrowser` se presentan las categorías de los métodos de una clase). El panel C observamos un listado con todos los **`RTemplatesMethod`** ya definidos. Por último, el panel D permite ver y editar la precondición del **`RTemplateBehavior`** seleccionado[22].

Al seleccionar un **`RTemplateMethod`** veremos en el panel E su definición, la cual podrá ser editada tal como se realiza con cualquier otro método de clase en Squeak.



[22] Recordar que esta expresión debe componerse con la de los padres de este comportamiento en la jerarquía.

Figura 64. RTemplateBehaviorBrowser. Browser de patrones de comportamiento 2

3.3.3 Inspector de Relación

En el capítulo anterior presentamos el **`RRelationBrowser`** capaz de crear, editar relaciones haciendo uso del **`RRelationInspector`**. Luego describimos cómo un **`RRelationInspector`** delega al **`RRolInspector`** la responsabilidad de inspeccionar un rol en el marco de la relación.

Así fue cuando expusimos las dos primeras solapas –*Type* y *Perspective*- de las tres que utiliza el **`RRolInspector`** y optamos por describir la tercera solapa en este capítulo dado que tiene por objeto operar contra el comportamiento de dicho rol en la relación.

Como ya hemos mencionado, luego de crear una relación ésta se registra en el ambiente, y al hacerlo se crean los comportamientos de Rol correspondientes en base a los patrones de comportamiento predefinidos.

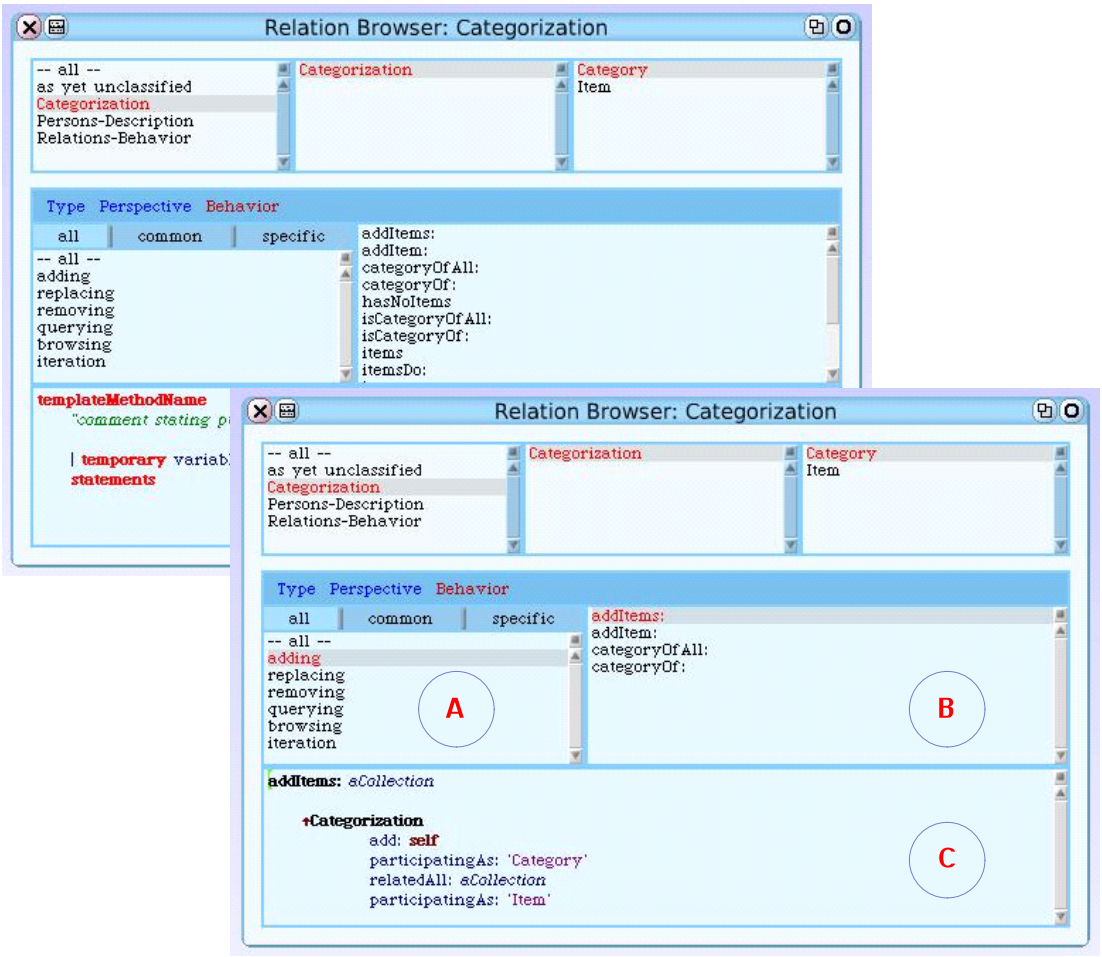


Figura 65. RRelationBrowser. Browser de comportamiento de Rol

En la figura podemos observar que en el panel A se presentan las categorías de los métodos, mientras que en el B, los métodos concretos post implementación de cierto patrón. Finalmente, al cliquear sobre uno de estos selectores, en el panel C observamos la implementación del método correspondiente.

En nuestro trabajo hemos optado por ofrecer comportamiento por grupos basado en los **`RTemplateBehavior`** (es decir se modifica un **`RTemplateBehavior`** y este cambio afecta

transparentemente a todos los comportamientos de Rol generados por este), no permitir modificar la implementación de aquellos métodos generados por el **`RRoleBehaviorBuilder`**.

Sin embargo este browser nos permite crear, editar y borrar –al igual que un `ClassBrowser`- nuevos métodos, específicos para el rol seleccionado[23].

3.3.4 Inspector de Comportamiento

Los inspectores son una importantísima herramienta que permiten a usuarios del ambiente interactuar y dialogar con los objetos directamente.

Si un objeto adquiere nuevas capacidades, cualquiera que fuesen, éstas deberían ser presentadas a través de su inspector.

Al presentar el inspector basado en relaciones durante el capítulo anterior, justificamos el inspector clásico no era suficiente, ya que sólo presentaba variables de instancia, cuando también era necesario mostrar las relaciones en las cuales el objeto estuviera en condiciones de participar.

De la misma manera, al brindar a los objetos del ambiente la posibilidad de recibir comportamiento accidental es necesario que nuestro inspector asimile dicha capacidad. Por ello agregamos al **`RInspector`** una nueva solapa llamada *Behavior*, que al ser seleccionada presenta el inspector de comportamiento de un objeto.

En la figura siguiente presentamos al inspector de comportamiento del *`pablo`*

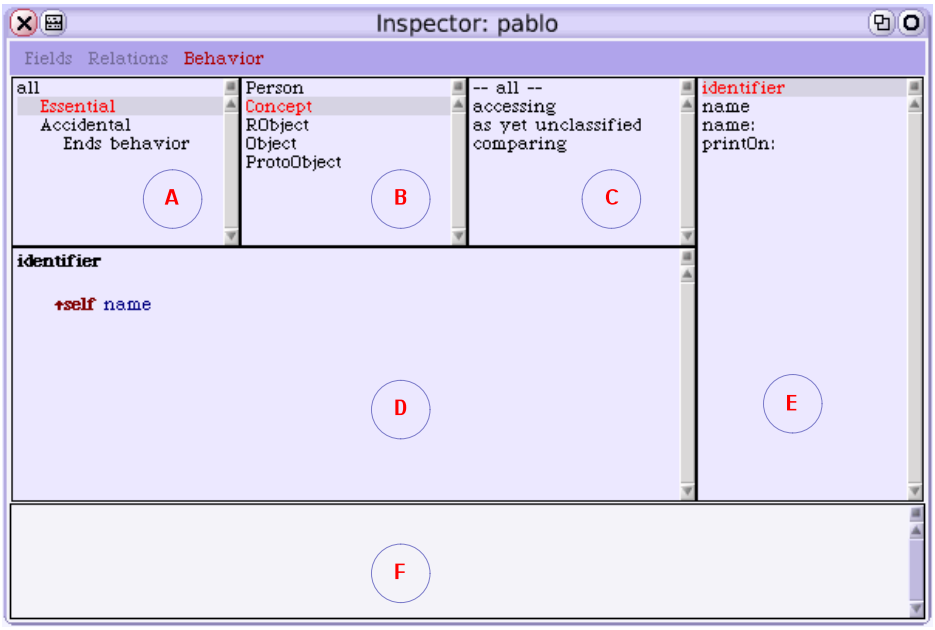


Figura 66. RObjectBehaviorInspector. Inspector de comportamiento de Rol.

En el panel A se presenta una primera clasificación de los comportamientos de un objeto: los *esenciales* y los *accidentales*. Al seleccionar los esenciales de despliega en el panel B una lista ordenada que presenta la jerarquía que clases que dan comportamiento esencial a *`pablo`*.

[23] Recordemos la necesidad del pattern Observer para extender los comportamientos de Rol con los métodos `changed:` y `update:`

Luego al hacer clic sobre una de estas clases, se observa el comportamiento definido en ella. Así se actualiza el panel C con las categorías de métodos de la clase, y el panel E con los métodos de dicha clase (similar al `ClassBrowser`).

Por último, seleccionando uno de los métodos se muestra en el panel D la implementación de dicho método.

Por el otro lado, partiendo de los comportamientos accidentales podemos apreciar en la próxima figura que éstos pueden a su vez encontrarse subcategorizados. Haciendo clic en *Role Behavior* se despliegan en el panel B todos los comportamientos de rol en los cuales el objeto podría potencialmente participar[24].

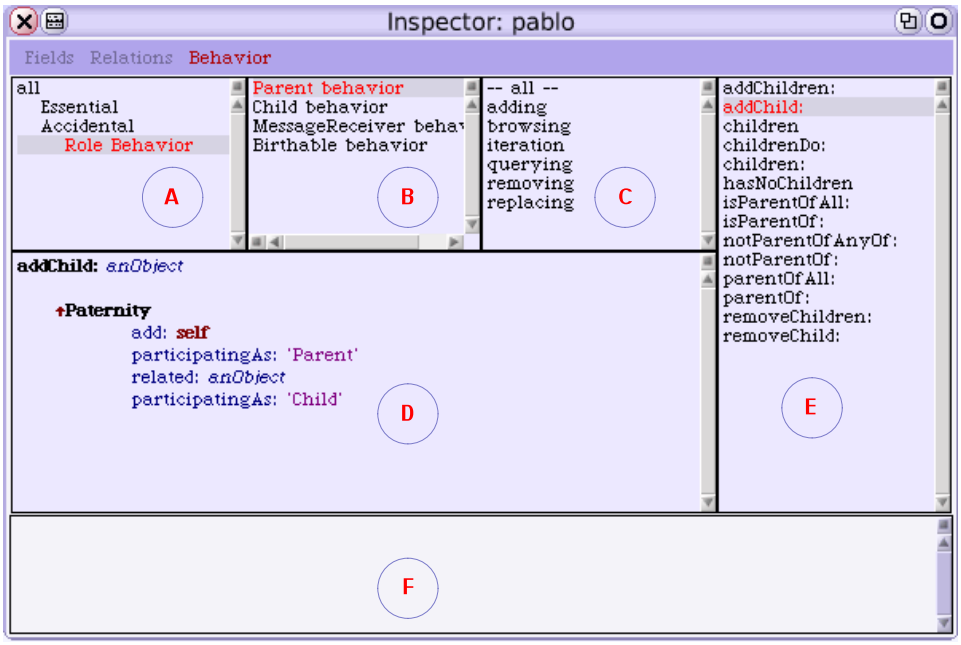


Figura 67. RObjectBehaviorInspector. Browser de comportamiento.

Al seleccionar uno de estos comportamientos, el panel C presenta las categorías con que fueron clasificados los **`RTemplateMethod`** en sus correspondientes **`RTemplateBehavior`**. Luego, en el panel E se presentan los métodos resultantes de la implementación de cierto patrón de método en el comportamiento de Rol.

3.4 Consecuencias

3.4.1 Mecanismo de sharing

3.4.1.1 Sharing y Relaciones

Como ya mencionamos, todos los mecanismos de sharing utilizan relaciones y, a pesar de sus diferencias, [Bobrow and Stefik 1986][Stein et al. 1988], tienen los siguientes puntos en común[Bardou and Dony 1996]

[24] Estos son los comportamientos que han sido generados automáticamente por el **`RRoleBehaviorBuilder`** basado en los patrones de comportamiento definidos por el usuario.

- Están basados en una o varias relaciones[25]
- Utilizan estas relaciones para lograr la herencia[Bobrow and Stefik 1986]
- Requieren que una semántica clara y precisa de las relaciones sobre las que operan.
- Tienen como fin lograr algún tipo de sharing entre objetos

Nuestro modelo no sólo cumple, sino que también requiere de estos principios.

3.4.1.2 Sharing Anticipado vs. No anticipado

Stein, Lieberman y Ungar distinguieron - en [Stein et al. 1988]- dos tipos de motivaciones para introducir el sharing en un sistema de objetos: Sharing Anticipado y No Anticipado.

Mientras que el *Sharing Anticipado* es aquel que se aplica cuando durante la fase conceptual de diseño, un diseñador puede anticipar propiedades en común entre diferentes partes del sistema; el *Sharing No Anticipado* es el que resulta de la necesidad -de un diseñador- de agregar funcionalidad a un objeto, sin que esta haya sido prevista originalmente.

Dados los caminos –muchas veces impredecibles- que sigue la evolución del software, sin duda es importante disponer de un ambiente que soporte *Sharing No anticipado*. Los lenguajes con *Delegation* o la *DinamycInheritance* logran esto al permitir que los nuevos objetos reutilicen el comportamiento de otros sin necesidad de una especificación previa de la relación [Stein et al. 1988].

Nuestro modelo también presenta beneficios en este sentido. Por un lado porque un objeto adquiere comportamientos en forma dinámica en función de las relaciones de comportamiento en que participe. Y por el otro, porque no es necesario definir dichas relaciones en tiempo de diseño, permitiendo especificarlas y agregarlas durante el ciclo de vida de los objetos.

Por tal motivo, el modelo propuesto no sólo permite ambos tipos de *Sharing*, sino que también a la vez ofrece mayor poder semántico que otros ambientes limitados a las relaciones *InstanceOf* y *SubclassOf*, o *InheritFrom*.

3.4.1.3 Empatía

En los lenguajes de programación orientados a objetos, la *empatía* es un mecanismo que permite a un objeto tomar prestado atributos (métodos o variables) de otro.

Decimos que un objeto `a` es empático con otro objeto `b` para un mensaje `m` si `a` no tiene un método propio para responder a `m`, sino que cuando se le envía `m` responde como si hubiera tomado prestado el método de respuesta de `b`. `a` toma prestado sólo ese método, y ninguna otra cosa de `b`, o sea que si el método indica que se mande un mensaje a self, lo recibirá `a`.[Stein et al. 1988].

Todas las variantes de delegación e inheritance incluyen alguna forma de empatía, pero difieren en cuándo y cómo se establece la relación. Puede ser explícita o por defecto, dinámica o estática, por objeto o por grupo. A continuación hacemos un breve resumen sobre estos tipos o formas de empatía y analizamos cómo nuestro modelo se adapta a ellas.

La empatía Estática o Dinámica se define al responder la pregunta ¿cuándo se requiere que sean definidos los patrones de sharing (comportamientos)? En un sistema estático, esto debe ser cuando se crea un objeto. En uno dinámico, cuando un objeto recibe un mensaje. Nuestro modelo permite definir lo mínimo necesario en tiempo de diseño, posibilitando la incorporación de nuevos comportamientos (patrones de sharing) una vez creado el objeto. Por ello que decimos que posee empatía dinámica.

[25] Por ejemplo, en los lenguajes basados en clases las relaciones *InstanceOf* y *SubclassOf*, y en Self la relación *InheritFrom*. [UNG87]

En cuanto a si es Implícita o Explicita, debemos preguntarnos ¿hay una operación con la que el programador indique explícitamente los patrones de sharing, o el sistema lo hace automática y uniformemente? Hacerlo explícito permite delegar un único método (ej: ejecuta thisobject thismethod). Nuestro modelo sigue el espíritu y las convenciones de Smalltalk, realizando un tratamiento implícito de la empatía.

Por último, para responder si es por objeto o por grupo, tenemos que preguntarnos: ¿La conducta se especifica para un grupo de objetos de una vez, o se puede poner comportamiento en un objeto individual? ¿Se puede especificar/garantizar la conducta de un grupo de objetos? Como en nuestro modelo ambas alternativas son posibles decimos que es posible contar tanto con empatía de objetos, como de grupos.

3.4.1.4 Templates

Un template -lo que hasta aquí hemos llamado Comportamiento- es una especie de molde de galletas para objetos: contiene todas las definiciones de métodos y variables necesarias para definir un nuevo objeto de cierto tipo. Si una vez definido el objeto no puede ganar ni perder atributos, lo llamamos un template estricto. En algunos lenguajes el template es un objeto común, mientras que en otros es un objeto generador distinto y especial, como la clase. También hay lenguajes que no utilizan templates, y en estos no hay un concepto inherente de grupo o especie para los objetos [Stein et al. 1988]

Nuestro modelo utiliza templates, aunque con un enfoque particular. A diferencia de la mayoría de los lenguajes basados en clases, el comportamiento de un objeto no está dado por un único template, con quien está vinculado a través de la relación *InstanceOf*. Por el contrario, un objeto, además de relacionarse con su clase, puede también –aunque no es necesario- vincularse con otros templates vía otras relaciones.

3.4.2 El problema de *self*

Debido al tratamiento que realizan sobre la pseudo-variable `self`, los ambientes basados en clases impiden una adecuada implementación para fowardear mensajes (delegación).

A menudo, un método al llevar adelante un mensaje puede necesitar que el objeto que originalmente recibió el mensaje realice algún servicio. Para permitir esto en ambientes con delegación, cuando se delega un mensaje, el método a ejecutar recibe un componente llamado “el cliente” junto al mensaje, el cual posee el objeto que originalmente recibió dicho mensaje.

En los ambientes basados en clase, con Herencia, la variable `self` es automáticamente ligada al receptor del mensaje durante la ejecución del código de un método. Cuando la búsqueda del método pasa de la clase original a la superclase, el valor de `self` no cambia, de manera tal que los métodos de la superclase pueden responder al mensaje “como si” ellos mismos fueran métodos del objeto original. Pero, cuando un usuario envía un mensaje, la variable `self` es automáticamente re-ligada al nuevo receptor, lo que hace generalmente imposible para el usuario designar que otro objeto responda en lugar del objeto que originalmente recibió el mensaje [Kristensen 1996].

Un ejemplo de esta incapacidad de los ambientes basados en clases sería un `wrapper` que quiera contar (o loggear) los envíos de mensajes a un objeto. Supongamos entonces que este `wrapper` implementa el `doesNotUnderstand` actualizando la cuenta y enviando el mensaje a su `wrapee`. Luego, si durante la consecuente ejecución del método, el wrapee envía mensajes a sí mismo vía `self`, estos mensajes no podrán ser contabilizados por el `wrapper` (no percibirán su existencia).

Otro ejemplo encontramos en el pattern de State. En éste se presenta nuevamente la idea de foward de mensajes. Como el `state` puede necesitar usar los servicios del `context`, la solución debe brindar los

medios para que desde un `state` se pueda referenciar el `context` que delegó el mensaje. En [Alpert et al. 1998] se discuten dos alternativas para lograr esto. Una de ellas propone que cada `state` mantenga una variable de instancia referenciando a `context`. La otra, que, en cada implementación de un método de `state` se contemple un colaborador externo para referenciar dicho `context`.

El modelo de comportamiento propuesto en nuestro trabajo no soluciona el *problema de self* presentado en este apartado. Sin embargo minimiza la posibilidad de sus ocurrencias al presentar soluciones alternativas a implementaciones donde comúnmente se utilizaría delegación, cómo es el caso del pattern State.

Quedará para trabajos futuros analizar que pasaría si modeláramos una nueva relación *Delegation*, complementando con ellas las relaciones de comportamiento, y adaptáramos el mecanismo de ML para que la contemple.

Nuestra intuición nos indica que es sólo un problema semántico que la presencia de relaciones puede ayudar a resolver.

3.4.3 Nuevas alternativas y soluciones

Con el modelo de comportamiento que hemos descripto, se nos presentan nuevas posibilidades o herramientas que permiten

- definir comportamientos fuera del marco de una clase
- vincular comportamientos con diferentes objetos, sin importar su clase
- un objeto puede poseer –además de su comportamiento esencial- muchos comportamientos accidentales

Es por ello que un diseñador o programador de un ambiente de clases y relaciones puede enfrentar un dominio de problema desde nuevos puntos de vista.

El pattern State puede ser uno de estos problemas. Este pattern apunta a resolver cómo modelar el comportamiento dinámico de un objeto, teniendo en cuenta que este dinamismo depende de su estado interno.

Como hemos mencionamos, podemos plantear una solución alternativa donde los estados sean meros comportamientos. Si este es el camino a seguir, necesitaremos una relación de comportamiento (supongamos *StateOf*) que brinde la semántica y reglas necesaria de los vínculos entre un objeto (`context`) y su estado (`state`).

3.4.4 Menor sobrecarga semántica de *SubclassOf*

Como ya dijimos, en los ambientes basados en clases es común observar implementaciones donde se sobrecargue la relación *SubclassOf* con distintas responsabilidades como por ejemplo promover el reuso de código, organizar y clasificar clases, definir protocolos estándar y hacer extensiones a clases existentes [Johnson and Foote 1988].

Los comportamientos basados en relaciones ayudan a disminuir este problema. Por un lado porque naturalmente separa las implementaciones de clase y de rol, ayudando la organización y clasificación entre elementos del dominio.

Por otro lado, la posibilidad extender los comportamientos de un objeto en función de las relaciones en que participa evita sobrecargar la semántica de *SubclassOf* para extender las clases a nuevos protocolos.

Por último la definición de patrones de comportamiento desde la perspectiva de un rol, ofrece a todo objeto que participe de una relación un protocolo estándar y coherente.

3.4.5 Protocolos con más semántica

Imaginemos que deseamos implementar el método `hasChildrenDescription` para que una persona responda un string indicando *'Si, tengo hijos'* y *'No, no tengo hijos'*.

Un programador podría implementar el método como se muestra en el caso A.

```
Person >> hasChildrenDescription
    ^self
      children isEmpty
        ifFalse: ['Sí, tengo hijos']
        ifTrue: ['No, no tengo hijos'].
```

Código 1 – Caso A "Person >> hasChildsDescription"

Si bien el método se encuentra correctamente programado, la mayoría de los programadores Smalltalk experimentados, hubiese utilizado el protocolo de Collection `ifEmpty:ifNotEmpty` en lugar de `isEmpty` para luego aplicar `ifTrue:ifFalse:`

```
Person >> hasChildrenDescription
    ^self
      children
        ifNotEmpty: ['Sí, tengo hijos']
        ifEmpty:  ['No, no tengo hijos'].
```

Código 2 – Caso B "Person >> hasChildsDescription"

En otras palabras, la comunidad de Smalltalk ha percibido la abstracción subyacente –posiblemente por su misma recurrencia en diferentes dominios de problema- e implementado el nuevo método con mayor semántica: *Si la colección esta vacía, hace esto, sino esto otro*.

Así como propusimos el caso B, también podría presentarse la alternativa que el programador decida abstraer la consulta sobre si tiene hijos en un método dedicado a ello *`hasChildren`*.

```
Person >> hasChildrenDescription
    ^self
      hasChildren
        ifTrue:   ['Sí, tengo hijos']
        ifFalse:  ['No, no tengo hijos'].
```

Código 3 – Caso C "Person >> hasChildsDescription"

Podemos observar que en el caso B y C utilizamos 2 mensajes en lugar de 3, como hizo el programador del caso A.

Existe una cuarta alternativa, que es la de conjugar o fusionar los casos B y C, mediante la utilización de un nuevo método con mayor semántica. Este método podría llamarse `ifHasChildren:ifHasNoChildren`: *Si tenés hijos hace esto, sino esto otro.*

```
Person >> hasChildDescription
    ^self
      ifHasChildren:  ['Sí, tengo hijos']
      ifHasNoChildren: ['No, no tengo hijos'].
```

Código 4 – Caso D "Person >> hasChildsDescription"

Como puede observarse, las implementaciones A, B, C y D son equivalentes y responden con la misma semántica. Sin embargo, utilizan implementaciones más o menos abstractas para lograr este fin.

Si bien la implementación del caso D resulta la más atractiva –se utiliza un mensaje en lugar de dos o tres, no rompemos con la ley de Demeter y es más legible- es difícil ver soluciones en Smalltalk donde se utilice tal nivel de expresividad. Esto puede encontrar su causa en que el programador no siempre privilegia una mayor semántica en los métodos por sobre el tiempo que insume su creación y posterior testeo.

Por ello, es posible observar implementaciones donde se expone al emisor del mensaje la estructura interna del objeto.

```
aCategory items: Set new.
```

Creamos que lo ideal sería reflejarlo con un mensaje con mayor poder semántico, que evite transferirle tal responsabilidad al emisor, permitiendo que éste sólo se preocupe por expresar su intención a través del envío del mensaje. Así, será exclusivamente la categoría quien decida como reflejar que no tiene ítems.

```
aCategory noItems
```

Pero como decíamos, es difícil que la autodisciplina de un programador (o de muchos) logre este tipo de protocolo donde a la vez se cumpla que sean completos, que perduren a lo largo de todo el ciclo de vida de un software, y que además sean coherentes, uniformes y estándares entre sí.

Cabe preguntarnos qué pasaría si los métodos ya estuvieran ahí, disponibles, y no fuera necesario esfuerzo alguno en desarrollarlos y testearlos. Nuestra intuición y experiencia nos indica que serían rápidamente adoptados por la comunidad.

En este sentido, nuestro modelo ofrece los medios para responder dicha pregunta, porque deja al alcance del programador estos protocolos en los objetos participantes en una relación.

Y lo que es más, en caso de que sea necesario, un usuario del Framework tendrá la posibilidad de definir nuevos patrones de comportamiento que serán natural y transparentemente extendidos a todos los objetos que lo requieran.

Capítulo 4

Pruebas de concepto

En este capítulo presentamos tres casos que sirven como prueba de concepto de distintos resultados obtenidos durante nuestro trabajo.

El primero de ellos nos permite observar y comprender el potencial de Relaciones como base central de otras herramientas, como por ejemplo un Framework de Interfaz de Usuario.

El segundo se enfoca en mostrar cómo Relaciones ayuda a una rápida prototipación y continua iteración en fases tempranas del proyecto.

Por último, en el tercero dejamos de lado los aspectos estructurales para introducirnos en el área de representación de comportamiento. Aquí exponemos una implementación alternativa al pattern State utilizando Relaciones.

4.1 Edición de objetos de negocio

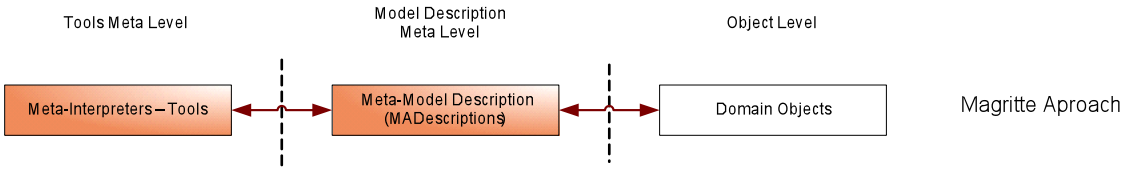
La mayoría de las aplicaciones comerciales requieren construir, presentar y validar manualmente un gran número de ventanas de diálogo y reportes. A menudo estos diálogos permanecen estáticos luego de que la fase de desarrollo ha finalizado y no pueden ser modificados sin requerir de nuevos desarrollos. Este cuadro empeora ya que con frecuencia los usuarios finales necesitan que sus sistemas se adapten rápidamente a sus necesidades de negocio[Renggli 2006]

Como ya hemos mencionado, describir dominios de problemas con Relaciones potencia la creación de nuevas herramientas, tanto como posibilita la integración con otras ya existentes. Con este objetivo en mente, nos proponemos integrar nuestro Framework con el de Magritte[Renggli 2006] a los efectos de disponer y hacer uso de sus herramientas para construir en forma automática vistas, reportes y editores con validaciones.

4.1.1 Arquitectura de Magritte

Magritte tiene la capacidad de describir objetos del dominio desde un nivel meta. A partir de esto, dispone de un Framework que facilita un conjunto de herramientas que interpretan el meta-modelo y permiten en forma automática construir vistas, reportes, editores con validaciones y mecanismos de persistencia sobre estos objetos de dominio [Renggli 2006].

A continuación presentamos la arquitectura propuesta por Magritte.



Para representar la descripción del modelo, Magritte utiliza el concepto Descripción, implementado con `MADescription` y sus subclases. Una Descripción se encarga de especificar o detallar las características de un aspecto de la clase de dominio bajo descripción, como ser un variable de instancia, sus propiedades y relaciones. [Renggli 2006].

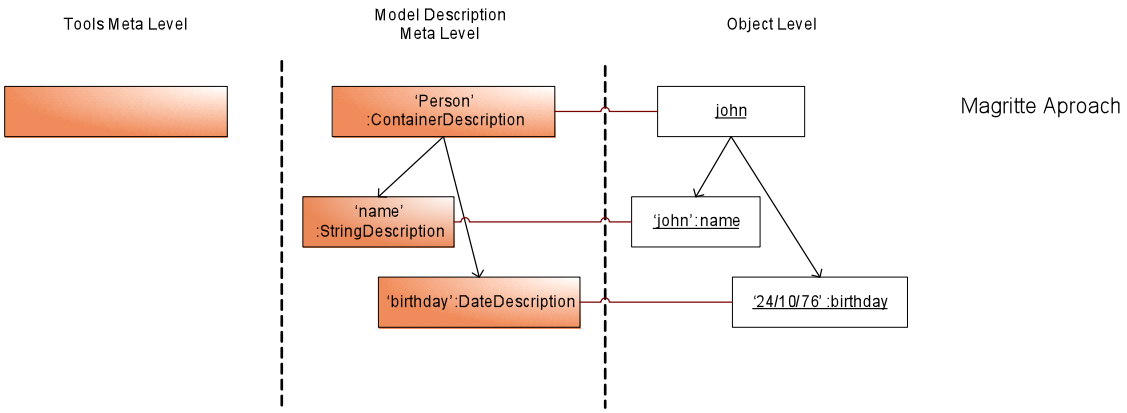


Figura 68. Magritte. Esquema de Niveles

En el ejemplo anterior, podemos apreciar la descomposición del objeto `john`, junto con la descripción correspondiente del mismo en el meta nivel.

4.1.2 Arquitectura de la integración

Nuestra propuesta para modelar la integración de Relaciones con Magritte fue la siguiente:

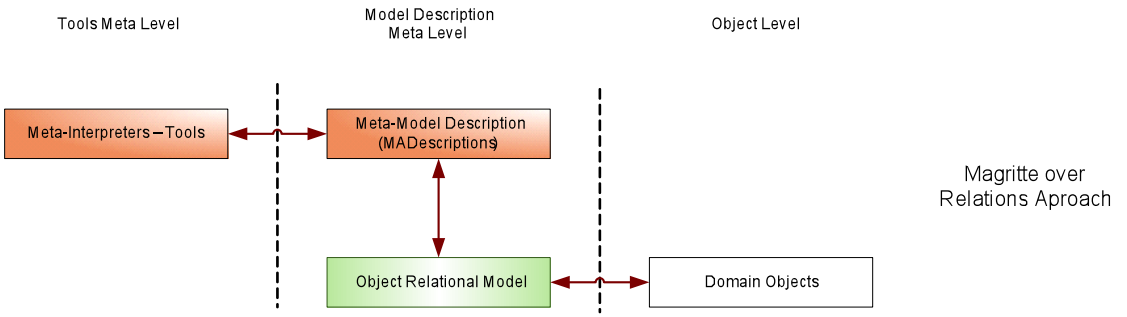


Figura 69. Magritte sobre Relaciones. Prueba de Concepto

Damos por supuesto que toda la información necesaria sobre la descripción del modelo se encuentra en el mismo modelo de relaciones.

Nuestra idea es que la integración permita derivar (inferir) las descripciones (**`MADescription`**) a partir de las perspectivas de participación de un objeto en una relación. Por ejemplo, derivar la descripción 'birthday' de una persona a partir de la relación *BirthdayOf entre Personas y Fechas.*

4.1.3 Adaptación e Integración

Magritte requiere que todos los objetos de negocio (Domain Objects) a ser descriptos provean el protocolo `description`. De manera predeterminada un objeto delega en su clase este mensaje.

```
Object >> description
    "Return the description of the reciever. Subclasses might override
    this message to return instance-based descriptions."

    ^ self class description
```

Pero en nuestro caso, tomamos la clase `RObject` y redefinimos el método `description` de la siguiente manera:

```
RObject >> description
    "Return the description of the reciever.
    Each instance has it's own description"

    ^ description ifNil: [self buildMADescription. ].
```

A modo de ejemplo, imaginemos definida la relación *FatherChild* y que el Magritte consulta a *juan* por su descripción. En este caso se daría una secuencia de colaboraciones similar a como muestra el diagrama de secuencia.

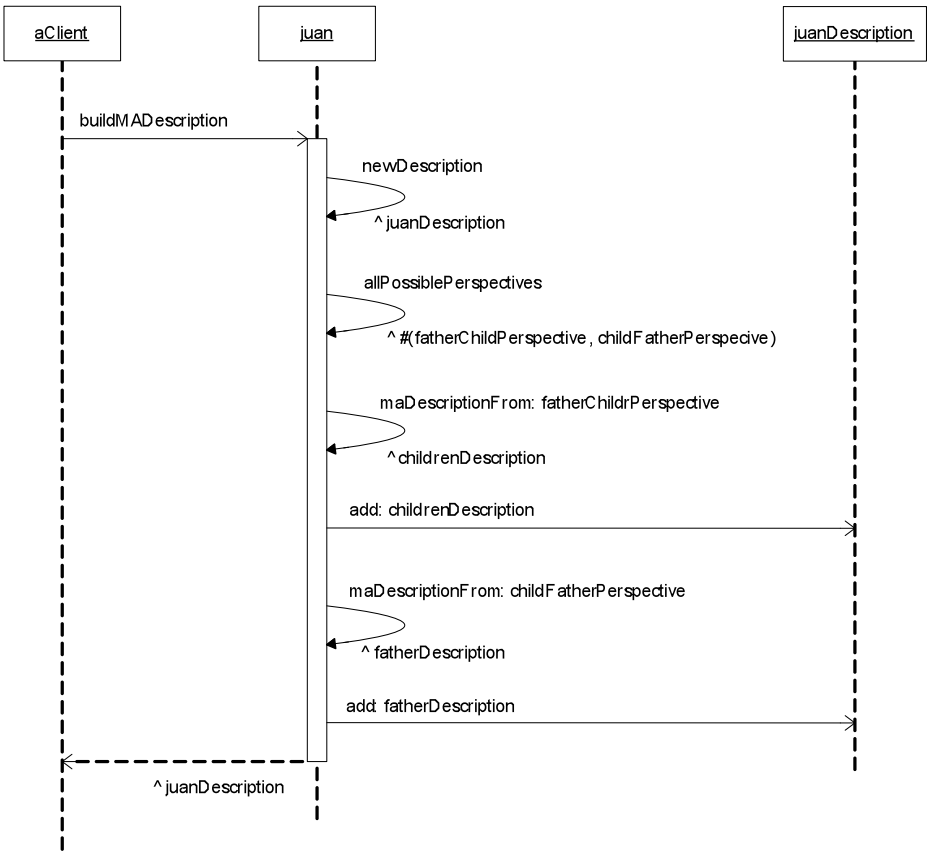


Figura 70. Magritte sobre Relaciones. Prueba de Concepto 2

Como puede observarse, para cada posible perspectiva en que el objeto pueda participar[26] se define su correspondiente `MADescription`.

El método `maDescriptionFrom:` es el responsable de realizar el mapeo de modelos de Magritte con el de Relaciones. Para graficar los aspectos a tener en cuenta durante este mapeo presentamos los casos:

Dada una perspectiva *p* tal que *p* `cannotBeRelatedToManyBs`[27]

Si…	**Creo una descripción instancia de…**
`p domainB kind = Duration`	`MADurantionDescription`
`p domainB kind = Date`	`MADateDescription`
`p domainB isEnumerable and: [ p isInmutable ]`	`MASingleOptionDescription`
`p domainB isEnumerable not or: [ p isInmutable not ]`	`RMAToOneRelationDescription`

Tabla 4. Adapación con Magritte. Especificación de Tipo de Descriptor.

Luego debe asignarse a la instancia de descripción obtenida el valor de sus atributos. Por ejemplo:

[26] Esto es, cada posible perspectiva dentro de cada posible relación definida en el ambiente.

[27] Esto significa que desde la perspectiva del objeto en la relación (rol A), la multiplicidad del otro rol (rol B) le indica que no podrá relacionarse muchas veces, es decir, podrá hacerlo 0 o 1 vez.

Asigno a…	El valor de…
`default`	`p defaultValueB`
`required`	`P hasAToBeRelatedWithAtLeastOneB`
`readOnly`	`p relation isReadOnly`
`visible`	`p isRoleBKnownByRoleA`

Tabla 5. Adaptación con Magritte. Especificación de MADescription.

Nota Para más información sobre cómo mapear Relaciones con Magritte, y conocer las particularidades de su integración ver el protocolo *Relations-Magritte-Example en la clase RObject, pudiéndose complementar con la sección "Trabajos relacionados > Magritte".

4.1.4 Resultado de la integración

En la sección anterior mostramos los principios de integración entre el Framework Magritte con el de Relaciones. Ahora mostraremos el resultado obtenido.

4.1.4.1 Primer paso: Universo de personas

Tomamos la clase `Person` la cual solo tiene definido una variable de instancia, name.

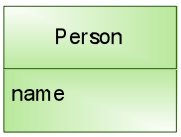


Figura 71. Magritte sobre Relaciones. Clase Person

Luego inicializamos el ambiente con las personas *juan*, *juan jr*, *juan jr jr*, *pablo*, *augusto*

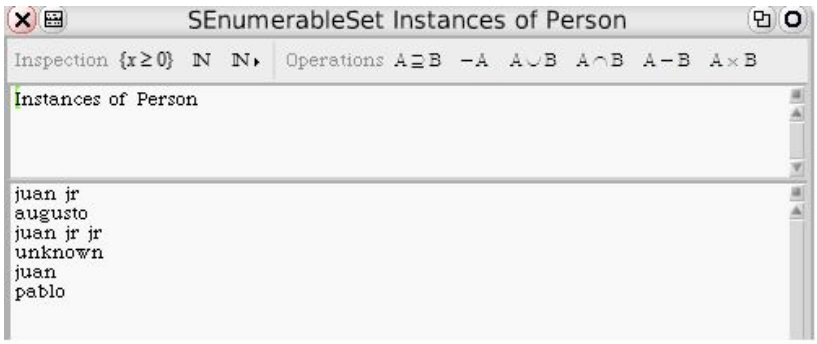


Figura 72. Magritte sobre Relaciones. Prueba de Concepto 3

A los efectos prácticos de esta prueba de concepto, creamos en la integración de Magritte para Seaside una aplicación que muestra un reporte de las personas definidas en el ambiente. El mismo esta definido en el path seaside/relations/persons.

De esta forma, si exploramos este path observamos el mismo listado de personas que las del conjunto anterior.

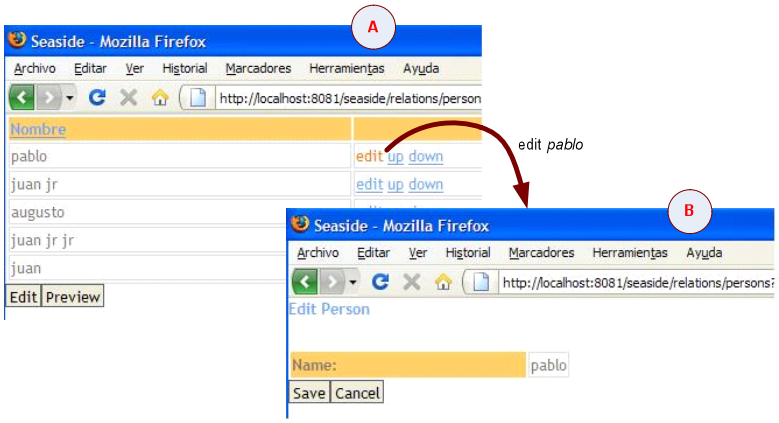


Figura 73. Magritte sobre Relaciones. Prueba de Concepto 4

Al hacer clic en "edit" observamos cómo en la ventana B, la descripción de `pablo` sólo nos indica que es persona y que posee un atributo `name` cuyo valor es el `String 'pablo'`.

4.1.4.2 Segundo paso: Relaciones *Birthday*, *Age* y *FatherChild*

Veamos qué ocurre cuando creamos una serie de relaciones que afectan a todas las personas y por ende a `pablo`. Entonces definimos las relaciones *Birthday*, *Age* y *FatherChild*

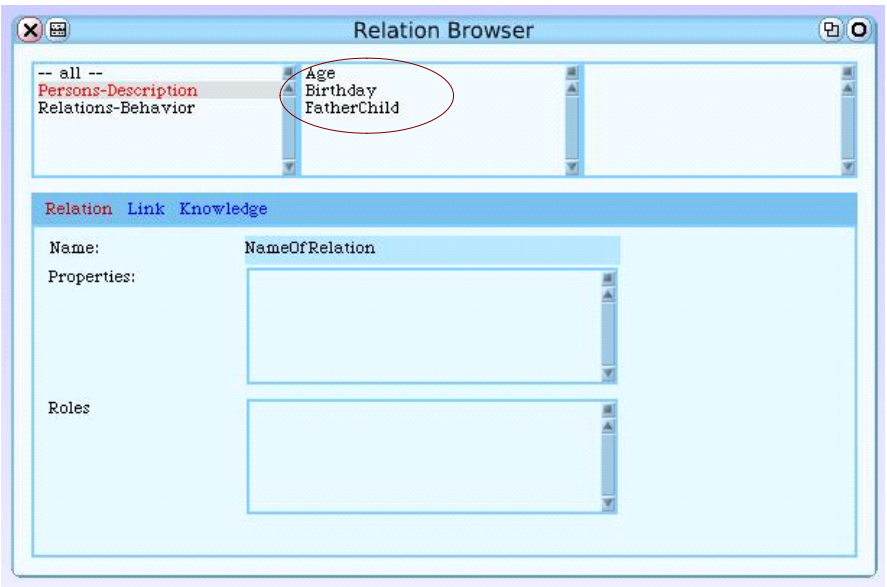


Figura 74. Magritte sobre Relaciones. Prueba de Concepto 5

Una vez definidas, volvemos al explorador de Internet y refrescamos la página donde estamos editando a `pablo` (F5). Al hacerlo obtenemos la siguiente vista:

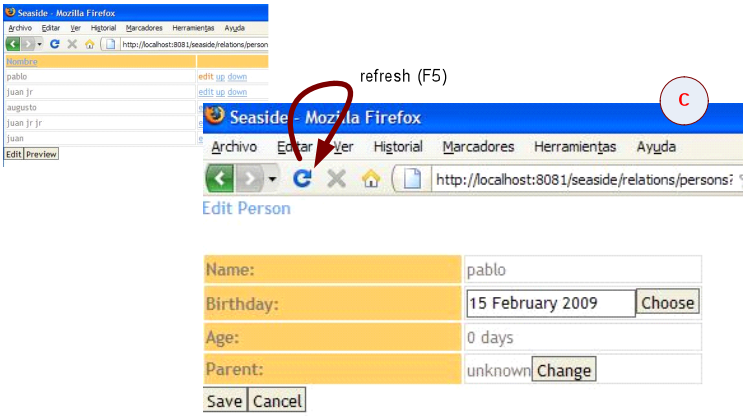


Figura 75. Magritte sobre Relaciones. Prueba de Concepto 6

Observamos que `pablo` acaba de tomar conciencia o bien aprendió sobre los conceptos

- fecha de nacimiento (birthday),
- sobre edad (age) y
- sobre que tiene un padre (aunque no sabe quien es aún).

Pero ¿y los hijos? Ocurre que debido a la definición de la relación *FatherChild*, una persona se encuentra en condiciones de ser padre cuando posee más de 12 años de edad y podemos observar que `pablo` aún no los tiene.

4.1.4.3 Tercer paso: Pablo nació el 4 de Diciembre de 1976

Actualicemos entonces su fecha de nacimiento.

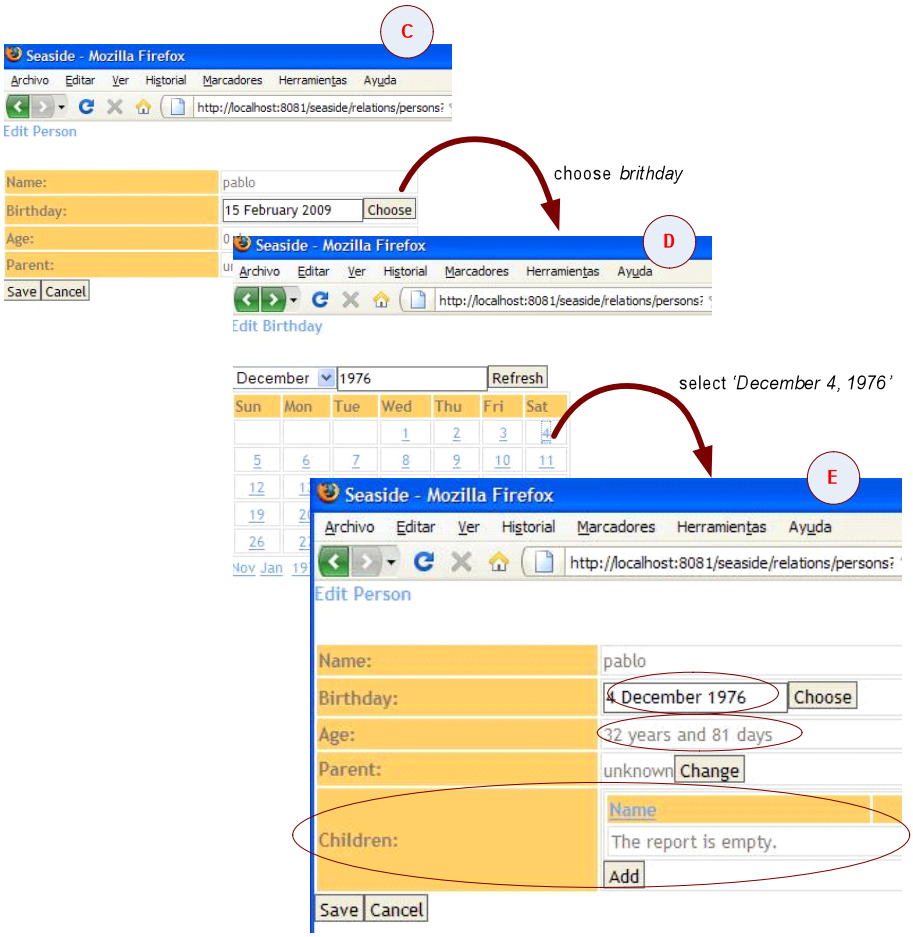


Figura 76. Magritte sobre Relaciones. Prueba de Concepto 7

Aquí es interesante detenerse a analizar los diferentes sucesos que se dieron lugar.

En primer lugar, al optar por cambiar la fecha de nacimiento (C), se desplegó un Editor de Fechas. Esto se debe a que el framework brinda a Magritte la información suficiente para que éste interprete y conozca el tipo del objeto a editar a los efectos de utilizar el Editor que corresponda.

En segundo lugar (D), al seleccionar el `4 de Diciembre de 1976` como fecha de nacimiento de `pablo` el framework nos retorna nuevamente a su editor (E). Pero como podemos apreciar en la figura, existen algunos cambios con respecto al estado de dicho editor en (C).

Veamos qué es lo que ha ocurrido. Como era de esperar, la fecha de nacimiento ha cambiado: `4 de Diciembre de 1976`, pero al hacerlo, también se ha actualizado -en forma transparente para nosotros- la edad de `pablo`, que ahora es de `32 años`. Esto se debe a que la relación *Age* es una relación derivada de la relación *Birthday*. Por último también observamos en D otro cambio. Al pasar a tener más de 12 años, `pablo` es ahora un elemento del dominio de Father en la relación *FatherChild*, y en consecuencia puede tener hijos.

4.1.4.4 Cuarto paso: *Pablo* es padre de *Augusto*

Ahora relacionemos a `pablo` con sus hijos.

Para agregar un hijo, hacemos clic sobre “Add”, lo cual despliega el reporte de todas las personas “candidatas” a ser hijos de `pablo`. Esta es una característica que no debemos pasar por alto. Los

candidatos que se presentan son aquellas personas en condiciones de ser hijo, es decir, las personas que forman parte del dominio de Hijo en la relación *FatherChild*[28].

En F, seleccionamos a `augusto` y observamos el cambio en la nómina de hijos de `pablo` en G.

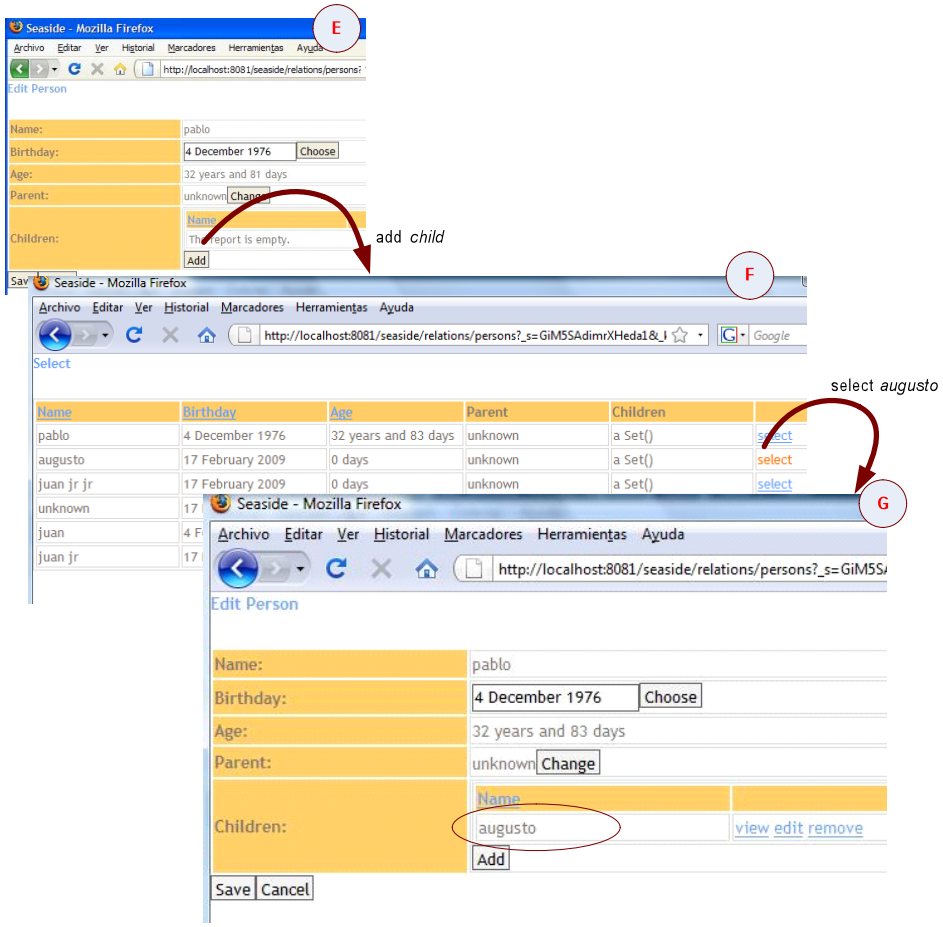


Figura 77. Magritte sobre Relaciones. Prueba de Concepto 8

4.1.4.5 Cuarto paso: *Pablo* no puede ser hijo de *Juan*

Ahora es el turno de seleccionar el padre de `pablo`.

En primer lugar observamos que la Herramienta utiliza el concepto de multiplicidad y cardinalidad para determinar qué tipo de widget utilizar. Así, a diferencia de Children donde es posible agregar varias personas, en Father solo podemos indicar una y sólo una. Luego de esta aclaración podemos proceder haciendo clic en "change" lo que despliega un reporte con todas las personas candidatas a ser padre.

Podemos observar que solo hay 2, ya que son las únicas mayores a 12 años. Esto nuevamente se debe a la noción "candidatos" que el modelo de relaciones ofrece a Magritte en función de los integrantes del dominio del rol Father.

Al seleccionar a `juan` se produce un error.

[28] Este algoritmo podría ser más sofisticado, por ejemplo al evitar listar las personas ya relacionadas como hijo con `pablo`

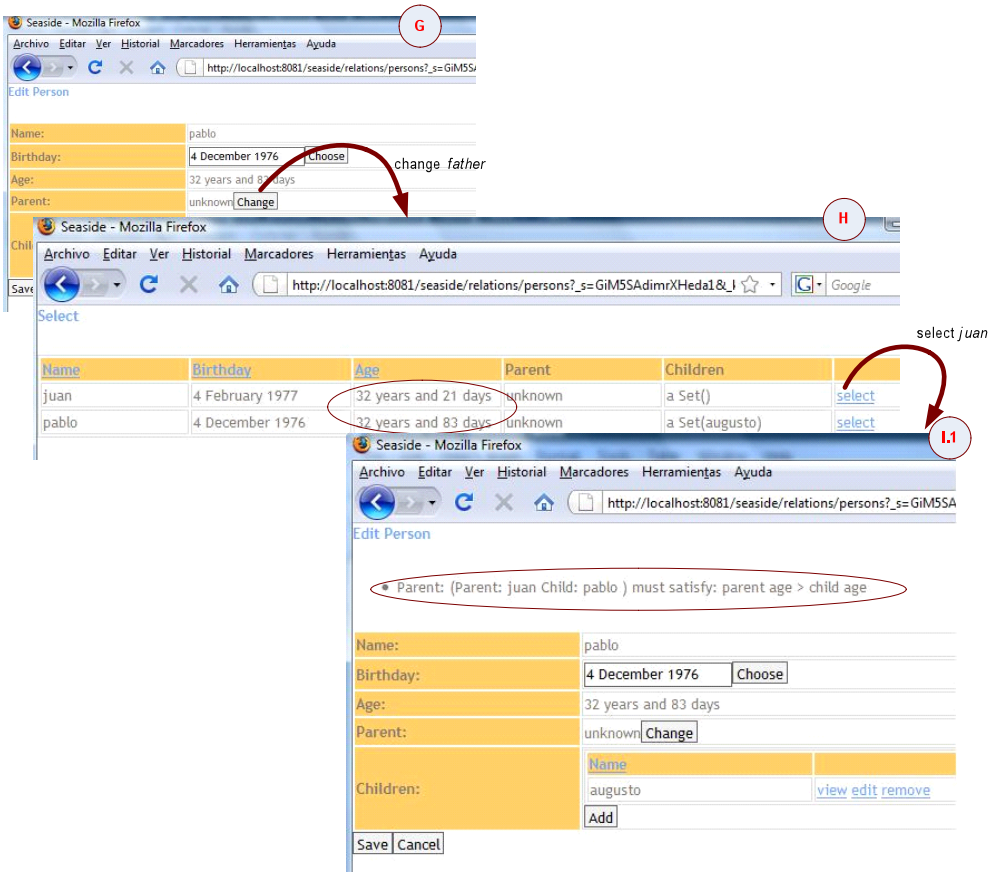


Figura 78. Magritte sobre Relaciones. Prueba de Concepto 9

El error se produce al no cumplirse con el invariante de "parent age > child age", es decir, la edad del padre es mayor que la del hijo. Este error fue detectado en la relación *FatherChild* mientras se intentaba agregar el link `(juan as Father, pablo as Child)` y atrapado por el editor en Magritte.

4.1.4.6 Quinto paso: *Pablo* no puede ser padre de sí mismo

En forma similar repetimos el proceso, y adrede (ya que no tenemos otra alternativa) hacemos la prueba de seleccionar a `pablo` como su propio padre.

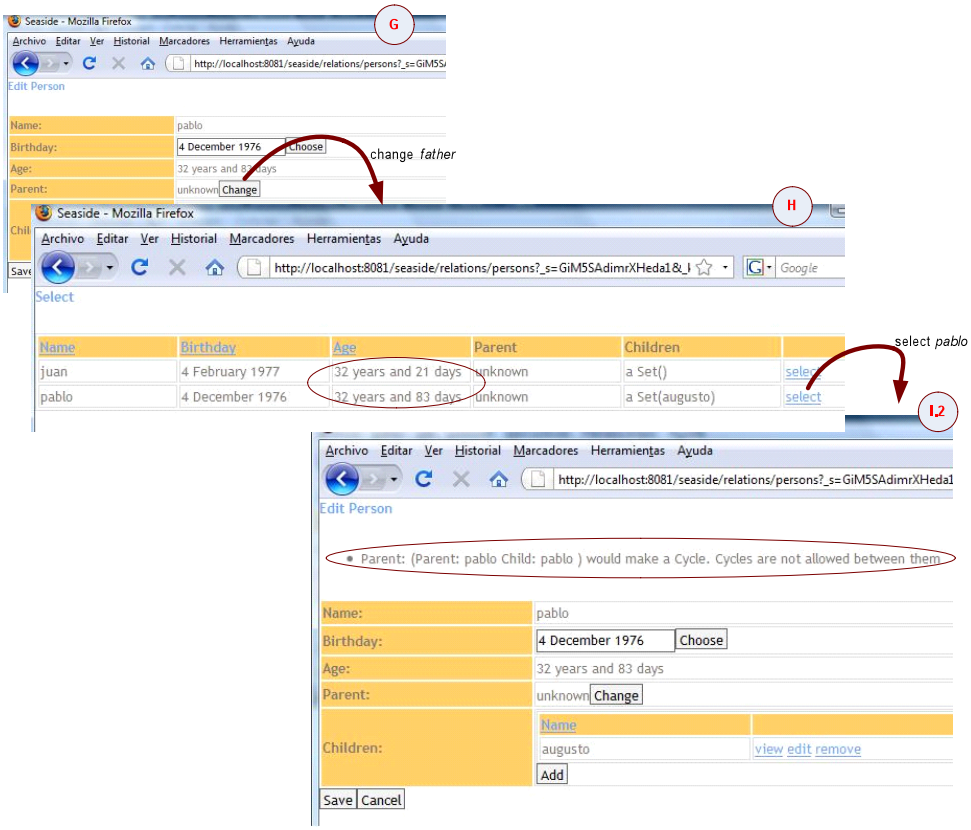


Figura 79. Magritte sobre Relaciones. Prueba de Concepto 10

Nuevamente volvemos a obtener un error detectado por la relación *FatherChild*, pero en este caso la causa es producto de que no se cumple la restricción "Sin ciclos" en el grafo de conocimiento resultante.

4.2 ¿Categorías o etiquetas?

Ocurre frecuentemente que al analizar el dominio para el desarrollo de un software específico, nos encontramos con que una porción de ese dominio es el de permitir categorizar elementos.

Así es que entonces, en nuestra primera reunión de relevamiento, nuestro cliente nos comenta que toda tarea pueda ser catalogada: "Quiero que toda tarea se catalogue con una categoría".

Imaginamos pues que todo elemento tiene que pertenecer a una y solo una categoría. Con esto, desarrollamos un prototipo, para el que utilizamos la siguiente relación:

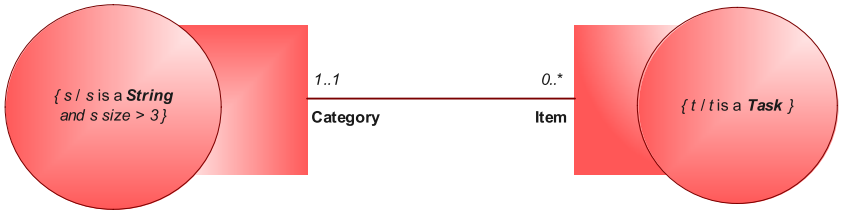


Figura 80. Categorization. Prueba de Concepto

Cuando se lo presentamos a nuestro cliente, este expresa: “Ummm, ok, pero a mí me gustaría que exista una jerarquía de categorías y que sólo las hojas puedan categorizar elementos”.

Para ofrecer esta solución primero necesitamos agregar una nueva relación, que jerarquice las categorías. Definimos entonces la relación *SubCategoryOf*. Luego tenemos que expresar que una categoría sólo se encuentra en condiciones de categorizar cuando es hoja de una jerarquía. Para ello modificamos el dominio del rol Category en la relación *Categorization*.

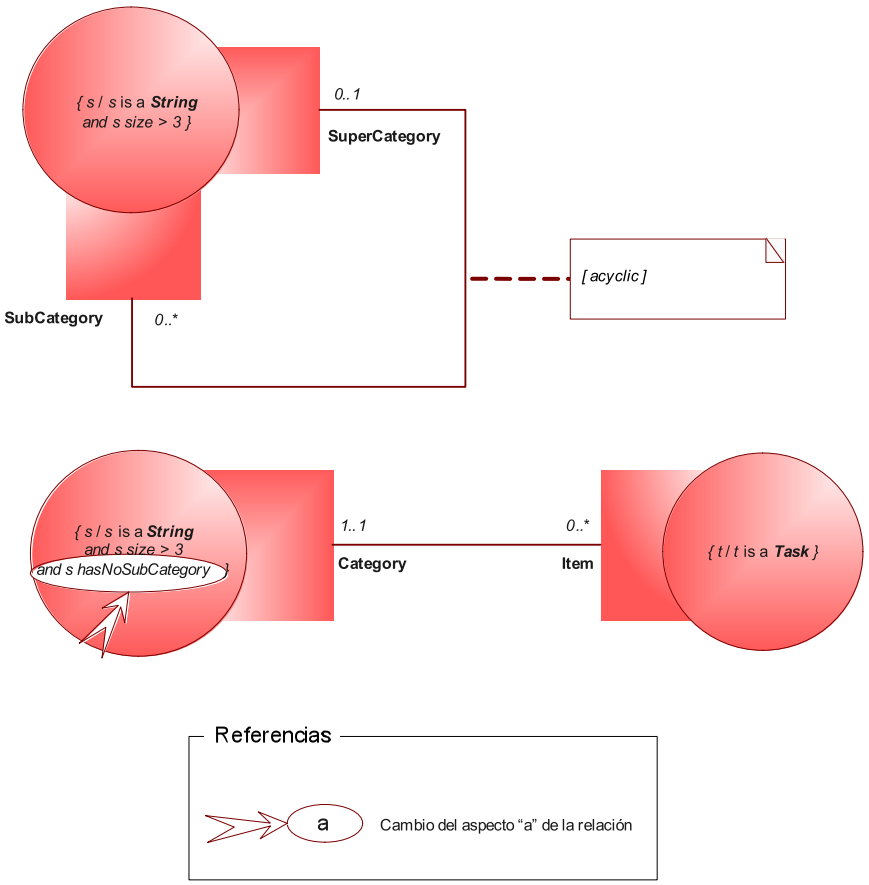


Figura 81. Categorization. Prueba de Concepto 2

Conformes con nuestra solución volvemos a presentarlo, pero nuevamente nuestro cliente exige otro cambio: “Sí, yo les había dicho que sólo las hojas pudieran categorizar, pero ahora que lo veo pienso que un elemento podría se categorizado por cualquier nivel de la jerarquía”.

Entonces, volvemos para atrás el cambio que habíamos realizado en la definición del dominio de Category. Con esto logramos su OK final para desarrollar el producto de software.

Pero una vez implementado y operativo comenzamos a recibir quejas de los usuarios del sistema porque no pueden categorizar un elemento con más de una categoría. Como nos encontramos en la etapa de soporte y mantenimiento postventa, nuestro cliente nos pide que realicemos esta adaptación.

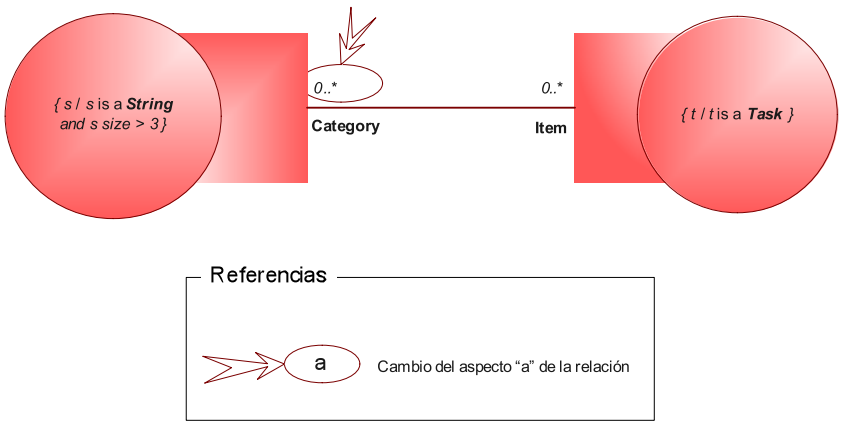


Figura 82. Categorization. Prueba de Concepto 3

Como puede observarse en la figura anterior, nos basta con modificar la multiplicidad de la relación para ajustar nuestro modelo a los nuevos requerimientos del usuario.

> **Nota** Es importante aclarar que al cierre de este trabajo no contamos con las herramientas de refactoring necesarias para modificar el modelo vivo, es decir, de un programa en ejecución. Sin embargo, consideramos que a los efectos de este trabajo alcanza con saber que existirán estas herramientas para aceptar las alternativas de solución propuestas en este ejemplo.

4.3 State Pattern con Relaciones

Tal vez uno de los patterns más utilizados es el State[Alpert et al. 1998]. La aplicación de este pattern trae ventajas al modelar el comportamiento de un objeto cuando éste depende de su estado interno. Sin embargo, presenta algunos inconvenientes propios de modelar delegación en un ambiente de clases.

Este pattern modela como una jerarquía de Clases los posibles estados de un objeto. En realidad lo que implementa es el comportamiento que deberá tener un objeto cuando se encuentre en dicho estado. En un ambiente de clases tradicional el programador se encuentra obligado a implementarlos como Clases.

A su vez, aunque propone algunas alternativas y facilita la toma de decisiones, no especifica una única forma respecto a cómo y/o quién es el responsable de ejecutar las transiciones entre estados.

En la figura siguiente se presenta un diagrama de clases donde se refleja la aplicación del pattern State para modelar el cambio de comportamiento de una conexión TCP[Alpert et al. 1998]

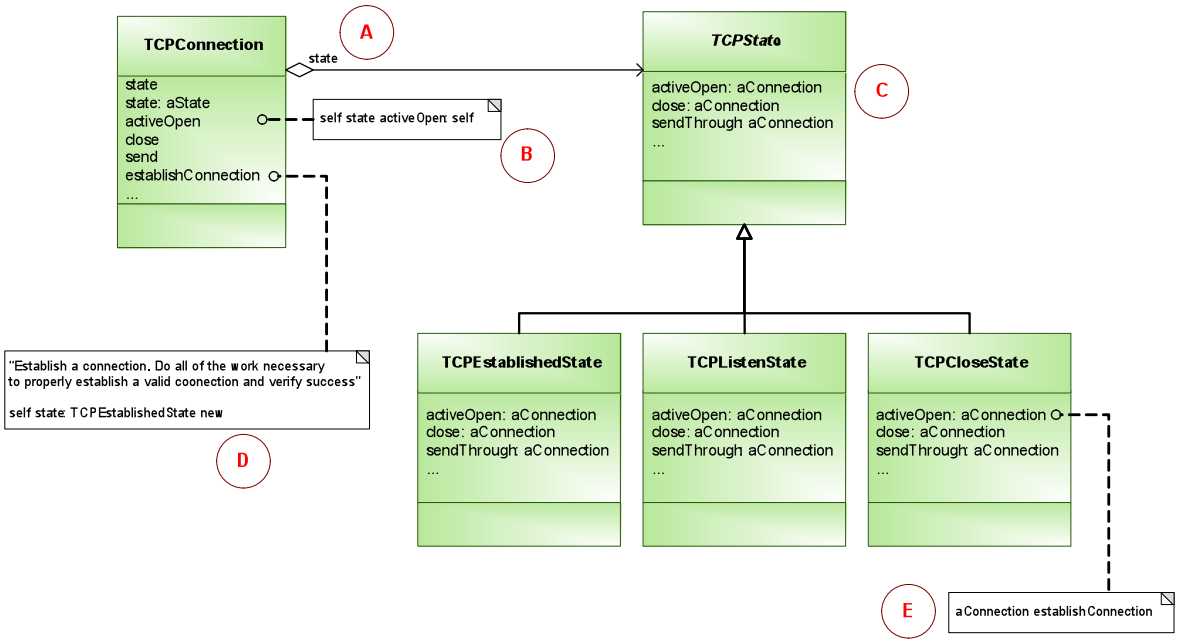


Figura 83. Pattern State. Ejemplo TCPConnection

En el diagrama de clases anterior podemos observar diferentes decisiones de diseño. En A, se desprende que existe una relación que podríamos llamar *StateOf*, aunque más tarde ésta se diluya en el código a implementar. En el extracto de código B, se observa la solución para "delegar" al estado tanto el método como el contexto (es necesario que la `TCPConnection` se pase como colaborador externo, así el estado puede operar sobre ésta[29]). Siguiendo hacia la derecha, en C vemos cómo un contexto (en este caso `TCPConnection`) delega cierta responsabilidad en su estado (`TCPState`). En la implementación del método `establishConnection` (D), vemos que es necesario que al menos exista una instancia de cada uno de dichos estados[30]. Finalmente, en E, observamos cómo una `TCPState` opera haciendo uso de los servicios que exporta el contexto.

Aun cuando el pattern brinda el marco general y acotado, todas estas alternativas y decisiones deben ser adoptadas por el programador.

En lo que resta de esta prueba de concepto, presentaremos una solución alternativa al pattern State utilizando relaciones.

4.3.1 Descripción de la solución alternativa

4.3.1.1 Primer paso: Modelar los comportamientos

Como hemos explicado al describir el Modelo de Comportamiento, no todo debe ser modelado como una clase.

En función de ello, en primer lugar modelamos una única clase -la inherente al dominio de problema- **`TCPConnection`**.

[29] Otra posiblidad analizada es que la instancia de TCPState sea dedicada a una conexión, en cuyo caso podría tener un colaborar interto llamado connection or context.

[30] Podrían implementarse como Singleton.

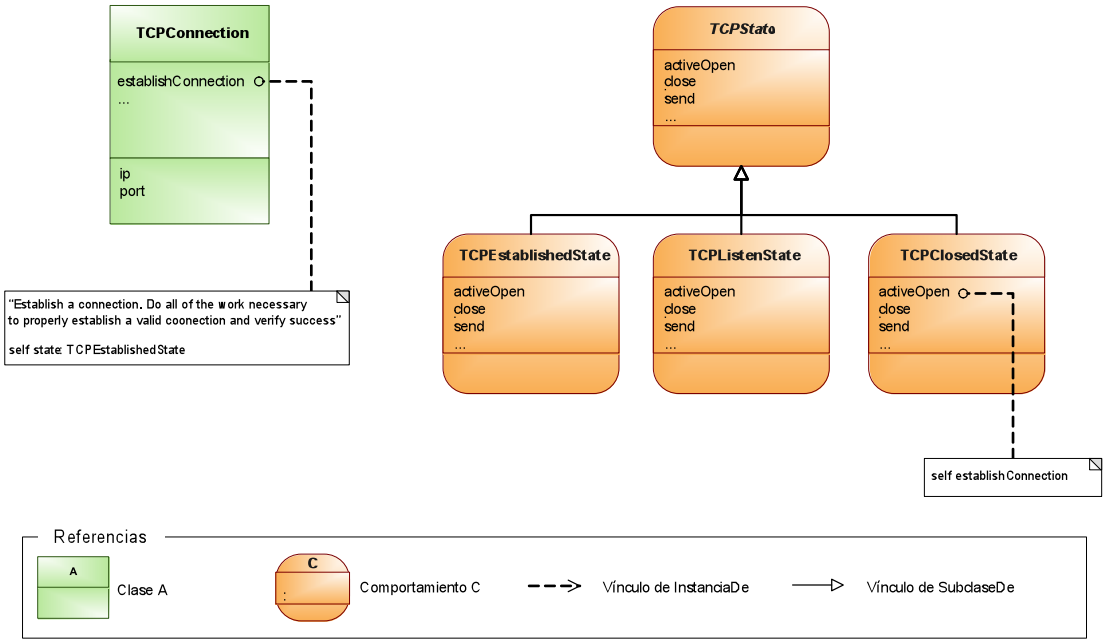


Figura 84. Patten State con Relaciones. Prueba de Concepto

Tabla 6. Ejemplo TCPConnection con Relaciones.

Luego, confeccionamos una jerarquía de estados, reflejando el comportamiento que tendrá un **`TCPConnection`** de encontrarse en alguno de ellos. Estos Comportamientos no son clases, y por lo tanto no son instanciables.

Podemos observar que en el modelo propuesto ya no contamos con los puntos de decisión B, C, D, E debido a que representamos cada uno de los estados como si correspondiera al comportamiento de una `TCPConnection`. No hay contexto que pasar de un lado a otro, no hay colaboradores externos en los métodos, y no hay que instanciar ningún estado.

Notemos además que ya no es necesario implementar los métodos `activeOpen`, `close`, `send`, `state`, `state:` en `TCPConnection`. Los primeros 3 debido a que estos serán adquiridos del estado en que se encuentre. Los últimos 2 porque es responsabilidad del comportamiento de Rol, el que será automáticamente generado por el framework de comportamiento.

4.3.1.2 Segundo paso: Reificar la relación *TCPStateOf*

Al tener implementado estos comportamientos estamos en condiciones de reificar la relación *TCPStateOf*. Al hacerlo, comienzan a aparecer ciertos resultados dignos de mencionar.

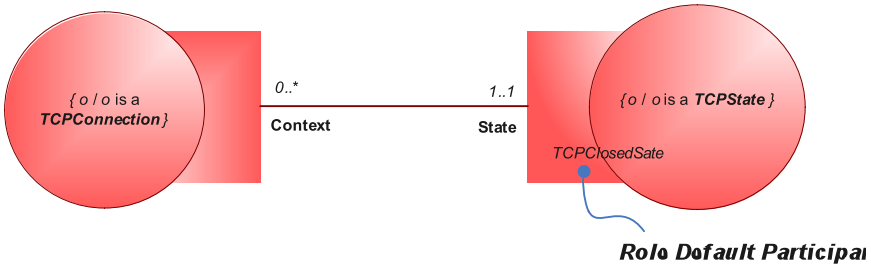


Figura 85. TCPStateOf. Diagrama de Relación

Como se muestra en la figura anterior, al describir la relación *TCPStateOf* hemos tenido que definir a `TCPClosedState` como el valor predeterminado (default) para el rol TCPState.

```
| rel |

rel := (RRelation new)
     name: 'TCPStateOf';

     "Roles"
     withRoleNamed: 'Context';
     withRoleNamed: 'State';

     "Dominios de participantes"
     all: TCPConnection asDomain
            canParticipateAs: 'Context';
     all: TCPState asSubclassesDomain
            canParticipateAs: 'State';


     "Multiplicidad"
     a: 'Context'
            mustBeRelatedAtLeast: One time
            atMost: One time
            withOthersParticipatingAs: 'State';

     a: 'Context'
            isRelatedByDefaultWith: TCPCloseState
            participatingAs: 'State';


     "Comportamiento"
     a: 'Context'
            extendsBehaviorWithParticipant: 'State';

     yourself.
```

Al disponer de esta relación, evitamos el punto de decisión A, sobre cómo implementar en forma ad-hoc esta relación.

4.3.1.3 Tercer paso: Instanciación de *TCPConnection*

A modo de ejemplo, imaginemos el siguiente escenario

```
conn := TCPConnection ip: '192.168.1.103' port: 8087.
```

Cuando creamos la instancia de `TCPConnection` *conn*, esta es detectada la relación `TCPStateOf` e inmediatamente relacionada al comportamiento de rol `Context` a través de la relación *CouldParticipateAs*.

A continuación la relación *TCPStateOf* determina que toda conexión debe estar vinculada a un estado, y como el estado predeterminado es `TCPClosedState`, lo vincula a este.

En el gráfico a continuación se muestran los vínculos de las relaciones *TCPStateOf*, *InstanceOf* y *CouldParticipateAs* que afectan al objeto *conn*.

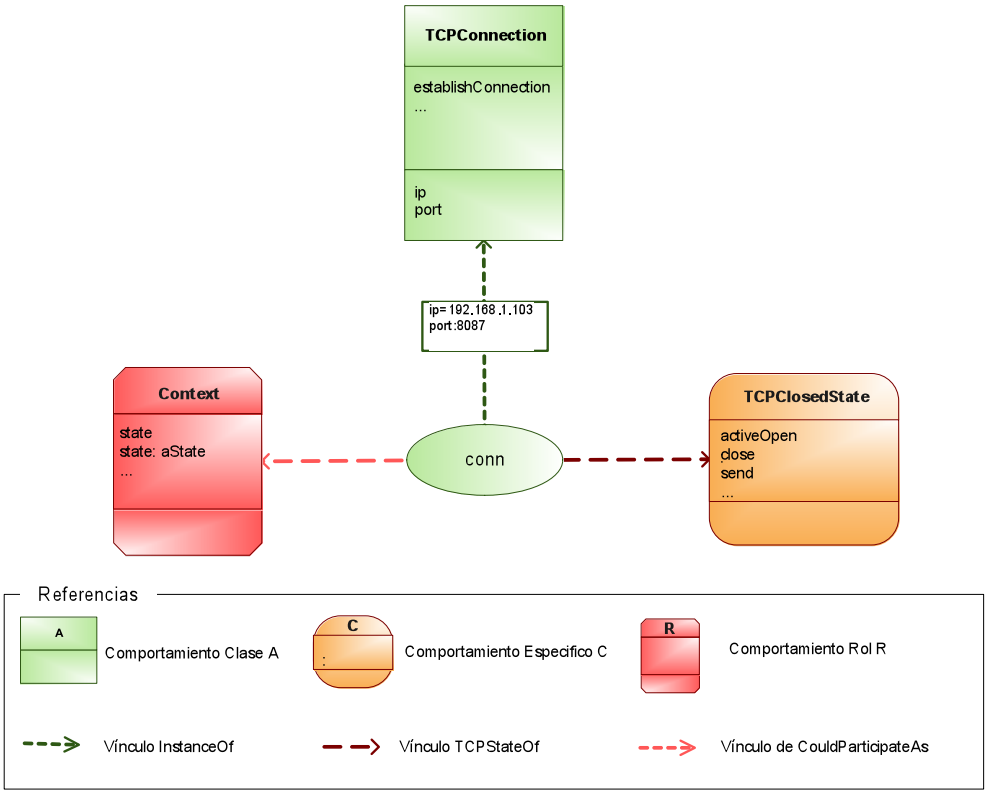


Figura 86. conn. Vínculos de Comportamiento.

Recordemos que por sus características, estas tres relaciones son aquellas que llamamos relaciones de comportamiento. Por tal motivo, serán consideradas cuando la relación *BehaveAs* calcule el conjunto de comportamientos del objeto `conn`.

De esta manera, el comportamiento de `conn`, desde la visión de *BehaveAs* será como se muestra a continuación:

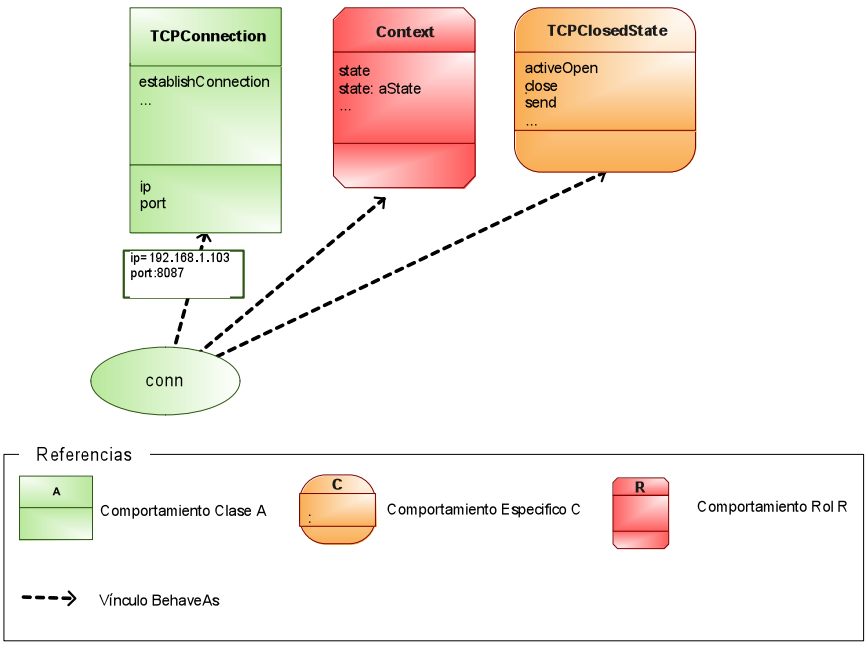


Figura 87. conn. Vínculos de BehaveAs.

4.3.1.4 Cuatro paso: Establecimiento y cierre de conexión

En A vemos cómo al recibir el mensaje `activeOpen`, el MethodLookup busca y ejecuta el método del comportamiento `TCPCloseState` –actual estado de *conn*- en el contexto de *conn*, es decir, bindeando la variable *self* a *conn*.

Luego dentro de las colaboraciones de este método se envía el mensaje `establishConnection` a *self*. El ML encuentra este método en la clase de *conn* y lo ejecuta.

Durante la ejecución de `establishConnection`, se envía el mensaje `state:` a *self*. El ML no encuentra este método en la clase del conn (ni en ninguna de sus superclases), pero sí lo encuentra en el comportamiento de rol Context. Luego ejecuta este método bindeando *self* a *conn*.

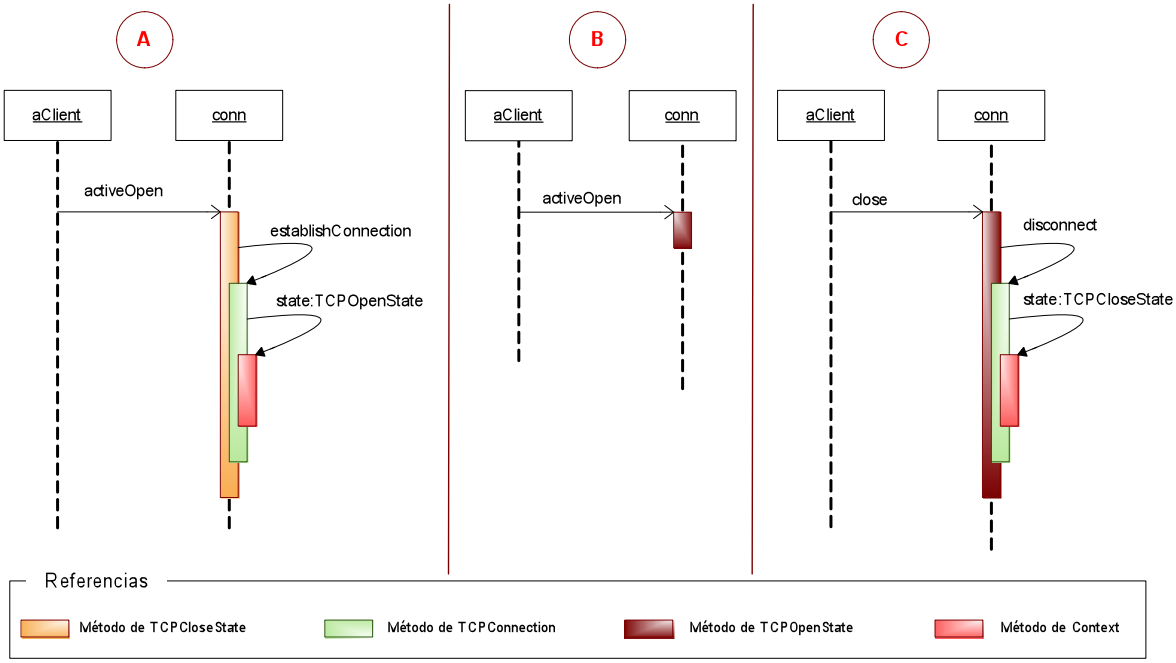


Figura 88. conn activeOpen. Diagrama de Secuencia

En B, notamos cómo al recibir el mensaje `activeOpen`, el MethodLookup busca y ejecuta el método del comportamiento `TCPOpenState` –actual estado de *conn*- en el contexto de *conn*, es decir, bindeando la variable *self* a *conn*.

Por último, en C ocurre una situación similar a como se presenta en el A, pero esta vez para cerrar la conexión.

4.3.1.5 Ventajas

Esta alternativa de solución ofrece un conjunto de beneficios con respecto a las implementaciones que describe el pattern. En general porque abstrae muchas de las decisiones que debe tomar un programador. Por ejemplo, el desarrollador ya no debe preocuparse por

- Instanciar un estado
- Determinar el modo en que debe instanciar un estado (Singleton, cada vez que lo necesito)
- Resolver si el contexto es dueño del estado, o el estado es global para todos los contextos
- Definir si pasa como colaborador externo al contexto, o si define una variable de instancia en el estado que referencie al contexto

Sin embargo, el programador aún deberá definir si la transición entre estados la lleva adelante el Context, si la distribuye entre los estados o, si debido a su complejidad, la delega a un tercer objeto.

Pero es aquí donde esta solución ofrece una nueva ventaja. En el caso que decida delegar esta responsabilidad a un tercer objeto, es muy probable que no tenga que preocuparse por quién será este objeto y tampoco como implementarlo, ya que la misma relación se encuentra en un lugar privilegiado para realizar esta tarea.

4.3.2 Implementación de la solución alternativa

4.3.2.1 Primer paso: Modelar los comportamientos

En el browser A se observa la implementación de la clase `TCPConnection`. Las implementaciones de los estados se muestran en los browsers B y C.

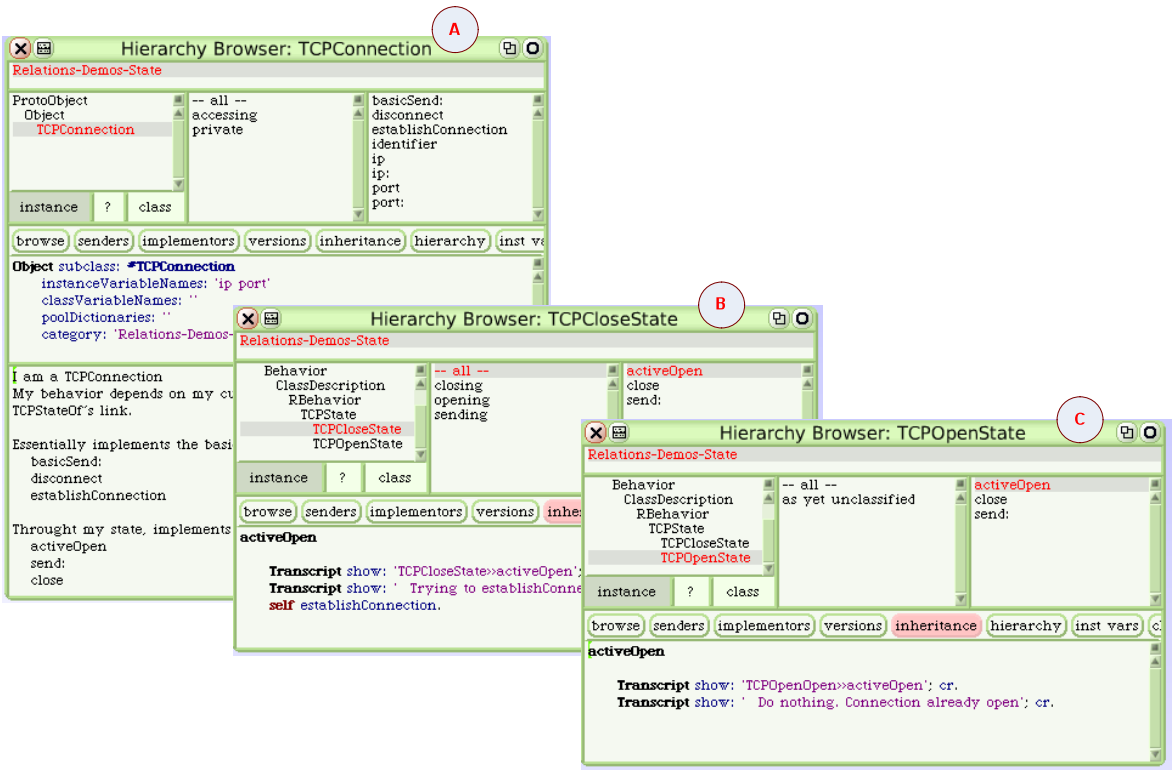


Figura 89. Pattern State sobre Relaciones. Prueba de Concepto

Como ya dijimos, los comportamientos de los estados no son implementados como clases, sino como meros comportamientos.

Es importante notar que en el código fuente de `TCPCloseState>>activeOpen` (B) se especifica la colaboración `self establishConnection` a pesar de que este método no se encuentra implementado para `TCPCloseState` o sus súper comportamientos. Aunque a priori parezca una incongruencia, podemos hacerlo ya que estos comportamientos formarán parte del comportamiento una `TCPConnection`, la cual sí dispondrá de esta implementación.

4.3.2.2 Segundo paso: Definir la relación *TCPStateOf*

Ya hemos desarrollado tanto la clase como los comportamientos en función de los posibles estados en que puede encontrarse una `TCPConnection`. Es el turno ahora de definir la relación *TCPStateOf*.

Describimos entonces a esta relación con dos roles, Context y State, tal como se muestran en los browsers de relaciones D y E.

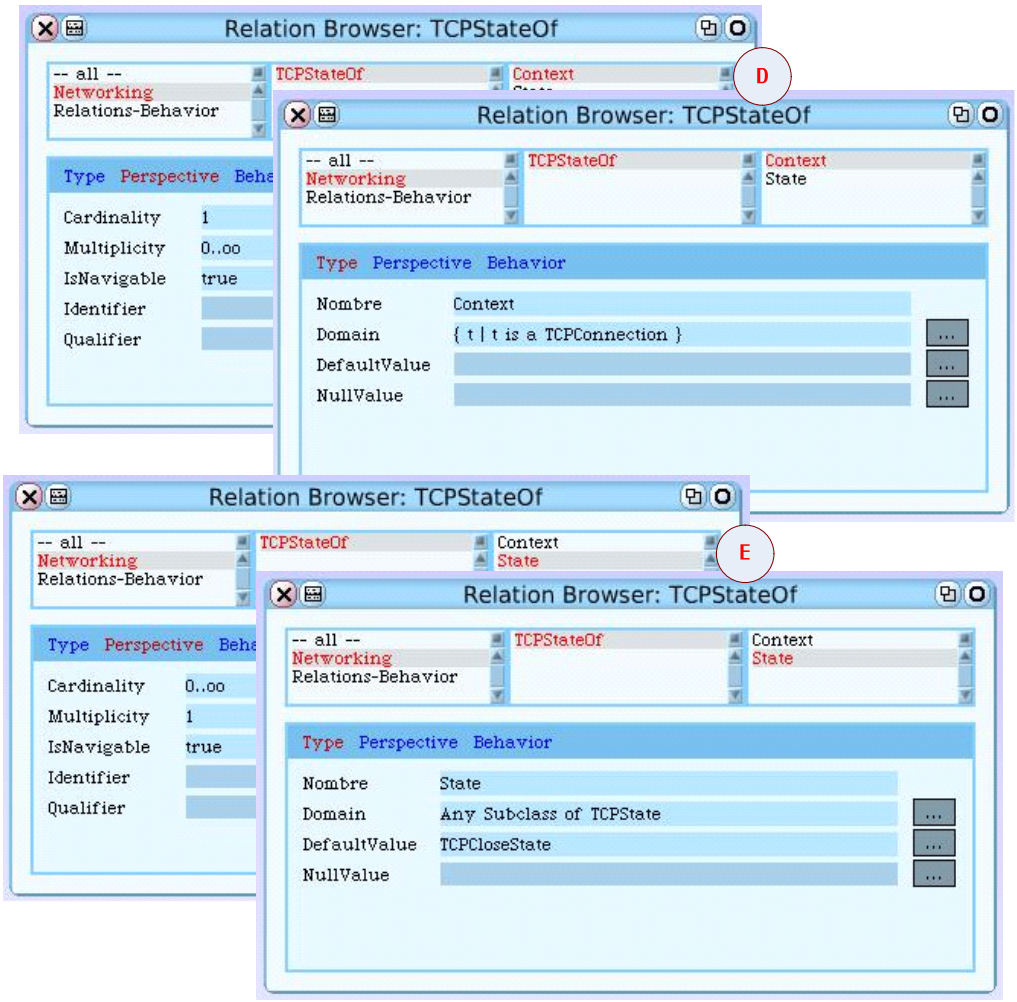


Figura 90. Pattern State sobre Relaciones. Prueba de Concepto 2

Además de las definiciones de cardinalidad y multiplicidad, resulta de interés detenerse en la especificación de dominio para uno de estos roles. En el browser D observamos que podrán participar como Context todos los objetos *t* que sean del tipo de `TCPConnection`. Similarmente, en E notamos que podrá participar como State cualquier objeto subclase (subBehavior) TCPState.

4.3.2.3 Tercer paso: Instanciar una nueva conexión

Ya hemos definido la relación. Creemos ahora una nueva conexión e inspeccionémosla con el inspector de relaciones.

```
conn := TCPConnection ip: '192.168.1.103' port: 8087.
conn rInspect.
```

Al abrir el inspector observamos (en F) las variables de instancia de *conn*. Luego, al inspeccionar el comportamiento *conn* vemos que éste compuesto por `TCPConnection` -y el de todas sus superclases- (Figura G).

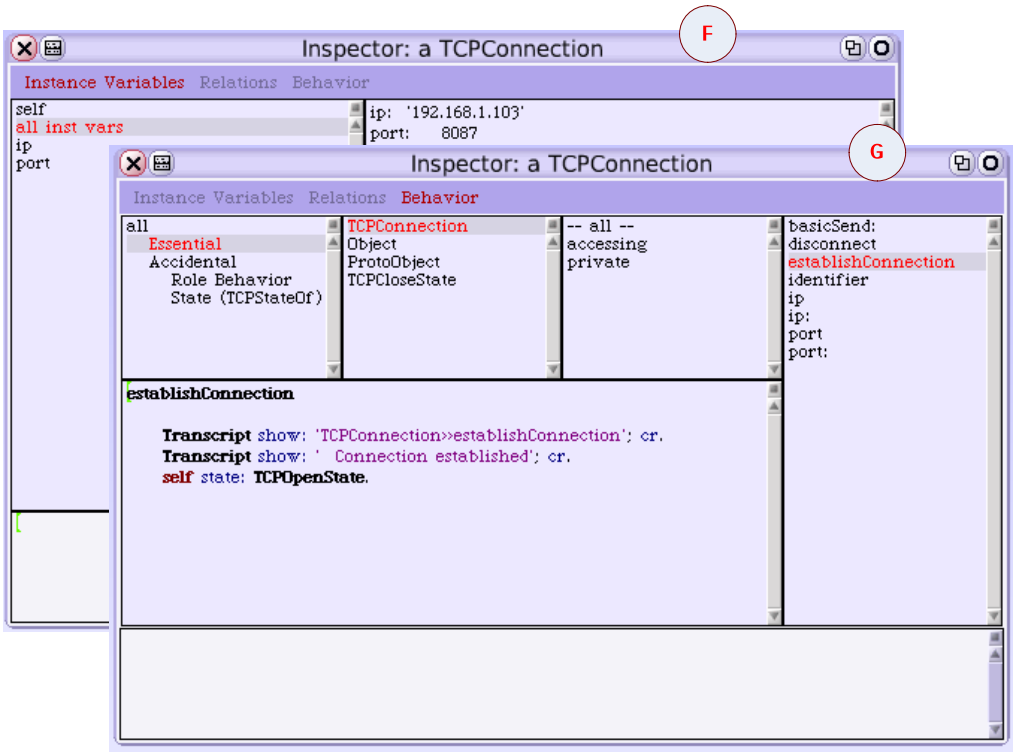


Figura 91. Pattern State sobre Relaciones. Prueba de Concepto 3

Pero, a su vez, también dispone de una serie de comportamientos accidentales. En el inspector H podemos observar el comportamiento de Rol Context, que `conn` adquiere por encontrarse en el dominio del Context de la relación *TCPStateOf*.

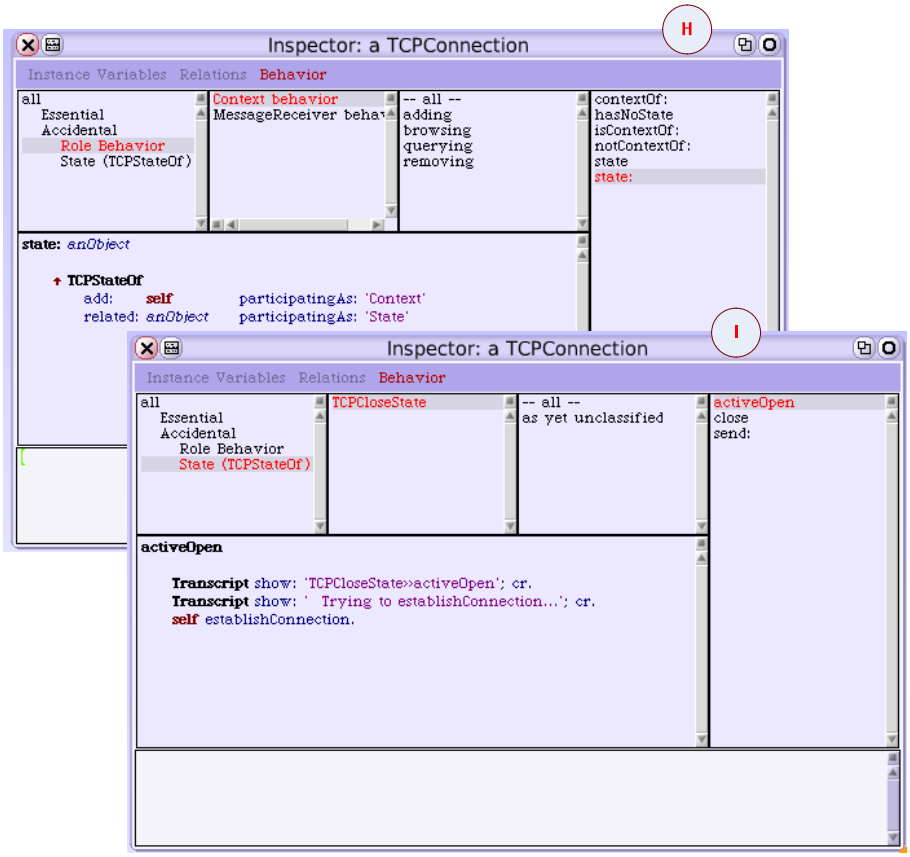


Figura 92. Pattern State sobre Relaciones. Prueba de Concepto

Pero también vemos -en el inspector I- que `conn` adquirió el comportamiento de `TCPCloseState`. Esto se debe a que en la relación *TCPStateOf* definimos a éste como el valor predeterminado para el Rol State. Luego, el Framework vinculó en forma automática a `conn` con éste `TCPCloseState`[31]. Más tarde, la relación *BehaveAs* infirió a partir de este vínculo que `TCPCloseState` es parte del comportamiento de `conn`.

4.3.2.4 Cuarto paso: Establecimiento y cierre de la conexión

Ya hemos observado e inspeccionado el estado y comportamientos de `conn`. Solo nos resta abrir la conexión.

```
Transcript
     show: 'Caso A. Abrir conexión';
     cr;
     cr.
conn activeOpen.
```

Al hacerlo, podemos observar (en el inspector J) cómo el comportamiento relativo al estado cambió de `TCPCloseState` a `TCPOpenState`. Esto se produjo porque durante la ejecución del método `TCPConnection>>establishConnection` se vinculó a `conn` con su nuevo estado. Automáticamente la relación *BehaveAs* detectó el cambio y actualizó el comportamiento de `conn` en función de ello.

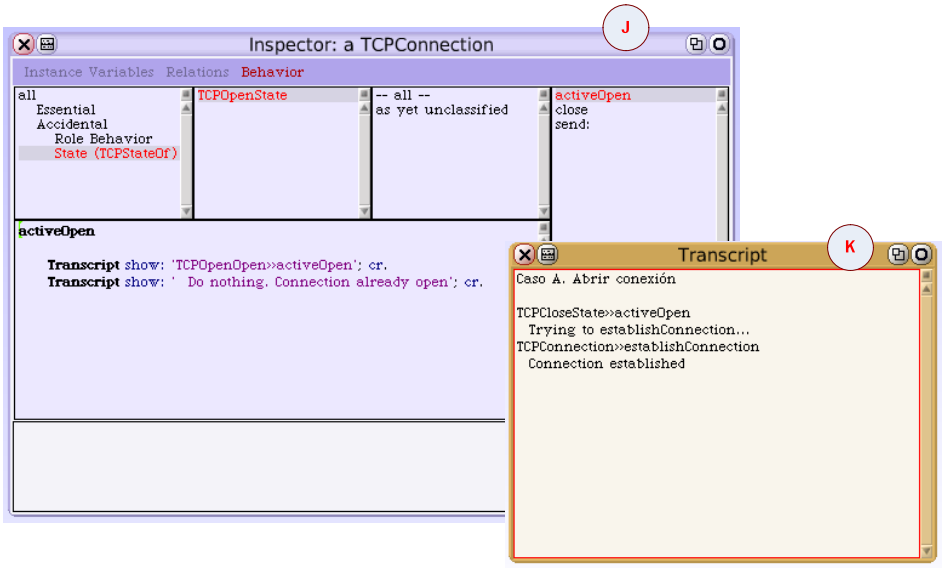


Figura 93. Pattern State sobre Relaciones. Prueba de Concepto

En el `Transcript` K podemos observar un log de las invocaciones realizadas.

Para verificar el correcto funcionamiento de la solución del pattern, intentamos abrir nuevamente la conexión.

[31] Para satisfacer las restricciones de cardinalidad y multiplicidad que indican que todo context debe participar siempre, una vez y vinculado a un context no nulo.

```
Transcript
    cr;
    show: 'Caso B. Abrir conexión (pero conexión ya abierta)';
    cr;
    cr.
conn activeOpen.
```

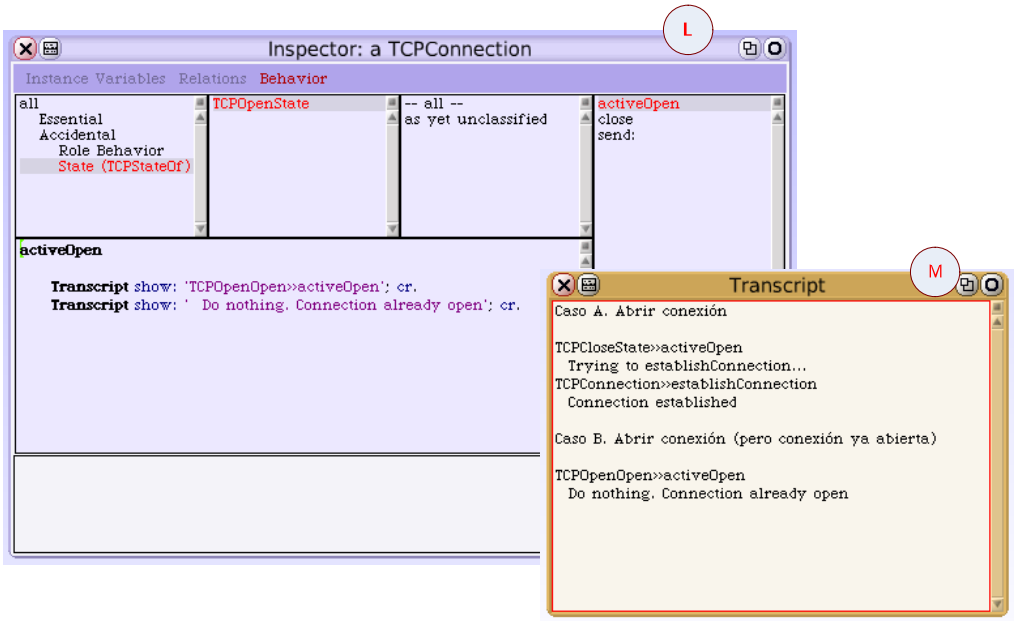


Figura 94. Pattern State sobre Relaciones. Prueba de Concepto

Podemos observar en el Transcript M que sólo se invoca al método `TCPOpenState>>activeOpen`. Esto prueba que efectivamente cambió el comportamiento de *conn* luego del establecimiento de la conexión.

Por último, cerramos la conexión.

```
Transcript
    cr;
    show: 'Caso C. Cerrar conexión';
    cr;
    cr.
conn close.
```

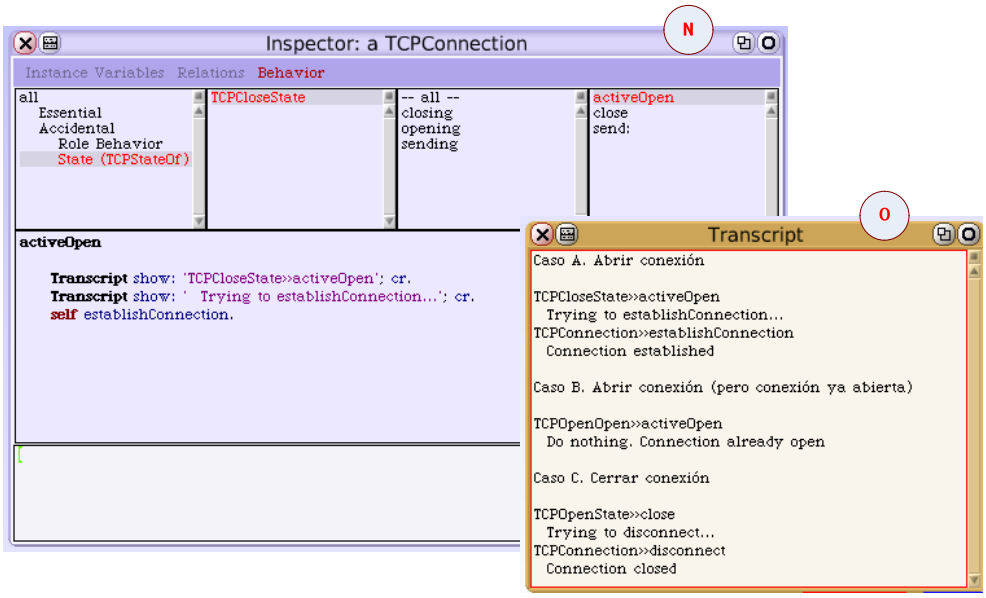


Figura 95. Pattern State sobre Relaciones. Prueba de Concepto

Aquí volvemos a apreciar los distintos métodos que son utilizados y el nuevo cambio de estado de `conn` hacia `TCPCloseState`.

Capítulo 5

Trabajos Relacionados

Durante el proceso de elaboración, construcción y formalización del Modelo de Relaciones y Comportamiento nos ha resultado de particular interés la diversidad de áreas de estudio, lenguajes y/o tecnologías que, a pesar de su heterogeneneidad, poseen cercanía con los modelos -y sus consecuencias- expuestos en este informe.

En las próximas secciones intentaremos exponer el por qué de esta "cercanía" con lenguajes de modelado como UML, trabajos sobre Relaciones, frameworks como Traits y Magritte, lenguajes de POO como Self, áreas de estudio como Programación basada en Roles, y lenguajes de programación del paradigma lógico como Prolog, entre otras.

5.1 Lenguajes de Modelado

Entre finales de los setenta y finales de los ochenta, los metodólogos comenzaron a experimentar con nuevos enfoques de análisis y diseño, con el fin de adaptarse a los nuevos lenguajes de programación orientados a objetos y a aplicaciones cada ves más complejas. Fue por entonces donde aparecieron los lenguajes de modelado orientados a objetos. Aprendiendo de estas experiencias, hacia mediados de los noventa ya habían surgido una nueva generación de metodologías, entre los que se encontraban el de Booch, OOSE (Object-Oriented Software Engineering) de Jacobson y OMT (Object Modeling Technique) de Rumbaugh.

Dado que cada uno de éstos estaban evolucionando independientemente hacia los otros dos, sus creadores decidieron unificarlos. Esta unificación se llamó Lenguaje Unificado de Desarrollo (UML) y en su versión UML 1.1 fue aceptada por la OMG (Object Managent Group) como estándar.

En las secciones siguientes analizamos similitudes entre la capacidad descriptiva de UML y Relaciones.

5.1.1 Semántica de Clases, Relaciones y Vínculos en UML

Una Relación se encuentra al mismo nivel que las Clases. Es un objeto cuya responsabilidad es especificar cierto tipo de vínculos que se da entre sus participantes. A su vez, es responsable de hacer cumplir las reglas y las restricciones que pueden darse entre dichos vínculos.

Un Vínculo se establece entre 2 o más objetos, cada uno de ellos cumpliendo un rol específico. La semántica de éste vinculo se define a través de la Relación a la cual pertenece. Entre los objetos vinculados, pueden existir atributos propios de la semántica por la cual se están vinculado. Estos atributos también son especificados en la relación.

UML combina estos conceptos y los utiliza en los diagramas de clase. Sin embargo, luego de la maduración de nuestro trabajo encontramos ciertas ambigüedades o abusos de notación que es conveniente exponer.

En [Booch et al. 1999] se definen varias relaciones (*Dependecy*, *Generalization*, *Use*, ect) y se define para cada una de ellas el tipo o estereotipo de línea gráfica que representará sus vínculos (instancias de asociación). En forma similar definen también el estereotipo gráfico con que se modelará una nueva Relación (Asociación).

Un diagrama de clases permite conectar dos clases con estas líneas. Ocurre entonces que cuando se conectan dos clases mediante un vínculo (Ej. *isSubclassOf*) las clases toman una semántica de "objeto" participante. Pero cuando se las conecta con una línea que representa una Relación, toman la connotación de "dominio de objetos"

Como ejemplo tomemos el diagrama de clases pattern Composite en [Gamma et al. 1995].

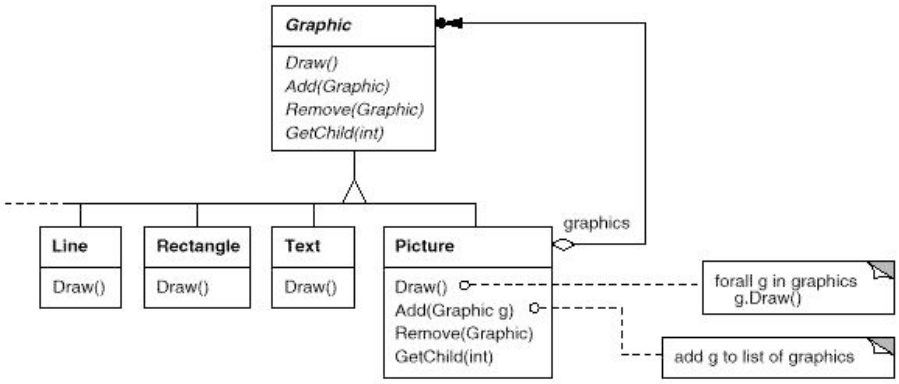


Figura 96. Pattern Composite. Diagrma de Clases

Si nos enfocamos en diferentes aspectos del diagrama obtenemos las siguientes interpretaciones

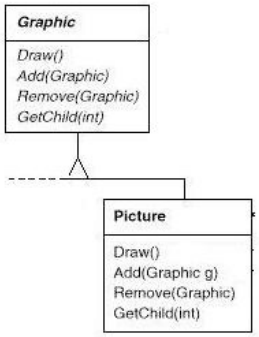
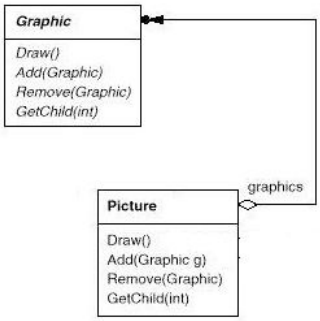
Gráfico		
Interpretación	La clase `Picture` es subclase de la clase `Graphic`	Las instancias de la clase `Picture` pueden estar compuestas por subinstancias de la clase `Graphic`
Semántica de las clases	Son consideradas como los objetos `Picture` y `Graphic`	Son considerados como dominios *{ o | o is a Picture}* y *{ o | o is a Graphic }*
Semántica de la línea	Representa al vínculo *isSubclassOf*	Representa Relación *Composition*

Tabla 7. Análisis semántico de Clases y Relaciones en un diagrama de clases de UML.

En consecuencia, cuando las clases son conectadas con líneas que representan Vínculos, deban ser interpretadas como objetos, pero cuando se las conecta con líneas que representan Relaciones, cumplen las veces de descripción de dominio.

5.1.2 UML y Relaciones

En general, los lenguajes de modelado son semánticamente más ricos que cualquier lenguaje de programación orientado a objetos actual. De hecho, según [Booch et al. 1999] esta es una de las razones por las que se necesitan modelos además de código.

Dada las características de nuestro trabajo especializado en un ambiente ejecutable con relaciones, el alcance de nuestra comparación se limita a los elementos disponibles en UML para la representación de las mismas.

5.1.2.1 Dominios

A diferencia de nuestro modelo, UML no ofrece la posibilidad de definir relaciones sobre dominios de rol ad-hoc como `{ s | s is String and s size < 4}`.

En su lugar utiliza Clases e Interfaces que se usan para describir el conjunto de todos los objetos que las satisfacen y lo complementa con restricciones predicadas en OCL sobre la relación.

5.1.2.2 Propiedades y Restricciones

En Relaciones damos la posibilidad describir la relación a través de alguna de sus propiedades matemáticas como por ejemplo Reflexiva, Simétrica, Transitiva, etc.

También permitimos definir propiedades que restrinjan la posibilidad relacionarse, como la Asimétrica o Acíclica. Pero cuando el dominio de problema lo requiera, Relaciones permite describir restricciones ad-hoc, en forma similar a como UML hace con OCL[OCL]

5.1.2.3 Atributos de una Relación

UML permite adornar una relación con diferentes atributos con el fin de describir el tipo y las características de una relación. Entre estos atributos se encuentran la Navegavilidad, la Visibilidad, Calificadores, Multiplicidad, Tipo de Agregación, entre otros.

El modelo de Relaciones presentado en este trabajo contempla alguna de estas posibilidades.

5.1.2.4 Multiplicidad y Cardinalidad

Mientras que UML utiliza solamente la Multiplicidad, en nuestro trabajo vimos necesario contar también con Cardinalidad para describir adecuadamente los requisitos y restricciones de participación y vinculación de objetos en una relación. Esto se debe a que si bien en relaciones binarias uno de estos conceptos puede inferirse del otro, en las relaciones de grado mayor a 2 son necesarios ambos.

5.1.3 Ingeniería directa e inversa sobre Relaciones

La ingeniería directa es el proceso de transformar un modelo en código a través de una correspondencia con el lenguaje de implementación. La ingeniería directa produce una pérdida de información porque los modelos estrictos son semánticamente más ricos que cualquier lenguaje de programación orientado a objetos actual.

La ingeniería inversa es el proceso de transforma código en un modelo a través de una correspondencia con el lenguaje de programación específico. Este proceso es incompleto. Hay pérdida de información por lo que no se puede recrear completamente un modelo a partir de código[Booch et al. 1999].

Nuestro lenguaje de programación con Relaciones reduce considerablemente este problema al introducir todos los elementos semánticos que permiten mapear ambos modelos en forma directa. Incluso, tal como se expresa en [Booch et al. 1999] y en la sección "Capacidad auto explicativa" la necesidad de construir diagramas y modelos previo a desarrollo e implementación podría perder relevancia.

5.2 Relaciones de primera clase en OO

5.2.1 Breve descripción

Los lenguajes de modelado como UML y diagramas de ER proveen asociaciones y relaciones como abstracciones elementales. Sin embargo, como hemos mencionado anteriormente, los actuales lenguajes orientados a objetos no ofrecen el soporte necesario para representarlas directamente.

Rumbaugh [Rumbaugh 1987] fue pionero en señalar el importante rol de las relaciones en los lenguajes orientados a objetos. Así, además de presentar informalmente un modelo, lo trasladó en un lenguaje al que llamó DSM [Shah et al. 1989]. Sin embargo, salvo un trabajo sobre propagación de métodos, y a pesar de una interesante lista de trabajos futuros, no encontramos durante nuestra investigación una continuidad a este trabajo.

Kolp [Kolp 1997] definió un modelo formal de relaciones en objetos y presentó una posible arquitectura de solución para implementar y reificar el concepto de Relación como construcción de primera-clase en un lenguaje con reflexividad como CLOS.

Noble presentó una serie de patterns para programar relaciones en lenguajes orientados a objetos[Noble 1997], y junto a Grundy [Noble and Grundy 1995] sugirieron que las relaciones deberían ser explicitas en los programas orientados a objetos. A pesar de ello, no brindan en estos trabajos detalles concretos sobre cómo dar soporte a relaciones en lenguajes orientados a objetos.

Bierman y Wren [Bierman and Wren 2005] trabajaron en un modelo formal y matemático con el fin de que permita que lenguajes fuertemente tipados (Ej. Java) puedan soportar relaciones de primera-clase. En este sentido desarrollaron RelJ, un lenguaje prototípico basado en un subconjunto funcional de Java.

Otros autores también han estudiado el tema. Suscheck y Sandén en [Suscheck and Sandén 2003]. Barbier y Henderson-Sellers describieron un modelo semántico sobre las relaciones WholePart [Barbier and Henderson-Sellers 1999].

5.2.2 Comparación con Relaciones

Nuestro trabajo encuentra en los anteriores muchas similitudes. Así, dado que su objeto de estudio es el mismo, las distintas perspectivas desde las cuales se han enfocado conducen -en muchos casos- a resultados o consecuencias complementarias entre sí.

En nuestro caso, concebimos nuestro modelo en tanto pragmática y paradigmáticamente. Nos permitimos llevar nuestras ideas rápidamente al ambiente de Smalltalk, sin la necesidad o preocupación del contar con el formalismo previo requerido en otros trabajos. Esto, sumado a las facilidades de

Smalltalk para obtener rápidas prototipaciones, nos permitió hacer evolucionar libremente el modelo subyacente.

Tal vez por ello, de la comparación con otros trabajos surgen algunas diferencias, las cuales trataremos de presentar a continuación.

5.2.2.1 Modelo de Relaciones

En general, todos los trabajos que hemos analizado utilizan argumentos similares para plantear el problema a resolver. Estos argumentos coinciden en su gran mayoría con aquellos que motivaron la concepción de nuestro trabajo.

También existe cierto consenso (posiblemente influidos por UML) sobre los principales aspectos para describir una nueva Relación o Asociación. Entre éstos se presentan los roles, la multiplicidad, navegabilidad, visibilidad, tipo de agregación, clases de asociación, entre otros.

A pesar de ello, al introducirnos en el mismo modelo de relaciones comenzamos a encontrar ciertas diferencias.

La Relación y sus vínculos

La primera de estas diferencias es cómo se representa una Relación. Todos los trabajos anteriores describen una nueva Relación con una nueva Clase, subclase de la clase Relation, donde los vínculos entre objetos son expresados como instancias de dicha Relación (Clase)

Nuestro modelo no sigue este principio. Tal como lo hemos descripto, existe una única clase **`RRelation`**, donde cada nueva instancia refleja una nueva relación. Luego, los vínculos son representados como instancias de de **`RLink`**, a menos que durante la definición de la relación el usuario haya optado por un nuevo tipo de Vínculo, por ejemplo `Marriage` para la relación *MarriedWith*.

Si bien en este trabajo no hemos profundizado las consecuencias -a favor y en contra- del camino adoptado, consideramos el nuestro ofrece una gran flexibilidad al privilegiar los mecanismos de composición por sobre los de herencia y subclasificación.

Administración y Control del Conocimiento

Los trabajos analizados, al modelar los vínculos como instancia de una Clase, dan a suponer que la administración de dichos vínculos se realiza en forma similar a como se hace con las instancias de una Clase, cuando en realidad esto no es así. A pesar de ello, y más allá de descripciones formales sobre cómo debería comportarse una relación, estos trabajos no permiten inferir cual es el modelo subyacente que da sustento a dicha administración.

En nuestro trabajo dimos esa responsabilidad al **`RKnowledge`**. Cada relación posee su propio **`RKnowledge`**, el cual trabaja en conjunto con ésta para hacer cumplir todas y cada una de sus propiedades, restricciones y reglas que deben regir entre los vínculos. En términos generales, el **`RKnowledge`** es responsable de garantizar el Invariante de la Relación.

Representación del Conocimiento

Al concentrar en un **`RKnowledge`** por cada relación la administración de los vínculos, es posible representarlos en diversas maneras y estrategias. Así, los vínculos podrán ser representados en forma explícita o implícitamente, con hash-tables o conjuntos, sobre un motor prolog, persistiendo en bases de datos relacionales, con noción de orden, combinaciones de los anteriores u otras alternativas.

Los modelos analizados presentan dificultades para ofrecer esta flexibilidad, principalmente debido a que éste mecanismo esta internalizado en otro nivel de abstracción superior.

Relaciones Derivadas

Según Kolp [Kolp 1997], una relación derivada es una relación inferida de una combinación de otras relaciones. Rumbaugh las definió entre los trabajos futuros de [Rumbaugh 1987], diciendo que a menudo es conveniente construir objetos y relaciones que sean estrictamente deducidos de un conjunto independiente de objetos y relaciones. Los restantes trabajos apenas si las mencionan y si lo hacen, no incursionan en un modelo que describa cómo pueden implementarse.

La evolución natural de nuestro modelo nos ha permitido expresar e implementar relaciones derivadas de manera ágil, incluso mediante diferentes enfoques y estrategias. Así es como al pretender derivar una relación en función del estado del ambiente es posible seguir alguna de las siguientes alternativas

- Crear un **`RKnowledgeProducer`** que a través de observaciones sobre el ambiente detecte cambios y actualice el conocimiento de la relación derivada.
- Definir un **`RKnowledge`** ad-hoc con la lógica esperada.

Por ejemplo, en este informe introducimos al **`RKnowledgeProducer`** en la sección Adquisición del conocimiento del Capítulo 3, cuando presentamos el ejemplo la relación *GrandfatherGrandchild* deducida en base a *FatherChild*.

Otro caso se presentó al configurar *BehaveAs* con la implementación ad-hoc **`RKnowledge`**. Esto simplificó en gran medida el Method Lookup, lo que nos permitió validar aquella hipótesis de Rumbaugh [Rumbaugh 1987] en la que suponía "*si los lenguajes de programación orientados a objetos dispusieran de este tipo de relaciones, podrían obtenerse grandes simplificaciones en diferentes algoritmos*".

Reflexividad, Transitividad y demás propiedades matemáticas

A diferencia de los otros trabajos donde se ha mencionado brevemente este característica, el modelo de relaciones permite expresar o definir una relación mencionando que propiedades matemáticas cumple. Luego es responsabilidad de **`RKnowledge`** hacer cumplir este invariante, según la estrategia que se haya elegido.

Jerarquía de Relaciones

Salvo Bierman y Wren [Bierman and Wren 2005] que desarrollaron un modelo formal matemático sobre cómo representar una jerarquía de relaciones, y lo llevaron a su prototipo RelJ, los demás autores no han profundizado sobre la semántica y representación de esta jerarquía, salvo la de subclasificación (jerarquía que Rumbaugh consideró necesario clarificar su semántica).

Dentro del alcance de nuestro trabajo no hemos incluido este tema. Sin embargo, consideramos que al momento de incluirla el trabajo de Bierman y Wren puede ser un buen punto de partida.

5.2.2.2 Modelo de Comportamiento sobre Relaciones

Con excepción de Rumbaugh [Rumbaugh 1987] -que planteó la posibilidad de que un conjunto de métodos sean generados en las clases de los objetos participantes de una relación para acceder y actualizar la información de esta-, ninguno de los trabajos anteriores teorizó sobre cómo la aparición una relación afecta el comportamiento de los objetos participantes.

En la próxima sección compararemos este aspecto del modelo con diferentes trabajos sobre programación basada en roles, fuertemente ligado a Relaciones.

5.3 Traits

5.3.1 Breve descripción

Los Traits son unidades de comportamiento[32] que proveen una forma de estructurar programas orientados a objetos por encima de los métodos pero por debajo del nivel de clases[Scharli et al. 2003b]. Esencialmente, un Trait es un grupo de métodos[33] que sirve como bloque para construir clases y como unidad primitiva de reuso de código [Scharli et al. 2003a].

5.3.1.1 Problema motivador

A pesar de la importancia de la Herencia como el mecanismo fundamental de reuso en los lenguajes orientados a objetos[34], sus principales variantes –herencia simple, herencia múltiple y mixin- sufren de problemas conceptuales y prácticos [Scharli et al. 2003a], entre los que pueden mencionarse

- Código duplicado
- Conflictos de propiedades: de estado y de métodos
- Acceso a propiedades sobrecargadas o en conflicto
- Extracción de wrappers genéricos
- Orden Total
- Dispersión del Glue Code (código de pegamento)
- Estructuras frágiles

Estructurar clases con herencia simple puede terminar en código duplicado cuando distintas ramas de la jerarquía usa un mismo feature. La herencia múltiple y mixins alivian el problema, pero traen otras dificultades de cara a la evolución: cambios en las clases o mixins pueden romper (en modos inesperados) código de más abajo en la jerarquía [Scharli et al. 2003b].

Traits soluciona ambos problemas, extrayendo y factorizando el comportamiento compartido como conjunto de métodos que no dependen del estado. La composición de Traits es simétrica y los conflictos deben ser manejados en forma explícita; esto significa que cambios en los componentes no generan efectos inesperados [Scharli et al. 2003b].

5.3.1.2 Descripción del modelo de Traits

Los Traits proveen servicios (Ej. los métodos implementados) y requieren a su vez de otros servicios (ej, aquellos métodos que son invocados por `self` y `super`-sends) [Scharli et al. 2003b].

A diferencia de las Clases, los Traits no contienen variables de instancia, y como consecuencia los métodos no pueden referencias a campos. Sin embargo, un método de Traits puede acceder al estado de un objeto en forma indirecta, es decir, a través del getter provisto por la clase a dicha variable de instancia[Scharli et al. 2003b].

Los Traits cumplen las siguientes propiedades[Scharli et al. 2003a]

- Un Trait provee un conjunto de métodos que implementan un comportamiento
- Un Trait requiere un conjunto de métodos que sirven como parámetros del comportamiento provisto.

[32] Están al nivel de clases, o templates, o RBehavior

[33] No hay variables de instancia como en las clases

[34] Nos referimos aquí a los lenguajes basados en clases

- Los Traits no especifican ninguna variable de estado, y los métodos provistos nunca acceden a éstas directamente
- Las clases y los traits pueden ser compuestos por otros traits, pero el orden de composición es irrelevante. Los métodos en conflicto deben ser resueltos en forma explícita
- La composición no afecta la semántica de una clase: el significado de la clase es el mismo que si los métodos del trait se hubiesen implementado directamente en la clase.
- La composición no afecta la semántica de un trait: un trait compuesto es equivalente a un trait aplanado conteniendo los mismos métodos.

5.3.1.3 Method lookup

En cuanto a las reglas de precedencia utilizadas por el method lookup, los métodos de la clase de un objeto toman precedencia por sobre los métodos del los traits, mientras que estos últimos toman precedencia por sobre los de las superclases de los métodos.

5.3.1.4 Ejemplo

La figura siguiente muestra como el trait `TCircle` es compuesto por un trait `TGeometry` y un trait compuesto `TMagnitude`, el cual contiene un subtrait `TEquality`. Es importante notar que los servicios provistos por los subtraits son propagados al trait compuesto (ej., `max:`, `~=`, y `area`). En forma similar, los requisitos no satisfechos por los subtraits (ej., `center` an `radius:`) son considerados métodos requeridos del trait compuesto [Scharli et al. 2003a].

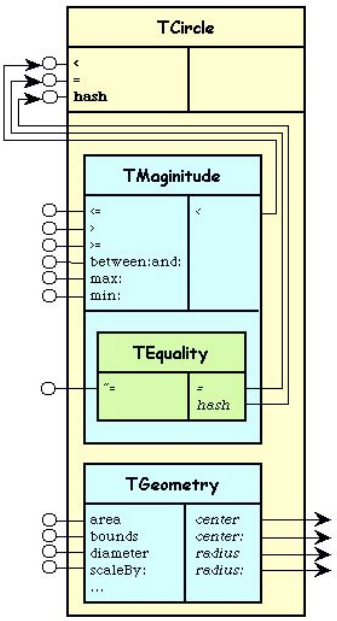


Figura 97. Tcircle. Trait

5.3.2 Traits y Relaciones

5.3.2.1 Composición entre Traits, vista como Relación

Aunque en sus trabajos [Scharli et al. 2003a][Scharli et al. 2003b] Nathanael Scharli, Oscar Nierstrasz, Stéphane Ducasse, Andrew P. Black hablan de la Composición entre Traits como una operación, luego del desarrollo de nuestro trabajo es posible imaginar una consecuente relación de Composición.

Supongamos que reificamos esta relación y la llamamos *TraitComposition*.

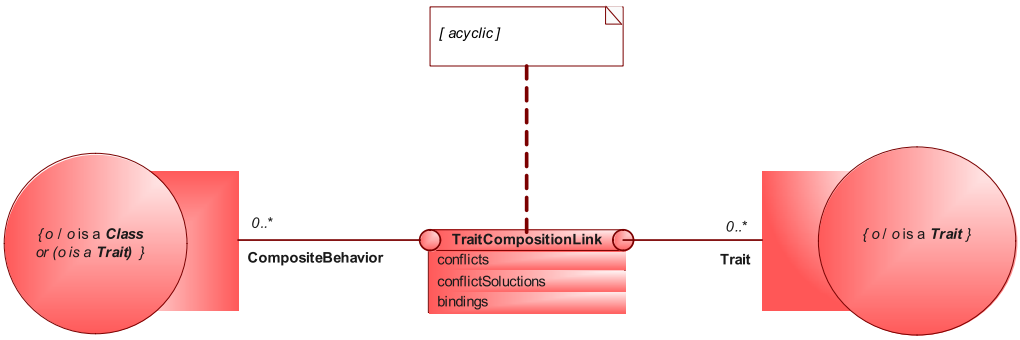


Figura 98. TraitComposition. Diagrama de Relación

Luego, cuando un Trait A quiere usar los servicios provistos por un Trait B, se lleva adelante la operación de composición de A por B normalmente. Pero una vez finalizado el proceso, expresamos que A ha sido compuesto por B mediante un nuevo vínculo de composición entre A y B. En dicho vínculo se encontrarán las decisiones del operador respecto a como han sido resueltos estos conflictos, los sinónimos utilizados, y los bindings realizados. Luego, el Framework de Traits podrá hacer uso de este vínculo para compartir o reutilizar comportamiento entre A y B.

5.3.2.2 Empatía por Grupo

Cuando una Clase c se compone con un Trait t, el comportamiento expresado por este es inmediatamente adquirido por todas las instancias de la clase c . De esta forma, Traits extiende el mecanismo de empatía por grupo, común a los lenguajes de programación orientados a objetos basados en clases.

Por lo anterior, decimos que los Traits se encuentran en un mismo nivel de abstracción que las clases. Pero a diferencia de éstas no se relacionan con objetos del nivel de abstracción inferior, es decir, las instancias. En consecuencia Traits no ofrece un mecanismo de empatía por objeto.

5.3.2.3 Interpretación Semántica

Sabemos que un Trait está compuesto por un conjunto de métodos que han de ser reutilizados por más de una clase. Pero, ¿qué representa este comportamiento y con que criterio deben ser agrupados estos métodos?

Scharli, Nierstrasz, Ducasse y Black definen traits como un bloque para construir clases y como unidad primitiva de reuso de código [Scharli et al. 2003a], pero no definen semánticamente qué es un trait (o qué debería ser), en la misma forma en que la literatura se ha encargado de explicar qué es una Clase.

El modelo de comportamiento sobre Relaciones propone una concepción diferente. En lugar de mera reutilización de código, los argumentos que guían la apración de un comportamiento se encuentran en

- La definición de los roles de una nueva relación
- Los objetos que pueden participar en una relación de comportamieno.

5.3.2.4 Resolución de Conflictos

Traits maneja en forma explícita los potenciales conflictos que puedan surgir durante la operación de composición. Esto, sumado a la empatía por grupos, ofrece plenas garantías y permite conocer qué método será invocado ante la recepción de cierto mensaje.

En contraposición, el modelo de comportamiento sobre Relaciones relaja este requisito y privilegia la empatía por objeto y dinámica. Esto dificulta establecer un criterio unívoco sobre cómo manejar los

conflictos; pero aún así es posible definir algoritmos que brinden un criterio uniforme. Por ejemplo, en la implementación realizada durante este trabajo, la relación *BehaveAs*[35] prioriza los comportamientos recientemente adquiridos por sobre los más antiguos, a excepción del esencial, que precede a todo el resto.

5.4 Split Objects

5.4.1 Breve descripción

5.4.1.1 Problema motivador

En los ambientes de programación sin clases –como Self- la relación de delegación provee naturalmente Property Sharing. Sin embargo, en algunas situaciones este mecanismo de Sharing genera problemas de identidad entre los objetos participantes de dicha relación.

En la figura siguiente se ha creado Turtle1 a partir del Point1. Si consultamos a la tortuga por su ubicación nos responderá 5@10 y todo funciona como esperamos. Pero si luego le indicamos a la tortuga que avance 4 puntos, esto hará que el punto Point1 sea 5@14 (cambia el valor de la propiedad y del punto)[Bardou and Dony 1996].

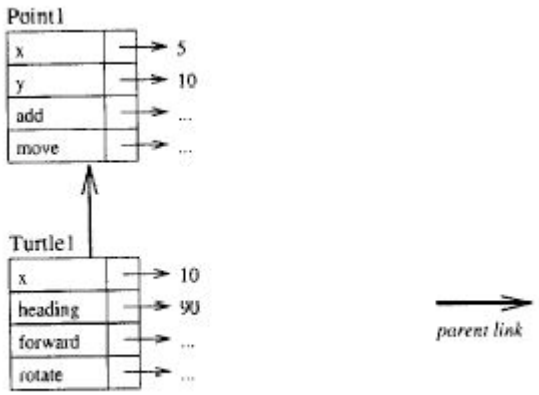


Figura 99. Split Object

En este caso, dado que el vínculo responde a una relación de delegación con Property Sharing, no solo se garantiza acceso de lectura a las propiedades de Point1, sino también de escritura.

5.4.1.2 Descripción del modelo de Split Objects

Daniel Bardou y Christophe Dony definen Split Objects[Bardou and Dony 1996] como un mecanismo para representar distintos puntos de vista (viewpoints) de un objeto.

Un Split Object es definido como una conjunto de piezas. Las propiedades de un Split Object se almacenan en sus piezas. Las piezas son organizadas dentro de un objeto en un jerarquía de delegación. Un Split Object representa una sola entidad del dominio a representar y sus piezas denotan los puntos de vista de dicha entidad. [Bardou and Dony 1996]

[35] Responsable de determinar los comportamientos de un objeto.

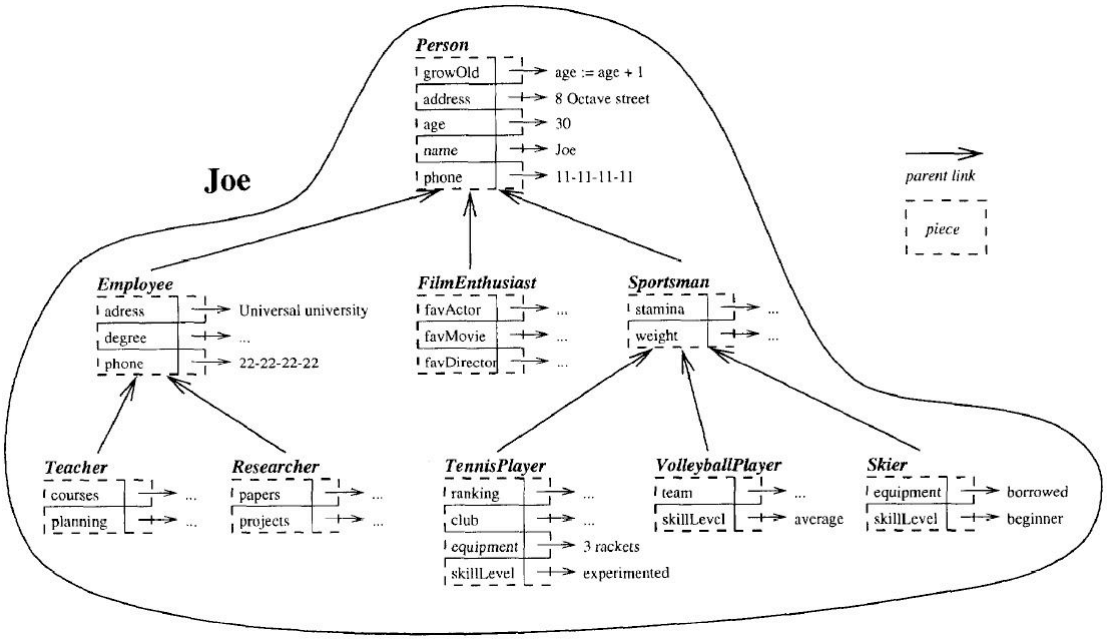


Figura 100. Split Objects. Viewpoints

Este mecanismo espera de los vínculos de delegación un determinada semántica. En primer lugar la delegación debe permitir property sharing[36]. En segundo lugar los vínculos se establecen entre viewpoints de un mismo objeto, y no a nivel de objetos. Un incorrecta semántica puede llevar a al problema de la identidad del objeto [Bardou and Dony 1996].

Los Split Objects son entidades de primera, y por lo tanto se acceden directamente, no así sus piezas. Pueden crearse por clonación o por ex-nihilo[37], y modificarse por piezas, agregando, quitando o modificando una de estas.

5.4.1.3 Method Lookup

Un mecanismo básico de envío de mensajes y consecuente Method Lookup permitiría a un usuario especificar el viewpoint que responderá al mensaje enviado. Este mecanismo da por supuesto que a) cada Viewpoint está representado por una y sólo una pieza y b) que el usuario que envía el mensaje conozca la estructura interna de piezas y nombre de Viewpoints posibles. Esto hace que este simple mecanismo sea insuficiente.

La realidad indica que un usuario puede querer interactuar con Joe (ver ejemplo) como un todo, o con un Viewpoint de Joe compuesto por las piezas Researcher y Theacher. De aquí se desprende que pueden existir muchos más viewpoints que piezas [Bardou and Dony 1996]

Se requiere entonces un mecanismo más complejo de envío de mensajes y method lookup, que permita que cualquiera de los posibles viewpoints de un Split Object reciba un mensaje. Las propiedades de un Split Object deben ser activadas sin necesidad de conocer las piezas que están almacenadas. Por esto, y

[36] Algunos sistemas restringen la delegación con property sharing a value sharing[SO]

[37] Ex nihilo es un término en Latín que significa “de la nada”. Generalmente es usado en conjunto con el término creación, como en creatio ex nihilo, que significa “creación de la nada”.

dado que es posible que existan mensajes ambiguos, una estrategia de method lookup debería detectar estas ambigüedades. A su vez, parece razonable esperar que esta estrategia, no realice lookup en piezas que no estén relacionadas con las piezas especificadas por el viewpoint en cuestión. En [Bardou and Dony 1995] se propone una estrategia que asegura una semántica uniforme de envío de mensajes, tanto a viewpoints explícitos como implícitos.

Split Objects utiliza dos pseudo-variables: `self` y `thisViewpoint`. Durante la activación de un método se bindea a `self` con el Split Object que recibió el mensaje, mientras que `thisViewpoint` lo hace con el nombre de la pieza que denota al viewpoint activado[Bardou and Dony 1996].

5.4.2 Split Objects y Relaciones

Una de las principales diferencias entre el Split Object y Relaciones es el ambiente en que fueron concebidas. Mientras que Split Object surge en un ambiente con delegación, Relaciones nace en un ambiente de clases como Squeak que no posee este mecanismo de sharing.

5.4.2.1 Empatía por Objeto

Al igual que en Relaciones, y como resulta natural de un ambiente con Delegación, Split Object permite la empatía por objeto, dinámica.

5.4.2.2 Interpretación Semántica

En Split Objects, a través de la relación *Delegation* se logra el sharing entre objetos. Sin embargo, al ser ésta la única relación en el ambiente, resulta difícil disponer de un mayor poder semántico entre piezas que “delega en”.

Es más, en su trabajo definen como elemento fundamental a la “pieza”, el cual permite componer y dar vida a un Split Object, pero que no es un objeto de primera clase como este último. Esto lleva a preguntarse cuáles serán los criterios o principios que definen qué es una pieza.

En este enfoque se persigue el fin de dar identidad a un objeto –representando por un Split Object- y que este se mantenga inmutable a lo largo de su vida.

Si bien los vínculos entre piezas se establecen con la relación *Delegation*, no queda claro cuál es el vínculo (y la relación semántica que lo rige) existente entre un Split Object y sus piezas.

Relaciones ofrece un ambiente donde es posible definir una nueva relación cuando la semántica del dominio de problema así lo indique. Si esta relación es de comportamiento, el Framework se hará responsable de hacer que los objetos adquieran los comportamientos con que se relacionen.

En Relaciones todo se limita objetos relacionados con otros enviándose mensajes. Es por ello que no existe una diferencia semántica como la que existe entre Split Object y sus piezas.

Por último, con respecto a la identidad de los objetos, Relaciones hace explícito que ésta es adquirida en función de la relación esencial *InstanceOf*, y de nadie o nada más. Cualquier otro atributo o comportamiento será adquirido permanente o temporalmente a través de vínculos con otros objetos.

5.4.2.3 Viewpoints

Split Objects plantea el concepto Viewpoint como una conjunción o agregación de de algunas (no necesariamente todas) piezas que componen a un Split Object. Esto permitiría, por ejemplo, que un objeto que cumpla las veces de alumno pueda enviarle mensajes al Joe profesor e investigador, es decir, al Viewpoint de Joe compuesto por las piezas Researcher y Theacher.

Este es un punto que en Relaciones no nos hemos detenido a analizar. En nuestro trabajo no contamos con mecanismos que permitan alterar el comportamiento de un objeto en función de la forma en que se invoca el mensaje. En otras palabras, no permitimos que un objeto nos diga "respondeme este mensaje desde tu perspectiva de Researcher y Teacher".

5.4.2.4 Resolución de conflictos

Split Objects se encuentra con conflictos similares (ambigüedades) a los que nos enfrentamos en Relaciones. En su enfoque es responsabilidad del MethodLookup definir la estrategia adecuada para su resolución. Así, en [Bardou and Dony 1995] se proponen una estrategia que asegura una semántica uniforme de envío de mensajes.

Si bien la implementación de nuestro modelo de comportamiento es simple, su pilares y fundamentos sobre MethodLookup y resolución de conflictos coinciden en forma con el enfoque de Split Object.

5.4.2.5 Ejemplo con Relaciones

En el ejemplo siguiente podemos observar la forma en que el modelo de Relaciones ofrecería a Juan un comportamiento y potencial similar al que el modelo de Split Object ofrece a Joe.

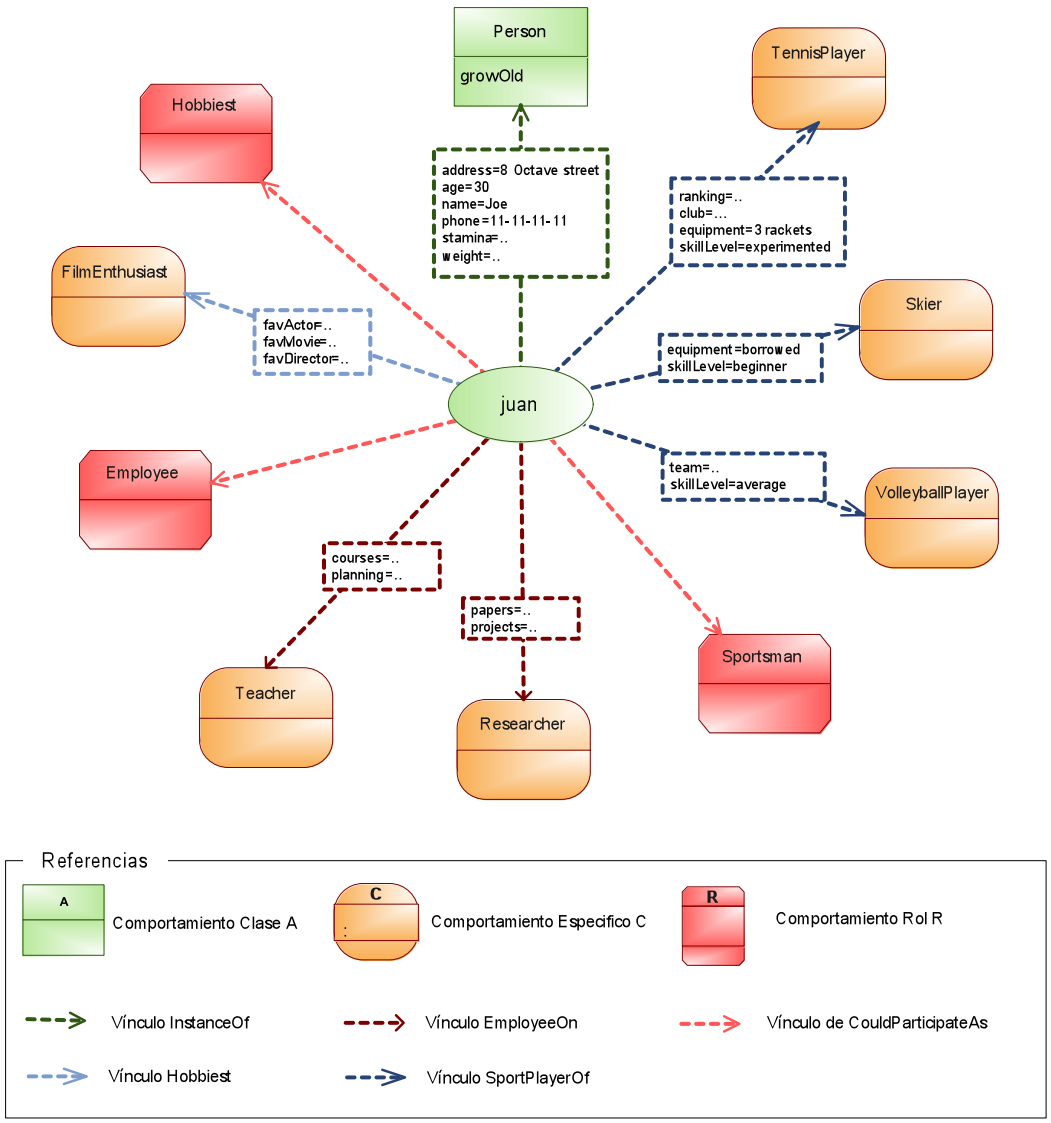


Figura 101. juan. Vínculos de comportamiento

Observamos que surgen las entidades o comportamientos de Rol que Juan cumple. Estas comportamientos de Rol se desprenden de las relaciones *EmployEmployee*, *InstanceOf*, *SportPlayer*, *HobbieHobbiest*. Son estas relaciones las que dan sentido a que Juan puede comportarse, actuar o disfrutar siendo Teacher, Researcher, VolleyballPlayer, Skier, TennisPlayer, FilmEnthusiastic.

Pero aunque el modelo de comportamiento ofrece este potencial, hemos decidido dejar para trabajos futuros el modelado de atributos inherentes a un comportamiento (como papers para Theacher o ranking para TennisPlayer).

5.5 Magritte

Ya hemos presentado como prueba de concepto la integración del modelo de relaciones con Magritte para desarrollar un Editor de Objetos de negocio. En esta sección nos proponemos profundizar aspectos más conceptuales sobre el alcance y potencial de ambos.

5.5.1 Breve descripción

Magritte es un meta-modelo integrado al meta-modelo reflexivo de Smalltalk [Renggli 2006] que tiene la capacidad de describir objetos del dominio desde un nivel meta. Dispone de un Framework con un conjunto de herramientas que interpretan el meta-modelo y permiten en forma automática construir vistas, reportes, editores con validaciones y mecanismos de persistencia sobre estos objetos de dominio [Renggli 2006].

A diferencia de otros meta lenguajes orientados a objetos utilizados para describir las entidades de dominio como MOF, EMOF y ECore [Renggli 2006], Magritte cuenta con la capacidad no sólo de describir estructuralmente el modelo, sino que permite especificar la semántica operacional del metamodelo.

Para lograr esto, utiliza el concepto Descripción, implementado con `MADescription` y sus subclases. Una Descripción se encarga de especificar o detallar las características de un aspecto de la clase de dominio bajo descripción, como ser un variable de instancia, sus propiedades y relaciones. [Renggli 2006].

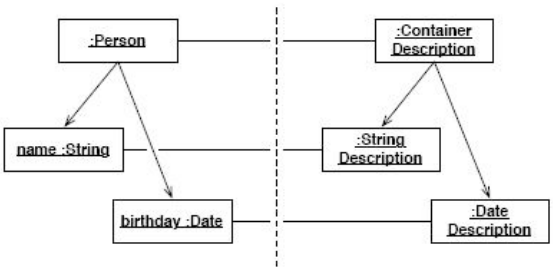


Figura 102. Instancia de Modelo Persona con sus descripciones asociadas. [Renggli 2006]

En la extracto de código siguiente se observa la instanciación la descripción del atributo name de `Person`.

```
(StringDescription selector: #name label: 'Name')
    beRequired;
    yourself
```

En cuanto al conjunto de herramientas, su funcionamiento se basa en consultar las descripciones de un objeto dado y en función de éstas operar.

5.5.2 Magritte y Relaciones

5.5.2.1 Descripción del dominio de problema

Tanto Maggrite (en una de sus funciones) como Relaciones proponen modelar o describir los objetos de negocio de cierto modelo de problema.

Cómo ya hemos mostrado durante una de las pruebas de concepto, es posible a partir del modelo de relaciones generar dinámicamente las descripciones que las herramientas de Magritte necesitan para operar. Esto nos permite comprender que Relaciones posee al menos las principales capacidades descriptivas requeridas por el meta-modelo de Magrittte.

5.5.2.2 Descripción pura vs. Descripción por extensión

Aunque en el Framework de Magritte no se encuentran separadas, existen dos grandes categorías que abarcan el universo de información que Magritte puede describir. Estas son

- Descripción pura del modelo
- Descripción adicional al modelo para adaptarlo al dominio de problema de los meta intérpretes. (Por ejemplo, un meta-intérprete responsable de presentar un editor de un objeto deberá conocer con que widget representar la fecha de cumpleaños)

Actualmente, aunque el modelo de Relaciones lo propone, el framework resultante no contempla atributos de extensión, propios a otras herramientas.

5.5.2.3 Magritte integrado sobre Relaciones

El Framework de relaciones permite que diferentes herramientas utilicen su información, accediéndola en forma estructurada, coherente y uniforme.

Fue así como durante a prueba de concepto “Editor de Objetos de negocio” luego de integrar el modelo de Relaciones con Magritte pudimos hacer uso de todas las facultades y herramientas.

5.5.2.4 Momento de la descripción del modelo

Una diferencia no menor entre la aproximación de Magritte y la de Relaciones es la etapa en la que se desarrolla el meta modelo. Mientras Magritte necesita de un modelo ya existente para luego tomar uno por uno los objetos del dominio de problema y describirlos, Relaciones permite construir el modelo a través de la definición de diferentes relaciones, facilitando el desarrollo y no requiriendo describir un modelo ya existente.

5.5.2.5 Objetivo o metas

Aunque más difícil de apreciar, tal vez la diferencia más importante entre Magritte y Relaciones se encuentra en el fin con que fueron creados.

Magritte se orienta a describir un modelo subyacente con el fin de resolver –ayudado por interpretes de nivel meta- la serialización, construcción de vistas con validación y de reportes.

Por el otro lado, Relaciones tiene como meta facilitar mecanismos para modelar un dominio de problema en forma más natural, con menor gap semántico entre usuario-programador. Como consecuencia de esto surge la descripción del modelo, y por lo tanto la posibilidad de construir herramientas sobre este, tal como Magritte.

5.6 Programación orientada a Roles

Muchos investigadores han argumentado las ventajas de los roles en el modelado e implementación de sistemas. Los roles permiten a los objetos evolucionar a lo largo del tiempo, permiten establecer interfaces independientes y concurrentes que ayudan a entender los distintos contextos de un objeto.

Generalmente los roles son un elemento natural en nuestra cotidiana formación de conceptos. Con roles los lenguajes de programación tienen la capacidad de brindar a un objeto la posibilidad de cambiar su comportamiento, tal como tal como lo observamos en la vida real –las cosas cambian a lo largo del tiempo, son utilizadas en forma diferentes en distintos contextos, etc.

5.6.1 Breve descripción

En el contexto de la programación de sistemas, Browne, Kane and Mahmood [Browne et al. 2005] consideran a los roles como especificaciones sobre las propiedades y comportamientos (semántica) de componentes. Entienden a la arquitectura de un sistema como la especificación de su estructura en término de estos roles, convirtiendo el desarrollo de programas en la mera selección de entidades que realizan los roles en la arquitectura. La especificación de los roles se logra en términos de ontologías del dominio de problema a resolver, donde la ontología define los comportamientos funcionales y de interacción de los componentes así como también un conjunto de atributos que permiten especificar las propiedades y comportamientos de los componentes y las relaciones/interacciones entre ellos.

Según Kendall, en el contexto de la programación orientada a objetos, los roles y los modelos de rol [Kendall 1999] son mecanismos de abstracción y descomposición. Mientras que las clases estipulan las capacidades de los objetos individuales, la noción de rol se enfoca en la posición y responsabilidades de un elemento dentro de un sistema o subsistema en su conjunto. Un modelo de rol identifica una estructura de elementos (objetos) y la describe como su correspondiente estructura de roles. Estos modelos capturan cómo los objetos interactúan entre sí; se ha demostrado que los modelos de rol son útiles tanto durante la conceptualización y análisis como en el diseño.

Kristensen plantea [Kristensen 1996][Kristensen 1997] un modelo conceptual entre un objeto y sus roles. El objeto al cual se asigna el rol es el objeto *intrínseco*; este tiene miembros intrínsecos (datos y métodos). Los roles agregan miembros *extrínsecos* (datos y métodos), y proveen perspectivas que pueden ser usadas por otros objetos como una forma selectiva de conocer y acceder al objeto.

5.6.1.1 Propiedades de Roles

Kristensen [Kristensen 1997] también define un conjunto de propiedades a satisfacer por los roles. Más tarde Kendall [Kendal99] basa su investigación en ellas.

- **Poder de Abstracción**. Los roles deben poder ser organizados en jerarquías.

- **Agregación/Composición**. Los roles deben poder componerse entre sí.
- **Dependencia**. Un rol no puede existir sin el objeto intrínseco.
- **Dinamismo**. Un rol se puede agregar o remover durante el ciclo de vida de un objeto.
- **Identidad**. El rol tiene la misma identidad que el objeto al que se encuentra asignado. El objeto y sus roles son vistos y manipulados como una misma entidad.
- **Herencia**. Un rol para una clase es también rol para cualquiera de sus subclases, y un super-rol es un rol para una clase si su sub-rol es un rol para la clase. (Existe un criterio alternativo que propone que un rol debería ser capaz de asignarse a cualquier clase).
- **Localización** (contexto). Un rol sólo tiene sentido en un modelo de rol.
- **Multiplicidad**. Varias instancias de un rol pueden existir para un mismo objeto. Un rol puede actuar en varios roles a la vez, incluyendo múltiples instancias del mismo rol.

5.6.2 Roles en la programación orientada a objetos

Varios trabajos con distintos enfoques se han presentado para implementar roles en lenguajes orientados a objetos [Kendall 1999].

Uno de los modelos más comunes es el propuesto por el pattern Role Object [Baumer et al. 1997]. Este pattern provee una clase individual para cada rol. Los roles son organizados en una jerarquía de clases. Un objeto central (una instancia de la clase `Core`) contiene en una variable de instancia los roles (subinstancias de `Role`). Tal como se menciona en menciona [Kendall 1999], otro de los enfoques para este propósito es el de Kristensen y Osterbye. Ellos plantean Role Object basado en el pattern Decorator como el mejor soporte para modelos de rol en los lenguajes orientados a objetos estándar.

Los casos anteriores presentan desventajas. En primer lugar porque no se cumple con la propiedad Identidad, ya que un objeto se compone de muchos otros (roles) con identidad propia. Luego tampoco ofrecen mecanismos que posibiliten la composición de roles. Y por último, en lenguajes fuertemente tipados, existe la necesidad de "inflar" la interfaz (interface bloat) para que el objeto soporte todo el protocolo de todos los posibles roles que pueda actuar.

Gottlob [Gottlob et al. 1996] presenta una variación del pattern Role Object para Smalltalk. Así, por ser un lenguaje dinámicamente tipado se puede construir una jerarquía de rol en forma independiente a la jerarquía de la clase `Core`, solucionando así el problema de *interface bloat*.

En base a la experiencia y resultados anteriores, Kendall [Kendall 1999] propone utilizar lenguajes orientados a objetos combinado con Aspect Oriented Programming (AOP) con el fin de obtener el poder expresivo necesario implementar Roles en AspectJ. Sin embargo, de las cinco estrategias investigadas ninguna logra ofrecer una solución completa y/o completamente práctica.

El lenguaje powerJava introduce roles como un modo de estructurar la interacción de un objeto con otros que invocan sus métodos. Los roles expresan así las posibilidades de interacción que un objeto ofrece a otros. En powerJava, estas posibilidades cambian en función de la clase del que invoca, y los roles mantienen el estado durante la interacción con cierto individuo. Los roles pueden a su vez ser adquiridos por diferentes tipos y clases [Baldoni et al. 2007].

En su trabajo [Baldoni et al. 2007], Baldoni, Boella y van der Torre dejan en evidencia la estricta relación entre las relaciones y el comportamiento de Rol. Esta cercanía entre Relaciones y Roles es también aceptada en diferentes áreas: desde lenguajes de modelado como UML y ER hasta la representación de conocimiento discutida en ontologías y sistemas multi-agentes. [Baldoni et al. 2007].

Baldoni et al coinciden con Kristensen y Kendall respecto de las propiedades que debe cumplir un rol en el modelo y adhieren a la argumentación de Steimann [Steimann 2000] que expone que el rol intrínseco de los roles es ser intermediarios entre las relaciones y los objetos en que ellas participan.

Con este background, los autores de [Baldoni et al. 2007] se propusieron investigar diferentes patterns para implementar en powerJava roles basados en relaciones. Debido a que dejaron fuera de alcance el definir relaciones como construcciones semánticas de primera clase, tuvieron que preocuparse por las diferentes estrategias de representación de relaciones en los lenguajes orientados a objetos estándar. Durante el desarrollo de su trabajo proponen 3 pattens. Mientras que los primeros presentaron desventajas (principalmente debido al acoplamiento entre Rol y Clase, o Rol y Relación) el tercero resolvió estos problemas, permitiendo definir *a)* roles abstractos durante la definición la clase que representa la Relación y *b)* roles que completen e implementen en las Clases de los objetos que pueden cumplir o actuarlos.

5.6.3 Roles y Relaciones

Cuando en el capítulo sobre el Modelo de Comportamiento basado en Relaciones introdujimos la forma en que habíamos concebido dicho modelo, planteamos cómo un objeto altera su comportamiento en función de las relaciones y vínculos en que participa.

En otras palabras, expusimos por qué el comportamiento de un objeto depende de *a)* el rol que dicho objeto cumple[38] en dicha relación y b) los objetos con que se vincula.

Esto coincide parcialmente con el enfoque presentado en [Baldoni et al. 2007] y se acerca aún más al argumento de Whitehurst, quien indica "*el comportamiento de un objeto puede cambiar dependiendo del rol en que actue. Cuando una asociación (vínculo) se forma entre dos instancias el comportamiento de estas instancias es alterado de alguna manera*".

Nuestra noción sobre qué es un comportamiento incluye aquello que comúnmente se considera rol. Es más, en nuestro modelo, los comportamientos de rol se encuentran incluidos dentro de un grupo más amplio que llamamos comportamientos accidentales.

Coincidentemente con el planteo de Steimann [Steimann 2000], nuestra concepción de Rol nació al considerar que su responsabilidad era ser intermediarios entre las relaciones y los objetos en que ellas participan.

Más tarde en nuestro trabajo consideramos diferenciar dos tipos de comportamientos: el esencial del accidental. Aquí encontramos una muy importante similitud con el modelo conceptual de Kristensen, donde existe lo intrínseco a un objeto (instancia de clase) y lo extrínseco (aquello que adquiere a través del rol).

También para nuestra sorpresa -aún habiendo adoptado un espíritu minimalista y simple al implementar el modelo de comportamiento- encontramos que muchas de las propiedades de Rol son soportadas naturalmente por nuestro modelo.

Propiedad	Nivel de soporte Modelo de Comportamiento
Poder de Abstracción	No Soportada actualmente. Nuestro modelo permite organizar jerarquías de comportamientos, pero aún no soporta jerarquía de roles. Esto será introducido cuando también se soporte jerarquías de Relaciones.

[38] O está en condiciones de cumplir.

Agregación / Composición	No soportada actualmente. De todas formas, tal como lo hemos desarrollado al analizar el mecanismos de composición de Traits, consideramos posible incorporarlo en trabajos futuros.
Dependencia (Un rol no puede existir sin el objeto)	Soportada. Esta propiedad apunta a que un Rol sólo sea actuado por un objeto intrínseco. En nuestro modelo un rol es un comportamiento. Este comportamiento pueden adquirirlo todos los objetos que participen de una relación. Sólo el objeto que se vincule a través de una relación de comportamiento pasará a actuar como tal.
Dinamismo	Soportada. Nuestro modelo de comportamiento cumple con los principios de empatía dinámica y por lo tanto, los roles pueden ser agregados o removidos las veces que se deseen a un objeto durante su ciclo de vida.
Identidad	Soportada. Un objeto adquiere comportamiento a través de las relaciones de comportamiento. El ML es responsable de navegar los vínculos en busca de este comportamiento, pero siempre a través del objeto receptor del mensaje. En consecuencia, la identidad del objeto receptor nunca se ve alterada.
Herencia	Deberíamos profundizar la semántica esperada por esta propiedad para comprender el grado de soporte que nuestro modelo aporta a la misma.
Localización (contexto)	Soportada. Los roles se generan automáticamente en función de las relaciones definidas. Por tal motivo todo Rol depende del modelo de relaciones. Podemos entonces suponer que el modelo de relaciones deriva en un modelo natural de roles.
Multiplicidad	No soportada actualmente. En la programación de roles se espera que un objeto pueda cumplir varias veces el mismo rol simultáneamente, y que cada uno de estos roles posea distintos estados. Consideramos que en futuros trabajos es posible lograr esta propiedad. Recordemos que en nuestro modelo los roles se adquieren a través de un vínculo en una relación de comportamiento. Un objeto puede tener varios de estos vínculos al mismo tiempo, y en ellos puede almacenarse la información de estado. Sin embargo, existe aún otro problema por resolver, relacionado con el contexto en que se activará una instancia de rol u otra. Esto puede depender del emisor, del estado del receptor o cualquier otra información de contexto. Esto requerirá una mayor profundidad de análisis y estudio sobre trabajos relacionados.

Tabla 8. Análisis de Propiedades de Rol que satisface el modelo.

Por otra parte, con excepción del completo modelo de Especificación de Rol propuesto en [Browne et al. 2005] y la breve introducción realizada en la investigación de [Baldoni et al. 2007], la literatura no parece dar particular interés a los mecanismos de que dispone un ambiente de roles para especificar cuándo o cómo un objeto adquiere (o libera) cierto rol. En este sentido, nuestro modelo presenta una semántica clara y precisa al definir que un objeto adquiere un comportamiento de rol cuando se encuentra en condiciones de participar en una relación, es decir, cuando se encuentra incluido el dominio de la relación.

Ya concluyendo este apartado, es necesario volver a mencionar que el modelo de comportamiento propuesto en nuestro trabajo aún requiere de mucha investigación y desarrollo. Pero sin embargo, dada la espontaneidad con que llegamos a este resultado, y la forma paradigmática en que se han resuelto muchos de los aspectos planteados en la literatura, vemos en él un auspicioso futuro en este campo.

5.7 Otros áreas de interés y tecnologías

5.7.1 Prolog

5.7.1.1 Breve descripción

Prolog es un lenguaje de programación lógico. Es un lenguaje de propósito general comúnmente asociado con inteligencia artificial y lingüística computacional. Posee un núcleo puramente lógico, llamado "Prolog puro", así como un gran número de características extra lógicas.

Sus raíces se encuentran en la lógica formal, y a diferencia de muchos otros lenguajes de programación, Prolog es declarativo: un programa lógico es expresado en términos de relaciones, y la ejecución llevada adelante por consultas sobre dichas relaciones. Las relaciones son definidas mediante cláusulas [Prolog].

5.7.1.2 Prolog y los lenguajes orientados a objetos

En los últimos años se han realizado diferentes trabajos con el fin de integrar Prolog con lenguajes orientados a objetos.

Así encontramos Logtalk[LogTalk], un lenguaje orientado a objetos que puede utilizar la mayoría de las implementaciones de Prolog como un compilador en back-end. Otro caso es Visual Prolog, también llamado Turbo Prolog, dialecto orientado a objetos de Prolog, considerablemente distinto al Prolog estándar.

Pero también existen trabajos que mediante frameworks permiten integrar a los lenguajes orientados a objetos puros, como Smalltalk y Java, con el potencial de Prolog.

Por ejemplo, en Squeak Map[SqueakMap] existen dos modelos. Por un lado el desarrollado por Mike Teng, llamado Prolog/V y portado a Squeak por Bolot Kerimbaev. Por el otro, el publicado por Satoshi Nishihara en 2008, AOKI Prolog for Morphic World.

En lo que respecta a Java, existen al menos dos trabajos. JPL es un puente bi-direccional entre Java Prolog y SWI-Prolog, que les permite interactuar entre sí. InterProlog es una librería que ofrece un puente entre Java y Prolog, implementado llamadas a predicados/métodos entre ambos lenguajes [Prolog]

5.7.1.3 Prolog y Relaciones

Existen varias razones que durante nuestro trabajo nos pusieron ante la necesidad de analizar una posible integración de Prolog con nuestro modelo de Relaciones.

La primer razón la encontramos en el objeto de estudio, la Relación, ya que ésta es justamente uno de los elementos centrales de Prolog. Sin embargo, descartamos esta primera situación ya que nuestra verdadera meta era reificar el concepto Relación en Smalltalk.

Más tarde, mientras desarrollábamos los elementos y las reglas de deducción volvió a surgir Prolog. En ese punto nos resultaba necesario resolver Transitividad, Refexibidad, Asimetría, Simetría, etc. Al

enfrentarnos a este problema, a pesar de nuestro escaso conocimiento en Prolog, vimos en este lenguaje un posible camino a seguir.

Por último, una posible integración con Prolog volvió a hacerse evidente cuando nos propusimos conceptualizar y modelar la Representación de Conocimiento. Durante esa etapa de nuestro trabajo concluimos que una relación puede mantener y representar su conocimiento de diversas formas y maneras. Fue entonces cuando visualizamos la posibilidad de que una de las implementaciones de **`KnowledgeRepresentation`** soporte en forma nativa la integración con un motor de Prolog.

A nuestro entender, esto último lograría una natural y suave transición entre ambos paradigmas. Se resuelve paradigmáticamente Relaciones en objetos, y se delega la lógica de su conocimiento a Prolog.

Capítulo 6

Conclusión

Nuestra tesis fue concebida al identificar la ausencia de una reificación del concepto “Relación”en los lenguajes orientados a objetos, y ciertos indicios sobre el potencial que -a éstos- podría aportar. Junto a la investigación de trabajos similares, iniciamos el proceso de reificación en Squeak.

La evolución de nuestro trabajo nos llevó a comprender la importancia de contar con elementos y mecanismos que permitan describir las relaciones de cierto domino de problema con un poder expresivo de características similares al propuesto en UML.

Mientras estudiamos el modo de especificar la Relación comprendimos que uno de sus aspectos centrales era el Rol, y quiénes son los objetos que pueden cumplir o participar de dicho Rol. Fue cuando afloró el Dominio de un Rol, inexistente en trabajos similares y sólo reflejado a través de la noción de Clase o Interfaz.

Pero a su vez, y gracias a la libertad de trabajar y madurar libremente este modelo, nos enfrentamos a nuevos problemas. Los modelos estáticos como UML sólo describen la Relación, y al no preocuparse por su implementación, tampoco lo hacen sobre cómo representar su Conocimiento. Nuestra investigación nos llevó a comprender la trascendencia de conceptualizar y modelar el Conocimiento de una Relación, junto a sus diferentes estrategias de representación.

Así, al modelar el Conocimiento surgió espontáneamente la reificación del Vínculo, y más tarde la entidad responsable de producirlo (Productor de Conocimiento). Evidenciamos además que a través del Vínculo es posible reflejar naturalmente mucha información que de otra forma hubiese resultado difícil de representar (ej. fuente, fecha de modificación, semántica).

Más tarde percibimos que un Rol es más que un nombre y un Dominio, y que trae consigo un comportamiento que en gran medida se infiere de la misma Relación, ofreciendo así las veces de nexo o puente entre un objeto y la relación en la que participa.

Dado ese Comportamiento de Rol teníamos que permitir que los objetos del Dominio pudieren adquirirlo. Pensamos en la idea de Relación de comportamiento, distinta a *InstanceOf*. Fue entonces que aparecieron en nuestra mente otras relaciones de comportamiento, no solamente ligadas al comportamiento de Rol (ej. *KnowsAbout*, *StateOf*).

Comprendimos que era necesario replantear el ML, no sólo en términos de las relaciones *InstanceOf* e *Inheritance*, sino también basado en estas nuevas relaciones de comportamiento. Al esbozar el nuevo algoritmo de ML, vimos cuánto más sencillo era expresarlo en términos de una nueva relación global de comportamiento, *BehaveAs*, calculada a partir del resto de las relaciones.

Así fue como arribamos al modelo de Relaciones y Comportamiento reflejado en este informe. En función de éstos logramos mostrar cómo resolver algunas falencias en los lenguajes orientados a objetos –duplicación de información, dispersión del conocimiento y el acoplamiento entre la abstracción y la implementación de una relación-. A su vez pudimos mostrar cómo un modelo de

relaciones ayuda a estandarizar, uniformar y extender no solamente los protocolos de los objetos participantes, sino también el acceso a la descripción del modelo por parte de meta-herramientas.

A pesar de estas bondades, no fue la comprobación de la hipótesis lo que mejor impresión nos causó sobre los resultados obtenidos, sino la naturalidad y fluidez con que el modelo evolucionó –en forma paradigmática- hacia otras áreas o dominios de problema. En este sentido, mientras por un lado vimos cómo facilita el desarrollo e integración con meta-herramientas o frameworks (ej. Interfaz de Usuario o Persistencia), por el otro notamos que disminuye la brecha entre el paradigma de la programación orientada a Roles y la programación orientada a objetos. Vimos que es posible una natural integración con Traits; que facilita una suave transición entre un modelo de objetos y una motor lógico como Prolog; y que reduce significativamente el gap semántico entre un lenguaje de modelado como UML y el lenguaje de programación.

Pero entonces, si por el simple hecho de plantearnos reificar el concepto Relación llegamos a conectarnos y acercarnos espontáneamente a tantas áreas de estudio, dominios de problema, paradigmas y tecnologías, cabe preguntarnos ¿cuáles serán los límites?, ¿qué nuevos resultados podemos esperar de continuar investigando y evolucionando los potenciales de Relaciones en los ambientes orientados a objetos?

6.1 Trabajos Futuros

En el marco de nuestro trabajo nos encontramos en la necesidad de recortar el alcance de muchos aspectos que nos hubiera resultado grato profundizar. En esta sección exponemos las principales ideas sobre el universo de posibles líneas de investigación a seguir.

6.1.1.1 Descripción de una relación

Mientras desarrollamos el modelo de relaciones entendimos la necesidad de contar con la posibilidad de especializar relaciones a partir de otras ya existentes. Así uno podría especializar la relación *FatherChild* y *MotherChild* de la relación *ParentChild*. En este campo, Bierman y Wren [Bierman and Wren 2005] desarrollaron un modelo formal sobre cómo representar una jerarquía de relaciones, el cual podría ser un posible punto de partida.

Al describir un rol explicitamos el dominio de los objetos que pueden participar, el valor nulo, su valor predeterminado y su nombre, delegando en la relación la validación de los elementos en condiciones de participar. En su momento intuímos la noción de Tipo. Más tarde, sobre el final de nuestro trabajo, esta noción fue sostenida por las sugerencias[39] de Fowler en su artículo “When to make a type” [Fowler 2003]. Por lo anterior, creemos relevante investigar sobre las semejanzas entre la descripción de un rol y la definición de tipo.

Otro aspecto interesante a incorporar a la descripción de una relación es poder especificar el elemento predeterminado de un Rol, no sólo en forma explícita sino también dependiente del contexto y momento en que se establezca el vínculo. Así por ejemplo, podrían especificarse en la relación *MentorChild* al padre del niño como Tutor predeterminado.

[39] Fowler sugiere evaluar la construcción de un nuevo tipo a) si el tipo tendrá algún comportamiento especial; b) si ciertas validaciones aplican sus posibles valores -ej. edad mayor a 300; y c) ante una posible evolución del sistema –ej. hoy la edad se mide en años con un Integer y mañana puede medirse en meses-.

Al escribir este informe nos ha resultado práctico y sencillo expresar las reglas y restricciones que guían los vínculos de una relación haciendo referencia al Invariante de la misma. Si bien en nuestro trabajo no hemos reificado este concepto, consideramos que es conveniente hacerlo en próximas evoluciones del modelo.

Aun así, hemos trabajado intensamente para hacer cumplir dicho Invariante. Fue cuando visualizamos una importante dependencia entre cardinalidad, multiplicidad, roles requeridos, roles que no pueden ser nulos, y las claves de un vínculo. Rumbaugh ya lo había anticipado en [Rumbaugh 1987]. Es por ello que consideramos interesante investigar y formalizar las dependencias, e inferencias que deben o pueden hacerse entre ellos.

Muchos trabajos se han realizado también sobre la formalización e interpretación semántica de la relación *WholePart* y *Composición*[Barbier and Henderson-Sellers 1999]. Entendemos que es necesario volcar muchos de los resultados de estos estudios en nuestro modelo.

6.1.1.2 Especificación y Representación del Conocimiento

Uno de los aspectos más importantes e interesantes para continuar la investigación iniciada en este trabajo es el estudio sobre lo que llamamos Representación de Conocimiento. Desde un punto de vista pragmático ofrece un buen punto de partida para mejorar considerablemente la performance del modelo actual, introduciendo diferentes estrategias y algoritmos para acceder a la información. Pero también porque entendemos que en el aspecto más conceptual posee un gran potencial, permitiendo implementaciones implícitas o explicitas, ordenadas, por comprensión o enumeración, con mecanismos de cache, con acceso y/o persistencia a fuentes de información externa, entre otras alternativas.

Siguiendo esta línea de investigación, vemos promisorio realizar una implementación de conocimiento integrada a un motor de Prolog. Esto ofrecería la posibilidad de delegar en este último toda responsabilidad de deducción y manejo lógico (ej: simetría, transitividad), permitiendo al modelo de relaciones preocuparse estrictamente por aspectos descriptivos del dominio de problema.

El modelo de relaciones también plantea la posibilidad de especificar y enchufar (plug) durante la definición de una relación un **`RKnowledge`** (conocimiento) específico, especialmente personalizado y configurado para la relación en cuestión. Si bien este modelo presenta una gran flexibilidad, es necesario madurarlo, principalmente a los efectos de un más claro y sencillo uso por parte de los usuarios del Framework.

Existe a su vez un conjunto de relaciones cuyo conocimiento se deduce del existente en otras relaciones. El modelo propuesto permite dar solución a este problema ya sea con **`RKnowledge`** específicos o definiendo productores de conocimiento particulares para la solución requerida. Sin embargo, también creemos que pueden ofrecerse al usuario mejores soluciones, que le permitan especificar en un lenguaje de más alto nivel cómo derivar una relación en función de otra u otras.

6.1.1.3 Descripción del Comportamiento

Diferentes trabajos también deben ser realizados en el área del modelo de comportamiento.

En primer lugar el de reificar el ML, previa reificación de las relaciones *InstanceOf* y *SubclassOf* en un ambiente basado en clases.

Al hacerlo será necesario evaluar la conveniencia de trasladar las actualmente llamadas variables de instancia al vínculo que se establece entre un objeto y su clase. Esto permitiría que en forma análoga y uniforme todos los comportamientos definidos que expresen estado lo mantengan también en los vínculos que con ellos se establezcan, logrando así la propiedad de Multiplicidad mencionada por Kristensen [Kristensen 1997] en la literatura de Roles.

Otro de los aspectos a estudiar es cómo representar adecuadamente las relaciones entre comportamientos (Ej: *SubclassOf*, *Composition*) para que puedan ser utilizadas por el ML. En cuanto a *Composition*, un punto de partida podría ser analizar la forma en que Traits [Scharli et al. 2003a] resuelve la operación de composición. Creemos interesante que durante este estudio se tenga también en cuenta la forma en que Kristensen describe la propiedad de Agregación/Composición [Kristensen 1997] (en la literatura de roles).

En cuanto a los roles, tal como lo mencionamos en la sección herramientas, es posible especificar el tipo de comportamiento de rol –común- que tendrán los objetos por participar de una relación en base a distintos **RTemplateBehavior**. Estos **RTemplateBehavior** se componen de **RTemplateMethod**. Los **RTemplateMethod** definen en términos abstractos un método, con su correspondiente selector e implementación. Por ejemplo, el selector `add!RoleB!:` aplicado a la perspectiva de `Father` en la relación *FatherChild* se materializará en `addChild:`. Pero existen situaciones donde sería deseable enriquecer el universo de nombres de métodos. Por ejemplo, si en la relación *Implication* deseáramos decir `Implied>>impliedBy:` en lugar de `Implied>>impliedOf:`, o `Implicant>>impliesTo:` en lugar de `Implicant>>implicantOf:`. Para lograr este objetivo comprendimos que es necesario proveer al modelo con información adicional, tal como verbos, preposiciones, que ayuden a expresar con mayor riqueza semántica la relación desde la perspectiva de cada una de los roles.

En cuanto a manejo de eventos, durante el trabajo presentamos el modelo de eventos de una relación. Sin embargo dejamos para trabajos futuros el potenciar el comportamiento de un rol con la capacidad de adaptarlos naturalmente a su perspectiva de la relación. Así, el evento *`LinkAdded:(juan as Father, pedro as Child)`* de la relación *FatherChild* se traduciría desde la perspectiva de *`juan`* al evento *`childAdded:pedro`*.

6.1.1.4 Herramientas

Si bien el conjunto de herramientas desarrolladas nos permiten operar con relaciones, vemos necesario complementarlas con mecanismos de refactoring.

En primer lugar para disponer de un Browser de Relaciones más sofisticado, que no sólo permita definir una nueva relación sino también ofrezca la posibilidad de modificar la estructura de relaciones ya registradas al ambiente y con conocimiento adquirido (vínculos). Aquí se deberá contemplar la detección y el tratamiento de posibles conflictos entre los vínculos existentes y la nueva definición de la relación.

También vemos interesante contar con una herramienta que permita materializar una relación perdida en el código hacia una relación del ambiente. En este sentido creemos que la abstracción **RKnowledgeRepresentation** podría cumplir las veces de adaptador.

Por último, una herramienta que resultaría cuanto menos atractiva sería un editor gráfico de relaciones -de las características de un diagrama de clases de UML- integrado directamente al ambiente, sin intermediación alguna, evitando la necesidad realizar ingenierías directas (o inversas) entre ambas representaciones.

6.1.1.5 Aplicación en otras áreas

A nuestro entender, la utilización de un ambiente de programación orientado a objetos con relaciones facilitaría el desarrollo en diferentes campos y áreas de estudio.

En el campo de investigación sobre el Cambio de Teorías, disponer de relaciones y vínculos reificados, mecanismos de deducción y representación del conocimiento podría ayudar a representar con mayor facilidad una teoría.

También consideramos apropiado conectar nuestro estudio con el progreso realizado en los últimos años en el campo de Ontología en ciencias de la información. Una ontología es una representación formal de un conjunto de conceptos en un dominio y las relaciones entre esos conceptos. La aplicación de ontologías es muy variada, desde inteligencia artificial, Web Semántica, ingeniería de software y arquitectura de la información (como una forma de representación del conocimiento).

6.1.1.6 Alcances y límites de modelar problemas con relaciones

Al trabajar con el modelo de relaciones y comportamiento hemos comprobado su potencial para modelar problemas en los ambientes de programación OO. Sin embargo consideramos necesaria una búsqueda de equilibrio entre las técnicas del estado del arte y relaciones.

Por ello debe abrirse una línea de investigación cuyo objetivo sea identificar las principales directrices sobre cómo enfrentar los problemas y ayudar a responder los siguientes interrogantes: ¿cómo modelo el ejemplo de apuesta de lotería?; si todo lo modeláramos con relaciones, los objetos dejarían de tener atributos y pasarían a ser solo una identidad, ¿cuál sería el límite?, ¿hasta donde las variables de instancia de un objeto dejan de ser variables y se transforman en vínculos de una relación?.

Referencias

[Alpert et al. 1998] Sherman R. Alpert, Kyle Brown & Bobby Woolf, 1998, "The Design Patterns Smalltalk Companion".

[Altman and Tylim 2007] Daniel Altman & Hernán Tylim, 2007, "SSET: Conjuntos con Semántica", Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires. Buenos Aires, Argentina.

[Arnold and Colmes 1996] K. Arnold, J. Gosling & D. Colmes, 1996, "The Java Programming Language", Addison Wesley.

[Baldoni et al. 2007] Matteo Baldoni, Guido Boella & Leender van der Torre, 2007, "Adding Roles to Relationship Patterns", *en WOA 2007, Genova, Italia*.

[Barbier and Henderson-Sellers 1999] Franck Barbier & Brian Henderson-Sellers, 1999, "Object Metamodelling of the Whole-Part Relationship", en proceedings TOOLS 32.

[Bardou and Dony 1995] Daniel Bardou & Christophe Dony, 1995, "Propositions pour un nouveau modele d'objets dans les langages & prototypes ", *en Actes de LMO 1995*.

[Bardou and Dony 1996] Daniel Bardou & Christophe Dony, 1996, "Split Objects: a Disciplined Use of Delegation within Objects", *en OOPSLA 1996*.

[Baumer et al. 1997] D. Baumer, D. Riehle, W. Siberski & M. Wolf. "Role Object", Proceedings of the 4th Annual Conference on the Pattern Languages of Programs, Monticello, Illinois, USA, September 2-5, 1997.

[Bierman and Wren 2005] Gavin Bierman & Alisdair Wren, 2005, "First-class relationships in an object-oriented language", *in ACM. 2005*.

[Black and Sharli 2004] Andrew P. Black & Nathanael Scharli, 2004, "Traits: Tools and Metodology".

[Bobrow and Stefik 1986] D. G. Bobrow & M. Stefik, 1986, "Object-Oriented Programming: Themes and Variations", *in AI Magazine (6)4, pp. 40-62, American Association for Artificial Intelligence. 1986.*

[Booch et al. 1999] G. Booch, J. Rumbaugh & I. Jacobson, 1999, "El Lenguaje Unificado de Modelado", Addison Wesley Iberoamericana.

[Bower and MacGlashan 2000] Bower & McGlashan, 2000, "Twisting the Triad, Object Arts Ltd". www.object-arts.com/papers/TwistingTheTriad.PDF.

[Bracha 1996] G. Bracha, 1996, "The Strongtalk Type System for Smalltalk", *en Proceedings of OOPSLA 1996, Workshop on Extending the Smalltalk Language*. http://bracha.org/nwst.html.

[Browne et al. 2005] James C. Browne, Kevin Kane & Nasim Mahmood, 2005, "Role Based Programming Systems", American Association for Artificial Intellignece 2005.

[Cambio Creencias] Eduardo Fermé, Sobre Cambio de Creencias, Departamento de Computación, UBA. http://www-2.dc.uba.ar/materias/Cambio Creencias/.

[Chambers et al. 1991] Craig Chambers, David Ungar, Bay-Wei Chang & Urs Hölzle, 1991, "Parents are Shared Parts of Objects: Inheritance and Encapsulation in SELF.

[Civello 1993] Franco Civello, 1993, "Roles for composite objects in object-oriented analysis and design", *en ACM 1993.*

[DemeterLaw] http://en.wikipedia.org/wiki/Law_of_Demeter.

[Elmasri and Navathe 2000] Ramez Elmasri & Shamkant B. Navathe, 2000, "Fundamentals of Database Systems", 3th Edition, Addison-Wesley, 2000.

[Encarta 1996] http://es.encarta.msn.com/encyclopedia_961536712/Reificaci%C3%B3n.html.

[ER] http://en.wikipedia.org/wiki/Entity-relationship_model.

[Evans 2004] Eric Evans, 2004, "Domain-Driven Design: Tackling Complexity in the Heart of Software", Addison-Wesley, 2004

[Fowler 2003] Martin Fowler, 2003, "When to make a type", publicado en IEEE Computer Society, IEEE Software.

[Gamma et al. 1995] Eric Gamma, Richard Helm, Ralph Johnson & John Vlissides, 1995, "Design Patterns. Elements of Reusable Object-Oriented Software", Addison Wesley.

[Goldberg and Robson 1983] Adele Goldberg & David Robson, 1983, "Smalltalk-80: The Language and Its Implementation", Addison Wesley.

[Gottlob et al. 1996] G. Gottlob, M. Schrefl, & B. Rock, 1996, "Extending Object-Oriented Systems with Roles", *en ACM Transaction on Information Systems 14, no. 3 (1996): 268-296*.

[Harrison et al. 1995] William Harrison, Harold Ossher & Hafedh Mili, 1995, *en Subjectivity in Object-Oriented Systems Worshop Summary en OOPSLA 1995*.

[J2SE API] Java 2 Platform Standard Edition 5.0 - API Specification. http://java.sun.com/j2se/1.5.0/docs/api/index.html.

[Johnson and Foote 1988] Ralph E. Johnson & Brian Foote, 1988, "Designing Reusable Classes", Department of Computer Science, University of Illinois, Urbana-Champaign, Journal of Object-Oriented Programming.

[Kendall 1999] Elizabeth A. Kendall, 1999, "Role Model Designs and Implementations with Aspect-oriented Programming", *en OOPSLA 99, ACM 99.*

[Kolp 1997] Manuel Kolp, 1997, "A Metaobject Protocol for Reifying Semantic Relationships into Reflective Systems", *In Proc. of 4 th doctoral workshop at the 9th Int. Conf. On Advanced Information Systems Engineering (CAiSE'97), Barcelona, Spain, pp. 89-100, June 1997.*

[Kristensen 1996] B. Kristensen, 1996, “Object-oriented ModeHing with Roles”, *in Proceedings of the 2nd International Conference on Object-oriented Information Systems, Dublin, Ireland, 1996*.

[Kristensen 1997] B. Kristensen, 1997, “Subject Composition by Roles”, *in proceedings of the 4th Intl. Conf. on Object-oriented Information Systems, Brisbane, Australia, 1997*.

[Lieberman 1986] Henry Lieberman, 1986, “Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems”, *in Proceedings of First ACM Conference on Object-Oriented Programming Systems, Languages and Applications, Portland, OR, September 1986.*

[LogTalk] http://logtalk.org.

[Morgan 1858] A. De Morgan, "On the syllogism, part 3", *in Heath, P., ed. (1966) On the syllogism and other logical writings. Routledge. P. 119.*

[Noble 1997] J. Noble, 1997, “Basic relationships patterns”, *In Proccedings of EuroPLOP, 1997*.

[Noble and Grundy 1995] J. Noble and J. Grundy, 1995, “Explicit relationships in object-oriented development”, *in Proccedings of TOOLs, 1995*.

[OCL] Object Constrain Language, Object Management Group. http://www.omg.org/technology/documents/modeling_spec_catalog.htm#OCL

[OMG 2003] OMG, “UML 2.0 OCL Specification”, October 2003. http://www.omg.org/docs/ptc/03-10-14.pdf

[Prolog] http://en.wikipedia.org/wiki/Prolog.

[Reddy 2003] P. V. Reddy, 2003, “Toward Better Logical Models in UML”, *in Journal of Object Technology, vol. 2, no. 5, September-October 2003, pp. 101-121.* http://www.jot.fm/issues/issue_2003_09/article2.

[Reification] http://en.wikipedia.org/wiki/Reification_%28computer_science%29.

[Relation] http://en.wikipedia.org/wiki/Relation_(mathematics)#Formal_definitions.

[Renggli 2006] Lukas Renggli, 2006, “Magritte: Meta-Described Web Application Development”.

[Rozenfarb 2001] Dan Rozenfarb, 2001, “Framework Construction Laboratory”, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires. Buenos Aires, Argentina.

[Ruby] http://www.ruby-lang.org.

[Rumbaugh 1987] Rumbaugh, 1987, “Relations as Semantics Constructs in OO Languages”, *in OOPSLA 1987.*

[Shah et al. 1989] Shah, Rumbaugh, Hamel & Borsari, 1989, "DSM, An Object-Relationship Modeling Language", *in Proceedings OOPSLA89.*

[Scharli et al. 2003a] Nathanael Scharli, Oscar Nierstrasz, Stéphane Ducasse & Andrew P. Black, 2003, "Traits: Composable Units of Behavior", *en ECOOP 2003.*

[Scharli et al. 2003b] Nathanael Scharli, Oscar Nierstrasz, Stéphane Ducasse, Roel Wuyts & Andrew P. Black, 2003, "Traits: The Formal Model".

[Squeak] http://www.squeak.org.

[Stein et al. 1988] L. A. Stein, H. Lieberman, & D. Ungar, 1988, "A Shared View of Sharing: The Treaty of Orlando", Object-Oriented Concepts, Applications and Databases, W. Kim and F. Lochovosky eds, Addison Wesley.

[Steimann 2000] F. Steimann, 2000, "On the representation of roles in object-oriented and conceptual modelling", *en Data and Knowledge Engineering, vol. 35, 2000.*

[Stroustup 1997] Stroustup, 1997, "The C++ Programming Language", Addison Wesley.

[Suscheck and Sandén 2003] Charles A. Suscheck & Bo Sandén, "A Construct for Effectively Implementing Semantic Associations", *in Journal of Object Technology, vol. 2, no. 3, May-June 2003, pp. 101-111*. http://www.jot.fm/issues/issue_2003_05/article1.

[UML] http://en.wikipedia.org/wiki/Unified_Modeling_Language.

[Ungar and Randall 1987] David Ungar & Randall B. Smith, 1987, "SELF: The Power of Simplicity", Sigplan Notices. http://research.sun.com/techrep/1994/abstract-30.html.

[van Rossum 2005] Guido van Rossum, 2005, Python Library Reference. http://docs.python.org/lib/lib.html.

Índice de Tablas

Índice de Figuras