5. Responsabilidades

Las responsabilidades, como ya se comentó en la introducción, son tareas asignadas a los distintos elementos que componen el software, que se detectan en la fase de especificación y se tratan en la fase de diseño del software.

En este proyecto se ha escogido un conjunto representativo de responsabilidades, de esta forma se podrá incrementar su número tomando éstas como ejemplo en la mayoría de los casos. No obstante se dedicará un apartado de este capítulo a explicar el proceso de implementación de las responsabilidades.

Las responsabilidades tratadas son:

- Clave. Las instancias de una clase pueden tener un identificador único definido por el desarrollador. La unicidad de la clave debe ser preservada por el programa. Aunque esta responsabilidad sea en realidad una restricción textual, debido a la solución que se a tomado para detectarla la trataremos de forma especial.

- Restricciones textuales. Todos los elementos de un modelo pueden tener especificadas restricciones textuales, cada una de éstas desemboca en una responsabilidad.

- Cardinalidades. En *UML* los extremos de las asociaciones tienen una cardinalidad asociada, lo que en algunos casos puede suponer una responsabilidad.

En general para poder detectar correctamente una responsabilidad se deben establecer unas consideraciones previas, lo que ayuda a acotar el problema y nos permite tener una visión general del mismo. Después de tener definido el marco de trabajo, hay que pensar qué estrategias podemos seguir para solucionar el problema.

5.1 Clave

Si dada una clase, es posible encontrar un conjunto de atributos tales que para toda las instancias de esa clase no existen otras con los mismos valores en dichos atributos, entonces este conjunto se llama clave. Hay tres tipos de claves: la clave propiamente dicha, claves alternativas y claves débiles.

Puesto que un conjunto de atributos que incluya una clave es también una clave, consideraremos sólo los conjuntos mínimos de atributos que de por sí son clave. Uno de estos conjuntos es la **clave propiamente dicha** de la clase.

Si existen dos o más conjuntos de atributos que son clave, y suponiendo que no mantienen una relación de inclusión, entonces los que no son clave propia de la clase serán **claves alternativas de la clase.**

A veces para una clase no existe un conjunto de atributos clave pero sucede que la composición de la clave de otra clase (la clase “fuerte”) con

un conjunto de atributos, la **clave débil**, de la clase actual (la clase "débil") definen una clave.

Los distintos conjuntos de atributos que forman una clave, deberán ser indicados por el diseñador.

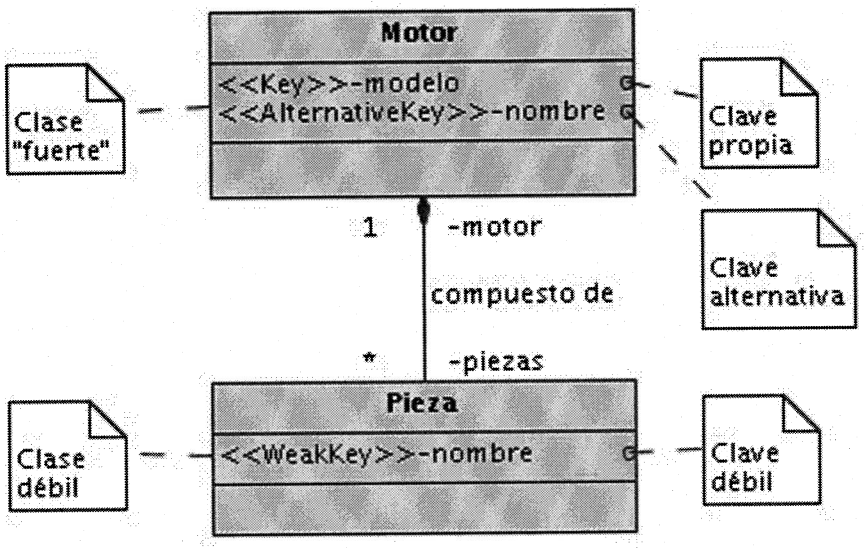


Fig. 1: Ejemplo de los distintos tipos de clave.

5.1.1 Consideraciones previas

- Elemento responsable: Clase
- Elemento donde se detecta la responsabilidad: Atributo
- Una clave puede estar formada por más de un atributo.
- Una clase con clave puede tener clave o clave débil, pero no ambas a la vez.

- Para definir una clave alternativa previamente hay que definir una clave.

- Sólo se permite una asociación de tipo composición en las clases débiles.

- La clase asociada por composición ha de ser otra clase débil, o bien una clase con clave.

- La última clase asociada por composición ha de ser una clase con clave. Esto evita ciclos de clases débiles.

5.1.2 Estrategias

- Crear una nota/comentario que indique las claves de una clase.

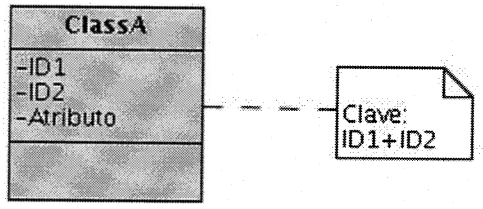


Fig. 2: Nota.

Tiene dos desventajas. Desde un punto de vista de implementación, no existe en el metamodelo un elemento que represente las notas, y desde un punto de vista de usabilidad, el diseñador tendrá que añadir una nota por cada clase que tenga clave, lo que en un diagrama grande podría ser un problema.

- Usando *stereotypes* a nivel de clase.

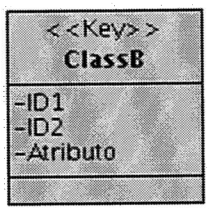


Fig. 3: Stereotype en clase.

En cuanto a la implementación sólo necesitaríamos tener definidos los stereotypes por cada tipo de clave. Y el diseñador sólo tendría que marcar las clases que tengan clave. El problema de esta solución es que perdemos información, no sabemos qué atributos forman la clave. Sólo podemos saber que en esa clase hay una clave. Por otro lado no se sabría cuántas claves alternativas hay.

- Usando *stereotypes* a nivel de atributo.

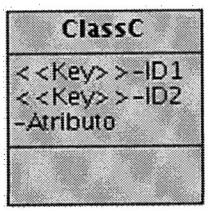


Fig. 4: Stereotype en atributo.

La parte de implementación merece los mismos comentarios que en el caso anterior. El diseñador tendrá que marcar los atributos que formen la clave propia con el *stereotype <<Key>>*, los que forman la clave alternativa con el *stereotype <<AlternativeKey>>* y en las clases débiles un conjunto con el *stereotype <<WeakKey>>*. El

único inconveniente de esta solución es que sólo podemos definir una clave alternativa (en realidad, raramente tendremos más de una clave alternativa), pero esta limitación podría solucionarse añadiendo nuevos *stereotypes*.

- Usando *tagged values.*

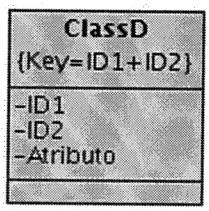


Fig. 5: Tagged value.

A diferencia de las notas, los *tagged values* sí pueden ser tratados por *AndroMDA*. El problema de esta solución es que el diseñador deberá introducir manualmente los nombres de atributos y conocer una sintaxis especifica, ambas cosas son fuentes de errores.

- Usando invariantes de clase expresados en OCL.

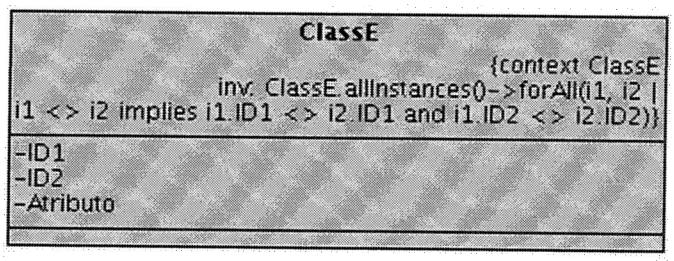


Fig. 6: Invariante de clase.

Esta solución es demasiado complicada para una tarea habitual, en el caso de clases débiles aun se complicaría más.

Después de tener en cuenta todas las posibilidades, la opción de usar stereotypes a nivel de atributo parece la más adecuada. No es demasiado costosa para el diseñador y nos da toda la información que necesitamos.

Esta responsabilidad servirá como ejemplo de responsabilidades donde es necesario declarar un *stereotype* para poder detectarla. A continuación se detallan los pasos a seguir en un caso similar.

5.1.3 Implementación de la responsabilidad

1. Crear el metafacade del elemento responsable.

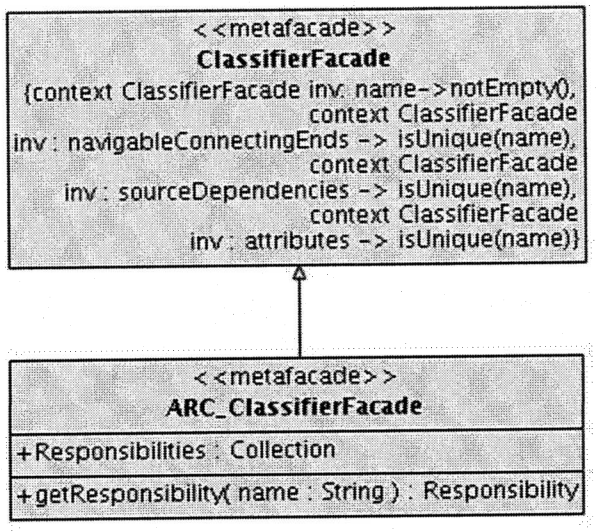
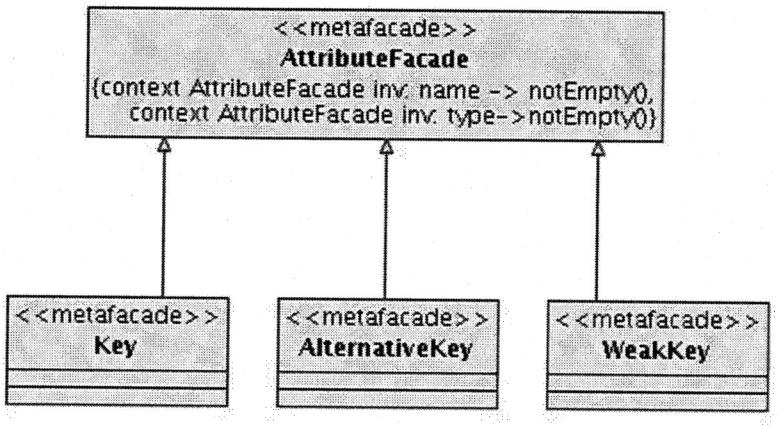


Fig. 7: Metafacade responsable de claves.

Ya que en este caso el elemento responsable es la clase, hay que crear un *metafacade* que extienda el *metafacade* de clase genérico. Éste contendrá una colección de responsabilidades, que serán

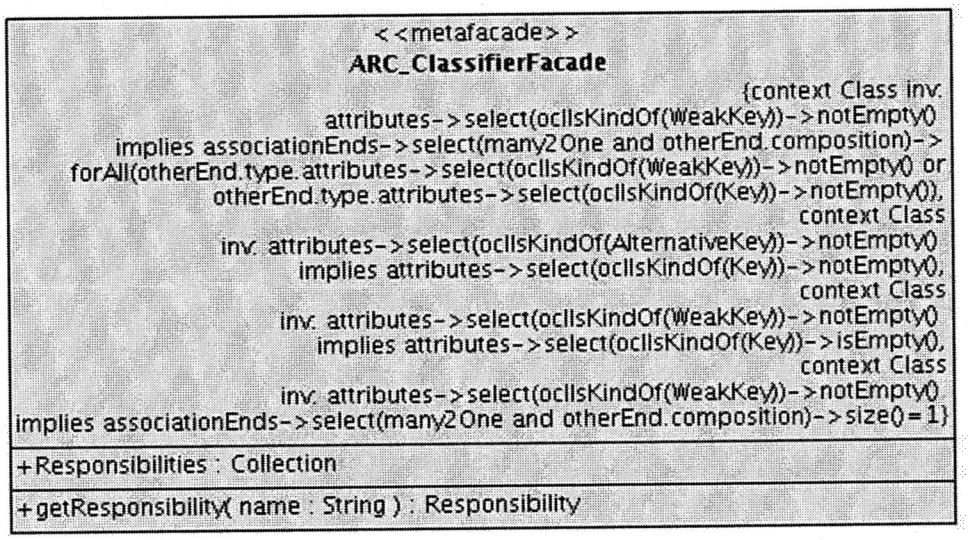
accesibles mediante los métodos "getResponsibility" y "getResponsibilities". Éste último es generado de forma automática ya que la colección es pública. Es importante seguir la misma nomenclatura, ya que esto facilita la mantenibilidad y extensibilidad del cartridge.

2. Crear los *metafacades* necesarios para los *stereotypes*.



Hay que crear los *metafacades* extendiendo el *metafacade* genérico de atributo, ya que los *stereotypes* se aplicarán sobre los atributos de la clase. Los *stereotypes* no necesitan tener un *metafacade* asociado. En este caso sí, ya que de otra forma sería complicado establecer restricciones en *OCL*.

3. Añadimos todas las restricciones textuales posibles usando OCL.



La única restricción que no se ha expresado en OCL es la que obliga a que la última clase asociada por composición a una clase débil, debe tener una clave. Se trata de un algoritmo recursivo difícil de expresar en OCL, por tanto se ha optado por otra solución que se verá más adelante, a la vez servirá para saber cómo actuar en los casos donde el OCL no es suficiente o no es indicado para expresar una restricción. Nótese que llegados a este punto sólo se ha trabajado sobre modelos.

4. Procesar el modelo. Usando Maven, simplemente se tiene que ejecutar el proyecto. Esto se hace para tener actualizados todos los archivos de código Java generados de forma automática y necesarios para continuar trabajando.

5. Implementar los métodos del metafacade. Ahora que se ha generado el código, podemos trabajar sobre él y añadirle las

funcionalidades.

```
**
* MetafacadeLogic implementation for
org.andromda.cartridges.arc.metafacades.ARC_ClassifierFacade.
*
* @see org.andromda.cartridges.arc.metafacades.ARC_ClassifierFacade
*/
public class ARC_ClassifierFacadeLogicImpl
extends ARC_ClassifierFacadeLogic
{
      public ARC_ClassifierFacadeLogicImpl (Object metaObject, String
context)
      {
            super (metaObject, context);
      }

      /**
      * @see
org.andromda.cartridges.arc.metafacades.ARC_ClassifierFacade#getRespo
nsibilities()
      */
      protected java.util.Collection handleGetResponsibilities()
      {
            ArrayList Responsibilities=new ArrayList();

            Responsibility resp=keyResponsibility();
            if(resp!=null) Responsibilities.add(resp);

            return Responsibilities;
      }

      /**
      * @see
org.andromda.cartridges.arc.metafacades.ARC_ClassifierFacade#getRespo
nsibility(java.lang.String)
      */
      protected Responsibility
handleGetResponsibility(java.lang.String name)
      {
            Iterator iter=handleGetResponsibilities().iterator();
            while(iter.hasNext()) {
                  Responsibility resp=(Responsibility)iter.next();
                  if(resp.getName() == name) return resp;
            }
            return null;
      }
      ...
}
```

Código 1: ARC_ClassifierFacadeLogicImpl.java

El código en negrita es el que se ha introducido, el resto fue

generado automáticamente. Es importante separar el código que detecta la responsabilidad de estos métodos, ya que cada elemento del modelo puede tener responsabilidades distintas. En este caso el método que detectaría la responsabilidad de clave seria "keyResponsibility".

6. En el mismo fichero, para solucionar la restricción pendiente, debemos implementar un método que es llamado por AndroMDA en el momento de la validación del modelo.

```
/**
* Check cicles of weak keys
*/
public void validateInvariants(java.util.Collection
validationMessages)
{
 super.validateInvariants(validationMessages);

 if(haveWeak()) {
  ARC_ClassifierFacadeLogicImpl currentClass=this;
  do {
   Iterator end_iter = currentClass.getAssociationEnds().iterator();
   while(end_iter.hasNext()) {
    AssociationEndFacade asso=(AssociationEndFacade)end_iter.next();
    if (asso.isMany2One() && asso.getOtherEnd().isComposition())
     currentClass=(ARC_ClassifierFacadeLogicImpl)asso.getOtherEnd().g
etType();
   }
  }while(currentClass.haveWeak()&&getName()!=currentClass.getName());
  if (getName()==currentClass.getName()) validationMessages.add(
   new org.andromda.core.metafacade.ModelValidationMessage(this,
"org::andromda::cartridges::arc::metafacades::ARC_ClassifierFacade::W
eakKey Cicle Check", "You can NOT set a cicle of weak classes."));
 }
}

/**
* haveWeak
*/
protected boolean haveWeak()
{
 boolean weak=false;
 Iterator attr_iter=getAttributes().iterator();
 while(attr_iter.hasNext() && !weak) {
```

```
  AttributeFacade atr=(AttributeFacade)attr_iter.next();
  if(atr.hasStereotype("WeakKey")) weak=true;
 }
 return weak;
}
```

Código 2: ARC_ClassifierFacadeLogicImpl.java (2)

En este método es importante ver que lo primero que se hace es llamar al mismo método de la clase padre, de esta forma todas las restricciones expresadas en *OCL* pasan a ser precondiciones de este método. Es obligatorio llamar antes o después al método del mismo nombre de la clase padre, en caso contrario no se evaluarían las restricciones *OCL*.

7. Asociar los *metafacades*.

```
<metafacades>
      <metafacade
class="org.andromda.cartridges.arc.metafacades.ARC_ClassifierFacadeLo
gicImpl" contextRoot="true"/>

      <metafacade
class="org.andromda.cartridges.arc.metafacades.KeyLogicImpl"
contextRoot="true">
            <mapping>
                  <stereotype>KEY</stereotype>
            </mapping>
      </metafacade>
      <metafacade
class="org.andromda.cartridges.arc.metafacades.AlternativeKeyLogicImp
l" contextRoot="true">
            <mapping>
                  <stereotype>ALTERNATIVE_KEY</stereotype>
            </mapping>
      </metafacade>
      <metafacade
class="org.andromda.cartridges.arc.metafacades.WeakKeyLogicImpl"
contextRoot="true">
            <mapping>
                  <stereotype>WEAK_KEY</stereotype>
            </mapping>
      </metafacade>
```

```
</metafacades>
```

Código 3: metafacades.xml

El formato de este fichero ya fue explicado en el capítulo anterior. El *metafacade* del elemento responsable no tiene ningún *mapping* ya que queremos que éste se aplique a todos los elementos del modelo. Los *metafacades* de los elementos donde se detecta la responsabilidad tienen el *mapping* con sus respectivos *stereotypes*.

5.2 Restricciones textuales

Las restricciones textuales, en AndroMDA, están compuestas por un nombre, una expresión *OCL* y una descripción. La expresión puede ser un invariante, una postcondición o una precondición.

Las múltiples aplicaciones de las restricciones textuales dentro del modelo *UML* provocan que sean una de las fuentes de responsabilidades más importantes. En algunos casos, como las claves, se puede optar por una solución simplificada, usando *tagged values* o *stereotypes*.

En este proyecto sólo detectaremos las responsabilidades derivadas de restricciones que sean precondiciones o postcondiciones de operaciones.

5.2.1 Consideraciones previas

- Elemento responsable: Operación

- Elemento donde se detecta la responsabilidad: Restricciones

- Las restricciones textuales pueden ser *PRE* o *POST* de operaciones.

- Las *PRE* y *POST* han de empezar por "*pre*:" o "*post*:" respectivamente sin indicar el "*context*". Cualquier otro formato no sera considerado como *PRE* o *POST*.

5.2.2 Estrategias

- Usando el código *OCL*

  Lo adecuado sería evaluar estas restricciones usando el código *OCL* y a partir de éste, determinar qué responsabilidad implica. Hacerlo de esta forma iría más allá del alcance previsto para este proyecto y probablemente sería materia suficiente para un proyecto entero.

- Usando el nombre de la restricción

  Esta solución implica tratar un conjunto limitado de restricciones y establecer un conjunto de palabras clave para identificarlas. Para el diseñador resultará más sencillo ya que no tendrá que especificar la restricción con *OCL*.

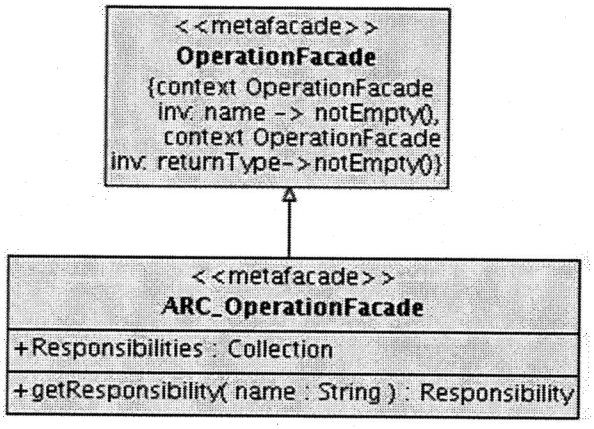
Se determinará qué responsabilidad tiene a partir del nombre de la restricción textual. Los nombres de las restricciones debe ser alguno listados a continuación:

- "*existsElement*", debe existir un elemento antes o después de la ejecución de la operación.
- "*notEmptyPopulation*", debe existir algún elemento de la población antes o después de la ejecución de la operación.
- "*deleteElement*", estrictamente para postcondiciones, indica que un elemento habrá sido eliminado al finalizar la operación.
- "insertElement", estrictamente para postcondiciones, indica que un elemento habrá sido insertado al finalizar la operación.
- "*listAll*", estrictamente para postcondiciones, indica que se listará un conjunto de elementos durante la ejecución de la operación.

Esta responsabilidad servirá de ejemplo para extender las responsabilidades derivadas de restricciones textuales, por ejemplo invariantes definidos para clases.

5.2.3 Implementación de la responsabilidad

1. Crear el metafacade del elemento responsable.



De la misma forma que en la responsabilidad anterior, creamos un metafacade siguiendo el mismo prototipo, en este caso extendiendo el metafacade genérico de operación.

2. Procesar el modelo. Al crear un nuevo metafacade, es necesario procesar el modelo para tener actualizados los archivos generados de forma automática.

3. Implementar los métodos del metafacade.

```
      ...
      /**
      * @see
org.andromda.cartridges.arc.metafacades.ARC_OperationFacade#getRespon
sibilities()
      */
      protected java.util.Collection handleGetResponsibilities()
      {
            ArrayList Responsibilities=new ArrayList();
            Responsibilities.addAll(contraintResponsibilities());
            return Responsibilities;
      }
      ...
```

Código 4: ARC_OperationFacadeLogicImpl.java

A diferencia del caso anterior, el elemento responsable obtiene una

colección de responsabilidades derivadas de restricciones textuales. En este caso también se separa el código encargado de detectar las responsabilidades por los motivos ya expuestos.

4. Asociar los *metafacades*.

```
      ...
      <metafacade
class="org.andromda.cartridges.arc.metafacades.ARC_OperationFacadeLog
icImpl" contextRoot="true"/>
      ...
```

Código 5: metafacades.xml

Simplemente hay que añadir el metafacade que queremos tener para todas las operaciones. AndroMDA determina a qué elemento del metamodelo está asociado gracias a la relación de herencia que se estableció en el primer paso.

5.3 Cardinalidades

Una cardinalidad es una limitación que expresa el mínimo y el máximo número de elementos de un determinado tipo. En este caso nos limitamos a las cardinalidades definidas en asociaciones, pero el mismo planteamiento se podría aplicar para cardinalidad de atributos, clases, o algún otro elemento.

Las asociaciones derivan en dos responsabilidades, una por cada extremo de la asociación. Las clases son responsables de mantener la limitación

expresada en el extremo opuesto de la asociación. Por ejemplo, en la figura siguiente, la clase "Empleado" es responsable de mantener un mínimo de 1 y un máximo de 3 asociaciones con la clase "Departamento" y la clase "Departamento" es responsable de mantener un mínimo de 2 asociaciones con la clase "Empleado".

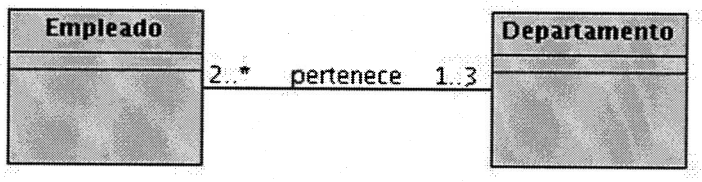


Fig. 8: Ejemplo de asociación.

Las responsabilidades de cardinalidades servirán de ejemplo para los casos donde las responsabilidades están expresadas gráficamente.

Las responsabilidades expresadas gráficamente nos evitan tener que definir restricciones, pues éstas ya están establecidas por el estándar *UML*.

5.3.1 Consideraciones previas

- Elemento responsable: Clase

- Elemento donde se detecta la responsabilidad: Asociación

5.3.2 Estrategias

Las únicas estrategias que se pueden abordar en este caso son referentes

al modo de implementación, pues la forma en que trabaja el diseñador ya está establecida.

Respecto al modo de implementación podría surgir la duda de dónde incluir el código que detecte la responsabilidad: crear otro metafacade que extendiera al metafacade de asociación genérico o trabajar sobre el metafacade que ya se ha creado, *ARC_ClasifierFacade*. Aunque parezca más eficiente la primera solución, para que el código sea más comprensible y así facilitar su extensibilidad, siempre se detectarán las responsabilidades desde el elemento responsable.

5.3.3 Implementación de la responsabilidad

Sólo queda un paso por hacer, la implementación de los métodos. El *metafacade* ya fue creado y asociado en la primera responsabilidad.

```
        ...
        /**
        * @see
org.andromda.cartridges.arc.metafacades.ARC_ClassifierFacade#getRespo
nsibilities()
        */
        protected java.util.Collection handleGetResponsibilities()
        {
              ArrayList Responsibilities=new ArrayList();

              Responsibility resp=keyResponsibility();
              if(resp!=null) Responsibilities.add(resp);

              Responsibilities.addAll(cardinalityResponsibilities());

              return Responsibilities;
        }
        ...
```

Código 6: ARC_ClassifierFacadeLogicImpl.java (3)

Aquí se puede ver la necesidad de separar el código de cada tipo de responsabilidad. De no ser así, se mezclaría el código de detección de las distintas responsabilidades de un elemento.

6. Tratamientos

Una vez se tienen detectadas las responsabilidades, hay que aplicarles un tratamiento. En el contexto de la ingeniería del software, un tratamiento es una posible solución de diseño para mantener una o más responsabilidades.

En este proyecto los tratamientos siempre serán los mismos para un mismo tipo de responsabilidad, esto es debido a que se aplican los tratamientos de forma incontextual, es decir que sólo se tienen en consideración las preferencias del usuario.

Una de las vías de investigación que ha quedado abierta es reconsiderar la forma en que se aplican los tratamientos. Si se continua el proyecto, se podrían aplicar tratamientos según parámetros contextuales, como por ejemplo la población de una clase. De esta forma una misma responsabilidad podría tener distintos tratamientos dependiendo del contexto además de las preferencias del usuario.

6.1 Estructura modular

Ya que los tratamientos son independientes de la plataforma que detecta las responsabilidades, y que uno de los requisitos del proyecto es la extensibilidad, se ha optado por la separación modular del software encargado de los tratamientos. De esta manera se podrá intercambiar la forma en que se detectan las responsabilidades y seguir usando el mismo módulo encargado de aplicar los tratamientos.

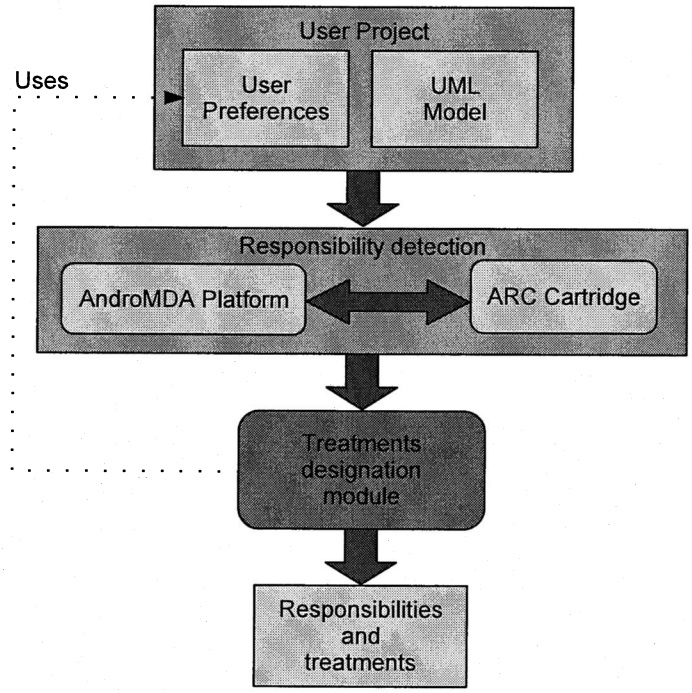


Fig. 1: Estructura modular

6.2 Tratamientos

Se ha tomado un conjunto de tratamientos para cada responsabilidad a modo de muestra del funcionamiento del módulo. Cada tratamiento tiene un identificador y una capa asignada, además se indica qué tratamientos se pueden aplicar a cada responsabilidad.

6.2.1 Diccionario de dominio

Un diccionario de dominio es un controlador de las instancias de una clase. Puede servir, por ejemplo, para saber mediante un identificador si la

instancia de una clase existe en el diccionario.

- Identificador: *"Dictionary"*
- Aplicable a las responsabilidades: Clave, RT[1]("*existsElement*"), RT("*notEmptyPopulation*"), RT("*insertElement*"), RT("*deleteElement*"), RT("*listAll*")
- Capa: Dominio

6.2.2 Hybernate

Hybernate es una tecnología que oculta la capa de persistencia, permitiendo automatizar la creación de las funcionalidades de consulta y persistencia a partir del esquema de la base de datos. Actualmente es compatible con aplicaciones hechas en Java y .NET.

- Identificador: *"Hybernate"*
- Aplicable a las responsabilidades: Clave, RT("*existsElement*"), RT("*notEmptyPopulation*"), RT("*insertElement*"), RT("*deleteElement*"), RT("*listAll*"), Cardinalidad
- Capa: Persistencia

6.2.3 Filtro de entrada

Los filtros de entrada son controles que residen en la capa de

1 Restricción textual

presentación, que permiten tomar los valores de entrada como válidos sin tener que efectuar comprobaciones adicionales. Un ejemplo podría ser una aplicación que en vez de permitir al usuario escribir en el campo de un formulario da a escoger una opción de un listado.

- Identificador: "*InputFilter*"
- Aplicable a las responsabilidades: RT("*existsElement*")
- Capa: Presentación

6.2.4 Control del tamaño de entrada en presentación

Este tratamiento consiste en añadir un control sobre el tamaño de los datos de entrada. Por ejemplo, limitar el número de elementos seleccionables de una lista.

- Identificador: "*InputSizeControl*"
- Aplicable a las responsabilidades: Cardinalidad
- Capa: Presentación

6.2.5 Control del tamaño de entrada en dominio

Se trata del mismo control que el caso anterior, pero esta vez se delega la responsabilidad a la capa de dominio.

- Identificador: "*CardinalityControl*"

- Aplicable a las responsabilidades: Cardinalidad
- Capa: Dominio

6.2.6 SQL

SQL es el lenguaje de consultas más usado para el acceso a bases de datos relacionales. Un ejemplo habitual del uso de SQL es obtener todas las tuplas de una tabla.

- Identificador: *"SQL"*
- Aplicable a las responsabilidades: Clave, RT("*existsElement*"), RT("*notEmptyPopulation*"), RT("*insertElement*"), RT("*deleteElement*"), RT("*listAll*"), Cardinalidad
- Capa: Persistencia

6.2.7 Disparador

Un disparador en una base de datos es un evento que se ejecuta cuando se cumple una condición establecida al realizar una operación de inserción, actualización o borrado.

- Identificador: *"Trigger"*
- Aplicable a las responsabilidades: Cardinalidad
- Capa: Persistencia

6.2.8 Procedimiento almacenado

Un procedimiento almacenado es un programa almacenado físicamente en una base de datos. Generalmente están escritos usando un lenguaje propio de la bases de datos. La ventaja de un procedimiento almacenado es que se ejecuta directamente en el motor de la base de datos, el cual normalmente está en un servidor separado.

- Identificador: "*StoredProcedure*"
- Aplicable a las responsabilidades: RT("*insertElement*"), RT("*deleteElement*"), RT("*listaTodos*")
- Capa: Persistencia

6.2.9 Borrado en cascada

Las bases de datos tienen un mecanismo para mantener la integridad cuando se borra un elemento referenciado por otra tabla, que consiste en borrar todos los elementos que hacen referencia al primero.

- Identificador: "*OnCascade*"
- Aplicable a las responsabilidades: RT("*deleteElement*")
- Capa: Persistencia

6.2.10 Esquema de base de datos

El esquema de base de datos define la estructura interna de la base de

datos, sus relaciones y sus reglas de integridad. Son éstas últimas las que nos permiten delegar responsabilidades ya que por ejemplo una clave puede ser fácilmente controlada por una regla de integridad de tipo clave primaria.

- Identificador: "*DBSchema*"
- Aplicable a las responsabilidades: Clave, Cardinalidad
- Capa: Persistencia

6.3 Tratamientos por defecto

Independientemente de las preferencias que el usuario haya establecido, existirán tratamientos por defecto para cada responsabilidad, éstos serán aplicables en cualquier circunstancia asegurando de esta forma que toda responsabilidad tenga al menos un tratamiento asignado. Estos tratamientos serán aquéllos que son implementables por el desarrollador y que no tienen dependencia tecnológica.

Ver "*Apéndice C: Asignación de tratamientos*".

6.4 Preferencias del usuario

En este proyecto, se ha optado por que las preferencias del usuario sean más genéricas que los tratamientos, ya que durante la asignación de tratamientos el módulo encargado de esta tarea actúa como sistema

experto. Por ejemplo, el usuario puede decidir que quiere priorizar el uso de la base de datos, pero no se le permite determinar para cada responsabilidad o tipo de responsabilidad en concreto, qué tratamiento aplicar.

Cada proyecto debe establecer sus preferencias en el fichero "*andromda.xml*" ubicado en el carpeta "*src/config*" de la misma forma que se puede ver en el siguiente ejemplo.

```
<andromda>
      <properties>
            <property name="tecnologicDependency">Medium</property>
            <property name="developLanguage">Java</property>
            <property name="database">PostgreSQL</property>
            <property name="interfaceComplexity">Low</property>
      </properties>
      ...
</andromda>
```

Código 1: andromda.xml

El valor de las propiedades debe ser uno de los valores válidos indicados en los siguientes apartados. Actualmente no se permiten propiedades multivaluadas, esta posibilidad quedará dentro de las posibles vías de extensión del proyecto. Una propiedad con un valor vacío o inválido será considerada una propiedad indefinida, por tanto no se aplicará ningún tratamiento que requiera un valor concreto en dicha propiedad.

6.4.1 Dependencia tecnológica

Indica el grado de dependencia que se quiere para aplicación. Un grado

bajo de dependencia evitaría seleccionar *Hybernate*, o tratamientos internos de la base de datos mientras que un grado alto propiciaría el uso de estos tratamientos.

- Identificador: "*technologicDependency*"
- Valores: "*High*", "*Medium*", "*Low*"

6.4.2 Lenguaje de desarrollo

Mediante esta variable o preferencia, indicamos qué lenguaje de programación se usará para el desarrollo de la aplicación. Esto nos permite descartar algunos tratamientos dependiendo del caso.

- Identificador: "*developmentLanguage*"
- Valores: "*C++*", "*Java*", "*.NET*"

6.4.3 Base de datos

Además del lenguaje de programación es importante saber con qué base de datos contará el programa. En caso de no tener base de datos muchos de los tratamientos no se pueden aplicar.

- Identificador: "*database*"
- Valores: "*None*", "*Oracle*", "*PostgreSQL*"

6.4.4 Complejidad de la interfaz de usuario

Cuanto mayor sea la proactividad de la interfaz de usuario, más responsabilidades podrán ser tratadas en la capa de presentación.

- Identificador: "*interfaceComplexity*"
- Valores: "*High*", "*Medium*", "*Low*"