GENERADOR DE REGLAS DE NEGOCIO

Informe de Proyecto de Grado

Estudiantes:
- Marcelo Belén
- Paula Blanco
- Leonardo Sbrocca

Tutora:
- Ana Erosa

Proyecto de Grado
INCO – Facultad de Ingeniería – Universidad de la República
Montevideo, Uruguay
Abril de 2005

Resumen

En este proyecto se presenta el desarrollo de una herramienta generadora de reglas de negocio. Informalmente, las reglas de negocio se pueden definir como restricciones a nivel operativo de un negocio. Un negocio tiene un conjunto de reglas de negocio único, el cual se aplica a la operativa diaria del mismo. Por ejemplo, una empresa que trabaja en el negocio de los medios de pago y que necesita validar transacciones financieras hechas desde un cajero automático podría tener una regla de negocio que verificara el estado operativo (operativo o no operativo) de la tarjeta con la cual se efectúa una transacción.

Se comienza estudiando la base teórica del problema, lo que incluye utilidad y aplicación de las reglas de negocio, categorizaciones de reglas existentes, mecanismos de especificación de reglas, motores de reglas de negocio existentes, etc. En base a la investigación mencionada y los requerimientos del cliente (empresa financiera *Cabal S.A*) desarrollamos el generador de reglas de negocio.

Dicho generador consta de 6 módulos. El módulo *Mapeo* permite construir el dominio del negocio a partir de la base de datos del mismo, sobre este dominio se especificarán las reglas de negocio. La especificación de las reglas es soportada por el módulo *Diseñador* que habilita la creación, modificación y eliminación de reglas y versiones de las mismas. Las reglas de negocio son especificadas en un lenguaje diseñado para tal fin, el cual soporta un conjunto predeterminado de tipos de reglas. A partir de la especificación de las reglas en el lenguaje mencionado, el módulo *Compilador* comprueba la correctitud sintáctica y semántica de las mismas. El módulo *GeneradorPLSQL* genera el código de las reglas de negocio creadas como procedimientos almacenados *PL/SQL*, mientras que el módulo *GeneradorJava* genera el código de las mismas como clases *Java*. Finalmente, el módulo *Evaluador* realiza el ordenamiento y ejecución de las reglas, utilizando el código generado para las mismas ya sea *Java* o *PL/SQL*, y árboles de evaluación creados para los objetos del dominio.

Se implementó un prototipo que cumple los principales objetivos que se pretende alcanzar en el desarrollo de un motor de reglas de negocio, estos son: independizar las reglas de negocio de las aplicaciones y que la realización de cambios en las reglas insuma un costo bajo. Además, se ha implementado una herramienta flexible en cuanto a que está habilitada para utilizarse sobre varios tipos de DBMS por la utilización de *JDBC* e *Hibernate* y es multiplataforma por estar desarrollada en *Java*.

Además, hemos obtenido buenos tiempos de respuesta en la evaluación, tanto para las reglas generadas como procedimientos *PL/SQL* como para las reglas generadas como clases *Java*. Siendo más rápida la evaluación en *PL/SQL*, lo que era de esperarse ya que los procedimientos se encuentran almacenados en la base de datos.

Palabras claves

Regla de negocio, Tipo de reglas de negocio, Lenguaje de especificación de reglas de negocio, Mapeo, Compilador, Diseñador, Generador de reglas *PL/SQL*, Generador de reglas *Java*, Evaluador de reglas de negocio.

TABLA DE CONTENIDO

1 Introducción

1.1 Motivación

El proyecto se enmarca en el área de las reglas de negocio. Informalmente, una regla de negocio es una restricción sobre algún aspecto del negocio. Durante mucho tiempo, si bien las reglas de negocio eran tenidas en cuenta, no se las consideraba en el diseño, sino que el diseño de un sistema se basaba exclusivamente en las funciones y/o en los datos del mismo. A raíz de esto, las reglas de negocio quedaban embebidas en el código de los procesos del sistema y como consecuencia de ello, a la hora de modificar las reglas había que modificar los procesos del sistema que la utilizaban, lo que además de ser improductivo posiblemente resultase en implementaciones diferentes de una misma regla.

Desde hace algunos años ha tomado fuerza la idea de diseñar y administrar las reglas de negocio en forma independiente de los procesos del sistema. Se busca una herramienta que permita independizar las reglas de negocio de los procesos, de modo que las modificaciones de las reglas tengan un mínimo impacto en los procesos, generando el código de forma automática y en consecuencia aumentando la productividad.

Teniendo en cuenta esta realidad, el desarrollo del *Generador de Reglas de Negocio* surge por la necesidad del cliente de disponer de un sistema que le permita administrar las reglas de negocio. Si bien la herramienta desarrollada busca solucionar la realidad de un negocio en particular, el problema es común a cualquier negocio. Por lo tanto, nuestra intención ha sido buscar una solución lo más genérica posible, o sea una herramienta de automatización de reglas de negocio aplicable a cualquier negocio.

1.2 Objetivos

Los objetivos del proyecto se dividen en dos grupos. El primero incluye los objetivos vinculados al estudio del área de las reglas de negocio tanto en la parte teórica como en el relevamiento de productos existentes. El segundo, incluye los objetivos vinculados al desarrollo de un sistema capaz de administrar las reglas de negocios, o sea el *Sistema Generador de Reglas de Negocio*.

Los objetivos pertenecientes al primer grupo son:

- Lograr un conocimiento profundo de los principales conceptos sobre los cuales se construye la teoría de las reglas de negocio.
- Elegir un modelo a partir del cual se diseñe el *Sistema Generador de Reglas de Negocio.*
- Tener una visión general de los productos y proyectos relacionados con las reglas de negocio existentes.

Con el desarrollo *Sistema Generador de Reglas de Negocio* se busca los siguientes objetivos:

- Administrar las reglas de negocio independientemente de los procesos del negocio.
- Identificar claramente los tipos de reglas de negocio que soportará el sistema.
- Crear un lenguaje formal para que el usuario pueda especificar las reglas de negocio.
- Generar código *PL/SQL* y *Java* a partir de las reglas de negocio especificadas.
- Desarrollar un evaluador de reglas de negocio, uno para *PL/SQL* y otro para *Java*.
- Comparar los evaluadores en lo que respecta al tiempo de ejecución, o sea *PL/SQL* versus *Java*.

1.3 Cronograma

A lo largo del proyecto se siguió un modelo de proceso iterativo e incremental, el cual consta de 16 fases.

Las fases 1 y 2 comprenden respectivamente la interiorización con el tema reglas de negocio y el análisis de requerimientos, la primera fase incluye el estudio de las reglas de negocio y de los motores de reglas de negocio existentes, la segunda fase incluye el relevamiento de las necesidades y expectativas del cliente.

En la fase 3 se realiza el análisis y diseño de la arquitectura del sistema. En las fases 3 y 4 se realiza respectivamente el diseño de los módulos *Mapeo* y *Diseñador*. La implementación de lo diseñado en estas fases se realiza paralelamente en las fases 6, 7 y 8 junto con la implementación del módulo *Compilador* (fase 9.)

La fase 10 consiste en la integración de lo implementado en las fases anteriores dando lugar a la primera versión del prototipo, en esta instancia el sistema permite la creación del dominio del negocio (Mapeo) y la especificación (y compilación) de reglas.

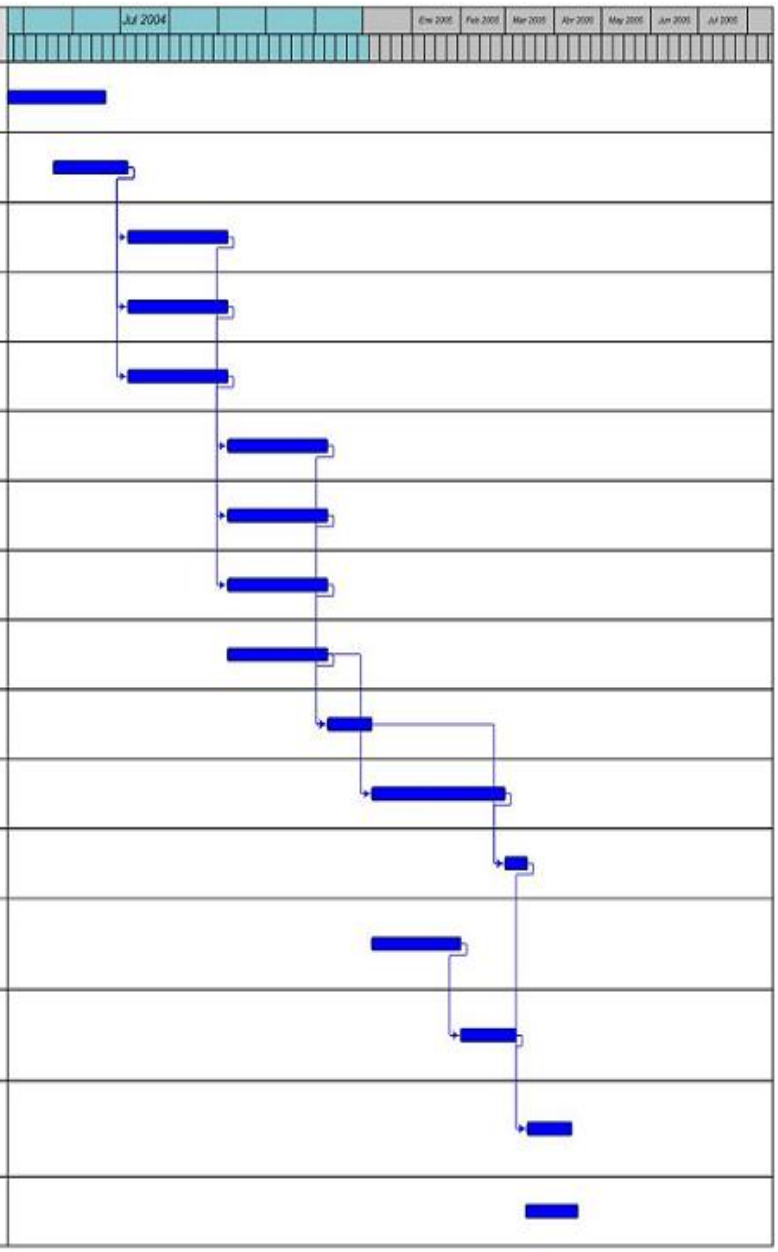
El diseño e implementación de las funcionalidades de generación de código *PL/SQL* y *Java* se realiza en la fase 11, en paralelo al diseño e implementación de los módulos *Evaluador* y *Secuenciador* (fases 13 y 14 respectivamente). La fase 12 consiste en la integración de los generadores de código al prototipo.

La fase 15 comprende la integración del Evaluador y Secuenciador al prototipo, el testeo del sistema y la comparación de tiempos de respuesta: *PL/SQL* versus *Java*.

Finalmente, en la fase 16 se prepara el informe del proyecto y la presentación del mismo.

En el siguiente diagrama se presenta el cronograma seguido:

Fase	Task Name	Start	Finish	Duration
1	Interiorización con el problema	21/04/ 2004	21/06/ 2004	8,86s
2	Definición y análisis de requerimientos	20/05/ 2004	05/07/ 2004	6,71s
3	Arquitectura - Análisis y Diseño	06/07/ 2004	06/09/ 2004	9s
4	Diseño del módulo "Mapeo"	06/07/ 2004	06/09/ 2004	9s
5	Diseño del módulo "Diseñador"	06/07/ 2004	06/09/ 2004	9s
6	Infraestructura del sistema central	07/09/ 2004	08/11/ 2004	9s
7	Implementación del módulo "Mapeo"	07/09/ 2004	08/11/ 2004	9s
8	Implementación del módulo "Diseñador"	07/09/ 2004	08/11/ 2004	9s
9	Implementación del módulo "Compilador"	07/09/ 2004	08/11/ 2004	9s
10	Integración fases 6, 7, 8 y 9	09/11/ 2004	06/12/ 2004	4s
11	Módulo "Generador" - Diseño e Implementación	07/12/ 2004	28/02/ 2005	12s
12	Integración de generación de código	01/03/ 2005	14/03/ 2005	2s
13	Módulos "Secuenciador" y "Evaluador" - Diseño	07/12/ 2004	31/01/ 2005	8s
14	Módulos "Secuenciador" y "Evaluador" - Implementación	01/02/ 2005	07/03/ 2005	5s
15	Integración - Testeo del prototipo - Comparación de arquitecturas	15/03/ 2005	11/04/ 2005	4s
16	Preparación de informe final y presentación	14/03/ 2005	15/04/ 2005	4,71s



1.4 Organización del documento

El documento está organizado en 8 capítulos. En el segundo capítulo se estudia el estado del arte de las reglas de negocio, y se presentan algunos productos disponibles en el mercado. En el capítulo tres se presenta el paradigma de reglas de negocio seleccionado como base teórica para el proyecto, mencionando las principales características del paradigma, como el modelo de conceptos y la categorización de reglas de negocios (tipos de reglas de negocios.) El capítulo cuatro expone las principales funcionalidades de la herramienta: administración de negocios, mapeo de modelo relacional a modelo de conceptos, diseño de reglas y evaluación de reglas. En el capítulo cinco se presentan las arquitecturas de los dos sistemas desarrollados, la arquitectura de la herramienta de generación de reglas de negocios (sistema central) y la arquitectura del evaluador de reglas. En el sexto capítulo se realiza un resumen de las principales decisiones de implementación tomadas. El capítulo siete muestra los resultados experimentales de las pruebas realizadas para comparar la performance del evaluador de reglas *PL/SQL* versus el evaluador de reglas *Java*. Finalmente, en el capítulo ocho se exponen las conclusiones del proyecto.

2 Estado del Arte

2.1 Introducción

Tradicionalmente, la atención en los sistemas de información ha estado enfocada en los datos, procesos y funcionalidades sin dar especial atención a las restricciones sobre los datos, las cuales generalmente aparecían en escena en el momento de la implementación.

La especificación de las reglas de negocio se encontraba embebida en las aplicaciones y por lo tanto estaba estrechamente ligada con factores técnicos como lenguajes de programación, manejadores de bases de datos, etc. Por lo tanto, un cambio en la realidad del negocio que ocasionara un cambio en las reglas implicaba modificar las aplicaciones que contenían dichas reglas, lo que podía llegar a ser muy poco productivo.

Pero la visión tradicional está cambiando y se busca hacer explícitas las restricciones sobre los datos, esto es, las reglas del negocio, desligándolas lo más posible de las aplicaciones y por lo tanto de las herramientas informáticas.

Los factores principales que impulsan la independencia de las reglas de negocio de las aplicaciones son los siguientes:

- El hecho de que las reglas de negocio estén directamente implementadas sobre las aplicaciones hace que el modificar una regla pueda requerir cambios en la estructura de los datos lo que puede ser muy costoso, en especial si las reglas cambian con frecuencia.
- Que las reglas estén especificadas en un lenguaje de programación determinado implica que solamente personal técnico pueda especificarlas, se busca revertir esta situación utilizando un lenguaje de alto nivel que permita que cualquier persona que conozca el dominio del negocio pueda especificar las reglas.
- Relacionado con el punto anterior, se pretende que las reglas del negocio sean especificadas por los analistas del negocio y no por el personal técnico, ya que los primeros son los que conocen mejor los procesos del negocio. Además, el utilizar un lenguaje simple y sin tecnicismos mejora la comunicación entre personal técnico y personal del negocio facilitando el análisis de requerimientos.
- Este enfoque, normalmente, facilita un mejor entendimiento y comunicación de las reglas del negocio las cuales permanecerán ordenadas, clasificadas y almacenadas en un repositorio, donde estarán disponibles para ser examinadas y reutilizadas por cualquier persona involucrada con el flujo del negocio.

En base a las distintas definiciones realizadas por distintos autores decimos que: ***"las reglas de negocio son restricciones que se imponen sobre los datos del negocio"***.

Veamos algunas características deseables de las reglas de negocio en cuanto a su independencia, las mismas fueron identificadas en el *Manifiesto de reglas de negocio* realizado por el *Business Rule Group* [6]:

- Se refieren a los requerimientos primarios del negocio.
- Se separan de los procesos, no están contenidas en ellos.
- Se especifican de forma declarativa, no procedural.

- Se construyen con expresiones bien formadas.
- Las reglas guían los procesos del negocio.
- Son motivadas por el negocio, no por la tecnología.
- Se construyen por y para gente del negocio, no para técnicos de Tecnología de la Información.
- Manejan la lógica del negocio, no plataformas de hardware o software.

En cuanto a los tipos de reglas de negocios, no existe una única categorización de reglas, sin embargo, hemos observado que las clasificaciones existentes abarcan en general estos tipos de reglas de negocio:

- **Relaciones**, entre objetos del negocio.
- **Restricciones**, sobre los datos del negocio.
- **Acciones**, a realizar cuando se cumplen determinadas condiciones.
- **Derivaciones**, de nuevo conocimiento.
- **Cálculos**, obtención de valores de atributos en base a fórmulas que contienen otros atributos.
- **Validación**, de los atributos de los objetos del negocio.
- **Presentación**, de los datos en los dispositivos de salida.

2.2 Especificación de reglas de negocio

En sistemas convencionales las reglas de negocio son expresadas en las etapas de requerimientos, análisis orientado a objetos y diseño orientado a objetos. En el modelado de objetos encontramos las reglas de negocio en las relaciones entre las clases, restricciones sobre las instancias, pre y post condiciones de las clases.

También encontramos en la implementación de las aplicaciones la implementación de las reglas, desde la interfaz de usuario a la base de datos. Por ejemplo, si a nivel de interfaz de usuario se restringe que el valor ingresado en un campo esté en un determinado rango se está especificando una regla de negocio. En cuanto a los Manejadores de bases de datos (DBMS), varios de ellos brindan soporte de reglas de negocio como control de integridad referencial, restricciones sobre los valores de las columnas, procedimientos almacenados que se ejecutan cuando los datos satisfacen determinadas condiciones, etc.

2.3 Arquitectura de los motores de reglas de negocio

Evidentemente no todos los sistemas de reglas de negocio tienen porque estar construidos sobre una misma arquitectura, pero se han identificado ciertos componentes que tienen los motores de reglas de negocios más populares. El siguiente es un esquema de dichos componentes:

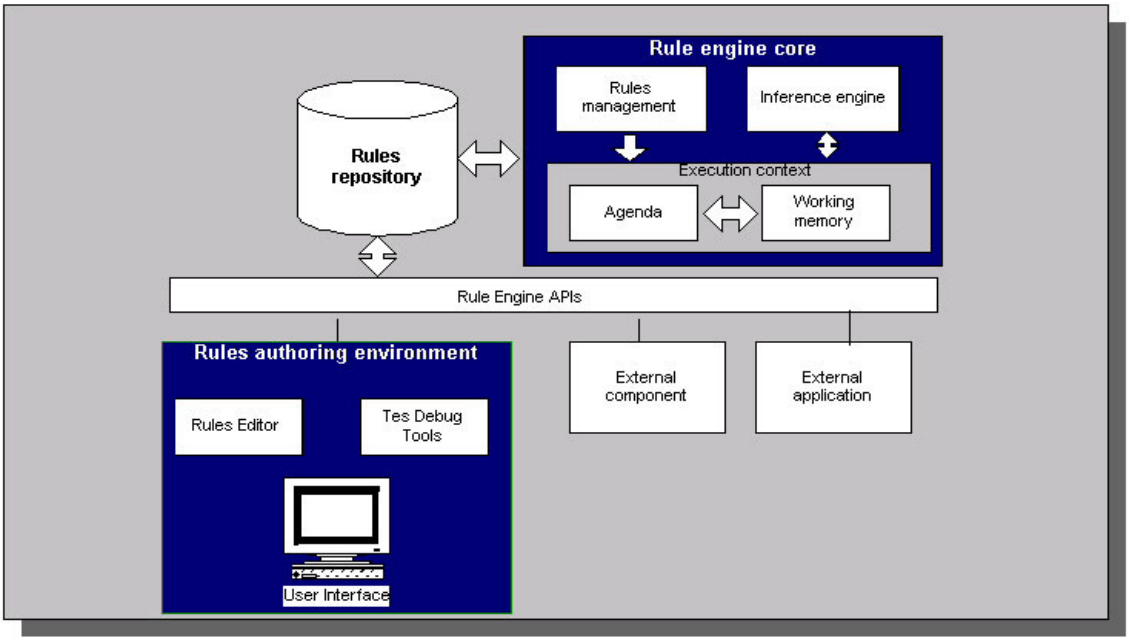


Figura 1. Arquitectura de un motor de reglas de negocio

Podemos visualizar en la figura anterior la existencia de tres componentes bien definidos, el repositorio de reglas, el núcleo del motor de reglas y el diseñador de reglas.

El repositorio de reglas almacena el dominio, el conocimiento del motor de reglas, que son las reglas de negocio. Las reglas generalmente son especificadas en un lenguaje de alto nivel como ya hemos mencionado, pero son almacenadas generalmente compiladas de forma de aumentar la performance.

El núcleo del motor de reglas cuenta con los siguientes componentes:

- **Módulo manejador de reglas:** Accede al repositorio y carga las reglas necesarias según se requiera.
- **Agenda:** Rastrea las reglas seleccionadas por el motor de inferencias, es decir, las reglas que se van a ir evaluando.
- **Memoria de trabajo:** Contiene el estado actual de hechos que llevaron a evaluar las reglas actuales en la agenda.
- **Máquina de inferencias:** Pone reglas en la agenda seleccionando de acuerdo a los hechos. Si una regla se ejecuta entonces se agregan más hechos, estos hechos se agregan en la memoria de trabajo, a su vez estos hechos son utilizados para obtener más reglas hasta terminar el ciclo, donde no se pueden obtener más reglas para evaluar.

El tercer componente, el diseñador, provee la interfaz de usuario la cual permite la especificación de reglas por parte del usuario, generalmente en un lenguaje de alto nivel. Además, lo más común es que se incluya en este componente un chequeo sintáctico de las reglas ingresadas por el usuario y en algunos casos herramientas de debug y testeo.

2.4 Productos existentes en el mercado

En esta sección se presentará las características básicas de algunos motores de reglas de negocio existentes en el mercado. No se pretende realizar un estudio exhaustivo de dichos productos, sino que el objetivo es identificar si cumplen las características mencionadas en este capítulo y realizar una comparación de los mismos.

2.4.1 Versata

Versata [3] es un producto de *Versata Corporation,* el mismo cuenta con un completo entorno de desarrollo y es usado generalmente como un generador para aplicaciones *Java* centradas en los datos.
Los tipos de reglas que soporta son: derivaciones, cálculos, relaciones, restricciones y acciones. Las reglas se especifican en forma declarativa, no procedural y tienen las siguientes características:

- Son atómicas.
- Son independientes de la implementación.
- Son eventos independientes.
- No necesitan ser ordenadas porque el sistema resuelve las dependencias y de esta forma determina el orden de ejecución.

Versata se basa en una arquitectura *J2EE*. Los elementos del dominio (Objetos del negocio) son representados como *entity beans* y las reglas son ejecutadas como componentes J2EE sobre servidores de aplicaciones como *WebSphere*, *WebLogic* y *JBoss*. Este diseño (*J2EE*) hace al sistema más general, flexible y reusable.
El repositorio de reglas que contiene *Versata* es un repositorio que contiene el conocimiento del negocio el cual se almacena en formato *XML*.
Observamos que *Versata* cumple con las características básicas que debe tener un motor de reglas de negocio ya que la implementación de las reglas es independiente de su especificación y además provee un lenguaje de especificación de reglas de alto nivel, lo que implica que las reglas pueden ser escritas por personal no técnico.

2.4.2 BRBeans

BRBeans [2] (*Business Rules Beans*) es un componente de *WebSphere Application Server* (*IBM*. Consiste en un framework utilizado para que las aplicaciones de negocio exterioricen sus reglas de negocio.

BRBeans provee dos interfaces de usuarios *Rule Management Application* y *Rule Management APIs*. La primera es una interfaz que permite a los usuarios administrar las reglas interactivamente, asociando a una regla determinada una implementación de regla existente, la segunda es una interfaz para ser utilizada por programadores que permite especificar el código de las reglas de negocio.

Las reglas de negocio implementan una interface llamada *rule implementor*. Tienen un nombre, folder, fecha de comienzo, y fecha de finalización. El nombre y el folder identifican a la regla. Las fechas determinan el intervalo de tiempo en que son válidas las reglas. Es obligatorio que tengan una fecha de comienzo, si no tienen fecha de finalización quiere decir que no expiran nunca.

El framework *BRBeans* contiene los siguientes componentes:

- *BRBeans Trigger Point Framework*: contiene la interfaz necesaria para que la aplicación del usuario pueda encontrar y disparar las reglas en tiempo de ejecución.
- *BRBeans Rule Implementors*: un *rule implementor* determina el comportamiento de una regla, *BRBeans* provee un conjunto predefinido de *rule implementors*, el usuario también puede escribir sus propias clases *Java* para implementar la lógica de las reglas.
- *BRBeans EJBs*: proveen la persistencia de las reglas de negocio. Son manejados por el *BRBeans Trigger Point Framework* y el *Rule Management APIs*, de modo que el programador no tiene que tratar directamente con ellos. Los *EJBs* están instalados en el servidor de reglas de negocios (*WebSphere application server*).
- *BRBeans Rule Management Application*: es una aplicación que se ejecuta independientemente, remota o localmente al servidor de reglas de negocio. Es utilizada para crear, actualizar, expirar y eliminar reglas de negocio, y puede ser usada para importar y exportar reglas de negocio desde/hacia *XML*.
- *BRBeans Rule Management APIs*: APIs para desarrollar un administrador de reglas de negocio propio.
- *BRBeans Rule Importer y Exporter*: para poder trasladar las reglas de negocio desde el ambiente de desarrollo al de producción.

La forma de ejecutar las reglas de negocio en una aplicación de usuario es a través de *trigger points*, estos son puntos en el código de la aplicación del usuario donde son disparadas las reglas de negocio. Más claramente, cuando se quiere ejecutar una regla en una aplicación se crea una instancia de la clase *TriggerPoint* y se ejecuta el método *Trigger* de la misma con los parámetros correspondientes a la regla que se desea ejecutar.

Vemos que si bien *BRBeans* independiza el código de la regla de su definición, las reglas deben ser especificadas por personas con conocimientos de programación ya que las mismas se especifican en lenguaje *Java*.

Una similitud de *Versata* y *BRBeans* es la utilización de *J2EE* que hemos observado es una tecnología muy utilizada en la implementación de motores de reglas de negocio.

2.4.3 Drools

Drools [4] es un motor de reglas que centraliza y administra el conjunto de reglas separándolo del código del sistema.

Esta herramienta utiliza un lenguaje basado en *XML*, *Drools Rule Language* para especificar reglas; *Drools Rule Language* no es declarativo en el sentido de que no especifica reglas a nivel operativo sino que soporta reglas del estilo *si <condición> entonces <consecuencia>*.

Drools provee un framework con los siguientes componentes:

- Una API (*Client*) para trabajar con los hechos (working memory), puede crear, eliminar o modificar hechos. Con este API también se pueden activar las reglas.
- Una API (*Administrative*) que es usada para cargar reglas que fueron creadas con *Drools Rule Language*. Básicamente lo que hace es leer archivos con formato *Drools Rule Language* y crear objetos *Java* o en otros lenguajes soportados.

Drools también resuelve los conflictos que pueden ocurrir al dispararse las reglas, ordena las reglas según distintas estrategias. Los conflictos pueden ocurrir ya que al disparar una regla la consecuencia de la misma puede manipular conocimiento. Como ya vimos, está característica es similar a la realización del orden de ejecución de reglas que realiza *Versata*.

Observamos que el lenguaje de especificación de reglas de *Drools* (similar a *XML*) es un intermedio entre el lenguaje de alto nivel que provee *Versata* y el lenguaje de programación (*Java*) que hay que utilizar para especificar reglas en *BRBeans*.

3 Paradigma de las Reglas de Negocio.

3.1 Introducción

Luego de realizado un estudio de los diferentes enfoques realizados por diferentes autores sobre reglas de negocios seleccionamos el paradigma de reglas de negocio expuesto por *Ronald Ross* en el libro "*Principles of the Business Rule Approach*" [1].

Las razones por la que se escogió dicho paradigma son las siguientes:

- En su libro, *R. Ross* expone una teoría clara y simple de las reglas de negocio, comprensible a todo el que esté interesado en el tema.
- Presenta una categorización de las reglas de negocio completa y bien definida.
- Observamos que las otras clasificaciones de reglas encontradas son un subconjunto de los tipos de reglas presentadas por *Ross* con ciertas variantes.
- Las reglas de negocio se especifican en un lenguaje de alto nivel, en forma declarativa sin estar ligada a aspectos de implementación.
- Las reglas se especifican sobre un modelo abstracto constituido por términos y hechos, sin hacer referencia a elementos de base de datos.

Lo que nosotros tomamos de este paradigma fueron dos elementos, el modelo de hechos y la categorización de reglas. El modelo de hechos lo tomamos como base para la construcción de un modelo sobre el cual especificar las reglas, de forma de no realizar dicha especificación directamente sobre la metadata de la base de datos del negocio. A partir de la clasificación de reglas realizada por *Ross*, definimos los tipos de reglas del generador de reglas implementado.

El enfoque que presenta *Ross* introduce las reglas de negocio en dos etapas, en la primera presenta los elementos que constituyen el conocimiento del negocio (*términos* y *hechos*), este conocimiento es la base sobre la cual se construye las reglas de negocio. En la segunda etapa expone las reglas de negocio en sí, definiéndolas y presentando una taxonomía de las mismas.

En lo que resta de este capítulo presentamos un resumen del paradigma junto a un ejemplo que se utilizará a lo largo del documento. El ejemplo se ubica en el contexto de una empresa que se dedica al negocio de los medios de pago.

3.2 Conceptos fundamentales de reglas de negocio

3.2.1 Definición de reglas de negocio

Definición: Las reglas de negocio son un conjunto de expresiones declarativas sobre aspectos operativos de un negocio.

En la definición anterior aparecen varios componentes que es necesario aclarar. Por **expresiones declarativas** se entiende oraciones en lenguaje natural, que no se escriben como procedimientos o acciones sino de manera declarativa. Con **aspectos operativos** quiere decirse que las reglas de negocio se encuentran a un nivel operativo del negocio en cuestión, o sea, no están a alto nivel

(políticas, directrices) ni a bajo nivel (código que implementa las reglas) sino que están a un nivel intermedio, el de las operaciones diarias del negocio. **Un negocio** significa que las reglas se inscriben en un contexto dado, al que llamaremos negocio; el conjunto de reglas debe ser el mismo para dicho negocio.

3.2.2 Conceptos básicos

Las reglas se construyen a partir de componentes, que forman el conocimiento básico del negocio. Estos componentes son conceptos a los que llamaremos **términos** y relaciones que llamaremos **hechos**. Para poder definir reglas, primero debemos definir los términos y hechos sobre los cuales se construirán. Los conjuntos de términos y hechos deben ser únicos dentro del contexto del negocio, de esta manera se mantiene la unicidad y coherencia de estos.

3.2.2.1 Términos

Definición: Un **término** es una palabra o expresión que tiene un significado preciso para el negocio.

Veamos algunas características inherentes a los términos. Primero, un término es **no ambiguo**, esto significa que dado el contexto del negocio, un término está representado una sola vez y no se solapa con otro término. Segundo, es **básico y atómico**, básico implica que no puede ser derivado ni calculado a partir de otros términos, atómico significa que es indivisible. Tercero, es **cognoscible**, representa una cosa que podemos conocer. Cuarto, debe estar **definido**, el concepto que un término representa no debe darse por sentado, debe ser definido. Quinto, un término se representa mediante un **sustantivo** o nombre en singular.

Definición: Un **Catálogo de Conceptos** es una colección con todos los términos de un negocio y sus definiciones.

3.2.2.2 Hechos

Definición: Un **hecho** es una expresión que representa la conexión lógica entre dos términos.

Algunas características de los hechos son las que siguen:

- **No es ambiguo**, dado el contexto del negocio un hecho está representado una sola vez y no se solapa con otro hecho.
- **Se basa en términos**, está construido a partir de términos y tiene la siguiente estructura: <Término> <Relación> <Término>, dónde *<Término>* indica la ocurrencia de un término del catálogo de conceptos y *<Relación>* indica un verbo que relaciona ambos términos del hecho. De lo anterior se infiere que en un hecho aparecen dos y sólo dos términos conectados por un verbo.

3.2.2.3 Modelo de Hechos

*Definición: El **Modelo de Hechos** de un negocio es un conjunto mínimo de términos y hechos para dicho negocio.*

El *Modelo de Hechos* estructura el conocimiento básico del negocio. Un experto en el dominio del negocio define el conjunto de hechos del negocio, estableciendo así un vocabulario común sobre el negocio. En el *Modelo de Hechos* no se representa ni cardinalidad ni opcionalidad.

Definición: El **Modelo de Conceptos** de un negocio es el Catálogo de Conceptos más el *Modelo de Hechos* de dicho negocio.

3.3 Ejemplo: Empresa de medios de pago

Para clarificar estos conceptos básicos utilizaremos un ejemplo que será utilizado a lo largo del documento. El ejemplo se inscribe en el contexto de una empresa que se dedica al negocio de los medios de pago. La empresa ofrece como producto distintos tipos de tarjetas mediante las cuales el cliente puede efectuar transacciones financieras (depósitos, transferencias, compras, etc.). El cliente tiene una cuenta bancaria que está asociada a su tarjeta. La transacción se efectúa o bien desde un comercio (Point Of Sale) o desde un cajero automático. Las tarjetas tienen un número de identificación personal, una fecha de vigencia, un estado que indica si está operativa o no, y número que identifica a la tarjeta. Las cuentas tienen un grupo de afinidad al cual pertenecen, un estado (estado de cuenta) y número de cuenta que las identifica.

Pasemos a enumerar los elementos del negocio. Primeramente tenemos los términos del negocio (el conocimiento básico del negocio), en este caso los términos son: *transacción*, *tarjeta*, *cuenta*, *cliente*, *comercio, pin de la tarjeta, fecha de vigencia de la tarjeta, estado de la tarjeta, grupo de afinidad de la cuenta, estado de cuenta* y *número de la cuenta*.

Estos términos debemos tenerlos en un catálogo de términos junto a sus definiciones. Algunas de las entradas del catálogo serían las siguientes:

Tarjeta:	producto que vende la empresa (tarjeta de débito, tarjeta de crédito).
Cliente:	persona que tiene una tarjeta.
Transacción:	transacción financiera.
Cuenta:	cuenta bancaria.
Comercio:	comercio con POS (Point of Sale).

Segundo, debemos definir los hechos del negocio. Los mismos son: Cliente tiene tarjeta, Cliente tiene cuenta, Transacción se efectúa con tarjeta, Transacción se efectúa desde comercio, Tarjeta tiene pin, Tarjeta tiene fecha de vigencia, Tarjeta tiene estado, Cuenta tiene grupo de afinidad y Cuenta tiene número.

La unión de los términos y hechos del negocio forma lo que llamamos *modelo de conceptos*.

3.4 Reglas de negocio

Las reglas controlan el comportamiento del negocio. Son oraciones declarativas construidas a partir de los términos y hechos del negocio. Las reglas tienen ciertas características que se enumeran a continuación. Son **independientes de los procesos o procedimientos**, esto significa que las reglas no se definen ni mantienen dentro del código de los procesos. Son **distintas e independientes de los eventos**, con evento se entiende un suceso que debe ser registrado por el sistema, como inserción, modificación o eliminación; esta independencia implica la no mención de eventos que puedan violar las reglas. Cada regla debe tener un **sujeto**, este sujeto debe estar explícito e indica el objeto sobre el que actúa la regla.

Algunos ejemplos de reglas:

- El estado de una tarjeta debe ser operativo si una transacción es efectuada por dicha tarjeta.
- El pin de una transacción debe ser igual al pin de la tarjeta que efectuó dicha transacción.
- La fecha de vigencia de una tarjeta debe ser mayor a la fecha de hoy si una transacción es efectuada con dicha tarjeta.

3.4.1 Categorías de reglas

Las reglas de negocio pertenecen a tres categorías principales: **Rechazadores**, **Productores** y **Proyectores**. A continuación se describen estas categorías y sus subcategorías derivadas.

3.4.1.1 Rechazador

Comúnmente llamada Restricción. Es aquella regla que impide (rechaza) un evento si éste resulta en una violación de la regla. Los rechazadores protegen al negocio de estados incorrectos, esto es, de información que viole las reglas del negocio. Por ejemplo se podría definir una regla que rechazase una transacción si el estado de la cuenta es no operativo para la cuenta vinculada a la transacción: *El estado de una cuenta debe ser operativo cuando una transacción opera sobre dicha cuenta*.

3.4.1.2 Productor

Es toda aquella regla que ni rechaza ni proyecta un evento sino que simplemente calcula o deriva un valor basándose de alguna función matemática. Dentro de esta categoría tenemos la regla de **Cálculo** y la regla de **Derivación**.

Regla de Cálculo es toda regla productora que calcula un valor mediante operaciones aritméticas estándares (por ejemplo, suma, multiplicación, promedio, etc.) especificadas explícitamente. Una regla de Cálculo provee una fórmula precisa de cómo se calcula el término computado. Por ejemplo: *El límite de crédito de una cuenta (actual) es igual al límite de crédito (anterior) de dicha cuenta más el importe de la transacción para la transacción que opera sobre dicha cuenta*.

Regla de Derivación es toda regla productora que deriva un valor booleano (verdadero o falso) mediante operaciones lógicas (por ejemplo, AND, OR, NOT, EQUAL TO, etc.) especificadas

explícitamente. Una derivación provee una definición precisa para un término derivado, esto es, un término booleano cuyo valor (verdadero o falso) siempre está establecido por las operaciones lógicas especificadas.

3.4.1.3 Proyector

Es toda regla que realiza alguna acción (excepto rechazar) cuando un evento relevante ocurre. Un proyector nunca rechaza eventos sino que los proyecta, esto es, causa un nuevo(s) evento(s) como resultado. Los proyectores prescriben generalmente el comportamiento automático del sistema. Ejemplo: se podría definir un proyector que enviara un e-mail al gerente de ventas (procedimiento envío mail por tarjeta vencida) si la fecha de vigencia de una tarjeta ha caducado: *Si la fecha de vigencia de una tarjeta es menor o igual a la fecha de hoy entonces enviar mail_por_tarjeta_vencida al gerente de ventas*.

Dentro de la categoría de proyector tenemos tres subcategorías: **Habilitador**, **Copiador** y **Ejecutivo**.

Habilitador es todo proyector que habilita o deshabilita algo. Dentro de esta subcategoría tenemos a las Reglas de Inferencia, los Habilitadores de Reglas o Reglas de Excepción, los Habilitadores de Procesos y los Habilitadores de datos.

Una **Regla de Inferencia** es un Habilitador que infiere algo verdadero bajo circunstancias apropiadas. Por ejemplo, una regla de inferencia podría decir que una tarjeta debe ser considerada vencida si la fecha de vigencia de dicha tarjeta ha caducado (regla de tarjeta vencida) o sea: *Tarjeta es tarjeta vencida si la fecha de vigencia de la tarjeta es menor o igual a la fecha de hoy*.

Un **Habilitador de Reglas** es una regla que habilita o deshabilita otra regla bajo determinadas circunstancias. Por ejemplo un habilitador de reglas podría ser una regla que apagara la "Regla de tarjeta vencida" si la cuenta tiene grupo de afinidad 1: *Si una transacción opera sobre una cuenta que tiene grupo de afinidad igual a 1 entonces deshabilitar regla_de_tarjeta_vencida*.

Un **Habilitador de Procesos** es una regla que habilita o deshabilita una operación, proceso, o procedimiento bajo determinadas circunstancias. Por ejemplo un habilitador de procesos podría ser una regla que inhabilitara el procedimiento "Envío mail por tarjeta vencida": *Deshabilitar proceso mail_por_tarjeta_vencida*.

Un **Habilitador de Datos** crea o elimina instancias de datos actuales bajo determinadas circunstancias. Por ejemplo un habilitador de datos podría eliminar las transacciones cuyas fechas sean anteriores a dos años de la fecha actual: *Borrar transacción si la fecha de la transacción es menor a la fecha de hoy menos 2 años*.

Un **Copiador** es un proyector que replica (copia) datos actuales. Hay dos clases de reglas copiadoras: **Regla de Impresión** y **Regla de Presentación**.

Una **Regla de Impresión** es un copiador que setea el valor de algo que persiste (por ejemplo, algo en la base de datos). Ejemplo: podría definirse una regla de impresión que actualizase el número de transacciones efectuadas por una tarjeta. *Cuando se efectúa una transacción con una tarjeta entonces el número de transacciones de dicha tarjeta debe incrementarse en uno.*

Regla de Presentación es un copiador que establece un valor o parámetro relacionado a cómo van a ser presentados los datos (en una pantalla, informe, etc.).

Por último tenemos los **Ejecutivos**, comúnmente llamados **Disparadores**. Son proyectores que causan que una operación, proceso, o procedimiento se ejecute, o que una regla se dispare. Tenemos dos tipos de ejecutivos: **Disparador de Proceso** y **Disparador de Regla**.

Un **Disparador de Proceso** es un proyector que causa la ejecución de una operación, proceso o procedimiento. Ejemplo: podría definirse un disparador para ejecutar el procedimiento *mail_por_tarjeta_vencida*.

Un **Disparador de Regla** es un proyector que dispara otra regla.

Resumimos la clasificación de reglas presentada en el siguiente esquema:

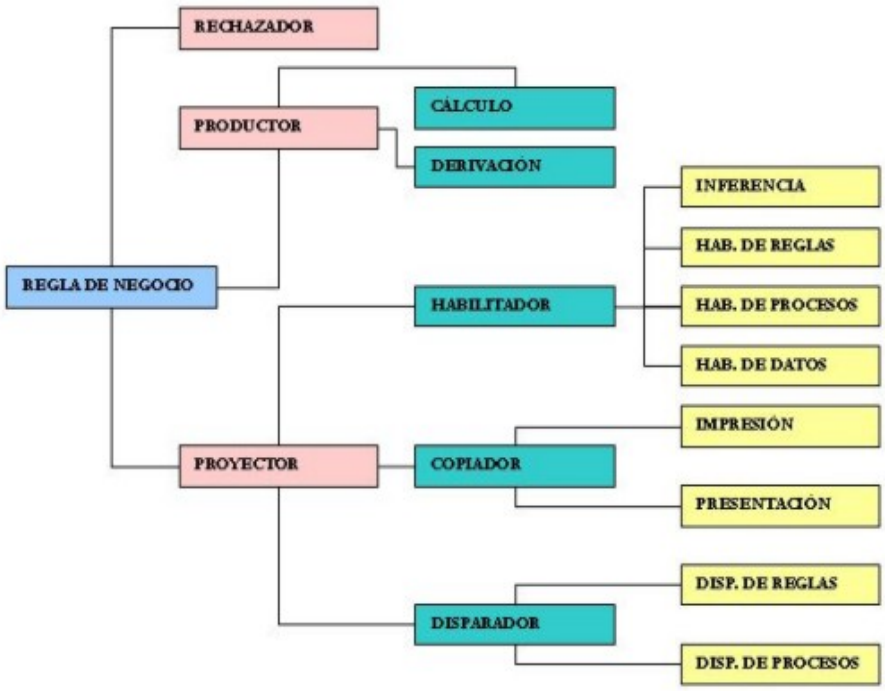


Figura 2. Clasificación de reglas de negocio de Ronald Ross

4 Funcionalidades del Sistema

4.1 Introducción

El sistema desarrollado permite la especificación de reglas sobre un modelo de conceptos (dominio) creado a partir de la base de datos del negocio. Las reglas son especificadas por el usuario en un lenguaje formal creado para tal fin, y luego son compiladas, de esta forma se determina su correctitud sintáctica y semántica. Para cada regla se genera código en *PL/SQL* y código *Java*. Finalmente, las reglas pueden ser evaluadas sobre instancias de los objetos del dominio utilizando los evaluadores *PL/SQL* y *Java*.

A continuación, realizamos un esquema de las funcionalidades que abarca el sistema las cuales detallaremos en lo que resta del capítulo. Dichas funcionalidades son:

- **Administración del sistema:** comprende las funcionalidades de administración de usuarios, administración de drivers y administración de los distintos negocios registrados.
- **Mapeo:** brinda la funcionalidad de la realización de una abstracción sobre la base de datos del negocio, habilitando la construcción de un modelo de conceptos sobre la misma, a partir del cual se diseñarán las reglas de negocio
- **Diseño y compilación de reglas:** contempla las funcionalidades de creación de conjuntos de reglas, reglas y versiones de reglas proveyendo una interfaz de usuario para tal fin. Incluye también la compilación de las reglas.
- **Generación de código de reglas:** incluye la generación del código de las reglas en lenguaje *Java* y *PL/SQL*, a partir de la especificación de las reglas en el lenguaje formal de especificación de reglas ya mencionado.
- **Evaluación de reglas:** permite evaluar las reglas sobre instancias de objetos del modelo de conceptos a partir de la construcción de árboles de evaluación creados para tal fin.

Para clarificar la estructura del sistema presentamos el siguiente diagrama del mismo:

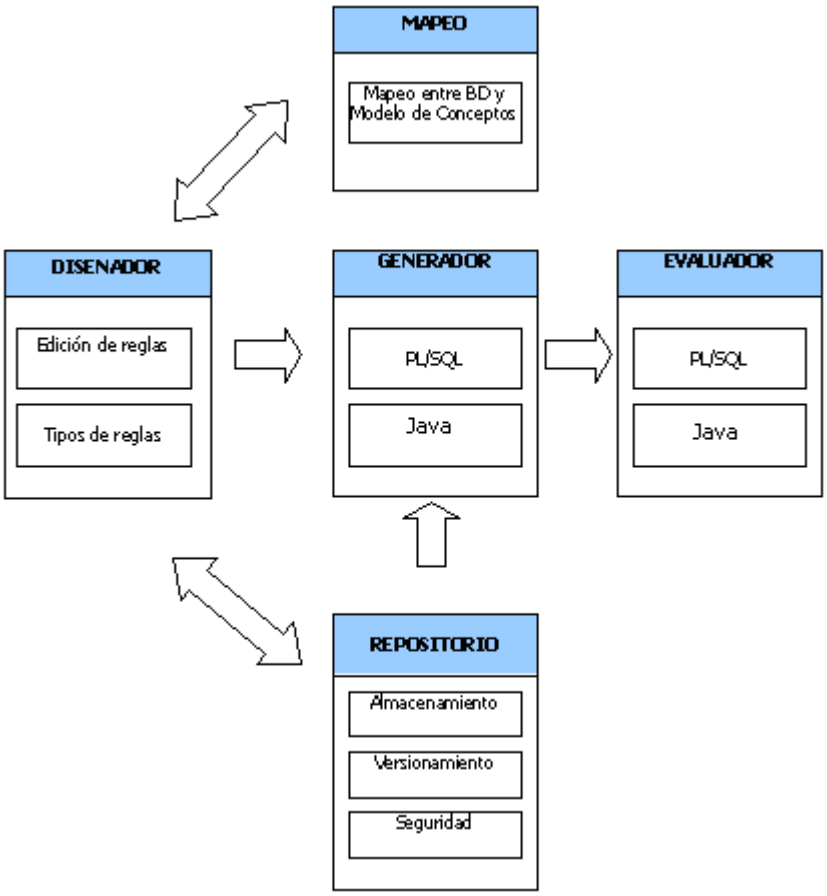


Figura 3. Módulos del generador de reglas de negocio

4.2 Funcionalidades relacionadas con la administración

Las funcionalidades de administración del sistema serán realizadas por un usuario administrador cuyas funciones principales son:

- Administración de usuarios:

  El usuario administrador es quien deberá dar de alta a los usuarios del sistema, al darles de alta deberá asignarles un perfil. El perfil es la posibilidad de leer y/o escribir en los diferentes módulos e indica para cada uno de los módulos (Mapeo, Diseñador y Evaluador) el nivel de permisos del que se dispone, el cual puede ser de lectura, escritura o ninguno de estos.

- Administración de drivers:

  El punto de entrada del generador es la base de datos de un negocio específico sobre la cual se creará un modelo de conceptos y luego se diseñarán las reglas. El sistema no impone restricciones sobre el DBMS al cual debe pertenecer la base de datos del negocio por lo que puede ser cualquier DBMS soportado por JDBC (*Oracle*, *MySql*, etc.). Por lo tanto, el administrador deberá dar de alta los distintos DBMS a los cuales pueden pertenecer las bases de datos de los negocios a crear.

- Administración de negocios:

La creación y eliminación de negocios es tarea exclusiva del usuario administrador, quien al momento de crear un nuevo negocio deberá especificar los datos para conectarse a la base de datos a partir de la cual se creará el negocio.

4.3 Mapeo

El "mapeo" consiste en la construcción de un modelo de conceptos a partir de la base de datos de un negocio, este es el primer paso a realizar luego de la creación del negocio ya que las reglas se definen sobre el modelo de conceptos. Si bien el modelo de conceptos debe definirse al inicio, el mismo puede modificarse en cualquier instancia, por ejemplo, en caso de que se hayan creado nuevos objetos en la base de datos del negocio correspondiente.

Veamos en primer lugar cuales son las ventajas de especificar las reglas de negocio sobre un modelo abstracto (modelo de conceptos) frente a lo que sería la especificación de las reglas directamente sobre la metadata de la base de datos del negocio, es decir, porque incluimos la realización de un mapeo en el sistema generador de reglas. Algunas de estas ventajas son:

- La representación de la realidad del negocio como un modelo de conceptos permite cierta independencia con respecto al DBMS de la base de datos del negocio, en el sentido de que el usuario que construye dicho modelo no necesita saber a que manejador de base de datos corresponde la base de datos a partir de la cual construye dicho modelo, sólo necesita conocer la metadata de la misma.
- A partir del punto anterior se desprende otra ventaja, el usuario que administra el mapeo no necesita tener conocimiento de ningún manejador de base de datos en particular, solamente tiene que tener nociones básicas de base de datos, como ser, que son las tablas, columnas y claves primarias en dicho contexto.
- El modelo de conceptos permite la utilización en cada caso de una nomenclatura propia que no tiene porque coincidir con la de la base de datos, es decir, los componentes del modelo de conceptos pueden ser nombrados diferente a sus correspondientes en la base de datos.
- Abstracción de los tipos de datos, los que pueden diferir según el DBMS de la base de datos del negocio. Por ejemplo, en *MySql* existe el tipo de datos *String* que denomina una cadena de caracteres, pero en *Oracle* no existe este tipo de datos sino que los tipos de datos utilizados para representar una cadena de caracteres son *Varchar* y *Varchar2*. En el modelo de conceptos existirán solamente cuatro tipos de datos y el sistema se encarga de mapear los tipos de datos utilizados en la base de datos del negocio con estos, manteniendo la consistencia necesaria.
- Fácilmente, se puede tomar "parte" de la base de datos del negocio utilizando sólo lo que nos interesa de la misma para construir las reglas, además se pueden construir varios proyectos sobre una misma base de datos. Por ejemplo, para una misma empresa se podrían construir dos modelo de conceptos distintos uno sobre la "porción" de la base de datos que mantiene la información del personal y otro sobre la "porción" que mantiene la información propia del negocio, esto es útil porque seguramente los tipos de reglas a aplicar sobre estos elementos serán diferentes.
- Como veremos más adelante el modelo de conceptos es fácil de construir y mantener por lo que su utilización no es costosa en estos aspectos.

En lo que resta de esta sección describiremos los dos componentes sobre los cuales se realiza el mapeo, estos son, la base de datos relacional del negocio y el modelo de conceptos, para luego detallar como se realiza el mapeo entre ambos.

El *Modelo de Conceptos* se construye sobre dos elementos, *términos* y *hechos*. En este proyecto nos basamos en el modelo de conceptos de *Ross* descrito en el capítulo 3, pero con algunos cambios. A los *términos* los clasificaremos en dos categorías disjuntas, *objetos* y *atributos*. Un *objeto* es un *término* que puede contener cero o más atributos. Un *atributo* es un *término* que debe pertenecer a un solo objeto.

Un *hecho* es una relación entre dos términos. Se lo representa como una sentencia con tres elementos, *sujeto*, *verbo* y *objeto* (en el sentido gramatical), y con el siguiente orden: <*Sujeto*> <*Verbo*> <*Objeto*>. Dónde *Sujeto* y *Objeto* son términos y *Verbo* representa la relación entre ambos términos. Este patrón no es válido para todo término, ya que algunas combinaciones no tienen sentido. Las combinaciones válidas se muestran a continuación:

- <*objeto1*> <*Verbo*> <o*bjeto2*>. Dónde objeto1 y objeto2 son dos términos de tipo objeto.
- <*objeto*> *tiene* <*atributo*>. Representa la última posibilidad, la relación de un objeto con su atributo. Sobre esta combinación se aplica la restricción de que el atributo sólo puede aparecer en un hecho.

Para poder trabajar con el *Modelo Relacional*, debemos identificar qué elementos del mismo participarán en el mapeo. Existen dos elementos útiles al mapeo, ellos son, *tabla* y *atributo*. *Tabla* es una tabla o relación del modelo, *atributo* es un atributo (columna) que participa en una o más relaciones del modelo. Resumiendo se tienen los siguientes elementos:

- Tabla, relación que vincula uno o más atributos: *Tabla*(a_1,…, a_n), dónde a_1,…, a_n son atributos del modelo.
- Atributo, elemento que aparece en una o más Tablas.

El mapeo entre los modelos tendrá como dominio el conjunto de elementos del modelo de conceptos y como codominio el conjunto potencia de los elementos del modelo relacional.

- Dominio: MC, tq. MC = objetos U atributos U hechos; objetos, atributos y hechos son respectivamente los objetos, atributos y hechos del modelo de Conceptos.
- Codominio: Σ^{MR}, tq. MR = tablas U atributos; tablas y atributos son respectivamente las tablas y atributos del modelo Relacional.
- Mapeo: *mc2mr*: MC -> Σ^{MR}.

A continuación se detallan los tipos de mapeos que se pueden realizar en el sistema, estos casos fueron definidos de forma tal de que todo negocio con una base de datos relacional pueda ser representado en un modelo de conceptos y como consecuencia de ello se puedan formular reglas de negocio sobre él.

Tipo	Dominio	Codominio
1	Objeto	Tabla
2	Atributo	Atributo
3	Hecho	Tabla y atributos de la tabla

4.3.1 Creación de objetos

Un objeto del modelo de conceptos se crea a partir de una tabla del modelo relacional, esto es lo que llamamos tipo de mapeo 1. No pueden existir dos objetos del modelo de conceptos que hayan sido creados a partir de una misma tabla y no pueden existir dos objetos con el mismo nombre en un modelo de conceptos. Cuando se crea un objeto, se agregan como atributos del mismo, todos los atributos de la tabla que forman la clave primaria, mediante un mapeo de tipo 2, de forma de no perder "la identidad" de la relación extendiéndola al objeto.

4.3.2 Creación de atributos

El tipo de mapeo 2 permite agregar atributos a un objeto del modelo de conceptos. Un atributo se crea a partir de un atributo (columna) de una tabla y dicho mapeo debe cumplir que la tabla a la que pertenece la columna con la cual se va a mapear el atributo coincide con la tabla a partir de la cual se creó el objeto, y que no existe ya un atributo del objeto mapeado con dicha columna. Además, no pueden existir dos atributos con el mismo nombre para un objeto dado.

De lo anterior se deduce que:

- La cantidad de atributos de un objeto será menor o igual a la cantidad de columnas de la tabla a partir de la cual se creó el objeto.
- Como mínimo, la cantidad de atributos de un objeto es igual a la cantidad de columnas que forman la clave primaria de la tabla con la que mapea el mismo.
- Al agregar un atributo a un objeto se está creando de forma implícita el hecho que representa la relación de un atributo con el objeto al que pertenece, aunque esta relación no será considerada explícitamente como hecho en nuestro modelo de conceptos.

4.3.3 Creación de hechos:

Para crear un hecho en el modelo de conceptos se deben especificar los siguientes elementos:

- Los dos objetos del modelo sobre los cuales se construirá el hecho.
- Tabla del modelo relacional que mapea con el hecho a crear.
- Atributos del hecho, que corresponden a los atributos de los objetos que definen el hecho y su correspondencia con las columnas de la tabla a partir de la cual se creará el hecho.

Los atributos de los objetos que definen el hecho deben cumplir que:

- Son la unión de los atributos que forman la clave primaria de los dos objetos.
- Dado que la tabla que define el mapeo es la misma con la que mapea uno de los objetos, los atributos de este objeto se corresponden con las columnas que forman la clave primaria de la tabla.
- Para el otro objeto todos sus atributos que forman la clave primaria tienen que corresponderse con una columna distinta de la tabla, que podría eventualmente ser alguna de las columnas con las cuales se mapean atributos del otro objeto (pero no de sí mismo.)

Observar que esta correspondencia puede estar o no explicitada en la base de datos mediante claves foráneas.

Puede existir más de un hecho (definidos sobre la misma tabla) entre dos objetos, pero deben diferir en los atributos que definen el mapeo por parte del objeto que no es el que mapea con la tabla que define el hecho.

4.3.4 Ejemplo de mapeo

Para ilustrar la creación de un modelo de conceptos presentaremos un ejemplo utilizando parte de la realidad del negocio de medios de pago presentado en el capítulo 3.

Modelo relacional:

TABLA	COLUMNAS	CLAVE PRIMARIA	CLAVE FORÁNEA
Cuentas	emisor sucursal_emisor numero_cuenta grupo_afinidad	emisor sucursal_emisor numero_cuenta	
Tarjetas	tarjeta pin emisor sucursal_emisor numero_cuenta	tarjeta	(emisor,sucursal_emisor,numero_cuenta) correspondientes a Cuentas(emisor,sucursal_emisor,numero_cuenta)

Modelo de conceptos:

OBJETOS	
Objeto	**Tabla**
Cuenta	Cuentas
Tarjeta	Tarjetas

Se crean dos objetos en el modelo de conceptos, *Cuenta* y *Tarjeta* que mapean con las tablas *Cuentas* y *Tarjetas* respectivamente.

ATRIBUTOS			
Atributo	**Objeto**	**Tabla**	**Columna**
emisor	Cuenta	Cuentas	emisor
sucursal_emisor	Cuenta	Cuentas	sucursal_emisor
numero_cuenta	Cuenta	Cuentas	numero_cuenta
grupo_afinidad	Cuenta	Cuentas	grupo_afinidad
tarjeta	Tarjeta	Tarjetas	tarjeta
pin	Tarjeta	Tarjetas	Pin
emisor	Tarjeta	Tarjetas	emisor
sucursal_emisor	Tarjeta	Tarjetas	sucursal_emisor
numero_cuenta	Tarjeta	Tarjetas	numero_cuenta

En la tabla anterior se listan todos los atributos creados. Para cada atributo se detalla el objeto al que pertenece y la columna de la tabla con la cual se mapea, columna que pertenece a la tabla con la cual se mapeó el objeto. Notar que los atributos que mapean con columnas que forman parte de la clave primaria de la tabla serán creados de forma automática por el sistema al crear los objetos correspondientes.

HECHOS					
HECHO	**TABLA**	**OBJETOS**	**ATRIBUTOS QUE DEFINEN EL HECHO**		
tarjeta_tiene_cuenta	Tarjetas	Tarjeta y Cuenta	**ATRIBUTO**	**OBJETO**	**COLUMNA** (de la tabla que mapea con el hecho)
			tarjeta	Tarjeta	tarjeta
			emisor	Cuenta	emisor
			sucursal_emisor	Cuenta	sucursal_emisor
			numero_cuenta	Cuenta	numero_cuenta

El hecho *tarjeta_tiene_cuenta* relaciona los objetos *Tarjeta* y *Cuenta* y es mapeado con la tabla *Tarjetas*. Dado que la tabla a partir de la cual se construye el hecho (Tarjetas) coincide con la tabla de mapeo del objeto *Tarjeta*, todos los atributos (sólo el atributo *tarjeta* en este caso) que forman la clave primaria de dicho objeto participan en la definición del mapeo, los cuales se corresponden con las columnas de la tabla que forman la pk. Observamos también, que para cada atributo que forma parte de la pk del objeto *Cuenta* (*emisor, sucursal_emisor* y *numero_cuenta*) se ha establecido su correspondencia con una columna de la tabla Tarjetas.

Notar que el hecho creado es explícito en la base de datos ya que las columnas *emisor, sucursal_emisor* y *numero_cuenta* son claves foráneas en la tabla *Tarjeta*s que referencian a la tabla *Cuentas* a partir de la cual se mapea el objeto *Cuenta*. Sin embargo, no es restricción del sistema la existencia de la clave foránea en estos casos por eso debe especificarse la correspondencia de columnas con atributos, en la cual los tipos de datos deben coincidir para que esto sea válido. Otra observación es que en caso de querer definir otro hecho entre los mismos objetos y la misma tabla los atributos que forman la clave primaria del objeto *Cuenta* deberían corresponderse con otras columnas diferentes que las que se corresponden en este caso, si esto fuera posible y tuviera sentido hacerlo.

Un diagrama del modelo de conceptos creado es el siguiente:

Figura 4. Modelo de conceptos

Los asteriscos (*) antes de un atributo denotan que el mismo forma parte de la clave primaria del objeto.

4.4 Diseñador

El diseñador es el módulo que permite la especificación de reglas de negocio sobre un modelo de conceptos correspondiente a un negocio.

Los puntos de entrada del diseñador son:

- Modelo de conceptos del negocio, el cual fue creado en el mapeo.
- Tipos de reglas, los cuales ya están determinados y se detallarán más adelante.
- Lenguaje de especificación de reglas.

Las reglas se agrupan en conjunto de reglas tal que toda regla pertenece a un único conjunto y a su vez un conjunto puede contener varias reglas y conjuntos. Además, para cada regla se pueden crear varias versiones, de las cuales sólo una puede estar activa en un momento dado (es la versión que se utilizará en la evaluación.)

Como hemos mencionado en el capítulo 3 la clasificación de reglas que utilizamos en nuestro proyecto se basa en la categorización de reglas de negocio realizada por *Ronald Ross*, la cual adaptamos al modelo de conceptos definido y creamos un lenguaje que permite su especificación formal. A continuación, detallaremos los tipos de reglas de negocio soportadas por el sistema y la definición del lenguaje formal creado para especificarlas.

4.4.1 Definiciones previas

4.4.1.1 Objeto evaluado

Llamaremos **objeto evaluado** en una regla al único objeto (objeto del modelo de conceptos) sobre el cual se evaluará la regla, esto significa que este objeto es un parámetro de la evaluación por lo tanto se considerará que dicha instancia no existe en la base de datos y las consultas y/o actualizaciones de datos sobre el objeto se harán en el mismo objeto recibido como parámetro. En cambio, las consultas y/o actualizaciones sobre cualquier otro objeto que aparezca en la regla sí se realizarán en la base de datos. Como se verá más adelante no todos los tipos de reglas requieren la especificación de un objeto evaluado, pero los que la requieren no permiten la evaluación de la regla sobre otro tipo de objetos del modelo de conceptos.

4.4.1.2 Especificación de funciones predefinidas, funciones de usuario, condiciones y fórmulas

Las **funciones predefinidas** son funciones que brindará el generador de reglas de negocio para facilitar la especificación de reglas. En primer lugar vamos a definir el tipo de datos colección que se utilizará en la especificación de reglas.

Las características de una **colección** son:

- Está compuesta por objetos del modelo de conceptos (Ej.: Comercio, Cuenta, etc.) de un mismo tipo (Ej.: colección de comercios, colección de cuentas, etc.)
- No tiene elementos repetidos.
- No está ordenada.

En el siguiente recuadro detallamos las **funciones predefinidas** existentes:

- **Seleccionar**(colección| nombre_objeto, nombre_instancia, condición): colección
  Permite seleccionar de una colección los elementos que cumplen una determinada condición. Se debe especificar un nombre para referenciar a los elementos de la colección en la condición.
  Se especificará más adelante la estructura de las condiciones.

  Para aclarar el funcionamiento veamos un ejemplo: Supongamos que tenemos el objeto Tarjeta en el modelo de conceptos y queremos obtener una colección con todas las tarjetas cuyo grupo de afinidad es 1, esto se especificaría de la siguiente manera:

  Sea C1 la colección de todas las tarjetas existentes en la base de datos.
  C2 = Seleccionar(C1,t, t.grupo_afinidad = 1)
  Donde C1 y C2 son colecciones.

  Otra opción es especificar el nombre de un objeto existente en el dominio en lugar de una colección, en este caso la función retornará unja colección con todas las instancias almacenadas del objeto que cumplen la condición.
- **Sum**(colección, atributo): Numérico
  Esta función retorna la suma del atributo especificado de todos los objetos de la colección. El atributo debe ser de tipo numérico.
- **Prom**(colección, atributo): Numérico
  Retorna el promedio del atributo especificado de todos los objetos de la colección. El atributo debe ser de tipo numérico.
- **Min**(colección, atributo): Numérico o Fecha
  Retorna el valor mínimo de un atributo de los objetos de la colección, el mismo puede ser numérico o fecha.
- **Max**(colección, atributo): Numérico o Fecha
  Retorna el valor máximo de un atributo de los objetos de la colección, el mismo puede ser numérico o fecha.
- **Cont**(colección): Entero
  Retorna la cantidad de elementos de la colección especificada.
- **EsVacia**(colección): bolean
  Retorna verdadero si la colección es vacía, falso en caso contrario.
- **Intersección**(colección, colección): colección
  Retorna una colección con los elementos que se encuentran en ambas colecciones, esto es, la intersección de las colecciones.
- **Iguales**(colección, colección): boolean
  Retorna verdadero si las colecciones son iguales, es decir si tienen los mismos elementos sin considerar el orden de los mismos en la colección.
- **Pertenece**(colección, objeto): boolean
  Retorna verdadero si el objeto especificado se encuentra en la colección
- **Largo**(string): Entero
  Devuelve el largo del string.
- **Concat**(s1: string,s2: string): String
  Retorna un string que es el resultado de concatenar s2 a s1.
- **Abs**(Numérico): Numérico
  Calcula el valor absoluto de un número.
- **Hoy**(): Fecha
  Retorna la fecha del día.
- **Dia**(Fecha): Entero
  Retorna el día correspondiente a la fecha especificada.
- **Mes**(Fecha): Entero
  Retorna el mes correspondiente a la fecha especificada.
- **Anio**(Fecha): Entero
  Retorna el año correspondiente a la fecha especificada.

Las **funciones de usuario** son funciones externas al sistema implementadas por el usuario y pueden ser utilizadas en el diseñador ingresando su nombre (string), sus parámetros y el valor de retorno. Los parámetros y el valor de retorno de la función pueden ser del tipo de cualquiera de los objetos del modelo de conceptos y de los tipos básicos numérico, string, fecha o booleano.

Un **procedimiento de usuario** se especifica de igual forma que una función de usuario, la diferencia es que no retorna ningún valor.

Una **fórmula** al evaluarse puede dar un resultado de tipo numérico, string o fecha y resulta de una combinación válida de elementos de los siguientes tipos:

- Operadores numéricos: +, -, / ,*
- Funciones predefinidas que retornan literales numéricos, fechas o strings
- Funciones de usuario (que retornen numéricos, fechas o strings)
- Atributos (numérico, fecha o string)
- Literales (numérico, fecha o string)

Una **condición** es una expresión booleana que resulta de combinaciones válidas de elementos de los siguientes tipos:

- Operadores lógicos: and, or y not.
- Operadores relacionales: =, <, >, <=, >= y <>.
- Fórmulas
- Objetos
- Atributos
- Literales booleanos: verdadero o falso
- Funciones predefinidas que retornan booleanos
- Funciones de usuario (que retornen booleanos)

4.4.1.3 Definición de objetos y colecciones

Antes de especificar una regla se deben "definir" todos los objetos y colecciones de objetos que aparezcan en la misma, la definición de los mismos es similar a la declaración de variables en un lenguaje de programación.

Los **objetos** se pueden definir de dos formas:

- El objeto se define como el objeto evaluado

  Esta definición sólo es válida en los casos en que se define un objeto evaluado, el mismo es único y obligatorio en algunos tipos de reglas como veremos más adelante.

- Un objeto se define por un camino de hechos a partir del objeto evaluado

  Los objetos se pueden definir a partir de un camino de hechos a partir del objeto evaluado, pero hay que tener en cuenta que esto sólo es válido cuando los caminos permiten definir de forma única a los objetos.

Veamos más claramente este último punto: supongamos que tenemos los objetos *Cliente* y *Tarjeta* y el hecho "*Cliente tiene tarjetas".* En este caso no sería correcto definir un objeto Tarjeta como el camino de hechos (que contiene un solo hecho) que va desde Cliente a Tarjeta ya que el mismo no define un objeto sino más bien una colección de objetos. Si bien esto no se

puede identificar en el modelo de conceptos de forma explícita porque el mismo no maneja cardinalidad, se utilizará información de la base de datos para realizar este tipo de controles. Más específicamente, se permitirá ir de un objeto A hacia un objeto B en un camino de hechos sólo si la tabla que define el hecho es la tabla a partir de la cual se creó el objeto A.

Las **colecciones** se pueden definir de dos formas:

- Mediante un camino de hechos que parte del objeto evaluado

  Este caso es similar a la definición de objetos mediante caminos de hechos controlando que los objetos intermedios del camino queden definidos de forma única utilizando la estrategia ya explicada. Además, hay que controlar que el camino en efecto termine en una colección, esto se realizará de la siguiente forma: sean los últimos dos objetos del camino los objetos A y B en ese orden, el hecho del camino entre los objetos A y B debe tener como tabla de mapeo la tabla a partir de la cual se creó el objeto B.

- Mediante la invocación de funciones que retornan colecciones

4.4.2 Tipos de reglas

Luego de definidos los elementos básicos del lenguaje y la forma de definir objetos y colecciones se explicará la funcionalidad de cada uno de los tipos de reglas disponibles en el generador de reglas y su especificación formal. Los tipos de reglas que proveerá el generador son: rechazadores, cálculos, y disparadores de procesos.

4.4.2.1 Rechazador:

Una regla de este tipo rechaza acciones si no se cumplen determinadas restricciones sobre los datos. En este tipo de reglas **siempre** existe un objeto evaluado que es justamente el objeto que se evalúa.

Existen dos tipos de rechazadores, rechazador 1 y rechazador 2.

4.4.2.1.1 Rechazador 1

El rechazador 1 tiene la siguiente estructura: **[<condición>] <consecuencia>**

Este tipo de rechazador evalúa primero una condición, si la misma se cumple evalúa la consecuencia y retorna "verdadero" si esta se cumple y "falso" en caso contrario. Si la condición no se cumple la consecuencia no se evalúa y el retorno es "verdadero". La tabla de verdad en este caso sería:

Condición	Consecuencia	Verdadero	Falso
Verdadero		Verdadero	Falso
Falso		Verdadero	Verdadero

Podemos observar entonces que en el único caso en que se rechaza es cuando la condición es verdadera y la consecuencia falsa.

Sabemos ya cómo puede estar formada la condición, resta definir que es una consecuencia.

Existen 3 tipos de consecuencias posibles:

- **Relación binaria**: <hecho> <cardinalidad>

Esta regla se utiliza para restringir las cardinalidades de una relación, específicamente restringe la cardinalidad en un solo sentido, en el extremo que corresponde al segundo objeto del hecho.
La cardinalidad se define dando los dos extremos numéricos que definen el rango y operadores relacionales, por ejemplo: 1>=, 5<=. Su definición se formalizará más adelante.
El objeto evaluado debe ser uno de los dos objetos especificados en el hecho.
Si se cumple la restricción de cardinalidad en la instancia que se evalúa, retorna verdadero.

Ejemplo:

Regla: "Una transacción debe tener una tarjeta asociada"

Especificación de la regla	
Objeto evaluado:	Transacción
Variable del objeto evaluado:	Tx
Declaración de variables:	Tas coleccion Tarjeta := camino Tx : Transaccion_es_hecha_con_Tarjeta;
Código de la regla:	Transacción_es_hecha_con_Tarjeta(Tx:Tas) tiene cardinalidad = 1

Analicemos ahora el ejemplo presentado.

El objeto evaluado de la regla es Transacción esto implica que esta regla se evaluará siempre sobre instancias de dicho objeto. Siempre que se define un objeto evaluado se debe especificar un nombre para referenciar a dicho objeto en el código de la regla, esto es, la variable del objeto evaluado, Tx en este caso.

Asumimos que fue declarado el hecho Transacción_es_hecha_con_tarjeta, el que relaciona los objetos Transacción y Tarjeta mediante un mapeo de hechos donde la tabla con la que mapea el hecho es la tabla Transacciones.
Se declara la variable Tas de tipo colección de Tarjeta, como el camino definido por el hecho Transacción_es_hecha_con_tarjeta.

Finalmente, que una transacción deba tener siempre una tarjeta asociada implica que la cardinalidad de la relación entre ambos sea 1, que es lo que se especifica en el código de la regla.

Por último, observamos que este rechazador no tiene condición.

- **Comparación**: <atributo> <op. relacional> (<atributo>|<literal> | null)

Las comparaciones se utilizan para controlar la correctitud de los valores de los atributos. Si la comparación se cumple, la evaluación retorna verdadero.
El objeto evaluado es en este caso el objeto al que pertenece el atributo que se compara.

Ejemplo:

Regla: "El pin de una transacción debe coincidir con el de la tarjeta asociada"

Especificación de la regla	
Objeto evaluado:	Transacción
Variable del objeto evaluado:	Tx
Declaración de variables:	Ta instancia Tarjeta := camino Tx : Transaccion_es_hecha_con_Tarjeta;
Código de la regla:	Tx.pin es = Ta.pin

Esta regla realiza la validación de pin, lo que implica que el pin de la transacción debe coincidir con el de la tarjeta asociada a la transacción.

- **Inclusión:** <atributo> <lista de literales>

Está regla tiene similar función que la anterior, la diferencia es que utiliza una lista de valores para comparar y no un solo valor, y determina si el atributo pertenece o no a la lista de literales.

Al igual que en el caso anterior el objeto evaluado es el objeto al que pertenece el atributo que se compara.

Ejemplo:

Regla: "Si el límite de crédito de una tarjeta es mayor a 15000 el grupo de afinidad de la misma debe ser 1, 2 ó 3"

Especificación de la regla	
Objeto evaluado:	Tarjeta
Variable del objeto evaluado:	Ta
Declaración de variables:	No tiene
Código de la regla:	Condición: Ta.limite_credito > 15000 Consecuencia: Ta.grupo_afinidad en {1,2,3}

Este ejemplo, a diferencia de los anteriores, tiene una condición lo que implica que la consecuencia sólo se evaluará en caso de que la condición sea verdadera.

4.4.2.1.2 Rechazador 2

El rechazador 2 tiene la siguiente estructura: **<consecuencia> sólo si <condición>**

Este rechazador tiene un significado diferente al anterior forzando a que si la condición no se cumple la consecuencia no puede cumplirse, en el caso de que esto sucediera se rechaza, es decir, retorna falso.

Condición	Consecuencia	Verdadero	Falso
Verdadero		Verdadero	Verdadero
Falso		Falso	Verdadero

La consecuencia tiene prácticamente el mismo formato que en los rechazadores de tipo 1:

- Relación binaria: <objeto>

  En este caso se va a rechazar el objeto especificado si la condición no se cumple, además este mismo objeto es el objeto evaluado. En este caso como la "consecuencia" es un objeto no tiene sentido su evaluación booleana, por lo que siempre es "verdadero".

Ejemplo:

Regla: "Una tarjeta no puede tener más de tres transacciones aprobadas en un mismo día"

Especificación de la regla	
Objeto evaluado:	Transacción
Variable del objeto evaluado:	Tx
Declaración de variables:	Ta instancia Tarjeta := camino Tx: transaccion_tarjeta; Txs_aprobadas coleccion Transacción := seleccionar (@Transaccion, inst, inst.numero_tarjeta = Ta.numero AND inst.fecha = Tx.fecha);
Código de la regla:	Condición: cont (Txs_aprobadas) < 3 Consecuencia: Transacción

Analicemos la especificación de la regla presentada como ejemplo:

En primer lugar, se obtiene la tarjeta (Ta) a la que está asociada la transacción (Tx) sobre la que se evalúa la regla. Luego, utilizando la función predefinida "Seleccionar" se obtiene la colección de todas las transacciones almacenadas que corresponden al mismo número de tarjeta (Ta) y que fueron realizadas en el día. Finalmente, si la colección obtenida tiene 3 o más elementos se rechaza la transacción.

- **Comparación:** <atributo> <op. relacional> (<atributo> | <literal> | null)

Análogo al primer caso.

- **Inclusión:** <atributo> <lista de literales>

Análogo al primer caso.

4.4.2.2 Cálculo:

Las reglas de cálculo tienen la siguiente estructura:

[<condición>] (<atributo> (<fórmula> | <exp.booleana>))+

Este tipo de reglas se utiliza para calcular el valor de uno o más atributos en base a una fórmula o en base a una expresión booleana en el caso de atributos de tipo booleano. En caso de especificar

una condición, se evaluará esta en primer lugar, si el resultado es falso no se realizará el cálculo, en caso de que la evaluación de la condición sea verdadero o no exista condición entonces sí se realizará el cálculo. El atributo puede ser de tipo numérico, booleano, string o fecha.

La regla no retorna ningún valor, solamente modifica el valor del atributo.

Este tipo de reglas debe tener un objeto evaluado siempre.

Ejemplo:

Regla: "El atributo dis_planes de una Transacción se setea con el atributo limite_financiacion de la Cuenta cuya Tarjeta es la Tarjeta de la Transacción."

Especificación de la regla	
Objeto evaluado:	Transacción
Variable del objeto evaluado:	Tx
Declaración de variables:	Cta instancia Cuenta := camino Tx : Transaccion_es_hecha_con_Tarjeta :Tarjeta_tiene_Cuenta
Código de la regla:	Tx.disp_planes := Cta.limite_financiacion;

4.4.2.3 Disparador de procesos:

Los disparadores de procesos tienen la siguiente estructura:

<condición> <procedimiento de usuario>

Esta clase de regla se utiliza para ejecutar un procedimiento cuando se cumple determinada condición. Se deben especificar los parámetros del procedimiento. La especificación de un objeto evaluado es opcional.

Ejemplo:

Regla: "Si la fecha de vigencia de una tarjeta es menor o igual a la fecha de hoy entonces enviar mail_por_tarjeta_vencida al gerente de ventas"

Especificación de la regla	
Objeto evaluado:	Transacción
Variable del objeto evaluado:	Tx
Declaración de variables:	Ta instancia Tarjeta := camino Tx : Transaccion_es_hecha_con_Tarjeta;
Código de la regla:	Condición: Ta.fecha_vigencia <= Tx.fecha Consecuencia: mail_por_tarjeta_vencida (Ta)

En este ejemplo asumimos que ha sido definido por el usuario el procedimiento *mail_por_tarjeta_vencida.*

A modo de resumen en el siguiente cuadro se presenta un esquema de las reglas de negocio que pueden definirse en el sistema:

TIPOS DE REGLAS DE NEGOCIO:

- **Rechazador**
  - **Rechazador 1:** [<condición>] <consecuencia>
  - **Rechazador 2:** <consecuencia> sólo si <condición>
- **Cálculo:** [<condición>] (<atributo> (<fórmula> | <exp.booleana>))+
- **Disparador de procesos:** <condición> <procedimiento de usuario>

Figura 5. Tipos de reglas de negocio que soporta el generador de reglas

4.4.3 Lenguaje de especificación de reglas

En la administración de las reglas de negocio se manejará el versionamiento de las mismas, por lo tanto, para cada regla pueden existir una o más versiones de las cuales sólo una estará activa en un momento dado, la versión activa es la que se utilizará en la evaluación. Además, las reglas se organizarán en conjuntos de reglas, dicha organización es similar a una estructura de directorios donde un conjunto puede contener otros conjuntos y reglas y una regla pertenece a un único conjunto.

Para la especificación de las reglas se utilizó *EBNF*. Los no terminales se representan entre "<" y ">", los terminales en negrita, "::=" define los no terminales, "|" es para elegir entre alternativas, "[]" es para opcionalidad, "()" es para lista de alternativas, "*" y "+" para clausura de Kleene, y "" es para encerrar caracteres. A continuación se especifica el lenguaje:

Regla de negocio y versiones

<regla_negocio> ::=<nombre_regla> <tipo_regla> [<objeto> <variable>] <version_regla>+
<nombre_regla> ::= **identificador**
<tipo_regla> ::= **rechazador_1** | **rechazador_2** | **calculo** | **disparador_proceso**
<objeto> ::= **identificador**
<variable> ::= **identificador**
<version_regla> ::= <id_version> <nombre_regla> <especificacion_version>
<id_version> ::= **secuencial**
<especificacion_version> ::= [<declaracion_variables>] <implementacion_regla>
<declaracion_variables> ::= <declaracion_variable>+

Declaración de variables

<declaracion_variable>::=<variable> (**instancia** | **coleccion**) <objeto> ":=" <valor_variable> ";"
<valor_variable> ::= (**camino** <variable> ":" <resto_camino> | <funcion_coleccion>)
<resto_camino> ::= (<hecho> | <resto_camino> ":" <hecho>)
<hecho> ::= **identificador**

Implementación de la regla

<implementacion_regla> ::= (<rechazador_1_relacion> |
<rechazador_1_comparacionacion> |
<rechazador_1_inclusion> |
<rechazador_2_relacion> |
<rechazador_2_comparacionacion> |
<rechazador_2_inclusion> |
<calculos> |
<disparador_proceso>)
<rechazador_1_relacion> ::= [<condicion>] <consecuencia_relacion>
<rechazador_1_comparacionacion> ::= [<condicion>] <consecuencia_comparacion>
<rechazador_1_inclusion> ::= [<condicion>] <consecuencia_inclusion>
<rechazador_2_relacion> ::= <condicion>
<rechazador_2_comparacionacion> ::= <condicion> <consecuencia_comparacion>
<rechazador_2_inclusion> ::= <condicion> <consecuencia_inclusion>
<consecuencia_disparador_proceso>

<calculos> ::= [<condicion>] <consecuencia_calculos>
<disparador_proceso> ::= [<condicion>] <consecuencia_disparador_proceso>

Consecuencia y Condición

<consecuencia_relacion> ::= <relacion> **tiene cardinalidad** <cardinalidad>
<consecuencia_comparacion>::= <atributo> **es** <comparacion>
<consecuencia_inclusion> ::= <atributo> **esta en** <lista_literales>
<consecuencia_calculos> ::= <calculo>+
<consecuencia_disparador_proceso> ::= <funcion_usuario>

<relacion> ::=<hecho> "(" <variable> ":" <variable> ")"

<atributo> ::=<variable> "." <nombre_atributo>

<nombre_atributo> ::=**identificador**

<cardinalidad> ::=(("<"|"<="|">"|">="|"="|"<>") **natural** |
(">"|">=") **natural and** ("<"|"<=") **natural**)
<comparacion> ::= (("<"|"<="|">"|">="|"="|"<>") <atributo> |
("<"|"<="|">"|">="|"="|"<>") <literal> |
("<"|"<="|">"|">="|"="|"<>") <funcion_usuario> |
("<"|"<="|">"|">="|"="|"<>") <funcion_predefinida> |
(">"|">=") <literal> **and** ("<"|"<=") <literal> |
("="|"<>"|) **null**)

<lista_literales> ::= "{" <literal>+ "}"
<literal> ::= (<literal_booleano> | <literal_numerico> | <literal_string> | <literal_fecha>)
<literal_booleano> ::= **true** | **false**
<literal_numerico> ::= **natural** | **decimal**
<literal_string> ::= **string**
<literal_fecha> ::= **fecha** | **fecha** ":" **hora**
<condicion> ::= <expresion_booleana>| <literal_booleano>
<calculo> ::= <atributo> ":=" (<expresion_booleana>| <formula>) ";"

Expresión booleana y fórmula

<expresion_booleana> ::= (<formula> ("<"|"<="|">"|">="|"="|"<>") <formula> |
<expresion_booleana> (**and**|**or**) <expresion_booleana> |
not <expresion_booleana> |
"(" <expresion_booleana> ")")
<formula> ::= (<literal> | **null** | <atributo> | <variable> | "-" <literal>| "-" <atributo> |
<formula> ("+"|"-"|"*"|"/") <formula>| <funcion_predefinida> | <funcion_usuario> | "("
<formula> ")")

Funciones que retornan colecciones

<funcion_coleccion>::= (**seleccionar** "(" <variable> "," <nombre_instancia> ","
<expresion_booleana> ")" |
seleccionar "(" "@" <objeto> "," <nombre_instancia> ","
<expresion_booleana>)"| **interseccion** "(" <variable> "," <nombre_atributo> ")")
<nombre_instancia>::= <variable>

Funciones predefinidas y de usuario

```
<funcion_predefinida>        ::=   (sum  "(" <variable> "," <nombre_atributo> ")"  |
                                    prom "(" <variable> "," <nombre_atributo> ")" |
                                     min  "(" <variable> "," <nombre_atributo>  ")"  |
                                    max  "(" <variable> "," <nombre_atributo> ")"  |
                                    cont "(" <variable> ")"                                  |
                                    largo "(" <formula> ")"                                  |
                                    concat "(" <formula> "," <formula> ")"                   |
                                    abs "(" <formula> ")"                                    |
                                     hoy "("  ")"                                            |
                                    anio "(" <formula> ")"                                   |
                                    mes "(" <formula> ")"                                    |
                                    dia "(" <formula> ")"                                    |
                                    hora "(" <formula> ")"                                   |
                                     minuto "(" <formula> ")"                      |
                                     segundo "(" <formula> ")"                               |
                                    esVacia "(" <variable> ")"                    |
                                    iguales "(" <variable> "," <variable> ")"                |
                                    pertenece "(" <variable> "," <variable> ")" )
<funcion_usuario>     ::=     ["::"<paquete> "@" <clase> "@"]  <nombre_funcion> "(" [<lista_parametros>] ")"
<paquete>             ::= (identificador  | <paquete> "." identificador)
<clase>               ::= identificador
<nombre_funcion>  ::=          identificador
<lista_parametros> ::=         (<parametro>  | <lista_parametros> "," <parametro>)
<parametro>           ::= (<variable>  | <atributo> | <literal>)
```

Terminales

```
Identificador          ::=      ("LETRA" | "_")+
Natural                ::=      "DIGITO"+
Decimal                ::=      "DIGITO"+  "." "DIGITO"+
String                 ::=      "COMILLA DOBLE" "SIMBOLO"+ "COMILLA DOBLE"
Fecha                  ::=     "COMILLA SIMPLE"  día "/" mes "/" año "COMILLA SIMPLE"
hora                   ::=      "COMILLA SIMPLE"  hora ":" minuto ":" segundo "COMILLA SIMPLE"
```

4.5 Evaluador

El *Evaluador*, es el componente que permite ejecutar las reglas diseñadas, sobre instancias de objetos del modelo de conceptos definido para un negocio en particular.

Como veremos más adelante previo a su utilización el *Evaluador* debe ser parametrizado y dicha parametrización depende totalmente del modelo de conceptos y las reglas definidas, por lo que debe realizarse para cada negocio existente en el sistema, esto es, para cada modelo de conceptos definido.

Primeramente, listaremos las funcionalidades relacionadas con la parametrización (o administración) las cuales realizará un usuario administrador del evaluador, para luego explicar el proceso de evaluación.

4.5.1 Parametrización del evaluador

Como se mencionó anteriormente, previo a la utilización del *Evaluador* se debe realizar la parametrización del mismo lo que consiste en asignar para cada objeto (o para los objetos que sea necesario) del modelo de conceptos las reglas que se van a aplicar.

Ya hemos mencionado que al diseñar una regla se puede (se debe en algunos casos) definir un objeto evaluado al especificarlas, esto significa que estas reglas solamente van a ser evaluadas sobre ese objeto del modelo. En el caso de reglas que no poseen un objeto evaluado las mismas pueden aplicarse sobre cualquier objeto del dominio, de todas formas en ambos casos hay que realizar la asignación de la regla al objeto explícitamente.

Además de poder asignar las reglas a los objetos, se permite modificar el orden en que se evaluarán y crear árboles de evaluación en base a condiciones sobre los atributos del objeto.

Brindaremos en primer lugar las características del árbol de evaluación:

- Un objeto del modelo de conceptos puede tener asignado un único árbol de evaluación, pudiendo no tener ninguno en cuyo caso no se aplicarán reglas sobre él.
- Cada nodo del árbol consiste en una lista ordenada de reglas, en la cual para las reglas que tengan objeto evaluado, el mismo tiene que ser el objeto para el que se construye el árbol. Además, se puede asociar a cada nodo una etiqueta de forma de facilitar la comprensión del árbol.
- Cada rama del árbol contendrá una expresión booleana que puede contener condiciones sobre los atributos del objeto.
- Cada nodo puede tener 0 o más hijos.
- A cada ocurrencia de una regla de tipo *Rechazador* en un nodo del árbol se le podrá asignar una lista de reglas. Estas reglas se aplicarán en el caso de que el resultado de la evaluación sea falso, es decir en el caso de que se rechace. Por lo tanto, si un rechazador tiene una lista de reglas asociadas antes de terminar la evaluación (porque rechazó) se evaluarán dichas reglas e inmediatamente se finalizará la evaluación.

La lista de reglas asociada al rechazador no puede contener reglas de tipo *Rechazador*.

Las condiciones de evaluación que se encuentran en las ramas del árbol, tienen la siguiente forma:

<condición evaluación> ::= true | false | (<condición evaluación>) | <condición evaluación> (“or” | “and”) <condición evaluación> | “not” <condición evaluación> | <atributo> (“<” | “>”| “<=” |”>=” | “=” | “<>”) (Literal | <atributo>) | <atributo> (“=” | “<>”) null

En donde los atributos que participan en la misma sólo pueden ser atributos del objeto sobre el cuál se está construyendo el árbol.

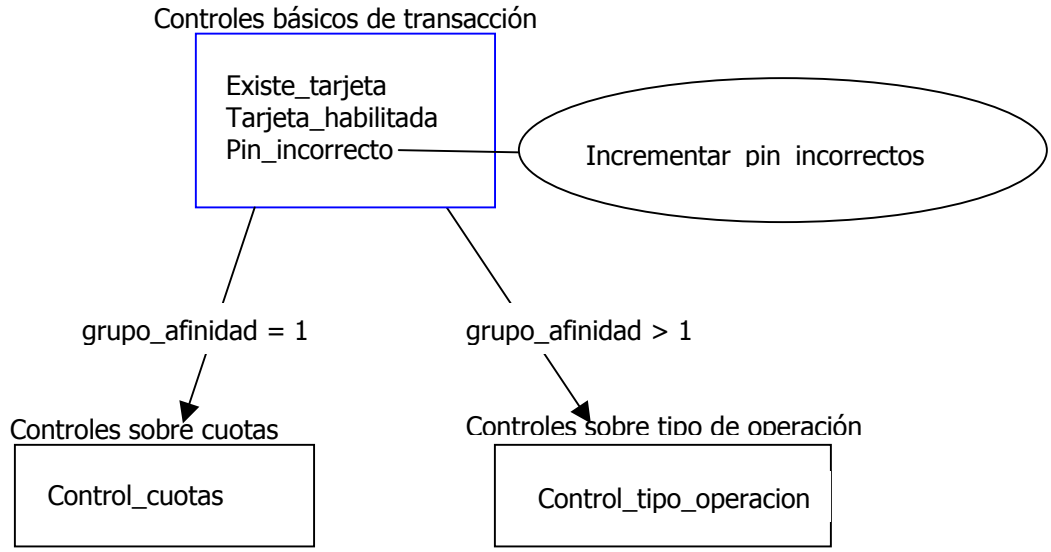
4.5.2 Ejemplo de árbol de evaluación

Veamos un ejemplo sencillo de árbol de evaluación en el contexto del ejemplo utilizado del negocio de medios de pagos:

Las reglas disponibles son: *Existe_tarjeta*, *Tarjeta_habilitada*, *Pin_incorrecto*, *Incrementar_pin_incorrectos*, *Control_cuotas* y *Control_tipo_operación*.

Para que el árbol sea correcto vamos a suponer que todas las reglas cumplen que si tienen un objeto evaluado el mismo es el objeto *Transaccion*.

El siguiente es el árbol de evaluación para el objeto *Transaccion*:



Inicialmente, se aplican tres reglas de tipo rechazador en el primer nodo, la primera rechaza si la tarjeta asociada a la transacción no existe, la segunda comprueba que la tarjeta esté habilitada y rechaza en caso de que no lo esté y la tercera valida el pin de la transacción. A continuación, se evalúan las condiciones de las aristas adyacentes en orden y según el atributo *grupo_afinidad* de la *Transaccion* se determinará cual será el próximo nodo.

Observar que la ocurrencia de la regla *Pin_incorrecto* en el primer nodo, tiene una lista de reglas asociada que se evaluarán si el resultado de evaluar *Pin_incorrecto* es falso, es decir, si el pin de la transacción no coincide con el de la tarjeta asociada. En este caso la lista tiene una sola regla (*Incrementar_pin_incorrectos*), la misma incrementa la cantidad de validaciones fallidas para la tarjeta en un día. Por lo tanto, en el caso de que se rechace la transacción porque el pin es

inválido, se evaluará la regla *Incrementar_pin_incorrectos*, e inmediatamente después se finalizará la evaluación.

4.5.3 Proceso de evaluación

En esta sección describiremos el proceso de evaluación de un árbol de reglas sobre una instancia del objeto asociado al árbol. En el siguiente recuadro presentamos un pseudocódigo del algoritmo de evaluación y a continuación del mismo explicamos dicho proceso.

```
Evaluación (Objeto obj){
    Arbol ar = getArbol(obj.getClass().getName())

    If ar == null return;

    Nodo actual = ar.getRaiz();

    While ( not actual.esvacio() ){

            reglas = actual.getReglas();

            while ( not reglas.esvacio() ){

                    regla       = reglas.next();
                    resultado = regla.run(obj);

                    if  ( resultado.error == true and regla.getTipo() <> rechazador )
                            return;

                    if( resultado.error == true ){
                            regla.runReglasRechazador();
                            return;
                    } // if

            } // while reglas

            actual = getNextNodo(ar,obj);
            /* evalua las condiciones de las ramas del nodo sobre el objeto y decide
            porque rama seguir */
    } // while nodos

    return;

}
```

Figura 6. Algoritmo de evaluación

La función de evaluación recibe como parámetro la instancia del objeto del modelo de conceptos que se va a evaluar y retorna el mismo objeto que recibió como parámetro con las modificaciones que puedan haber sido realizadas al objeto a través de la evaluación de las reglas.

En primer lugar se verifica que el objeto pertenezca al modelo de conceptos especificado, de no ser así retorna un resultado de error y termina la evaluación.

Si el objeto especificado no tiene un árbol de evaluación asociado la función de evaluación retorna sin realizar ninguna acción.

La evaluación comienza por el nodo raíz del árbol asociado al objeto recibido como parámetro, realizando el siguiente proceso hasta retornar por una condición de fin:

- Se evalúan las reglas habilitadas del nodo en el orden en que aparecen listadas. Al evaluar una regla puede pasar una de las siguientes cosas:

    - La regla se evalúa exitosamente, retorna "verdadero" si es un rechazador y si es otro tipo de regla simplemente retorna luego de realizar la acción correspondiente al tipo de regla (no retornan ningún valor). En este caso se continúa evaluando la próxima regla de la lista.
    - La regla es un rechazador y la evaluación retorna "falso", se interrumpe la evaluación. Si la regla es un *Rechazador* y tiene una lista de reglas asociada se evaluarán dichas reglas y luego se finalizara la evaluación.
    - La regla es de cualquier tipo y su evaluación da error porque alguno de los objetos o atributos que aparecen en la regla no existen en la instancia actual, en este caso se detendrá la evaluación.

Para aclarar un poco más este caso veamos un ejemplo, supongamos que tenemos los objetos *Transaccion* y *Tarjeta*, estamos evaluando el objeto *Transaccion* y diseñamos una regla que rechaza la transacción si el pin de la misma no coincide con la tarjeta asociada. Ahora supongamos que estamos evaluando una instancia de *Transaccion* que no tiene un identificador de *Tarjeta* válido, en este caso la regla no tiene sentido porque la tarjeta no existe y por lo tanto la evaluación de la regla retorna error.

- Luego de evaluar todas las reglas del nodo se busca un nodo hijo por el cual seguir, para ello se evalúan las condiciones de las ramas de todos los hijos del nodo, en el orden en que fueron definidas, con la instancia que se está evaluando. Al encontrar la primer condición que evalúa en verdadero (las ramas tienen un orden) se selecciona el nodo correspondiente para continuar la evaluación. Puede pasar que ninguna condición retorne verdadero, en este caso se termina la evaluación. También puede pasar que exista un error al evaluar una condición porque se menciona en la misma un atributo que no tiene ningún valor, en este caso se termina la evaluación.
- Cuando se llega a un nodo que no tiene hijos se termina la evaluación.

La evaluación de cada regla depende de su tipo, y ya fue especificado que es lo que cada tipo de regla hace, lo que es bueno recordar es que las reglas que tengan un objeto evaluado y además realicen modificaciones al mismo las realizarán sobre el mismo objeto que se especificó como parámetro, retornando entonces el objeto modificado (igual las consultas). En cualquier otra situación en la que se actualizan o consultan atributos de un objeto en una regla y el mismo no es el objeto evaluado dichas consultas o actualizaciones se realizan en la base de datos.

Finalmente, es oportuno recalcar que el éxito de la evaluación de las reglas dependerá en gran medida de una adecuada construcción del árbol de evaluación, pueden existir reglas para todas las restricciones pero si las mismas no están bien ubicadas en el árbol, la evaluación no será correcta. Por ejemplo, supongamos que tenemos dos reglas, una fija el valor de un atributo en un valor por defecto y la otra consulta ese valor para realizar otras acciones. Si colocáramos en primer lugar en el árbol la regla que consulta el valor del atributo y luego la que lo fija, la evaluación siempre daría un resultado erróneo ya que el atributo siempre tendría un valor equivocado, y nunca se ejecutarían las acciones de la regla que consulta su valor.

5 Arquitectura

En este capítulo se describe la arquitectura que se utilizó para el desarrollo del sistema. Con ese fin, se mostrarán los distintos componentes, las distintas responsabilidades de cada uno y cómo se relacionan para alcanzar la solución al problema planteado.

5.1 Introducción

La descripción del sistema expresada en un modelo de distribución de nodos físicos (*Deployment*) es la siguiente.

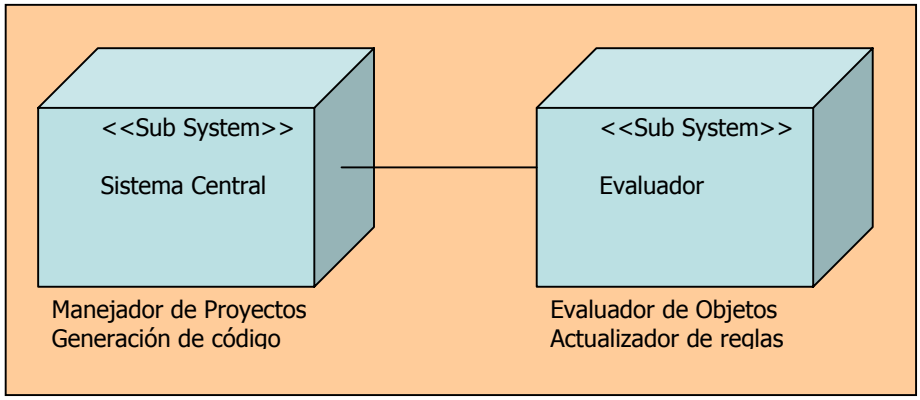


Figura 7. Descripción del sistema

5.1.1 Descripción de cada subsistema:

En el subsistema *Central* se encuentran el *Mapeo*, el *Diseñador* y el *Repositorio* de reglas, conjuntamente con el manejo de usuarios y los generadores de código. Este subsistema también se encarga de publicar las reglas en el evaluador.

En el subsistema *Evaluador* se encuentra el evaluador de reglas de negocio. Éste es el encargado de conectarse con las distintas fuentes de información (del negocio propiamente dicho) y ejecutar las reglas diseñadas por el usuario y generadas por el generador de reglas (componente del sistema central).

5.1.2 El Sistema Central

5.1.2.1 Introducción

Se construye básicamente como aplicación de ventanas. Su funcionamiento está basado en la configuración de las distintas fuentes de datos, mapeo de los datos al modelo de conceptos, edición de reglas basadas en un modelo de conceptos y generación de los distintos códigos ya sea *Java* para las reglas y evaluación en J2EE (no se usa toda la infraestructura J2EE) o *PL/SQL* para la evaluación como procedimientos almacenados en *Oracle*. Por las características de este proyecto y las posibles modificaciones se definió una estructura de Capas, entre las cuales encontramos las siguientes:

- Capa de Presentación
- Capa Lógica
- Capa de Acceso al Repositorio
- Repositorio

A continuación se presenta una breve descripción del objetivo de cada capa y cual es el lugar que ocupa en este sistema.

Capa de Presentación: Soporta la interacción con los usuarios del sistema. Permite la definición de proyectos y creación de usuarios, y manejos de proyectos. Dentro de lo que es el manejo de proyectos nos encontramos con los usuarios registrados y con permisos sobre los proyectos. Estos usuarios podrán realizar mapeos de los datos que maneja el negocio con el modelo de conceptos, esto es, que el usuario debe definir objetos para luego poder definir reglas sobre ellos. También, un usuario podrá definir reglas y generar los distintos códigos (*Java* y *PL/SQL*) para luego, al momento de la evaluación de un objeto poder ejecutar el código generado.

Capa Lógica: Soporta la lógica del sistema. Es quien se encarga de que las reglas, proyectos, modelos de conceptos, funciones auxiliares y demás sean consistentes y tengan una estructura correcta.

Capa de Acceso al Repositorio: Contiene los componentes que permiten a la capa Lógica conocer los distintos datos persistentes en el repositorio.

Repositorio: Almacena los datos del sistema. Esta capa está compuesta por las distintas bases de datos que serán manejadas por el sistema y los distintos archivos que serán respaldados a modo de configuración del sistema. En esta capa se incluye las bases de datos de los negocios con las que trabaja el sistema.

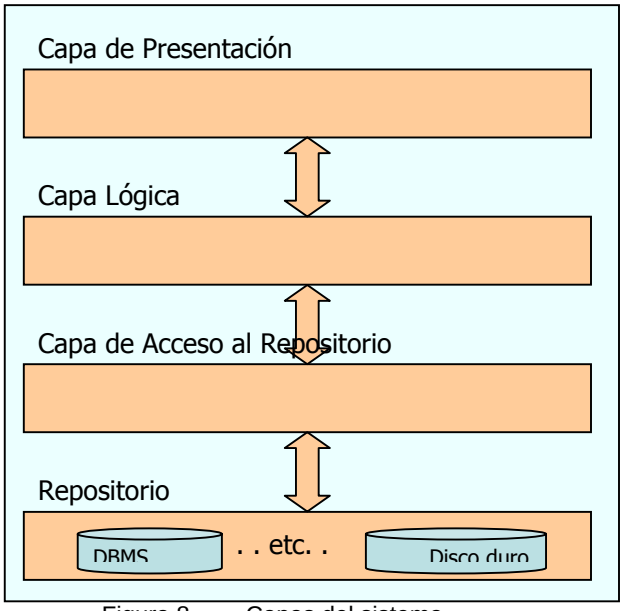


Figura 8. Capas del sistema

5.1.2.2 Primer nivel de la vista lógica.

5.1.2.2.1 Capa de Presentación

Está compuesta por los componentes de interfaz de usuario: UIAdministrador, UILogin, UIMapeo, UISecuenciador y UIDiseñador. Cada uno de estos componentes soporta la interacción del usuario del sistema con los componentes lógicos del mismo.

5.1.2.2.2 Capa Lógica

Está compuesta por los componentes necesarios para soportar la lógica del sistema. En total son ocho componentes: **Sesión, Administrador, Negocio, Mapeo, Diseñador, Compilador, Generador y Secuenciador**

- **Sesión:** Brinda las funcionalidades necesarias para manejar la sesión de un usuario. Su principal cometido es chequear los permisos de los usuarios, autorizando únicamente las operaciones que su perfil de usuario le permite.

- **Administrador:** Soporta las funcionalidades utilizadas para administrar los usuarios del sistema y para crear y/o eliminar negocios. Estas funcionalidades son de uso exclusivo del usuario administrador del sistema.

- **Negocio:** Soporta el estado de un negocio. Es utilizado por todos los usuarios del sistema, y se encarga de mantener la consistencia de los distintos componentes de un negocio.

- **Mapeo:** Soporta las funcionalidades necesarias para crear el modelo de conceptos del negocio.

- **Diseñador:** Brinda las funcionalidades necesarias para poder especificar las reglas de negocio en el *Lenguaje de Especificación de Reglas* conociendo el modelo de conceptos definido para el negocio.

- **Compilador:** Soporta el proceso de compilación mediante el cual se comprueba la correctitud de las reglas especificadas mediante el *Lenguaje de Especificación de Reglas*.

- **Generador:** Está compuesto por tantos componentes como lenguajes soporte el sistema. En nuestro, caso hay dos componentes: uno para generar código *PL/SQL* y otro para generar código *Java*. Como entrada utilizan el código analizado por el *Compilador*, o sea la especificación de la regla en el *Lenguaje de Especificación de Reglas*.

- **Secuenciador:** Ofrece las funcionalidades que permiten establecer un orden para la evaluación de las reglas. Es el asignado a la definición del árbol de evaluación que debe recorrer el evaluador al momento de evaluar un objeto.

5.1.2.2.3 Capa de Acceso al Repositorio

Está compuesta por los componentes utilizados para acceder a los distintos repositorios. Dichos componentes intermedian con la *Capa Lógica* y el *Repositorio*. El cometido principal de este componente es que la *Capa Lógica* se vea independizada del mecanismo utilizado para el acceso al Repositorio (DBMS), y a los distintos archivos de configuración registrados en el sistema. Para resolver este objetivo identificamos los siguientes componentes,

- Acceso a BD del Sistema.
- Acceso a BD del Dominio.
- Manejador de Drivers y acceso a Disco.

5.1.2.2.4 Repositorio

El repositorio consiste en la Base de Datos del Dominio, la Base de Datos del Sistema Generador de Reglas de Negocio, y el repositorio de código *de* reglas *Java*. Cada negocio se construye a partir de la Base de Datos del Dominio (dicha base de datos puede ser diferente para dos negocios distintos.) Esta base de datos almacena los datos pertinentes al dominio del negocio, o sea las tablas que contienen los datos del negocio de la organización y es creada y administrada por fuera del Sistema Generador de Reglas de Negocio. La otra base de datos, Base de Datos del Sistema Generador de Reglas de Negocio, es la que soporta al sistema (todos los negocios existentes), la misma es construida y administrada por el sistema. En la base de datos del sistema se almacena los datos administrativos del sistema como el nombre y datos de cada de negocio, los usuarios de los mismos y sus perfiles. Además, se almacenan los datos operativos del negocio, por ejemplo las reglas, versiones de reglas, modelo de conceptos, código fuente de las reglas, etc. El código de las reglas creadas por el generador *PL/SQL* también es almacenado en la base de datos del sistema, en forma de procedimientos almacenados. Por último el repositorio de código almacena el código de las reglas creadas por el generador *Java* (clases Java).

5.1.2.2.5 Diagrama de componentes (sistema central)

En el diagrama de componentes se representan gráficamente los componentes del sistema y la interacción entre ellos.

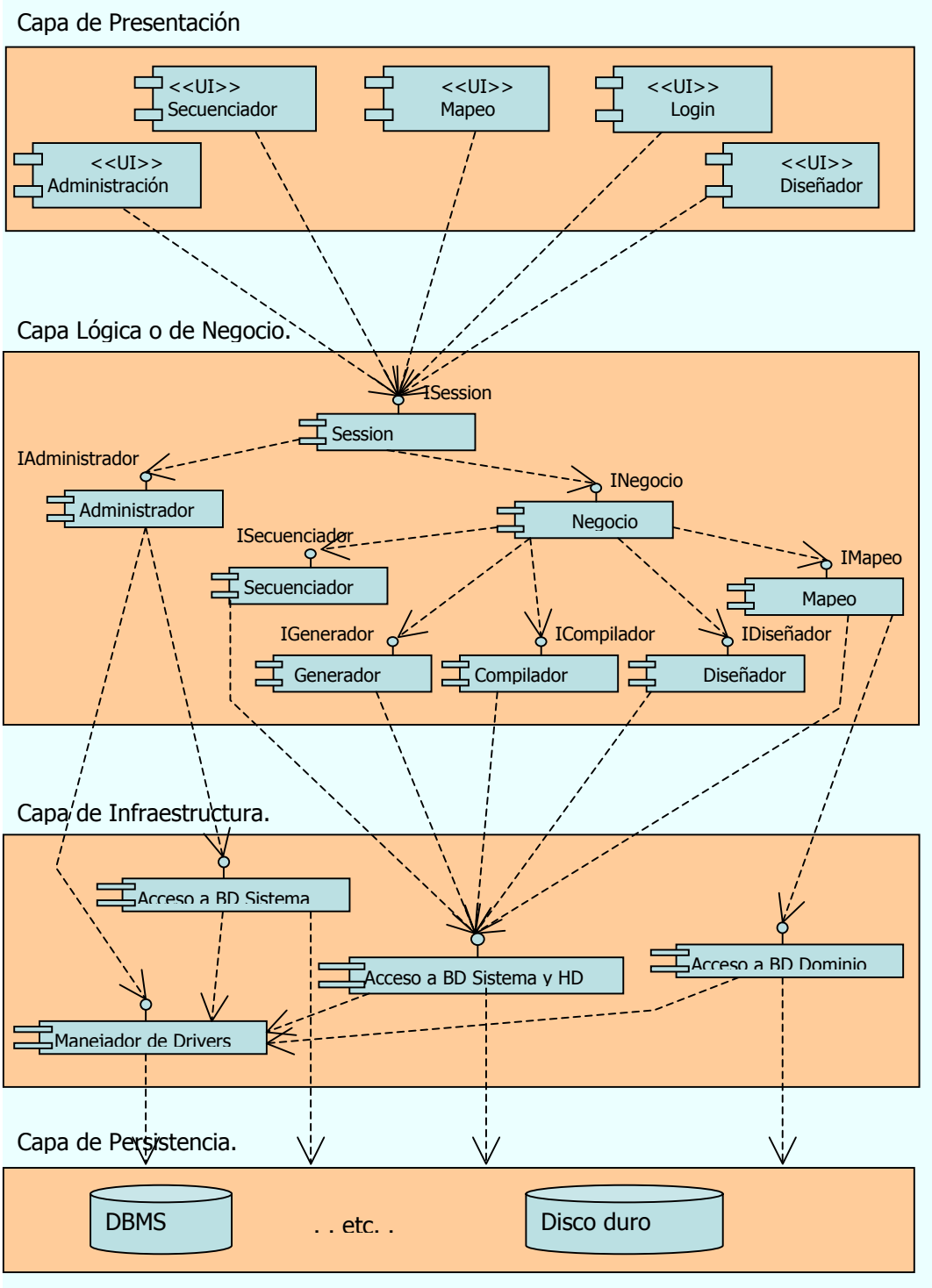


Figura 9. Diagrama de componentes

5.1.3 El Evaluador de Reglas

5.1.3.1 Introducción

Se construye sobre un proyecto web. Con el fin de hacer de forma rápida y sencilla un servicio Web (*Web Service*) este proyecto esta basado en capas siguiendo la recomendación de *Sun* en sus especificaciones de arquitecturas *J2EE*. En este proyecto la estructura de capas no es estricta sino que estamos hablando de capas relajadas, ya que la capa de lógica del sistema accede a sus dos capas inferiores indistintamente. Las capas que se necesitan en el evaluador son las siguientes:

- Presentación del Servicio
- Lógica de Sistema (Procesos y Dominio)
- Persistencia
- Fuentes de información
- Acceso al Sistema Central

A continuación se presenta una breve descripción del objetivo de cada capa y cual es el lugar que ocupa en este sistema.

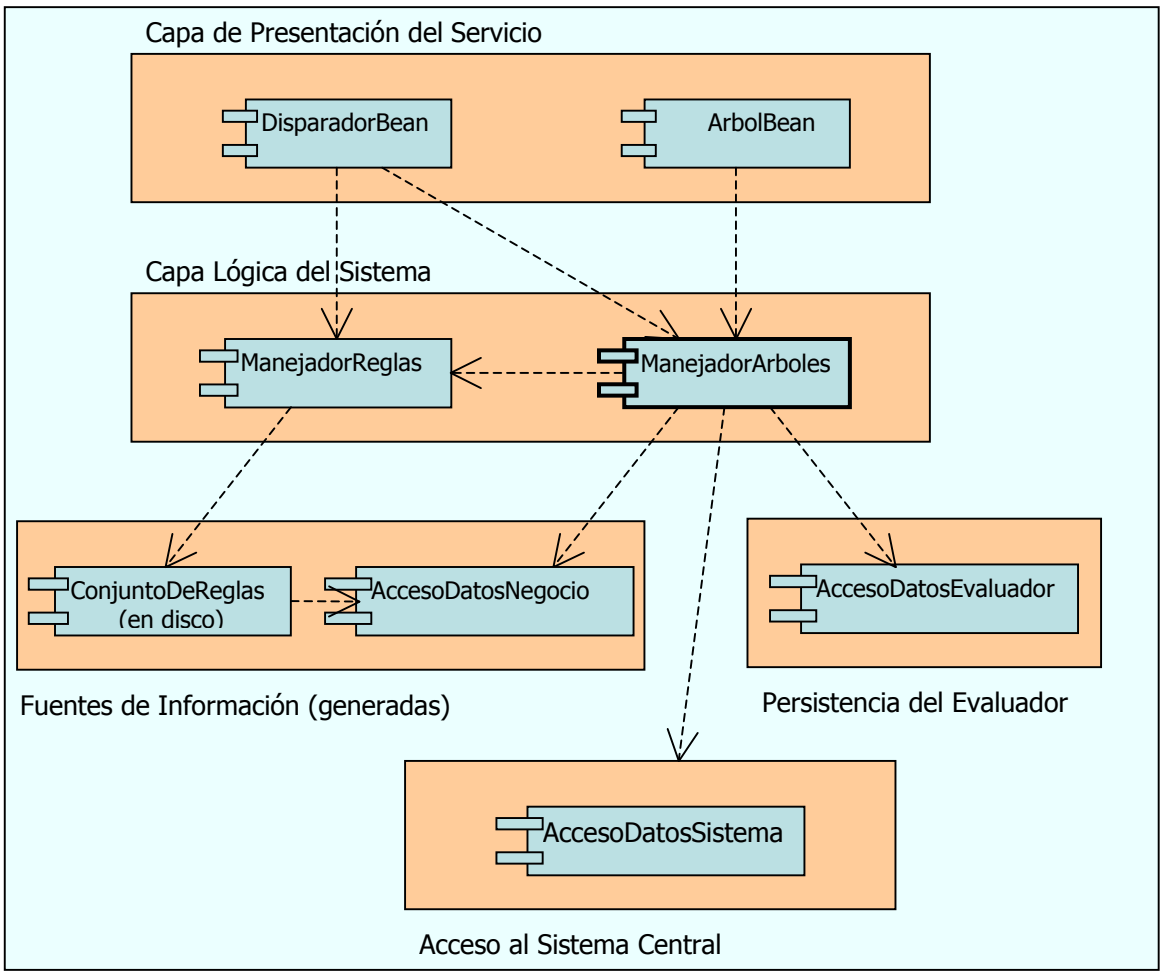
Presentación del Servicio: En esta capa es dónde se describen los distintos servicios del evaluador de reglas. Esto significa, que es la capa encargada de encontrar la mejor forma de presentar el servicio de evaluación de reglas. Esos servicios podrán ser presentados de distintas formas, WebServices, objetos *RMI*, simples páginas donde se describen los valores de un objeto a evaluar u otros.

Lógica de Sistema: Esta capa es la encargada del manejo del negocio, o sea, es la encargada de la evaluación de un objeto. Por lo tanto esta capa debe conocer todas las clases asociadas a cada regla del negocio que está en el sistema, debe permitir el cambio "on the fly" de las distintas clases asociadas a cada regla y hacer un manejo correcto y eficiente del árbol de evaluación de reglas asociado a cada tipo de objeto.

Persistencia (relacionada con lo que sería Lógica de Dominio): Esta capa es la encargada de conocer el dominio del evaluador, no el dominio de los distintos negocios. Está orientada al conocimiento de los datos persistentes y a un buen manejo de los mismos, pero también tiene un conocimiento muy fuerte de lo que significan estos datos y conoce cómo se relacionan. En nuestro caso, de esta parte se encargarán un conjunto reducido de clases que tienen la estructura establecida para ser montado en el framework *Hibernate 2.0* [8]. Este proyecto es manejado de forma similar a las fuentes de informaciones.

Fuentes de Información: Esta capa es la encargada de la comunicación con las distintas fuentes de información. Las fuentes de información son: la base de datos del negocio con el que se esté trabajando y el conjunto de reglas generadas para este negocio. Debe levantar dinámicamente el código ejecutable de cada una de las clases que representan a cada una de las reglas generadas por el generador *Java*.

Acceso al Sistema Central: Esta capa es la encargada de la comunicación con la base de datos del sistema central. Su objetivo es poder conocer los procedimientos *PL/SQL* almacenados en la base de datos del sistema, los mismos representan el código de las reglas generadas por el generador *PL/SQL*. Debe conocer estos procedimientos para habilitar su ejecución en la evaluación.



5.1.3.2 Primer nivel de la vista lógica.

5.1.3.2.1 Capa de presentación del servicio:

Para la implementación de esta capa se usaron *Session Beans*. Estos componentes son los que presentan el conjunto de operaciones que utiliza el evaluador. Dichas operaciones están separadas en dos grupos:

- El grupo de la evaluación propiamente dicha, que implementa las siguientes funcionalidades:

  - Evaluar un objeto por medio de un método de acceso remoto que recibe como argumento el objeto a evaluar y retorna el resultado de la evaluación.
  - Cambiar el código ejecutable de una regla que está en producción, modificación "on the fly". Esto significa que la regla puede estar usándose en ese momento, pero para las próximas evaluaciones debe ejecutarse el código nuevo de la regla. Esta operación debe recibir el código (identificador de la regla original) de la regla a suplantar, código binario actualizado de la regla (contenido del archivo .class) el nombre de la clase que se está publicando para poder ser levantada dinámicamente y utilizarla en la evaluación

de los siguientes objetos que se deseen evaluar y deban ejecutar esa regla. Esto solo es utilizado en el caso de estar realizando la evaluación de reglas generadas en Java.

- El otro grupo de operaciones encapsuladas en el *Session Bean*, *ArbolBean*, son las operaciones que permiten definirle al evaluador cual es la estructura de reglas en la evaluación de los distintos objetos. Esto significa establecer la secuencia de reglas que se deben ejecutar al evaluar un objeto, para esto se publican los siguientes métodos:

  - Agregar y quitar un árbol de evaluación.
  - Agregar y quitar un negocio.
  - Agregar y quitar un nodo.
  - Agregar y quitar una regla.
  - Establecer o borrar una relación entre un nodo y una regla.
  - Publicar negocio (este recibe un archivo de formato XML con toda la especificación de evaluación)

De esta forma se puede definir en el evaluador el modo de evaluación de los distintos objetos.

5.1.3.2.2 Capa Lógica del Sistema

En esta capa se destacan dos componentes que realizan todas las tareas que enmarca el evaluador de reglas. Estos componentes tienen objetivos distintos pero necesitan colaboraciones para completar sus cometidos.

- Manejador de árboles (evaluador de objetos).

  - El evaluador de objetos en infraestructura, está basado en el uso de objetos *hibernate*. Estos son los que describen el árbol de ejecución para la evaluación de un objeto. Cada vez que se intenta evaluar un objeto se consulta a las estructuras levantadas en memoria si ya tienen cargado al árbol de evaluación para el tipo de objeto que se está por evaluar. En caso de no estarlo, se levanta en ese momento el árbol de evaluación para evaluar dicho objeto. A partir de este momento, dicho árbol queda en memoria para futuras ejecuciones.

    En la estructura cargada en memoria por este componente sólo se conocen los nombres de las clases que implementan el código de las distintas reglas que se deben ejecutar. Para la obtención del código binario de la regla se necesita del segundo componente que es el encargado de esto. Esto es válido para el caso de la evaluación de las reglas generadas en *Java*. Para la evaluación de las reglas en PLSQL solamente hay que invocar los procedimientos almacenados (código de las reglas) en la base de datos del sistema.

    Por tanto este componente es quien tiene que decidir cual es la ruta a seguir según los resultados obtenidos al evaluar cada una de las reglas, esto se hace respetando el árbol descrito en el *Secuenciador* (Componente del Sistema Central).

- Manejador de reglas.

  - Este componente como ya mencionamos es el encargado del manejo de los códigos binarios de las reglas. Tiene dos cometidos principales: el de levantar las reglas dinámicamente para poder permitir la ejecución de las mismas, y el de brindar la posibilidad de cambiar su código en tiempos de ejecución. Esto hace que el sistema no

tenga que bajarse cortando el servicio al momento de querer modificar una regla. ¿Cómo se resuelven estos dos puntos? Para el primer punto, levantar las reglas dinámicamente, se dispone de un *classloader* que hace referencia al repositorio de reglas y a medida que son necesitadas las clases son levantadas y almacenadas en una tabla de hash por cuestiones de performace. Para resolver el segundo punto, lo que el manejador de reglas hace es escribir la regla en el repositorio y luego decide dependiendo de si la regla cambió o no el nombre de la clase que la implementa. En caso de no cambiar el nombre, sólo borra la regla de la tabla de hash y luego cuando se necesite al no encontrarse en memoria es cargado el nuevo código, pero si cambia el nombre de la clase que la implementa, debe comunicarle al otro componente (manejador de árboles) que debe levantar nuevamente los árboles que involucren a la regla modificada.
Notar que esto solo se utiliza en la evaluación de reglas generadas en Java.

5.1.3.2.3 Persistencia:

La capa de persistencia es un componente de Infraestructura. Este componente es el encargado de la persistencia de los datos propios del evaluador. Se creó una base de datos con la estructura necesaria para almacenar toda la información referente a los árboles de evaluación de cada uno de los objetos que pueden ser evaluados.

Vista de Datos para la persistencia de este "modulo":

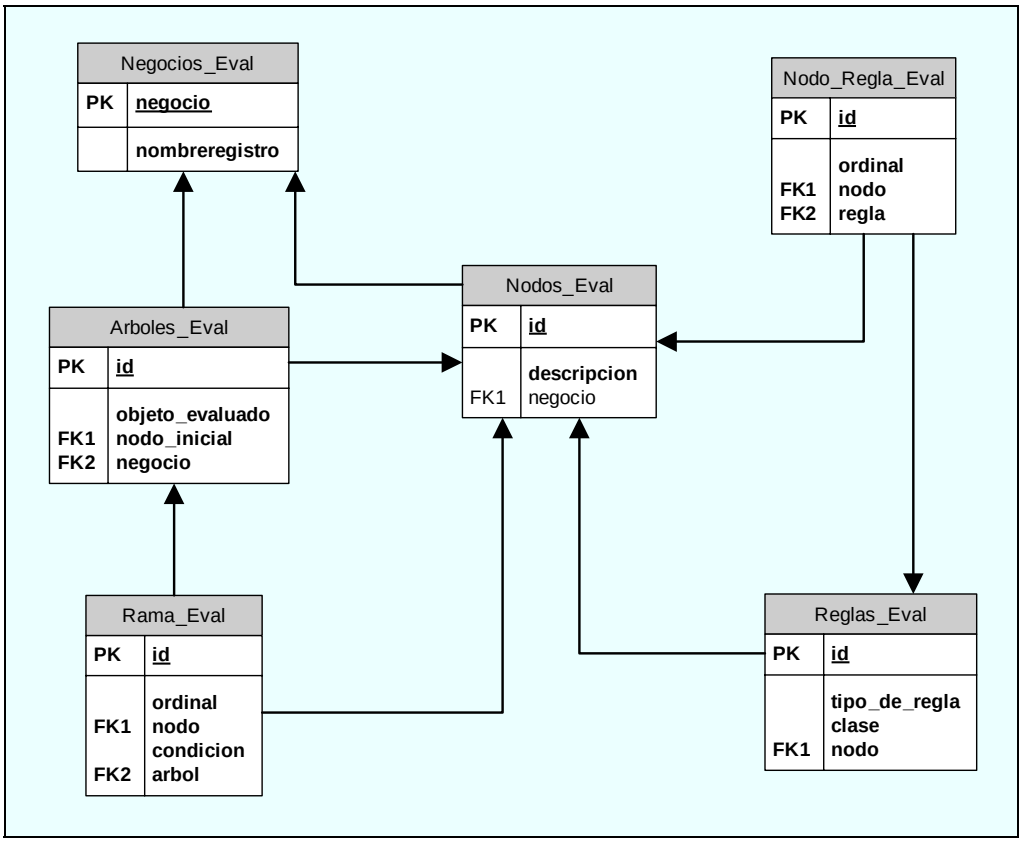


Figura 10. Persistencia del Evaluador

5.1.3.2.4 Fuentes de Información:

Este componente es el segundo de los que integran la infraestructura y resuelve la conexión con la base de datos del negocio. Esto sólo es necesario para la evaluación de las reglas generadas como clases *Java*.

Para resolver esta comunicación nos basamos en *Hibernate 2.0*. El modelo *hibernate* consiste en el mapeo de clases *Java* a tablas de una base de datos. En nuestro caso, mapeamos los objetos que representan el dominio del negocio creados al realizar el mapeo, con las tablas correspondientes en la base de datos de dicho negocio.

El proyecto *hibernate* creado (el que tiene los objetos del negocio) es publicado en el servidor de aplicaciones y accedido desde este módulo (Evaluador).

5.1.3.2.5 Acceso al Sistema Central:

Este componente es el tercero y último, en él se realiza el acceso a la base de datos del sistema central. La necesidad de que el componente evaluador conozca la base de datos del sistema central, está dada porque es en esta base de datos donde se almacenan como procedimientos almacenados el código *PL/SQL* de las distintas reglas de negocio existentes.

Por lo tanto, para realizar la evaluación en modo *PL/SQL* se invocan los procedimientos almacenados conociendo los nombres de los mismos y la ubicación de la base de datos que los contiene.

6 Decisiones de Implementación

En lo que refiere a decisiones de implementación, este proyecto se basó mucho en el uso de tecnologías de punta, siempre teniendo en cuenta cuales son nuestros objetivos y cuales son las necesidades del cliente. El proyecto tuvo instancias en donde se comenzó con el uso de ciertas tecnologías y con el transcurso del tiempo fueron apareciendo herramientas que se comportaban mejor y se adecuaban más a nuestras necesidades lo que nos introdujo un costo de tiempo de aprendizaje y estudio de si realmente nos servían o no.

6.1 Distintos puntos a destacar en el proyecto:

1 Discusión sobre el lenguaje a manejar para la especificación de reglas de negocio.
2 Manejadores de bases de datos, necesidades del cliente, extensibilidad del producto.
3 Resolución del cambio de reglas con el sistema en ejecución (cambio de reglas en caliente).
4 El uso de *Entity Beans*, planteado como requerimiento del cliente, discusión del uso de *Entity CMP* (Container Manager Persisten) y *BMP* (Bean Manager Persisten).
5 Problemas de las relaciones compuestas.
6 Estudio de *Hibernate*, comparaciones entre entity y objetos hibernate, temas relacionados con la performace y la generación de código.

6.2 Detalle de los puntos antes nombrados:

1 Como cualquier lenguaje de programación, el Lenguaje de Especificación de reglas de negocio requiere de las tres etapas clásicas para el análisis de la correctitud del código fuente, o sea, análisis lexicográfico, análisis sintáctico y análisis semántico. Para resolver estos puntos se decidió utilizar las herramientas *JLex* [22] y *Cup* [23], los motivos de esta decisión fueron:
Este par de herramientas era adecuado para el proyecto, ya que generan código *Java*, lenguaje con el que se implementó el prototipo. Además, estas dos herramientas trabajan juntas, integrando fácilmente las tareas de análisis lexicográfico y análisis sintáctico.

2 Como infraestructura nosotros nos basamos potencialmente en dos manejadores de bases de datos, *MySQL* y *Oracle 9i*. La solución que nosotros planteamos a este problema contempla lo solicitado por el cliente (Manejador de Bases de datos Oracle). Pero también sabemos que es posible usar otros manejadores como *Informix* o *ProstgreSql* u otros, ya que hibernate asegura que soporta la mayoría de los DBMS más conocidos[9].

Algunos de los soportados por este framework son los siguientes:

DB2 7.1, 7.2, 8.1
MySQL 3.23, 4.0, 4.1
PostgreSQL 7.1.2, 7.2, 7.3, 7.4
Oracle 8i, 9i, 10g
Sybase 12.5 (JConnect 5.5)
HypersonicSQL 1.61, 1.7.0, 1.7.2, 1.8
Microsoft SQL Server 2000

SAP DB 7.3

3 En lo que refiere al problema de levantar las clases (clases que se corresponden con reglas) en tiempo de ejecución del evaluador, para la solución del mismo se discutieron varias opciones.

La primera opción considerada para la carga de las clases, fue que se hiciese sola, es decir, el conjunto de reglas se encuentra entre las librerías al momento de levantar el servidor de aplicaciones. Pero esto tiene el problema de que las reglas pueden cambiar en cualquier momento y el sistema no puede detenerse para volver a cargar las nuevas reglas. Concluimos, por lo tanto que esta solución no era la más adecuada.

Por lo que se piensa en una segunda opción, esta consiste en levantar las clases dinámicamente en un cargador de clases de *Java* ("*URLClassloader*") que levantara las clases de una dirección URL especifica. Si bien, si se agregan reglas nuevas en tiempo de ejecución estas son levantadas, nos encontramos con el problema de que el cargador de clases tiene las clases levantadas en memoria por lo que no refresca las clases que fueron cambiadas en el disco. Por lo que esta tampoco es una solución adecuada.

Es por esto que buscamos una tercera solución, y esta es la que realmente cumple con lo solicitado aunque pensamos que tiene algunos problemas de performance mínimos. Lo que se resolvió es que el propio evaluador tenga una tabla con todas las reglas levantadas y aquellas que cambien son quitadas de la tabla. Cuando se necesite una clase y dicha clase no este en la tabla, se crea un cargador nuevo (con el fin de que se refresque lo que tengamos en el disco) y se carga la clase que se necesite, y luego de que la clase esté cargada se agrega a la tabla que contiene todas las clases cargadas. Esto hace que el evaluador sea más eficiente, y permita el cambio en tiempo de ejecución de las reglas.

4 En el proyecto se planteo como requerimiento o como objetivo comparar los tiempos de evaluaciones entre código ejecutado en *EJB* usando *Session Beans* y *Entity Beans* contra código en procedimientos almacenados en *Oracle PL/SQL*.
En el curso de nuestra investigación sobre *Entity Beans* nos encontramos con dos opciones los *CMP* y los *BMP*.

Los BMPs requieren la codificación manual de las operaciones de acceso a la base de datos, mientras que en el caso de los CMPs esto es realizado de forma automática. Por lo tanto, en *CMP* los programadores no deben escribir el código de las consultas sql.

Según los creadores la opción *CMP* es viable en modelos que no cambian. Es decir, si el modelo es cambiante lo mas adecuado es que el programador maneje el conjunto de consultas.

5 A mitad del proyecto nos encontramos con la existencia de relaciones compuestas entre distintas tablas de la base de datos del cliente. Con relaciones compuestas nos referimos a que en la base de datos existen tablas que están relacionadas y su relación es por más de un atributo. Podríamos resolver estoy haciendo que cada tabla tenga un campo que fuese identificador único, pero esto era inviable ya que requería modificar la base de datos del negocio. Por tanto, la solución es hacer lo que en *J2EE* se llama relaciones compuestas (*composite relationships*), pero esto no es un mundo muy explorado y ejemplificado en *J2EE* por lo que se nos torno bastante complicado.

6 En la investigación de cómo usar las relaciones compuestas para poder resolver este problema, nos encontramos con un framework para persistencia llamado *Hibernate*,

comenzamos a investigar y prometía una solución muy adecuada a lo que estábamos buscando.

Al investigar sobre *Hibernate* nos encontramos que el framework maneja mucho mejor el tema de objetos referentes a registros (idea de los entity) con respecto a como lo maneja *J2EE*. Nótese que nosotros comenzamos a investigar cuando estaba en producción la versión 2.1.8 de *Hibernate*. Actualmente con la versión 3.0 que ya esta lista para producción mas a nuestro favor el cambió estratégico que se busco para resolver las carencias de performance de *EJB*. Ya que esta última versión tiene una gran cantidad de mejoras (véanse [16], [17] y [18]).

7 Resultados experimentales

7.1 Introducción

Se realizó una prueba con el fin de medir la performance del código generado. La intención fue medir el tiempo que dura en evaluarse un objeto utilizando su *árbol de evaluación* y aplicando el algoritmo presentado en el capítulo 4. Se intenta medir la performance desde el punto de vista de la escalabilidad de los evaluadores en función de la cantidad de objetos evaluados en un determinado intervalo de tiempo.

Quiere compararse el desempeño de ambos evaluadores para un conjunto dado de objetos del mismo tipo a evaluarse en el mismo *árbol de evaluación*. Esto implica la creación de un negocio a partir de una base de datos que tendrá la metadata y los datos del negocio. Es necesario crear el *modelo de conceptos* correspondiente, diseñar y generar las reglas de negocio necesarias y construir el *árbol de evaluación*.

La evaluación se hace sobre una serie de conjuntos de objetos: 50 objetos, 100 objetos, 200 objetos, etc. Cada conjunto los objetos debe comenzar su evaluación dentro de un intervalo de tiempo determinado. La llegada de cada objeto al punto de entrada de la evaluación (nodo raíz del árbol de evaluación) será aleatoria dentro del intervalo de tiempo mencionado, pudiendo darse el caso que dos o más evaluaciones se ejecuten de manera concurrente.

7.2 Parámetros y medidas de performance

Antes de presentar los resultados es necesario definir las medidas y parámetros que se utilizarán.

7.2.1 Parámetros

7.2.1.1 Tolerancia para una evaluación

Dado un objeto O y su árbol de evaluación θ, la ***tolerancia λ*** para la evaluación E(O,θ) se define como el límite superior de tiempo (en milisegundos) tolerado para que E(O,θ) se lleve a cabo.

7.2.1.2 Intervalo de evaluación para un conjunto de objetos

Dado un conjunto C de objetos del mismo tipo y su árbol de evaluación θ, el ***intervalo de evaluación μ*** para el conjunto C se define como el intervalo de tiempo (en segundos) durante el cual debe comenzar la evaluación E(O,θ) para todo objeto O perteneciente al conjunto C.

7.2.2 Medidas

7.2.2.1 Tiempo de evaluación de un objeto

Dado un objeto O y su árbol de evaluación θ, el ***tiempo de evaluación T*** de E(O,θ) se define como el intervalo de tiempo (en milisegundos) necesario para que E(O,θ) se lleve a cabo.

7.2.2.2 Número de total evaluaciones

Dado un conjunto C de objetos del mismo tipo y su árbol de evaluación θ, el ***número total de evaluaciones Ω*** se define como la cardinalidad de C.

7.3 Resultados

7.3.1 Condiciones bajo las cuales se realizó la prueba

Antes de mostrar los resultados de la prueba es necesario mencionar las condiciones bajo las cuales se realizó la misma.

La infraestructura con la cual se realizó los experimentos consta de las siguientes características:

- PC Pentium 4 con procesador de 2800 MGhz y 512 MB de memoria RAM
- Sistema operativo Windows xp
- Motor de base de datos Oracle 9i
- Servidor de aplicaciones JBoss 4.0.0
- Hibernate 2.1.8
- JVM 1.4.2_06

Los parámetros de los experimentos son los siguientes:

λ = 1000 milisegundo.
μ = 60 segundos.

7.3.2 Resultados

Las siguientes tablas presentan los resultados de realizar la prueba con el evaluador Java y el evaluador PL/SQL.

Evaluador Java

Ω	Min T(E(o, θ)) tq. o ∈ C	Max T(E(o, θ)) tq. o ∈ C	Prom T(E(o, θ)) tq. o ∈ C
50	0	171	72
100	0	390	77
150	0	703	90
200	0	454	79
250	0	578	97
300	0	859	127
350	0	1422	119
400	0	531	112
450	0	1719	150
500	0	1188	172

Evaluador PL/SQL

Ω	Min T(E(o, θ)) tq. o ∈ C	Max T(E(o, θ)) tq. o ∈ C	Prom T(E(o, θ)) tq. o ∈ C
50	0	234	71
100	0	1203	103
150	0	594	65
200	0	1156	75
250	0	816	60
300	0	906	64
350	0	1165	54
400	0	860	58
450	0	969	51
500	0	1609	75

8 Conclusiones y Trabajo Futuro

Las conclusiones las podemos dividir en dos categorías, una relacionada con las reglas de negocio y otra relacionada con el proyecto.

Con respecto a las reglas de negocio, consideramos que es un campo muy reciente y que por lo tanto falta mucho por hacer. Los distintos productos existentes en el mercado utilizan distintos modelos para trabajar con reglas de negocio y lo que es más, existen diferentes puntos de vista en la parte teórica, sobre todo en lo que respecta a la categorización de las reglas. Por lo tanto esperamos que en un futuro próximo se llegue a un acuerdo en cuanto a la teoría de las reglas de negocio y se estandaricen definiciones, categorías, lenguajes de especificación y modelos para trabajar con ellas.

Con respecto al proyecto hemos alcanzado los objetivos propuestos, teniendo como producto un prototipo que puede ser útil a quienes estén interesados en las reglas de negocio, útil sobre todo en el enfoque utilizado, creando nuestro propio modelo para trabajar con reglas de negocio. Quien esté interesado puede adoptar otros enfoques, pero disponiendo de antemano de los resultados de trabajar con el enfoque utilizado por nuestro proyecto.

Se ha implementado una herramienta "flexible" en cuanto a que está habilitada para utilizarse sobre varios tipos de DBMS por la utilización de *JDBC* e *Hibernate* y es multiplataforma por estar desarrollada en *Java*.

Los tipos de reglas que soporta la aplicación incluyen los tipos de reglas más utilizados que son restricciones sobre los datos, cálculos y disparadores de eventos.

La generación de código realizada provee eficiencia para los negocios que utilizan bases de datos *Oracle*, ya que las reglas se almacenan como procedimientos almacenados en el caso del generador en *PL/SQL*. Por otra parte, la generación de código *Java* permite, como ya se mencionó, la utilización del generador de reglas sobre una amplia gama de DBMS.

El evaluador fue implementado de tal forma de habilitar al usuario a realizar un ordenamiento para la ejecución de las reglas muy flexible, ya que puede ser fácilmente modificado.

Consideramos que la realización de cambios, ya sea en el código de las reglas o en el orden de evaluación de las mismas sea sencillo es muy importante, porque uno de los objetivos de "separar" las reglas del código de las aplicaciones es que la modificación de las reglas produzca un impacto mínimo.

En cuanto a las comparaciones de los tiempos de evaluación utilizando los dos generadores implementados (*Java* y *PL/SQL*), observamos experimentalmente que en promedio los tiempos de evaluación en *PL/SQL* son menores. Este resultado es bastante razonable considerando que en el caso del generador *PL/SQL* las reglas se implementaron como procedimientos almacenados lo que se caracteriza por su eficiencia.

Hay muchas cosas que pueden mejorarse y/o extenderse. Por ejemplo las partes de mapeo (modelo relacional – modelo de conceptos) y de lenguaje de especificación de reglas de negocio pueden ser mejoradas y/o extendidas. Mejoradas, si se adopta un enfoque distinto que pueda simplificar tanto el mapeo como el lenguaje. Extendidas, si se extienden los tipos de mapeo y

reglas soportados, ya que podrían considerarse otros tipos de reglas y formas de representación de las mismas.

De cualquier manera esperamos que el trabajo realizado sea útil para otros interesados en las reglas de negocios.

Glosario

- **BNF**

BNF es una gramática de libre contexto desarrollada por Backus y Naur en 1962 para describir la estructura sintáctica del ALGOL.

- **Cup**

Cup es una herramienta para la construcción de analizadores sintácticos que genera parsers escritos en *Java*.

- **Deployment**

Distribución física que soporta un sistema, esto es la distribución de los componentes de un sistema y como se relacionan los mismos.

- **EBNF**

Un EBNF es un meta-lenguaje que modifica o agrega varios símbolos meta-lingüísticos a los que provee BNF, a fin de hacer el meta-lenguaje más expresivo.

- **Entity Beans**

Los entity beans son componentes EJB que modelan la información del negocio.

- **EJB (Enterprise Java Beans)**

EJB es un modelo de componentes para representar la lógica de negocio y correrla dentro de un servidor de aplicaciones.

- **Hibernate**

Hibernate es un motor de persistencia que realiza automáticamente el mapeo de objetos a bases de datos relacionales a la vez que sirve como servicio de consultas SQL. Con hibernate es posible crear objetos *Java*, con asociaciones, herencia, polimorfismo, composición, etc. y que éstos se correspondan con tablas de una base de datos.

- **J2EE**

J2EE es un grupo de *especificaciones* diseñadas por Sun que permiten la creación de aplicaciones empresariales, esto sería: acceso a base de datos (JDBC), utilización de directorios distribuidos (JNDI), acceso a métodos remotos (RMI/CORBA), funciones de correo electrónico (*JavaMail*), aplicaciones Web(*JSP y Servlets*), etc.

- **JDBC**

Java Database Connectivity (JDBC) es una API para el acceso a base de datos relacionales con soporte de pool de conexiones y transacciones distribuidas relacionales.

- **Jlex**

JLex es un generador de analizadores léxicos programado en *Java*.

- **PL/SQL**

PL/SQL es un lenguaje de programación estructurado que se utiliza dentro del administrador de base de datos Oracle.

- **Session Beans**

Los session beans son componentes EJB que modelan procesos de negocios, y pueden hacer cualquier cálculo necesario o acceder a otros sistemas.

Referencias bibliográficas y sitios de interés

[1] "Principles of the Business Rule Approach", Ronald G. Ross, Addison-Wesley Information Technology Series, ISBN 0-201-78893-4

[2] Business Rule Beans, WebSphere Application Server Enterprise Services, IBM Corp. http://www.ibm.com/
Visitada por última vez: 14/04/2005

[3] Versata Logic Studio, Versata Inc., http://www.versata.com/
Visitada por última vez: 14/04/2005

[4] Drools, http://drools.org/
Visitada por última vez: 14/04/2005

[5] Business Rule Solutions, http://www.brsolutions.com/
Visitada por última vez: 14/04/2005

[6] Business Rule Group, http://www.BusinessRuleGroup.org
Visitada por última vez: 14/04/2005

[7] Business Rule Community, http://www.brcommunity.com/index.php
Visitada por última vez: 14/04/2005

[8] Hibernate 2.1.8 -> http://www.hibernate.org/
Visitada por última vez: 14/04/2005

[9] http://www.hibernate.org/80.html
Visitada por última vez: 14/04/2005

[10] Ant 1.6.2 -> http://ant.apache.org/
Visitada por última vez: 14/04/2005

[11] XDoclet 1.1.2 -> http://xdoclet.sourceforge.net/xdoclet/index.html
Visitada por última vez: 14/04/2005

[12] Jboss AS 4.0.1 SP1 -> http://www.jboss.org/downloads/index#as
Visitada por última vez: 14/04/2005

[13] MySQL 4.1.10a -> http://dev.mysql.com/downloads/mysql/4.1.html
Visitada por última vez: 14/04/2005

[14] Oracle 9i -> http://www.oracle.com/index.html
Visitada por última vez: 14/04/2005

[15] iLog -> http://www.ilog.com/
Visitada por última vez: 14/04/2005

[16] http://www.theserverside.com/discussions/thread.tss?thread_id=27988
Visitada por última vez: 14/04/2005

[17] http://www.theserverside.com/discussions/thread.tss?thread_id=27037
Visitada por última vez: 14/04/2005

[18] http://www.theserverside.com/discussions/thread.tss?thread_id=31343
Visitada por última vez: 14/04/2005

[19] http://blog.hibernate.org/cgi-bin/blosxom.cgi/2005/03/ [evolución de hibernate]
Visitada por última vez: 14/04/2005

[20] http://hibernate.sourceforge.net/HowHibernate3MakesComplexThingsEasy.pdf
Visitada por última vez: 14/04/2005

[21] http://www.javaworld.com/javaworld/jw-08-2004/jw-0809-ejb.html
Visitada por última vez: 14/04/2005

[22] http://www.cs.princeton.edu/~appel/modern/java/JLex/current/manual.html
Visitada por última vez: 14/04/2005

[23] http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html
Visitada por última vez: 14/04/2005

Apéndice (Herramientas utilizadas)

Las herramientas utilizadas para el desarrollo y ejecución de la aplicación son las siguientes:

- Eclipse versión 3.0.0
- Oracle 9i
- Jboss 4.0.0
- Ant 2.1.8
- MySql 4.1.7