Mecanos y Análisis de Conceptos Formales como soporte para la construcción de Frameworks*

Félix Prieto[1], Yania Crespo[2], José Manuel Marqués[1], y Miguel Ángel Laguna[1]

[1] Dpto. de Informática. Unviersidad de Valladolid
{felix,jmmc,mlaguna}@infor.uva.es
[2] Dpto. de Ciencias de la Computación. Universidad de la Habana
yania@infor.uva.es

Resumen La creación y posterior evolución de frameworks es un proceso difícil. Esta dificultad es mayor en los frameworks de dominio, que deben adaptarse rápidamente a los cambiantes requisitos de las áreas de negocio. En este trabajo mostraremos cómo el concepto de mecano puede constituir un soporte adecuado para facilitar las tareas de creación, evolución e instanciación de frameworks. Para ello presentaremos una arquitectura inicial de mecano "bien formado como framework" que deberá ser refinada hasta obtener una gramática detallada. También mostraremos cómo las técnicas del Análisis de Conceptos Formales pueden orientar estos procesos, detectando generalizaciones y sugiriendo reorganizaciones de clases que pueden favorecer su reutilización. Basándonos en esto proponemos un proceso para asistir a la construcción evolutiva de frameworks. En este contexto el concepto de mecano nos proporciona el soporte para propagar las modificaciones necesarias desde el nivel de abstracción en que son detectadas.
Palabras clave: Framework, Mecano, Retículo de Galois, Análisis de Conceptos Formales.

1 Introducción

La reutilización del software ha sido uno de los principales objetivos de la Ingeniería del Software desde sus orígenes. El paradigma Orientado a Objetos se presentó en su momento como la herramienta definitiva para la obtención de este objetivo. Sin embargo, a pesar de que proporciona avances considerables en este sentido, parece ya evidente que las clases son un elemento reutilizable (*asset*) de grano demasiado fino para basar en ellas el proceso de reutilización [19]. Para resolver este problema se han propuesto elementos de grano más grueso, como pueden ser los frameworks [10] o los mecanos [7].

Un framework es un diseño reutilizable de todo o parte de un sistema software descrito por varias jerarquías de herencia de clases, generalmente algunas abstractas, y por las colaboraciones que se establecen entre las instancias de estas clases. La reutilización se produce mediante la instanciación del framework

* Este trabajo ha sido parcialmente financiado por la CICYT(TIC97-0593-C05-05) como parte del proyecto MENHIR

en una aplicación concreta, rellenando los puntos de variabilidad del diseño, los puntos calientes[1] [14], en los que el desarrollador con reutilización incluye la funcionalidad específica de su sistema.

Aunque se han propuesto diferentes clasificaciones para caracterizar los tipos de frameworks [3], [10], todas parecen coincidir en que se pueden distinguir dos tipos de frameworks denominados frameworks de aplicación y frameworks de dominio. Un framework de aplicación encapsula una capa de funcionalidad horizontal que puede ser aplicada en la construcción de una gran variedad de programas. Los frameworks que implementan las interfaces gráficas de usuario representan su paradigma. Por otra parte, un frameworks de dominio implementa una capa de funcionalidad vertical, correspondiéndose con un dominio de aplicación o una línea de producto. Estos frameworks deberán ser los más numerosos, y su evolución deberá ser también la más rápida, pues deben adaptarse a las áreas de negocio para las que están diseñados.

Sin embargo, a pesar de sus esperados beneficios, algunos obstáculos han impedido implantar, de manera exitosa y generalizada, modelos de reutilización basados en frameworks. Las principales dificultades han venido dadas porque éstos son difíciles de construir, no es sencillo aprender a utilizarlos, y ello es especialmente cierto en el caso de los frameworks de dominio.

Por otra parte se define mecano como un conjunto de assets clasificados en diferentes niveles de abstracción y relacionados entre sí, ya sea en el mismo nivel (relaciones intranivel) o en niveles diferentes (relaciones internivel), cumpliéndose la restricción de que exista al menos una relación internivel. Los mecanos, por su propia definición, dan soporte para la trazabilidad entre niveles de abstracción.

En el presente trabajo mostraremos una propuesta de integración de frameworks en el modelo de mecano y la forma en que este modelo, asistido por un repositorio como el mantenido por el grupo GIRO[2], soporta la creación, evolución y reutilización de frameworks. Para ello se analiza en primer lugar la arquitectura de un mecano como un framework y se introducen técnicas basadas en Análisis de Conceptos Formales (ACF) y refactorización, para dar soporte a la creación y evolución de los frameworks. Con las técnicas basadas en ACF se detectan generalizaciones y se sugieren transformaciones de clases, que pueden ser interesantes para incrementar su reusabilidad. El modelo de mecano nos permite propagar las modificaciones realizadas entre los distintos niveles de abstracción en que deben ser aplicadas las transformaciones, mientras que las refactorizaciones[3] se emplean para ejecutar dichos cambios.

En lo que sigue, este documento se organiza del siguiente modo: En la siguiente sección se expone la integración entre frameworks y mecanos. En la sección 3 introduciremos las definiciones y resultados del ACF, además de algunas de sus

[1] Puntos calientes, del inglés *hot spots*, también denominados *hooks*.

[2] "`http://giro.infor.uva.es/`".

[3] Refactorizaciones, del inglés *Refactorings*, son una forma especial de transformación de software Orientado a Objetos que se caracterizan por no depender de la semántica del software a modificar, tener como objetivo refinar diseños y ser realizadas mediante reestructuración y reorganización de clases y agrupaciones.

aplicaciones prácticas. En la sección 4 propondremos un proceso de construcción de frameworks asistido por herramientas basadas en el modelo de mecano, el ACF y las refactorizaciones. Terminaremos la exposición con algunas conclusiones y líneas de trabajo futuro.

2 Frameworks y Mecanos

Los frameworks son aceptados como un asset de grano adecuado para fomentar el proceso de reutilización. Sin embargo, como hemos indicado, su éxito en la práctica ha sido bastante limitado fuera del ámbito de las interfaces gráficas de usuario (frameworks de aplicación), en el que se originaron, y ello a pesar de que en estos días son escasos los sistemas completamente construidos desde cero.

En realidad esto no debería sorprendernos. Si diseñar un sistema es costoso, diseñar un sistema general reutilizable lo es mucho más. Un framework debe encerrar una teoría del dominio del problema y debería ser siempre el resultado de un análisis de dominio, sea explícito u oculto, formal o informal. El diseño de un sistema que cumpla sus requisitos y además encierre la solución para un amplio rango de problemas futuros, es un verdadero reto.

Debido a su coste y complejidad de desarrollo, los frameworks deberán ser construídos solamente cuando se advierte que muchas aplicaciones serán desarrolladas en un dominio específico, permitiendo que la inversión realizada en el desarrollo del framework se amortice al reutilizarlo. En resumen, por todas las razones anteriores, se estima que las situaciones en que es económicamente rentable afrontar la construcción de un framework son aquellas en que se dispone de varias aplicaciones del mismo dominio (o se deben producir en breve plazo) y se espera que se requieran nuevas aplicaciones con cierta frecuencia.

También se admite como cierto que el framework no puede ser considerado como un producto final, sino que debe ser esencialmente evolutivo. Por esa razón se han propuesto varias estrategias [4] para la fabricación de frameworks que parten del desarrollo de varias aplicaciones del dominio, a partir de las cuales se inicia la construcción de un primer framework. Posteriormente, con la información proporcionada por las sucesivas instanciaciones, dicho framework va refinándose y transformándose.

2.1 El papel de los mecanos

Un framework puede ser visto como un asset de grano grueso que aglutina elementos de diferentes niveles de abstracción, lo que nos conduce directamente al concepto de mecano. La descripción conceptual del dominio que subyace en el framework, se representa en el mecano como un asset complejo de análisis, mientras que el propio framework se distribuye entre assets de diseño e implementación, a los que se añaden guías y documentación de instanciación. Los assets complejos están formados por assets de grano más fino relacionados entre sí, y estas relaciones cobran especial importancia debido al carácter eminentemente evolutivo de los frameworks.

Por otra parte, al disponer en el repositorio de las aplicaciones iniciales del dominio como un conjunto de mecanos, la presencia de los assets de análisis, diseño e implementación de las mismas, puede, con las herramientas adecuadas, simplificar notablemente la tarea de abstraer la estructura inicial del framework. Partiendo de los elementos comunes de estas aplicaciones, y gracias a las relaciones internivel mantenidas en los mecanos, podremos recuperar los elementos que reifican las abstracciones que vayan a formar parte del framework. Las potenciales instanciaciones del framework que se consideren reutilizables por sí mismas, también estarán representadas como mecanos.

Con todo ello en el repositorio se almacenarán como mecanos las aplicaciones iniciales del dominio, el framework a que dan lugar junto con las instanciaciones que se consideren reutilizables, y todo ello manteniendo las relaciones existentes entre los diferentes assets.

Para que un repositorio basado en mecanos facilite de modo efectivo la construcción de frameworks a partir de aplicaciones existentes, es imprescindible disponer de herramientas de asistencia. Estas herramientas deben tener un doble papel, el de sugerir cuál debe ser la evolución y el de conseguir realizar las transformaciones necesarias de manera consistente. Para abordar la construcción de estas herramientas necesitamos cubrir tres aspectos fundamentales:

- Definir la estructura precisa de los elementos que formarán parte de los mecanos: los assets de todos los niveles de abstracción y las relaciones semánticas entre ellos. Esto, en última instancia, determina una arquitectura específica de los mecanos como soporte para el desarrollo basado en frameworks.
- Definir métodos de inferencia para indicar las tareas de evolución y adaptación, para construir frameworks e instanciar aplicaciones a partir de éstos.
- Definir y analizar las refactorizaciones que ejecutan estas transformaciones.

2.2 Estructura de los elementos

La arquitectura que se propone para los mecanos como soporte de frameworks se muestra en la figura 1. Esta arquitectura es una representación a grandes rasgos, que deberá ser refinada hasta una estructura final. En [7] se define una gramática de grafos tubo, grafos coloreados en que uno de los colores forma un árbol, para determinar la estructura de un mecano "bien formado". En este caso, a partir de dicha gramática deberemos obtener una definición de mecano "bien formado como framework". De la figura 1 cabe destacar lo siguiente:

- Como assets de análisis aparecen jerarquías de clases de análisis o jerarquías de conceptos asociados con sus propiedades.
- Como assets de diseño se distinguen el diseño del framework propiamente dicho y la guía de instanciación. Para representar los primeros se explora el lenguaje de patrones para la especificación y evolución de frameworks propuesto en [15], aunque actualmente asumimos assets de diseño representados mediante UML.
- Como assets de implementación del framework y sus instancias, se tienen paquetes y clases en un lenguaje de programación orientado a objetos.

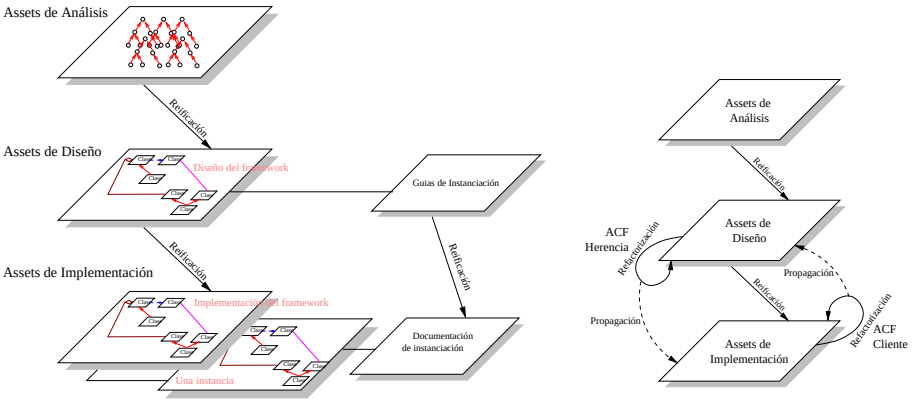

Fig. 1. Mecanos como frameworks y el proceso propuesto

En [2] se ha definido un modelo general con variantes para las reglas y estructura de los assets del nivel de implementación y las relaciones entre ellos. Dados los elementos enumerados anteriormente, aún falta especificar las relaciones semánticas que se establecen entre los diferentes assets y sus restricciones.

Como resultado final de todo lo anterior se obtendrá la gramática detallada de un mecano bien formado, específico para frameworks. De esta forma teniendo una gramática detallada donde no sólo se abarca la estructura general de un mecano como grafo tubo, sino que se reifican los nodos que representan los assets componentes y se define específicamente el significado (y restricciones) de las relaciones que se establecen entre éstos, se puede emprender una labor de definición y desarrollo del soporte automático a diferentes tareas. Las tareas propias del problema de la construcción evolutiva de frameworks son las que se tratan a continuación. Los métodos y herramientas de inferencia para indicar la necesidad de hacer evolucionar la estructura del framework estarán basados en el ACF, mientras que las transformaciones indicadas por este análisis se efectuarán mediante refactorización. En el presente trabajo se abundará en la parte relacionada con el ACF. Para el tema de la refactorización se pueden consultar [2], [4] y [5].

3 Análisis de conceptos formales

El Análisis de Conceptos Formales fue introducido por Wille en [18] y aparece totalmente desarrollado en [6]. Se trata de una técnica matemática que permite poner de manifiesto las abstracciones subyacentes en una tabla de datos, formalmente un contexto, mediante la construcción de un retículo de conceptos, también conocido como retículo de Galois, asociado a ésta. El ACF ha sido utilizado en trabajos relacionados con representación del conocimiento, [9] y en temas relacionados con la Ingeniería del Software, [16]. Comenzaremos por introducir las nociones básicas definidas por Wille.

Definición 1. *Llamaremos* ***contexto formal*** *a una terna,* (G, M, I)*, de conjuntos que verifican* $I \subseteq G \times M$*.*

Informalmente llamaremos a G conjunto de objetos y a M conjunto de atributos. La relación binaria I nos da la incidencia del conjunto de atributos sobre el conjunto de objetos, y nos permite definir un par de aplicaciones:

$$\begin{aligned} \varphi : \mathcal{P}(G) &\longrightarrow \mathcal{P}(M) \\ A &\longmapsto A^{\uparrow} = \{m \in M \mid (g,m) \in I \ \forall g \in A\} \\ \psi : \mathcal{P}(M) &\longrightarrow \mathcal{P}(G) \\ B &\longmapsto B^{\downarrow} = \{g \in G \mid (g,m) \in I \ \forall m \in B\} \end{aligned}$$

Estas dos aplicaciones nos permiten realizar la siguiente definición, en la que reflejamos la noción informal de concepto como conjunto de objetos y atributos que se determinan mutuamente.

Definición 2. *Llamaremos* ***concepto formal*** *a un par* $(A, B) \in \mathcal{P}(G) \times \mathcal{P}(M)$ *que verifica* $A^{\uparrow} = B$ *y* $B^{\downarrow} = A$*. Denotamos el conjunto de los conceptos formales asociado a* (G, M, I) *por* $\mathfrak{G}(G, M, I)$*.*

Sobre $\mathfrak{G}(G, M, I)$ podemos definir una relación de orden parcial dada por, $(A, B) \leq (A', B') \Leftrightarrow A \subseteq A' (\Leftrightarrow B \supseteq B')$ donde $(A, B), (A', B') \in \mathfrak{G}(G, M, I)$. El ***teorema básico de los retículos de conceptos*** nos garantiza entonces que $(\mathfrak{G}(G, M, I), \leq)$ forma un retículo completo al que denominaremos retículo de Galois o retículo de conceptos asociado al contexto, y que pone de manifiesto la estructura existente en los conjuntos de objetos y atributos.

Para representar de una forma clara la información recogida en el retículo de Galois, habitualmente se utiliza el diagrama de Hasse. En este diagrama cada punto representa un concepto formal y cada arco indica una relación de orden entre dos conceptos, donde el mayor de ellos está situado por encima del menor, con la restricción de que no exista ningún concepto intermedio. Cuando cada punto se etiqueta con la descripción del concepto asociado, el diagrama se vuelve difícil de leer. Por ello habitualmente cada objeto se coloca etiquetando el concepto más bajo en que está presente dentro del diagrama, $\gamma(g)$, mientras que los atributos etiquetan el más alto en que aparecen dentro del mismo, $\mu(m)$, como hemos hecho en las figuras 2 y 4.

Hay que resaltar que el retículo y el propio contexto pueden ser reconstruídos a partir de ésta representación puesto que los conjuntos de objetos y atributos aparecen como etiquetas y $(g, m) \in I \Leftrightarrow \gamma(g) \leq \mu(m)$. Además el retículo de Galois nos permite comprobar fácilmente cuándo varios objetos comparten un atributo, pues éste aparecerá en un concepto superior a todos ellos, o dualmente cuándo un objeto posee varios atributos. Así en el diagrama de la figura 2 es evidente a primera vista que el atributo `title` es común a todos los tipos de referencia bibliográfica de BibTeX salvo `MISC`. El mismo diagrama pone en evidencia que todos los tipos de referencia con el atributo `chapter` poseen también el atributo `pages`, lo cual indica que, estructuralmente, ambos atributos aportan la misma información, o que todo tipo de referencia con `editor` dispone también de los atributos `year` y `author`.

El interés práctico del ACF está garantizado por la existencia de diversos algoritmos y herramientas que permiten obtener el retículo de Galois de un contexto formal.

3.1 Análisis de herencia

El ACF nos permite estudiar la estructura dada por la herencia en un conjunto de clases [8]. En este estudio el conjunto de objetos G viene dado por los nombres de las clases, mientras que el conjunto de atributos M está formado por los nombres de sus características. La incidencia $(g, m) \in I$ nos indica que la característica m está presente en la clase g. Los conceptos ahora están definidos por conjuntos de clases y de características que se determinan entre sí.

El retículo de Galois nos permite detectar tanto la definición del mismo método en distintas clases como la posibilidad de definir clases abstractas que encapsulen el comportamiento de un conjunto de clases. Corregir estas situaciones en la jerarquía de herencia nos permitirá eliminar la redundancia y asegurar que la relación de herencia se utiliza básicamente en el sentido de especialización. Todo ello contribuye a aumentar la calidad de la jerarquía de clases [10].

Las redefiniciones de métodos requieren un tratamiento especial, que ponga de manifiesto su carácter incremental. Para ello recurriremos a la noción de contexto multivaluado, en que la tabla de incidencia recoge un conjunto discreto de valores en sus entradas, en este caso los distintos niveles de definición de los métodos. Esta alternativa, propuesta ya en [8], requiere la posterior reducción del contexto multivaluado mediante la introducción de una escala, que en este caso viene dada por la relación de orden parcial que se establece entre las distintas versiones del método.

Para ilustrar el potencial de estas técnicas consideremos un imaginario conjunto de clases formado por los tipos de referencia bibliográfica utilizados en BibTeX, cuyos atributos vienen dados por los campos de información obligatorios en cada tipo de referencia. Podemos construir una matriz de incidencia como la recogida en la figura 2, de la que se deduce el retículo de Galois reflejado en el diagrama de Hasse de la misma figura, sobre el que podemos hacer algunas consideraciones:

- La aparición de un nodo, que representa una clase, por debajo del que representa a otra, $\gamma(g_1) \leq \gamma(g_2)$, nos indica que todas las características de g_2 están presentes en g_1, por lo que podríamos considerar a la primera clase heredera de la segunda.
- Existen nodos que representan varias clases del conjunto original, y con ello ponen de manifiesto que, con la información recogida en el contexto, las clases son equivalentes.
- Algunos de los nodos no corresponden a clase alguna del conjunto original, pero su presencia proporciona un punto de entrada único para una característica presente en la jerarquía de herencia.
- Otros nodos que no corresponden a clase o característica alguna del conjunto original pueden constituir una abstracción útil, en el sentido de que recogen los ancestros comunes a varias clases.

	author	booktitle	chapter	editor	institution	journal	note	pages	publisher	shool	title	year
ARTICLE	×					×					×	×
BOOK	×			×					×		×	×
BOOKLET											×	
INBOOK	×		×	×				×	×		×	×
INCOLLECTION	×	×							×		×	×
INPROCEEDINGS	×	×									×	×
MANUAL											×	
MASTERTHESIS	×									×	×	×
MISC												
PHDTHESIS	×									×	×	×
PROCEEDINGS											×	×
TECHREPORT	×				×						×	×
UNPUBLISHED	×						×				×	

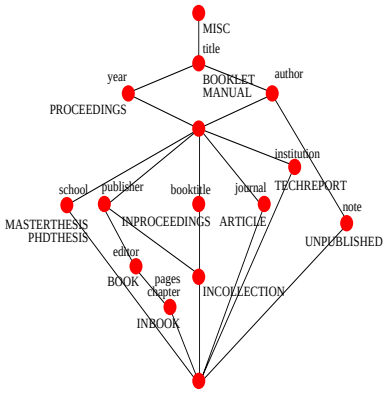

Fig. 2. Tabla de incidencia para el contexto formal extraído de los tipos de referencias bibliográficas de BibTEX y el diagrama de Hasse su retículo de Galois.

3.2 Análisis de colaboraciones entre clases

El ACF nos permite analizar y estructurar, además de la herencia, la información relativa a las relaciones de cliente que se establecen entre un conjunto de clases tal y como aparece en [17]. Para este tipo de análisis la elaboración de la tabla de datos inicial requiere más información y es más compleja que la anterior. Hay que resaltar que lo que queremos analizar en este caso no son tan solo las colaboraciones entre clases, sino también qué características en concreto de las clases proveedoras son utilizadas por las clases clientes y en qué modo.

De manera informal podemos decir que en este caso el conjunto de objetos G viene dado por las entidades[4] presentes en las clases a analizar. No se incluyen las entidades cuyo tipo viene dado por clases que no van a ser analizadas. El conjunto de los atributos M está formado por las características de las clases. Ahora la incidencia $(g, m) \in I$ pretende reflejar que la entidad g necesita disponer de la característica m.

Para realizar este análisis, a diferencia de lo que ocurría en el anterior, necesitamos la información contenida en el código fuente de las clases, pues de otro modo no podríamos detectar la presencia de relaciones de cliente provocadas por elementos de implementación como las variables locales a los métodos. Precisamente por ello este análisis debe variar ligeramente en función del lenguaje de programación utilizado en cada caso. Nuestra intención en este trabajo es fijar las pautas generales necesarias para un lenguaje de programación orientado a objetos[5], para el que supondremos que existe una entidad predefinida, denotada por `Current`, que expresa la autorreferencia de los objetos. Los detalles propios

[4] Una entidad es un nombre en el texto de una clase que denota un objeto en tiempo de ejecución, esto es, un atributo, una variable local a un método, un argumento formal de un método o el resultado de una función.

[5] Aunque la terminología utilizada conduce a la notación de Eiffel.

de cada lenguaje concreto se tendrán en cuenta en la fase de implementación de los prototipos.

En cualquier caso, determinar que la característica m necesita ser definida en la clase base de la entidad g no es una tarea fácil. Si en el código nos encontramos con una llamada como `g.m` la relación es evidente, sin embargo esa no es la única situación que debe ser reflejada en la tabla. Los mensajes no cualificados que aparecen en un método pueden ser considerados como cualificados por la entidad que recibió el mensaje que activó el método, de modo que debemos determinar qué métodos pueden ser invocados desde qué entidades y las incidencias determinadas por ello en la tabla.

```
class DIA
creation make1,make2
feature
 hora,minuto,segundo,
 dia,mes,anno:INTEGER
 make1(h,m,s:INTEGER) is
  do
   hora:=h; minuto:=m
   segundo:=s
  end -- make1
 make2(d,m,a:INTEGER) is
  do
   dia:=d; mes:=m; anno:=a
  end -- make2
 muestra1 is
  do
   print(hora)
   print(minuto)
   print(segundo)
  end
 muestra2 is
  do
   print(dia);print(mes)
   print(anno)
  end
end -- Class DIA
```

```
class AP
creation make
feature
 c1:CLIENTE1
 c2:CLIENTE2
 make is
  do
   !!c1.make(1,1,1,0)
   !!c2.make(1,1,1,0)
   c1.terminar(1,1,1)
   c2.terminar(1,1,1)
  end
end -- Class AP
```

```
class CLIENTE1
creation make
feature
 inicio,fin:DIA
 dato:REAL
 cerrado:BOOLEAN
 make(h,m,s:INTEGER;
     r:REAL) is
  do
   !!inicio.make1(h,m,s)
   dato:=r
  end
 termina(h,m,s:INTEGER) is
  do
   !!fin.make2(h,m,s)
   fin.muestra1
   cerrado:=False
  end
end-- Class CLIENTE1
```

```
class CLIENTE2
creation make
feature
 inicio,fin:DIA
 dato:REAL
 cerrado:BOOLEAN
 make(d,m,a:INTEGER;
     r:REAL) is
  do
   !!inicio.make2(d,m,a)
   dato:=r
  end
 termina(d,m,a:INTEGER) is
  do
   !!fin.make2(d,m,a)
   cerrado:=False
  end
end-- Class CLIENTE2
```

Fig. 3. Código de las clases objeto del análisis

Para conseguir este objetivo podemos añadir a la matriz de incidencia una fila extra por cada uno de los métodos definidos en las clases. Estas filas artificiales representan características utilizadas a través del `Current` en dicho método. Si añadimos reglas que obliguen a reflejar la incidencia $(g, m) \in I$ también cuando el método m sea invocado por una entidad g' que puede ser referenciada por g, conseguiremos reflejar la utilización de servicios a través de la entidad g que, de otro modo, quedaba oculta. Para obtener la información de que una entidad g' puede ser referenciada por g necesitaremos hacer un análisis de las conexiones que ocurren en el texto de las clases.

De un modo en cierto sentido dual, también debemos considerar implicaciones que garanticen la coherencia de los resultados desde el punto de vista de las características de las clases. Así si m' es producto de la redefinición de m debemos considerar que las entidades que deben conocer m' también deben conocer m,

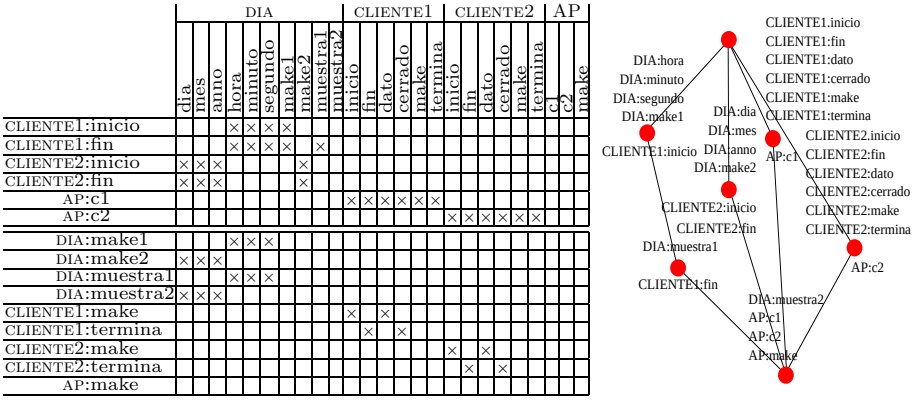

Fig. 4. Contexto asociado a las clases recogidas en la figura 4 y retículo asociado

pues en otro caso podríamos obtener resultados que nos inclinasen a considerar superfluo m cuando en realidad puede tratarse de un método importante para la definición incremental de m'.

Una vez que hemos construido la matriz de incidencia, las filas correspondientes a las entidades `Current` pueden ser descartadas, puesto que su información estará ya contenida en las filas relativa a otras entidades de las clases.

Con todas estas consideraciones el retículo de Galois que podemos construir a partir de la tabla de incidencia descrita nos permitirá obtener interesantes conclusiones sobre las relaciones de cliente que se establecen en el conjunto de clases. Así podremos detectar la presencia de características que no están siendo utilizadas por las instancias de la clase en que fueron definidas, y por ello pueden ser desplazadas hacia abajo en la jerarquía de herencia. También se pondrá de manifiesto la existencia de clases cuya funcionalidad puede ser repartida entre clases más pequeñas, puesto que las entidades que solicitan sus servicios lo hacen sistemáticamente a un subconjunto de ellos.

La detección de este tipo de situaciones supone la aplicación de refactorizaciones que deben ejecutar la evolución del conjunto de clases para aumentar su potencial de reutilización, tal y como se indica en [10] o [11]. En las figuras 3 y 4 se recoge un ejemplo de la utilización de esta técnica que ilustra su capacidad para estructurar la información sobre la relación de cliente. Sobre él podemos resaltar:

- La aparición de una característica por encima de una entidad, $\mu(m) \geq \gamma(g)$, indica que la característica m está siendo utilizada a través de la entidad g.
- La aparición de entidades con la misma clase base en ramas distintas del retículo nos indica la posibilidad de fraccionar dicha clase. Un claro ejemplo de esto es la clase DIA.
- Las características que etiquetan el concepto más bajo de este retículo, como `muestra2`, no están siendo utilizadas por ninguna entidad. La clase AP es un caso especial, puesto que ninguna clase es cliente de ella.

4 Proceso de construcción de frameworks

El ACF se perfila como la base para una buena herramienta de asistencia a la construcción de frameworks, a partir de un conjunto de aplicaciones del dominio, y para guiar su posterior evolución. Las técnicas presentadas en la sección anterior pueden ser útiles en varios sentidos:

Unificar las jerarquías de herencia presentes en las soluciones de problemas del dominio de las que partimos. El estudio del conjunto de clases, que forman parte de los assets de análisis y diseño de los mecanos que representan a las aplicaciones de partida mediante la técnica basada en herencia permite poner de manifiesto la existencia de clases que comparten el mismo conjunto de características. Cuando no sea posible unificar varias clases, el mismo tipo de análisis nos ayudará a poner de manifiesto las abstracciones comunes a las mismas, sirviéndonos además de índice sobre la idoneidad de las jerarquías de herencia.

Estudiar las relaciones de cliente existentes a partir de las assets de implementación de los mecanos que representan a las aplicaciones de partida, sugiriendo la posibilidad de eliminar características, mover su definición entre clases, o fraccionar una clase en clases más pequeñas y reutilizables.

El proceso de construcción de frameworks que se propone debe comenzar por el almacenamiento de algunas aplicaciones del dominio en un repositorio como el soportado por el grupo GIRO, estructurándolas como un conjunto de mecanos. El resultado final de todo el proceso es también un mecano que constituye un framework y posteriormente se pueden obtener otros mecanos fruto de instanciaciones de dicho framework. Las fases en que se divide el proceso son las siguientes:

1. Recopilar el conjunto de clases del dominio del problema que participan en la solución de aplicaciones de la línea de producto, a partir de los documentos de análisis y diseño disponibles. Esta recuperación se puede ver facilitada si se inicia la búsqueda desde el nivel de requisitos dentro del repositorio GIRO. Lo que nos interesa en esta primera fase son los elementos de análisis y diseño de estas aplicaciones, porque en ellos está contenida toda la información que necesitaremos en los tres siguientes pasos: Los nombres de las clases y sus características.
2. Normalizar los nombres de las características de las clases. El ACF nos sugerirá la posibilidad de unificar varias clases cuando compartan las mismas características, pero su único índice sobre la igualdad entre ellas viene dado por la información que le proporcione en este momento quien interactúe con el repositorio.
3. Determinar un orden parcial sobre las características que, tras la fase anterior, mantienen el mismo nombre. Este orden parcial debe reflejar la especialización progresiva de las características en la jerarquía de herencia, y será respetado por el algoritmo de ACF.
4. Aplicar el ACF para estudiar las jerarquías de herencia a nivel de análisis y diseño obtenidas en los pasos anteriores.

5. Una vez aplicadas las refactorizaciones para ejecutar los cambios que se consideren adecuados de entre los sugeridos por el paso anterior, propagar las transformaciones al nivel de implementación, obteniendo el código asociado con las clases analizadas mediante las relaciones internivel dadas por la estructura de mecano.
6. Aplicar el ACF al conjunto de clases de implementación obtenido en el apartado anterior para estudiar las relaciones de cliente que se establecen entre las clases objeto de estudio.
7. Una vez aplicadas las refactorizaciones para ejecutar los cambios que se consideren adecuados de entre los indicados por el paso anterior, propagar las transformaciones a los diseños asociados mediante las relaciones internivel.

Para que este esquema, ilustrado en la figura 1, sea plenamente operativo será imprescindible disponer de una herramienta versátil de obtención del retículo de Galois asociado a un contexto formal, así como de herramientas capaces de obtener el contexto formal a partir de la información contenida en el repositorio GIRO. En este sentido podemos basarnos en los trabajos de análisis de código EIFFEL y JAVA, desarrollados para los prototipos construidos durante la realización de [2], además de en la herramienta desarrollada en [1] para el análisis de diagramas UML realizados con Rational Rose.

5 Conclusiones y trabajo futuro

En el presente trabajo hemos mostrado una arquitectura inicial para el soporte a la creación, evolución e instanciación de frameworks mediante mecanos. Este enfoque permite además relacionar dos técnicas de inferencia de refactorizaciones basadas en ACF, que son aplicables a distintos niveles de abstracción, pero que pueden ser propagadas al resto del framework gracias a las relaciones internivel de los mecanos. Este enfoque nos permite proponer una estrategia de construcción de frameworks por generalización a partir de unas cuantas aplicaciones del dominio, cuyos patrones comunes serán detectados con ayuda del ACF.

La continuidad de este trabajo pasa por definir una gramática detallada de mecano "bien formado como framework" que permita abordar la definición y desarrollo de las herramientas de soporte automático a las diferentes tareas de creación y evolución de frameworks.

Estas herramientas deberán, en primer lugar, soportar la inferencia de las refactorizaciones. En este sentido las técnicas de ACF requieren de una captura de datos previa, para la que podemos apoyarnos en los trabajos sobre el análisis de código EIFFEL, JAVA y de diagramas UML descritos con Rational Rose que hemos desarrollado previamente. Además deberemos abordar la aplicación de las refactorizaciones inferidas mediante las técnicas anteriores. Para ello podremos basarnos en los trabajos que se describen en [2], [12], y [13].

La experimentación con todas estas herramientas se verá facilitada por la existencia del repositorio GIRO en el que disponemos ya de un conjunto de aplicaciones sobre tratamiento digital de imagen o gestión universitaria.

Referencias

1. F. Carretero y J. Quintana. Construcción de una herramienta para la obtención de métricas en assets de diseño. Proyecto de fin de carrera. dirigido por Manso, E., Universidad de Valladolid, Septiembre 1999.
2. Y. Crespo. *Incremento del potencial de reutilización del software mediante refactorización*. Tesis doctoral, Departamento de Informática. Universidad de Valladolid., Julio 2000.
3. M. Fayad y D.C. Schmidt. Object-Oriented application frameworks. *Communications of the ACM*, 40(10):32–38, Octubre 1997.
4. B. Foote y W. Opdyke. Lifecycle and refactoring patterns that support evolution and reuse. In *Proceedings of Pattern Languages of Programs PLoP'94, Monticello, Illinois*, Agosto 1994.
5. M. Fowler, K. Beck, J. Brant, W. Opdyke, y D. Roberts. *Refactoring : Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley, 1999.
6. B. Ganter y R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer, 1999.
7. F. García. *Modelo de reutilización soportado por estructuras complejas de reutilización denominadas mecanos*. Tesis doctoral, Departamento de Informática y Automática. Universidad de Salamanca, 2000.
8. R. Godin y H. Mili. Building and maintaining analysis-level class hierarchies using Galois lattices. *Proceedings of OOPSLA'93*, pág. 349–410, 1993.
9. R. Godin, G. Mineau, R. Missaoui, y H. Mili. Méthodes de clasification conceptuelle basées sur les treillis de galois et applications. *Revue d'Intelligence Artificielle*, 9(2):105–137, 1995.
10. R. E. Johnson y B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.
11. R.E. Johnson y W.F. Opdyke. Refactoring and aggregation.(invited paper). In S. Nishio y A. Yonezawa, editores, *Object Technologies for Advanced Software. Proceedings of the 1st JSSST International Symposium*, pág. 264–278. Springer-Verlag, LNCS 742, 1993.
12. I. Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *Proceedings of the Conference ACM SIGPLAN OOPSLA'96*, pág. 235–250. :31(10), Octubre 1996.
13. W.F. Opdyke. *Refactoring Object-Oriented Frameworks*. Tesis doctoral, Department of Computer Science, University of Illinois at Urbana-Champaign, 1992. Also Technical Report UIUCDCS-R-92-1759.
14. W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison Wesley, Wokingham, 1995.
15. D. Roberts y R. Johnson. Evolving frameworks: A pattern language for developing Object-Oriented frameworks. http://st-www.cs.uiuc.edu/~droberts/evolve.html, 1996.
16. M. Siff y T. Reps. Identifying modules via concept analysis. *IEEE Transactions on Software Engineering*, 25, Diciembre 1999.
17. G. Snelting y F. Tip. Reengineering class hierarchies using concept analysis. *ACM Transactions on Software Engineering and Methodology*, 5(2):146–189, Abril 1996.
18. R. Wille. Restructuring lattice theory: An approach based on hierarchies of concepts. In *Ordered Sets*, pág. 445–470. Reidel, Dordrecht–Boston, 1982.
19. R. J. Wirfs-Brock y R. E. Johnson. Surveying current research in Object-Oriented design. *CACM*, 33(9):105–124, Septiembre 1990.