Introducción a Hyper/J y la Separación Multidimensional de Competencias

Juan Manuel Nieto Moreno *
Escuela Técnica Superior de Ingeniería Informática
Universidad de Sevilla, España
nulain@yahoo.es

ABSTRACT
Existen diferentes metodologías de programación que pretenden conseguir una adecuada modularización de las aplicaciones y posibilitar una mejor separación de conceptos que las conforman. Gracias a ellas se pueden capturar los diferentes conceptos que componen una aplicación en entidades bien definidas, de manera apropiada en cada uno de los casos y eliminado las dependencias inherentes entre cada uno de los módulos. De esta forma se consigue razonar mejor sobre los conceptos, se elimina la dispersión del código y las implementaciones resultan más comprensibles, adaptables y reusables. Para referirse a estas tecnologías se habla hoy en día en general de programación orientada a aspectos [1], pero cada una de ellas encamina la solución del problema desde puntos de vista diferentes y por ello reciben otros calificativos como métodos adaptivos, filtros de composición, programación orientada a sujetos o separación multidimensional de competencias, a las que nos referimos en este documento. En el contexto de esta última es donde se sitúa la herramienta de IBM Hyper/J, objeto del estudio del presente documento.

1. INTRODUCCIÓN

IBM Hyper/J [2] es una herramienta que soporta la separación e integración multidimensional de competencias en el lenguaje Java. Con esto facilita la adaptación e integración de componentes Java y mejora la modularización y la remodularización no invasiva de las aplicaciones.

La separación de competencias [3] es un intento de descomposición del software en módulos, donde cada uno de los cuales representa y encapsula un problema particular de interés, que denominamos *competencia*. Los lenguajes OO se basan en la descomposición por clases. Pero además de éstas, existen otro tipo de competencias que no pueden ser encapsuladas de tal forma y cuya implementación, con las soluciones actuales, termina distribuida por toda la jerarquía de clases. Eso se debe a que el único tipo de competencia posible son las clases y por ello, otras características como pueden ser la persistencia, la distribución o el sincronismo –a las que llamaremos aspectos– no tienen una representación adecuada. Nos referimos a cada una de estos tipos de competencias –hasta ahora hemos indicado las clases y los aspectos– como dimensiones. La separación de competencias persigue la descomposición del software de acuerdo con una o más de estas dimensiones.

La mayoría de los formalismos actuales soportan la separación de competencias, pero con la tendencia hacia una única dimensión. La descomposición en clases de la POO facilita la evolución de las estructuras de datos, pues están definidas en una única o pocas clases. Pero, por el contrario, impide, o dificulta, la evolución de sus aspectos, pues éstos implican normalmente a varias clases. Usualmente es necesario tener de manera simultánea diferentes dimensiones de abstracción de conceptos, que además es posible que se solapen e interactúen. Se necesita la modularización respecto a diferentes dimensiones, ya sea, por clases, aspectos, roles o cualquier otro criterio. Este problema se conoce como la tiranía de la descomposición dominante (*tyranny of dominant descomposition* [3]), ya que sólo se dispone de una manera de descomponer el software y que además no deja lugar a otro tipo de descomposición. Ejemplos de esto son las, ya citadas, clases (en los lenguajes OO), las funciones (en lenguajes funcionales) o las reglas (en los sistemas basados en reglas).

2. SEPARACIÓN MULTIDIMENSIONAL

La solución de IBM para la separación multidimensional de competencias (SMC) se llama *hyperspaces* e Hyper/J es la herramienta que los hace posible en Java. En los siguientes apartados se describen éste y otros conceptos relevantes de la terminología de Hyper/J.

2.1 HYPERSPACES

Los *hyperspaces* o hiperespacios permiten la identificación explícita de cualquier dimensión y

* Este documento es parte del proyecto final de carrera de Juan M. Nieto, tutelado por Antonia M. Reina del departamento de Lenguajes y Sistemas Informáticos de la Universidad de Sevilla.

competencia en cualquier fase del ciclo de vida del software; la encapsulación de dichas competencias; la identificación y manejo de las relaciones entre dichas competencias; y la integración de las mismas. Formalmente un hiperespacio se define como una tupla *(U,M,H)* donde *U* es un conjunto de unidades recogidas en el hiperespacio; *M* es la matriz de competencias que simultáneamente organiza las unidades de *U* de acuerdo con las principales competencias; y *H* es un conjunto de hipermódulos que especifica cómo construir componentes con las unidades de *U*. Una unidad puede ser, por ejemplo, un aserto, una declaración, un diagrama de estados, una clase, una interfaz, requisitos de especificación o cualquier otra entidad que pueda ser descrita en cualquier lenguaje. Se distingue entre unidades primitivas (métodos, variables,...) y unidades compuestas (clases, paquetes, diagramas de colaboración...) Actualmente las unidades que soporta Hyper/J son paquetes, interfaces, clases, operaciones y atributos Java.

2.2 LA MATRIZ DE COMPETENCIAS

Las competencias (del inglés *concerns*) se pueden modelar como predicados, *c*, sobre un conjunto de unidades, *U*. Dichos predicados se usan para especificar cuándo una unidad está implicada en una competencia determinada. El conjunto de unidades implicadas en una competencia *c* puede representarse por: $U(c)=\{u \in U/c(u)\}$. Si sus conjuntos de unidades no son disjuntos las competencias pueden solaparse. Para organizar las unidades del hiperespacio, éstas pueden entenderse dispuestas en una matriz multidimensional (*concern matrix*), donde cada eje representa una dimensión de competencias y cada punto en un eje una competencia en esa dimensión. De esta manera quedan explícitamente representadas las dimensiones de interés, las competencias que pertenecen a cada dimensión y las unidades implicadas en cada competencia.

Formalmente dicha matriz de competencias se puede definir sobre un conjunto de unidades *U* como una tupla *(C, D)* donde *C* es un conjunto de competencias y *D* un conjunto de dimensiones de éstas, de modo que se cumple que:

- cada competencia en *C* está en una sola de las dimensiones de *D*; y
- las dimensiones de *D* particionan *U*.

De esta forma, cada unidad está involucrada solamente en una competencia en cada dimensión y sus coordenadas pueden entenderse como sus competencias asociadas. Si todas las unidades de la matriz están asociadas con alguna competencia de una misma dimensión, dicha dimensión particionará de manera natural *U*. En caso de no ser así, por conveniencia, en cada dimensión *d* está definida la competencia nula (*None concern*), N_d, que contiene las unidades que no son de interés para dicha dimensión.

2.3 HYPERSLICES

La matriz de competencias identifica las competencias y organiza las unidades de acuerdo con ellas y las dimensiones. Sin embargo, la matriz no soporta, en sí misma, la encapsulación de las competencias (sin ningún otro mecanismo, el conjunto de unidades no puede manejarse como módulos). Así surgen los *hyperslices* o hipersecciones, conjunto de competencias que son declarativamente completos, lo que significa que deben declarar todo aquello que referencian. La completitud de las declaraciones es importante ya que elimina el acoplamiento entre las hipersecciones. No se necesita dar la definición completa de las declaraciones, sino que se pueden utilizar declaraciones abstractas. En lugar de utilizar referencias a otras hipersecciones, el uso de las declaraciones abstractas favorece la reusabilidad e independencia de las hipersecciones y es vital para limitar el impacto de los cambios. Así un conjunto de unidades puede ser tratado como una hipersección, siempre y cuando sea declarativamente completo, y las competencias pueden ser encapsuladas en hipersecciones. Por tanto, independientemente de las limitaciones del lenguaje y la competencia, es posible sintetizar en una hipersección aquellas unidades necesarias para la competencia (más algunas declaraciones abstractas).

Formalmente las hipersecciones se definen como una competencia declarativamente completa $hs \in C$ (*C* es el conjunto de competencias). Así, si una unidad $u_1 \in hs$ referencia de algún modo a otra $u_2 \in U$ entonces $u_2 \in hs$. Suponer, por ejemplo, que una hipersección *X* contiene una unidad u_1 para su método *x()*, el cual usa una función *y()* de su unidad u_2 definida en la hipersección *Y*. Para hacer a *X* declarativamente completa, deberá contener otra unidad u_{2decl}, la cual declare *y()*, sin necesidad de implementarla, de tal forma que u_1 use u_{2decl} en lugar de u_2. Esto elimina el acoplamiento entre *X* e *Y*, entendiendo que u_{2decl} estará finalmente ligada a una implementación en cualquier otra hipersección.

2.4 HYPERMODULES

Los *hypermodules* o hipermódulos incluyen un conjunto de hipersecciones que se pueden componer entre sí, junto con un conjunto de reglas de composición que especifican como han de hacerlo. Los hipermódulos se pueden anidar, lo que permite modelar con ellos componentes funcionales o incluso aplicaciones completas. Las reglas indican las unidades de las diferentes hipersecciones que describen los mismos conceptos y cómo éstas han de integrarse. La sintaxis general para los hipermódulos en Hyper/J es la siguiente:

```
hypermodule hypermoduleName
hyperslices:
  dimensionName1.concernName1,
  dimensionName2.concernName2,
  ...

relationships:
  mergeByName
| nonCorrespondingMerge
| overrideByName;

reglas específicas
end hypermodule;
```

Las dos secciones fundamentales son las correspondientes a la definición de las hipersecciones que intervienen y la definición de las reglas (generales y específicas) de composición de las mismas.

Formalmente un hipermódulo se describe como una tupla *(HS, CR)* donde *HS* es un conjunto de hipersecciones y *CR* es un conjunto de relaciones de composición. Cada una de las reglas se describe como una tupla *(I, r, f, o)* donde *I* es una tupla de unidades de entrada de las hipersecciones del conjunto *HS*; *r* es una relación de correspondencia que define cómo las unidades en *I* se interrelacionan; *f* es una función de composición $f : (I \times r) \rightarrow U$, que indica cómo componer las unidades en *I* de acuerdo con *r*; y *o* es la unidad e salida que se obtiene usando *f*.

2.5 RELACIONES DE COMPOSICIÓN

Las hipersecciones pueden solaparse y compartir unidades o estar interrelacionadas por una o más relaciones de integración que indican cómo han de combinarse. Podemos identificar dos clases distintas de relaciones: las dependientes del contexto y las independientes del contexto. Las relaciones con solapamiento son un ejemplo de relación independiente del contexto, que existe cuando dos hipersecciones comparten unidades. Las relaciones de integración ejemplifican las dependientes del contexto, pues la relación sólo tiene sentido en un contexto determinado y no es inherente a la definición de los componentes. Otros tipos de relaciones son las generalizaciones, inclusiones y exclusiones. Los hiperespacios permiten la representación de ambos tipos de relaciones y su uso en el análisis e integración.

Una propuesta para las reglas de composición, basada en la programación orientada a sujetos [5], establece que estas reglas han de ser una combinación de una regla concisa y general y una serie de reglas específicas y más detalladas. Las últimas se usan para indicar de manera explícita aquellas excepciones a la regla general o para tratar casos especiales no contemplados en la misma. La aplicación de la regla general puede ser un proceso automatizado, pues se aplica a todo o parte de la composición. Los casos especiales que tratan las reglas específicas serán de los tipos siguientes:

- Emparejar unidades con diferentes nombres que describan el mismo concepto.
- Desemparejar unidades con el mismo nombre que describan conceptos diferentes.
- Mediar entre estructuras de módulos diferentes (como establecer correspondencias entre unidades anidadas a diferentes profundidades en diferentes hipersecciones).

Como se ha explicado, en Hyper/J primero se especifica una estrategia de integración general y después se definen las excepciones, o especializaciones, para casos concretos. Actualmente Hyper/J admite tres estrategias generales ([2]) llamadas:

- **mergeByName** indica que las unidades con el mismo nombre en diferentes hipersecciones se corresponden y que se fusionarán en una nueva unidad.
- **nonCorrespondingMerge** indica que las unidades con el mismo nombre en diferentes hipersecciones no se corresponden y por tanto no se fusionarán. Se utiliza cuando, de manera casual, esto ocurre y sin embargo las unidades no están relacionadas.
- **overrideByName** indica que las unidades con el mismo nombre se corresponden y, en lugar de fusionarse, la última de ellas, sobrescribe a las demás. El orden se define según el orden en el que se hayan declarado las hipersecciones en el hipermódulo (prevalece el último).

La sección de relaciones de cualquier hipermódulo empieza con alguna de estas tres estrategias. Si no fuera suficiente sólo con una de estas reglas, a continuación se detallan las reglas específicas. Las posibilidades que tenemos son las siguientes:

Equate:

```
equateRelationship ::= equate unitKind
unitName [, unitName]* [into newName];
unitKind ::= class | interface | operation |
action | field
```

La relación `equate` indica que los elementos de un conjunto de unidades se corresponderán aún cuando sus nombres no sean los mismos. El campo opcional `into` indica el nombre que se le dará a la nueva unidad compuesta. Se puede aplicar a cualquier clase de unidades, siempre y cuando aquellas que participen de la misma declaración sean del mismo tipo.

Order:

```
orderRelationship ::=
   order unitKind unitName [, unitName]*
(before | after)
   unitKind unitName [, unitName]*;
```

```
unitKind ::= hyperslice | class | interface |
operation | action
```

Cuando se combinan métodos, Hyper/J puede, por defecto, elegir el orden en el que ejecutar los métodos originales. O bien, si el orden es importante, con `order` se puede indicar cómo debe hacerse. Nótese que sólo se definen órdenes parciales, es decir, sólo para aquellas unidades en las que sea importante.

Rename:

```
renameRelationship ::=
  rename unitKind unitName to newUnitName;
unitKind ::= class | interface | operation |
action | field
```

Más que una relación, es una directiva de Hyper/J. Permite cambiar el nombre a una unidad concreta de una hipersección de un hipermódulo. Sólo puede cambiarse el nombre en las hipersecciones de salida, nunca en las de entrada (lo cual tampoco es necesario).

Merge:

```
mergeRelationship ::=
 merge unitKind unitName [, unitName]*
 [into newName];
unitKind ::= class | interface | operation |
action | field
```

Combina el conjunto de unidades especificadas, independientemente de si dichas unidades se correspondan o no según la regla general de combinación. Difiere de `equate` en que ésta no implica que las unidades se combinen, si no que sólo indica que se corresponden, dependiendo de la estrategia general el hecho de que se combinen o no.

NoMerge:

```
noMergeRelationship ::=
  noMerge unitKind unitName [, unitName]* ;
unitKind ::= class | interface | operation |
action | field
```

`NoMerge` tiene el efecto opuesto a `merge` (o `override`). Hace que unidades que se corresponden, no se combinen (o sobrescriban), a pesar de lo que implique la regla general de composición. Se utiliza para los casos excepcionales en los que la regla general no debe aplicarse.

Override:

```
overrideRelationship ::=
  override unitKind unitName [, unitName]*
  with unitKind unitName;
unitKind ::= class | interface | operation |
action
```

Esta relación indica que una unidad sobrescribe al conjunto de unidades indicadas en la declaración, en el sentido descrito para `overrideByName`. Puede aplicarse a cualquier tipo de unidad.

Match:

```
matchRelationship ::=
  match unitKind unitName with "pattern";
```

Se usa para indicar que una unidad debe corresponderse con un conjunto de unidades, que se especifican usando un patrón que hace referencia a los nombres de éstas. Al igual que ocurre con `equate`, con `match` las unidades que haga que se correspondan quedarán sujetas a la regla general de composición. Nótese, que la correspondencia sólo se hace entre unidades del mismo tipo. La sintaxis para los patrones se ilustra con los ejemplos siguientes:

`"foo"` indica solo aquellas unidades de nombre `foo`.
`"foo*"` unidades cuyo nombre comience con `foo`.
`"foo*bar"` unidades cuyo nombre comience con `foo` y termine con `bar`.
`"~foo*"` unidades cuyo nombre no comience con `foo`.
`"~foo"` cualquier unidad excepto aquellas de nombre `foo`.
`"{foo,bar}*"` unidades cuyo nombre comience con `foo` o `bar`.
`"*{foo,bar}"` unidades cuyo nombre termine con `foo` o `bar`.
"{f,~foo}*{~r}" unidades cuyo nombre comience con f, pero no con foo y que no termine con la letra r.

Bracket:

```
bracketRelationship ::=
   bracket [classMatchPattern .]
operationMatchPattern
      [from unitKind unitName [, unitName]* ]
      [before fullyQualifiedMethodName,]
      [after fullyQualifiedMethodName,]
unitKind ::= hyperslice | class | operation |
action
```

Esta relación indica que el comportamiento del conjunto de métodos indicados después de la palabra clave `bracket` debe ser ampliado ejecutando antes o después de su invocación los métodos especificados a continuación de `before` y `after`, respectivamente. Opcionalmente se puede restringir con la declaración `from` el contexto donde esta relación tiene efecto. Así puede restringirse a unidades del tipo `action`, `operation`, `class` o `hyperslice`. Esta relación sólo es aplicable a operaciones. En la actual versión de Hyper/J la única información contextual que puede pasarse a los métodos que se ejecutan antes o después es el nombre de la clase o de la operación ampliada. Para ello, se indica con `$ClassName` y `$OperationName`, entre paréntesis, seguido del nombre de las operaciones a invocar.

Summary function:

```
summaryFunctionRelationship ::=
  set summary function for unitType unitName
  to summaryFunction;
summaryFunction ::=
  external unitName
         | unitType unitName
unitType ::= action | operation
```

Cuando se componen métodos que devuelven valores, el método resultante debe devolver un único valor. Por defecto, Hyper/J devuelve el valor devuelto por el último de los métodos según el orden en el que los ha compuesto. En el caso de que esto no sea lo deseado, Hyper/J permite definir una función que sintetice el valor devuelto por cada una de los métodos compuestos. Esta toma como parámetro, la secuencia de valores devueltos y los usa para generar un único valor, que será el que se devuelva. Usando la palabra clave `external`, pueden usarse funciones que no estén en el hiperespacio definido. Nótese que este tipo de funciones han de ser siempre estáticas.

2.6 LIMITACIONES DE HYPER/J

Con Hyper/J se puede descomponer un programa de acuerdo con diferentes competencias, creando nuevos módulos separados que no interfieran unos con los otros. Posteriormente pueden integrarse para obtener un resultado final o de cualquier fase intermedia. Hyper/J ayuda a definir las interacciones entre las diferentes descomposiciones, proveyendo de una capacidad de composición. Por ejemplo, se puede usar para crear una versión de un sistema software que contenga solamente determinadas características y no otras, incluso cuando el sistema original no fue diseñado con dichas características de manera separada; o puede usarse para extender o adaptar un componente, incluso cuando dicho componente no se ha escrito facilitando los puntos de enlace necesarios.

Hyper/J puede usarse en cualquier fase del ciclo de vida del software, ya sea diseño, implementación, integración, evolución del sistema y reingeniería. Cuando se usa durante el diseño o la implementación, Hyper/J permite separar todos las competencias de importancia desde el principio. Durante la integración del sistema, los mecanismos de integración de Hyper/J pueden usarse para integrar el software que se ha desarrollado de manera separada, incluyendo los componentes reusables. Usado durante la evolución del sistema, los mecanismos de separación de competencias de Hyper/J permiten a los programadores centrarse únicamente en aquellas piezas del sistema relevantes en cada momento, no siendo necesario modificar el código existente. En la fase de reingeniería, se puede utilizar Hyper/J para introducir nuevas descomposiciones sin necesidad de cambios en el código actual.

Referente a las unidades de las que hemos estado hablando, Hyper/J soporta como unidades los paquetes, interfaces, clases, funciones y atributos Java. Soporta una matriz de competencias con esas unidades y permite definir hipersecciones con conjuntos de unidades para después integrarlos en hipermódulos. Hyper/J genera ficheros Java class para todos los hipermódulos definidos, que pueden ser ejecutados, si lo fueren, o usados como piezas de otras aplicaciones.

Las principales limitaciones actuales en la funcionalidad de la herramienta de composición de Hyper/J están detalladas en la sección 4.5 del manual de usuario e instalación de Hyper/J [2] y, por tanto, no se repetirán aquí.

3. EJEMPLO

Para poner en práctica los anteriores conceptos teóricos de Hyper/J se trabajará con un sistema de ejemplo para la gestión de personal. Partimos de un sistema orientado a objetos con dos funcionalidades principales: la gestión de la información del personal (nombre, identificador y cargo en la empresa) y la gestión de lo referente a los salarios. Además, gracias a un método `check()`, se cumple que el número de jefes por empleado es como mucho de 3. Originalmente tendríamos la siguiente jerarquía de clases:

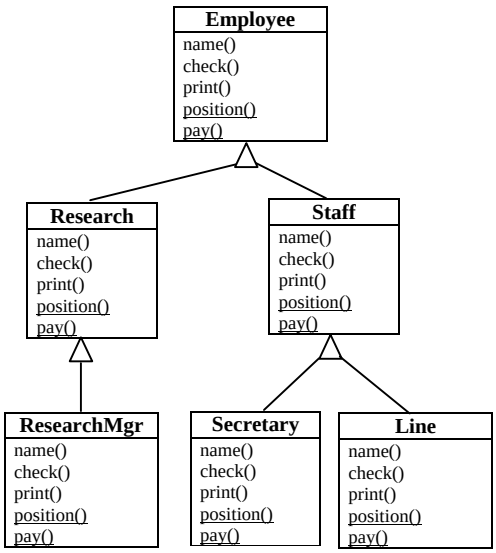

Figura 1 Jerarquía de clases del sistema

Supongamos que en este contexto nos aparecen nuevos requisitos. Por un lado, queremos eliminar la gestión de los salarios del sistema de gestión del personal y, por otro lado, modificar la actual regla referente a los jefes, pasando éstos a ser como mucho uno sólo por empleado. Gracias a Hyper/J podremos llevar a cabo estos cambios sin alterar el actual sistema de modularización y manteniendo las actuales relaciones entre las clases. En Hyper/J tendríamos la siguiente definición:

```
package Personnel:   Feature.Personnel
operation position:  Feature.Payroll
operation pay:       Feature.Payroll
```

Con la anterior definición, por defecto, tendremos que todos los miembros de las clases e interfaces del paquete Java `Personnel` pertenecen a la competencia `Personnel` en la dimensión `Feature`. Por sobre-

escritura, las declaraciones `operation` indican que cualquier método en cualquier clase que se llame `position()` o `pay()` pertenece a la competencia `Payroll` en la dimensión `Feature`. Gráficamente podemos visualizar la matriz de competencias como lo muestra figura 2:

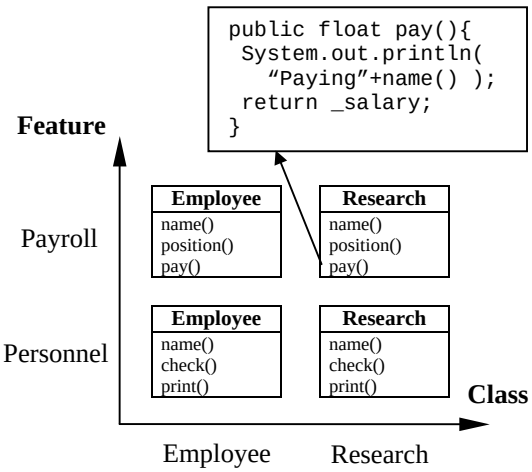

Figura 2 Matriz de comptencias con las dimensiones Feature y Class

En la figura pueden verse las dos dimensiones `Class` y `Feature` y dos hipersecciones, `Personnel` y `Payroll`. Cada hipersección encapsula su propia estructura de clases. Tal como se ha definido, `Payroll` contiene los métodos `position()` y `pay()`, además del método abstracto `name()` ya que éste es invocado desde `pay()`. Como se ha explicado en el apartado Hyperspaces cada hipersección debe ser declarativamente completa y es por ello que es necesario incluir el método abstracto `name()`, pues es referenciado en el método `pay()`. En la otra hipersección, `Personnel`, las clases contienen todos los otros métodos originales. Hyper/J inserta automáticamente una declaración abstracta para cualquier miembro al que se haga referencia, aunque no se de su implementación. De esta forma, las hipersecciones están desacopladas y pueden ser manejadas de manera independiente. `Personnel` es un programa completo, mientras que `Payroll` es un componente Java correcto, a falta de indicar una implementación para el método abstracto `name()`.

A continuación, para componer las hipersecciones usando Hyper/J es necesario crear un hipermódulo que indique las hipersecciones que se desea componer y cómo se llevará a cabo la composición. El siguiente fragmento de código define un hipermódulo con las hipersecciones requeridas:

```
hypermodule PayrollPlusPersonnel
   hyperslices: Payroll, Personnel;
   relationships:
        mergeByName;
end hypermodule
```

Como puede verse, primero se indica el nombre del hipermódulo, luego la lista de hipersecciones a componer y luego el tipo de composición que se llevará a cabo, que en este caso será por nombre. Ello implica que las entidades que en cualquiera de las hipersecciones tengan el mismo nombre, serán las mismas y habrán de fusionarse en una sola. Por ejemplo, `Payroll.Employee` y `Personnel.Employee` tienen el mismo nombre y se combinan para dar una nueva clase en `PayrollPlusPersonnel`. Los métodos concretos en la hipersección `Personnel` implementan los métodos abstractos de `Payroll`.

Otras competencias podrían modularizarse de modo separado. Puede ser el caso de querer establecer ciertas reglas de negocio, como la que se dijo al principio de restringir el número de jefes de un empleado a uno. Así por ejemplo, tendríamos las siguientes declaraciones para el método `check()` de `Employee`:

```
method Personnel.Employee.check:
BusinessRule.ThreeManagers

method SingleManagerPkg.Employee.check:
BusinessRule.OneManager
```

Así aparecería en la matriz de competencias una nueva dimensión `BusinesRule`, como se ve en la figura 3:

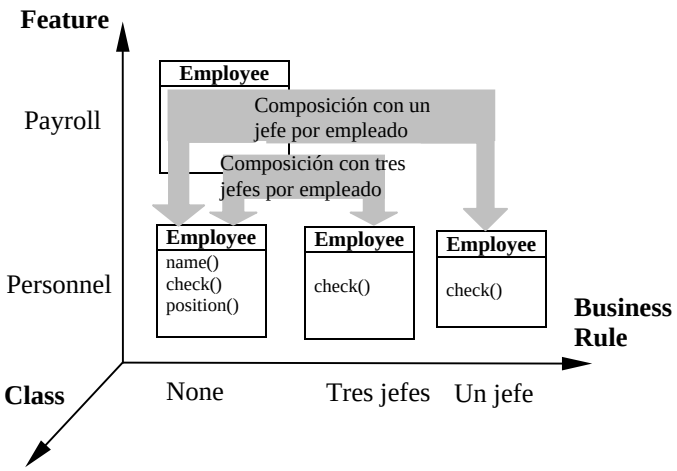

Figura 3 Nueva matriz de competencias

Como se observa en el gráfico, es necesaria la definición de la competencia `None` en las reglas de negocio, de lo cual se encarga automáticamente Hyper/J. `None` contiene todo el código que no tiene nada que ver con las reglas de negocio, pero que, como se explicó en el apartado referente a la matriz de competencias, es necesario incluir. Hyper/J automáticamente inserta una declaración abstracta para el método `check()`. Como resultado el desarrollador puede combinar las reglas de negocio en hipermódulos bien usando `BusinessRule.OneManager` o `BusinessRule.ThreeManagers`.

4. CONCLUSIONES

La filosofía de Hyper/J para soportar la separación de competencias es permitir la integración de módulos, denominados en la terminología *hyperslices*. Cada uno de estos módulos encapsula una competencia específica y se programa en Java estándar, por lo que no se requieren extensiones ni constructores nuevos para el lenguaje. Los *hyperslices* pueden solaparse o existir de manera independiente unos de los otros. Para hacer posible la integración, existen reglas de composición de modo que, en principio se sigue una regla general establecida de composición y después se utilizan reglas específicas, que se refieren a casos particulares.

En Hyper/J, a diferencia de otras implementaciones de metodologías de separación de conceptos, no existe el concepto de un programa base, sino que cada componente de código es independiente y provee de forma completa el código para la unidad particular de descomposición. Esa es la principal ventaja de Hyper/J, y es el hecho de utilizar componentes verdaderamente independientes entre sí, lo cual beneficia la reusabilidad de los mismos (llegando al caso de aplicaciones que se diseñaron por separado y que son capaces de funcionar independientemente).

Una ventaja de Hyper/J es el hecho de trabajar con el bytecode, por lo que no se necesita el código fuente, sino solamente conocer la interfaz (nombres de las clases, funciones...), lo cual se puede obtener fácilmente con compiladores inversos o con la documentación. Sin embargo, la herramienta de la que se dispone está aún en desarrollo, por lo que tiene algunas limitaciones.

El uso de Hyper/J es conveniente cuando se trata de combinar grandes componentes con muchas clases. Para combinar distintos componentes y probar diferentes configuraciones, en lugar de tener que trabajar añadiendo o quitando cada una de las clases, podemos beneficiarnos del uso de las hipersecciones. Éstas agrupan todas las clases y métodos asociados con una competencia específica, por lo que podemos trabajar con éstas combinándolas o eliminándolas según las necesidades, implicando una sola acción, en lugar de tener que manejar muchos módulos pequeños.

5. REFERENCIAS

1. Aspect Oriented Software Development. http://aosd.net.

2. Peri Tarr, Harold Ossher. *Hyper/J™ User and Installation Manual*, IBM Research, 2000.

3. P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. *N degrees of separation: Multi-dimensional separation of concerns*. In Proceedings of the 21st ICSE'99.

4. D. L. Parnas. On the criteria to be used in decomposing systems into modules. Communications of the ACM, December 1972.

5. H. Ossher, M. Kaplan, W. Harrison, A. Katz, V. Kruskal, *Subject-oriented composition rules*, OOPSLA'95, ACM, October 1995, pp. 235-250.

6. Barry R. Pekilis. Multidimensional Separation of Concerns and IBM Hyper/J. January 22, 2002.