Sistemas Informáticos
Curso 2004-2005

Statistics4j
Framework para el Desarrollo de Simulaciones Estadísticas

Ruth María Charro Henche
Pablo Lavín Mera
Arturo Mengotti Arribas

Dirigido por:
Prof. Luis Javier García Villalba
Dpto. Sistemas Informáticos y Programación

Facultad de Informática
Universidad Complutense de Madrid

Índice

1. OBJETIVO DEL PROYECTO

El objetivo de Staticstics4j es del desarrollo de un framework en java, para la realización de trabajos estadísticos en el ámbito de la biología.

El pensamiento de nuestra aplicación es el de la creación de un motor de trabajo y un entorno lo más general, rápido y reconfigurable posible, donde realizar cálculos estadísticos sobre cualquier tipo de muestra que haya sido anteriormente volcada en nuestra base de datos. Por lo tanto podríamos decir que nuestra filosofía es la de busca los datos "donde quieras", ejecuta sobre ellos las operaciones "que quieras" y por fin, muestra el resultado "del modo que quieras".

Una de las características más importantes debe ser la rapidez de la aplicación, pensada sobre todo para que corra en multiprocesadores, intentando para ello separar las tareas en hilos de ejecución paralelos siempre que sea posible.

Incluiremos en la distribución base una configuración por defecto que cargue los datos de un fichero de texto, los almacene del modo más eficaz en base de datos y tras la ejecución de ciertas tareas sobre ellos, muestre los resultados en forma de gráficos o valores, en un archivo PDF.

Dado el entorno de carácter académico en el que se desarrolla este proyecto, hemos pretendido que este proyecto sirva como excusa para el aprendizaje de diversas tecnologías emergentes usadas en el mundo de la empresa. SAX, JDOM, VELOCITY, HIBERNATE…

1.1 English Version

The target for Statistics4j is the development of a framework in java for the execution of statistic work in the environment of the biology.

Our thought in our application is the creation of a work engine and an environment so general, fast and reconfigurable as possible, where you can carry out some statistic calculations over any kind of sample previously inserted in our database. Thus, we could say that our philosophy is looking for the data "wherever you want", execute over them the operations "you want" and at the end, show the results "in the way you want".

One of the more important characteristics must be the speed of the application, developed thinking in being running in multi-processors, trying in that order to split the tasks in different parallel execution threads.

We'll included in our basic distribution, a default configuration which we'll get the data from a text file, save them in the most efficient way in the database and after the execution of certain task over them, show the result obtained in the way of charts or single values, in a PDF file.

Because of the academic environment in which this project is developed, we have tried that this project were useful for us as a reason for the learning of several rising technologies used in the world of the companies. SAX, JDOM, VELOCITY, HIBERNATE…

2. MANUAL DE USUARIO

2.1. Instalación de la aplicación

Como requisito antes de correr el programa, no necesariamente antes de la instalación de la aplicación, sólo debemos disponer de una máquina virtual de java instalada y una base de datos MySql en el sistema. El resto de librerías de terceros que el sistema utiliza vienen incluidas en la distribución.

La aplicación sólo debe desempaquetarse en el lugar que se requiera y cambiar el archivo de configuración **XMLConfSystem.xml** situado en statistics4j\src\es\ucm\fdi\statistics4j\conf\configuration, para que apunte a la base de datos instalada en el sistema e incluya su usuario y password.

2.2. Método de uso de la aplicación

En nuestra aplicación podríamos distinguir dos tipos generales de usuarios y por lo tanto el método de uso de la aplicación por parte de cada uno sería diferente:

- **Programador**: al que se le pide una aplicación que leyendo de un tipo de fuente concreta (por ejemplo, lea de archivos) y tras la ejecución de algunas acciones sobre esos datos se muestre de una forma concreta al terminar.

  A este usuario se le suponen conocimientos técnicos suficientes para tocar el código propio del framework en los lugares que ahora indicaremos para cambiar los módulos del programa, como son el módulo de persistencia (para cambiar la base de datos), el módulo de lectura (cambiando el tipo de fuente en el que se lee) o el de presentación (si por ejemplo quiere mandar por mail los resultados automáticamente). También conocimientos para cambiar los xml's de configuración del sistema donde se guarda la información sobre la base de datos, password, … etc.

- **Usuario normal**: podía imaginarse como un investigador con conocimientos medios o nulos sobre programación java.

  En el caso de conocimientos de programación podrá crearse sus propios .java que contienen las tareas a ejecutar por el programa con los cálculos para su investigación. En caso contrario contará con una pila de archivos .java suministrados por el primer tipo de usuario (programador) que sólo tendrá que seleccionar mediante la interfaz para que se usen como tareas del sistema y tras lanzar su ejecución esperar que se genere la salida que ha sido predefinida en la aplicación por el programador.

Por esta razón vamos a dividir este manual en dos submanuales, el primero para el usuario normal sin conocimientos o con conocimientos reducidos de programación y el segundo para un usuario mas avanzado que puede retocar la documentación.

Otra cosa a mencionar es el uso que haremos de la palabra *proyecto* durante este manual. Con proyecto nos referiremos al conjunto de tareas que queremos que se realicen sobre unos datos concretos. Por tanto si queremos al terminar estas tareas, mandar más operaciones sobre los mismo datos, seguirá perteneciendo al mismo proyecto que antes.

Utilización por parte de un usuario normal:

Antes de la realización de ninguna tarea suponemos que este usuario ya tiene localizadas la pila de tareas que quiere realizar sobre los datos que va o ha insertado previamente.

Si queremos saber cómo implementar esas clases Task debemos dirigirnos a la sección de la memoria que explica el módulo *Run-time* y ver la explicación de la tarea *Task* y un ejemplo de una clase que lo extienda, o ver directamente el código de algunos ejemplos de tareas que se incluyen con la distribución.

Tras tener las tareas que se van a lanzar (pueden haber sido implementadas por un tercero y conservarse como una pila de tareas), lanzamos la interfaz gráfica que nos guiará por la configuración del proyecto, y donde iremos completando en primer lugar el archivo desde el que se van a cargar los datos (la implementación por defecto sólo contempla la lectura de datos desde un archivo), segundo las variables con sus tipos que tendrá cada una de las muestras estadísticas que vayamos a insertar.

Siguiendo por la interfaz iremos marcando el resto de opciones para la aplicación incluyendo las tareas que serán ejecutadas por la aplicación y las dependencias entre ellas (si i depende de j, quiere decir que i debe esperar a que j termine su ejecución para comenzar la suya, ya sea por modificación en los datos o porque i necesite un valor suministrado por j).

Por último se lanzará la aplicación comenzando a lanzarse tareas contra los datos insertados o bien insertando los datos desde el archivo a la base de datos si es que marcamos la tarea de lectura como una de las tareas a ejecutar. Al terminar la aplicación a través de un mensaje por pantalla avisará de la finalización de las tareas con lo que podemos dirigirnos al pdf generado para el proyecto en /statistics_repository/NOMBRE_DEL_PROYECTO/ results.pdf

Para ver con profundidad como navegar a través de la interfaz y las opciones que nos propone, podemos dirigirnos a la sección de este documento dedicado al módulo Interfaz donde se describen las diferentes ventanas y su uso.

En ningún caso hará falta que este tipo de usuario recompile el programa, ya que el único código cambiante en la aplicación de un uso a otro será el de las tareas a ejecutar y éstas se compilan y cargan en tiempo de ejecución liberando al usuario de esa tarea (sólo deberá encargarse aquel que escriba el código, de que sean correctas y sigan la interfaz).

Utilización por parte de un programador:

Las acciones que puede realizar este usuario no están aisladas de las que podía realizar el anterior usuario, sino que son una ampliación de esas, ya que se supone que este usuario puede tocar el código en ciertas partes que indicamos para la introducción de un modo sencillo de otros módulos para añadir funcionalidad o bien cambiar la existente como podría ser el modo de presentación o la situación de la fuente de los datos (a través de Internet en una URL fija o como se desee)

Otras de las responsabilidades de este programador está durante la instalación, rellenando el xml de sistema con la base de datos instalada y eligiendo una base de datos dinámica o estática en función de los permisos que se quiera dar el usuario, para crear tablas en la base de datos (podremos elegir cualquier version de las dos) o sólo añadir valores (en este caso sólo es válida la opción de la base de datos estática)

- Añadido de un nuevo módulo de lectura:

  Para añadir un nuevo módulo de lectura debemos crear una instancia que implemente a la interfaz *Reader* (que es la que contiene el lector que deseamos cambiar).

  Luego debemos dirigirnos a la factoría ReaderFactory para que al ser invocado su método devuelva una instancia de la nueva clase que creemos.

  (Para más detalles consultar el módulo de lectura más atrás en este mismo documento).

- Añadido de un nuevo módulo de persistencia:

  Deberemos crear una nueva clase PersistenceModuleProject que implemente la función storeData(String)

  (Para más detalles consultar el módulo Persistencia más atrás en este mismo documento).

- Añadido de un nuevo módulo de presentación:

  Para añadir un nuevo módulo de presentación debemos crear una instancia que implemente a la interfaz *Presentation* (que es la que contiene el lector que deseamos cambiar). Solo debemos tener en cuenta cómo los datos serán guardados en la cosa de Resultados (ResultBeanList)

  Luego debemos dirigirnos a la factoría *PresentationFactory* para que al ser invocado su método devuelva una instancia de la nueva clase que creemos.

  (Para más detalles consultar el módulo de presentación más atrás en este mismo documento).

Después de cualquiera de estos cambios deberá ser recompilada la aplicación. El método que nosotros proponemos para facilitar todos estos cambios es mediante el IDE eclipse (www.eclipse.org) que es la herramienta con la que ha sido desarrollada esta aplicación.

2.3. Archivos de configuración

Como ya hemos explicado en varias ocasiones, *Ststistics4j* no es una aplicación que una vez completa todos los usuarios instalan y usan de la misma manera, sino que es un framework que sirve a los usuarios como base para que cada uno implemente una aplicación que se adapte a sus necesidades.

Para que el framework utilice las características deseadas por cada usuario, *Statistivs4j* utiliza unos ficheros de configuración, estos ficheros están escritos en lenguaje *xml,* y son leídos por el sistema. El usuario rellena estos ficheros de configuración, xo no lo hace directamente sobre el código xml, ya que puede ser que no tenga conocimientos suficientes para hacerlo, sino que los rellena a través de la interfaz de la aplicación, de una manera sencilla y amigable. A partir de estos datos introducidos por pantalla, nuestra aplicación crea los archivos de configuración dinámicamente utilizando la herramienta *JDOM*, y después a medida que va necesitando dichos datos, los lee de este archivo creado utilizando *SAX.* Como se utilizan y como funcionan las herramientas *sax* y *jdom,* es algo que explicaremos con detalle más adelante, en el apartado situado más atrás en la memoria, apartado que dedicamos única y exclusivamente al entendimiento de ambas.

De primeras se podría pensar que para que generamos un *xml* con los datos de configuración, si podríamos cargarlos directamente desde la interfaz en objetos java en memoria. Esto tiene una sencilla explicación, y tiene que ver con la división de la ejecución de la aplicación en tareas. La creación de los archivos de configuración pertenece a la tarea de lectura, y obviamente si queremos realizar esta lectura en una ejecución diferente que el procesamiento de los datos, necesitamos que los datos de

configuración se almacenen de forma persistente, para así poder ser utilizados en una ejecución posterior sin necesidad de realizar la lectura de nuevo.

Tenemos dos archivos de configuración comunes a toda implementación del framework. Los explicamos a continuación:

- **XMLConfSystem.xml**: En este archivo almacenamos la configuración del sistema, es decir, los datos relacionados con la instalación de la aplicación. Como se rellena durante la instalación de la aplicación, se rellena directamente sobre el código *xml*, sin interfaz, y no se crea dinámicamente, ya que es común a cualquier proyecto que se vaya analizar utilizando *statistics4j.*

Ejemplo de fichero de configuración de sistema:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE system SYSTEM "XMLConfSystem.dtd">
<system>
    <dataBase isDynamic="true" ip="localhost:3306" name="Statistics4j"
            user="root" pwd="root"/>
    <tasks>
            <task><taskName>Lectura y Almacenamiento</taskName>
                    <taskDescription>
                            Lee el contenido del proyecto y lo
                    almacena en la base de datos
                    </taskDescription>
            </task>
            <task>
                    <taskName>Calcular</taskName>
                    <taskDescription>Calcular</taskDescription>
            </task>
            <task>
                    <taskName>Presentación de Resultados</taskName>
                    <taskDescription>Presentación de
Resultados</taskDescription>
            </task>
    </tasks>
</system>
```

En la etiqueta <dataBase> se introducen los datos de configuración de la base de datos: nombre de la base de datos, dirección ip, nombre de usuario y contraseña para conectarse con ella, y si va a ser generada dinámicamente o estática (atributo *isDynamic*).

La configuración estática de la base de datos es una configuración creada para usuarios que no tienen permisos de usuario en su ordenador como para crear tablas en su base de datos. De este modo subsanamos el problema de los permisos aunque el rendimiento de inserción y búsqueda no será tan bueno como la dinámica.

A continuación en <tasks> tenemos la lista de tareas que se van a poder ejecutar en los proyectos analizados, con su nombre y descripción correspondientes. Son las tareas mínimas que se pueden ejecutar, a la espera de que el usuario incluya las suyas propias en cada proyecto, pero éstas ya no se guardarán en xml's pues son las tareas que queremos ejecutar inmediatamente y no tiene porque quedar guardado.

- **XML[NombreProyecto].xml**: Este es el archivo de configuración del proyecto. Su nombre depende del nombre del proyecto (XML + nombre +.xml). Se genera en tiempo de ejecución cuando cargamos un nuevo proyecto. El proyecto se carga desde un fichero de texto, el sistema crea a partir de los datos de este fichero el xml utilizando jdom, como ya hemos comentado antes. Aquí tenemos un ejemplo de xml de configuración de proyecto con el que iremos explicando los campos que contiene:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE project SYSTEM "XMLConf.dtd">

    <project>
        <root>projects/prueba.txt</root>
        <name>Prueba X Y</name>
        <description>Prueba con dos variables</description>
        <samplesSeparator />
        <variableList row="true" separator="&#x9;" number="2">
            <variable>
                <variableName>X</variableName>
                <type>java.lang.Integer</type>
            </variable>
```

```
                <variable>
                        <variableName>Y</variableName>
                        <type>java.lang.Integer</type>
                </variable>
        </variableList>
</project>
```

Este es el archivo correspondiente a un proyecto que consiste en general en un gran número de valores de dos variables, X e Y, enteras, (los valores no tiene porqué ser siempre enteros sino que sería posible cualquier tipo de Java como Double, String, Float …etc) sobre las cuales se realizarán operaciones estadísticas tales como la media, la varianza...etc.

Como se ve en el ejemplo, el archivo de configuración contiene la ruta del fichero de datos, el nombre del proyecto, su descripción, y la lista de variables de las que tenemos diferentes valores, con sus respectivos nombres y tipos (de java).

El valor **root** indica que los datos deben leerse desde el fichero prueba.txt situada en la carpeta projects del árbol de directorios de la aplicación. Este valor junto con el separador serán los que use el módulo de lectura para poder ir leyendo del archivo y obteniendo cada una de las muestras por separado. En caso de cambio en el módulo de lectura, estos campos podrían dejar de tener utilidad. (Para más información acerca de la lectura de los datos desde su fuente y método de guardado mirar las partes relacionadas con Modulo de Lectura y Modulo de Persistencia en esta mismo documento).

Name es el nombre que le pondremos al proyecto y que será guardado en la base de datos, en caso de volver a querer efectuar operaciones sobre los mismo datos tendremos que buscar este proyecto en la interfaz del sistema.

Descripción obviamente no será más que una breve descripción del proyecto para mejor entendimiento en el futuro de lo que aquí se ha guardado en la base de datos.

El valor **row**, indica si la lista de muestras están distribuidas por filas (una muestra en cada fila, o bien, si el valor es false que está distribuido por columnas (posible en el caso de estudios con muchos valores y pruebas efectuadas sobre un número pequeño de muestras)

El elemento "**separator**" se refiere al elemento que separa en una muestra cada uno de los valores (o experimentos) de una misma muestra. En este caso el valor será igual a "&#x9" lo que identifica una tabulación.

3. MÓDULOS DE LA APLICACIÓN

3.1. MODULO DE LECTURA

Introducción

Este será, como su propio nombre indica, el encargado de recuperar o leer los datos de un fichero, dirección web o allá donde quiera que estén ubicados, y mediante el módulo de persistencia volcar esos datos a la base de datos que el sistema tenga instalada para el posterior tratamiento de éstos. En la implementación por defecto que se ha incluido en el proyecto la lectura se hará desde un fichero de datos.

Active Object

El módulo de lectura se ha basado en este modelo para hacer más eficiente el transvase de datos desde la fuente original a la base de datos.

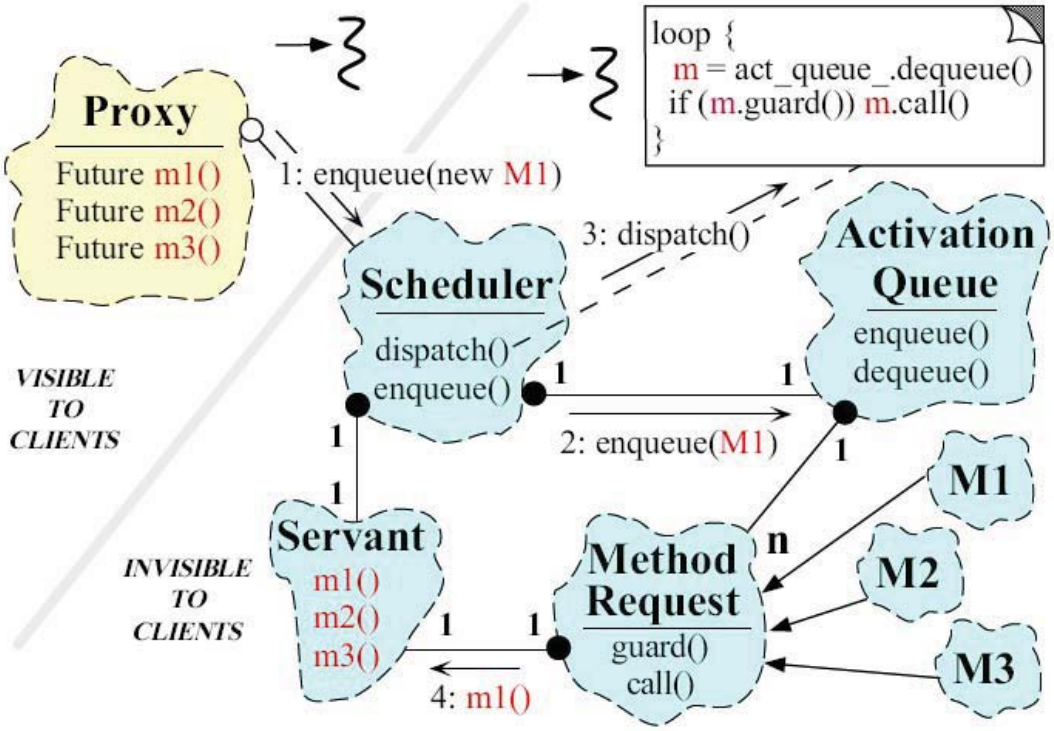
La razón para la elección de este modelo de diseño es que, con el avance de las telecomunicaciones en el caso de que la fuente de datos fuera un servidor remoto o más aún en el caso que los datos se obtengan directamente de disco en un archivo u otro soporte físico; el tiempo que la base de datos tardará en la inserción y procesamiento de los datos será superior al de la lectura en sí. Teniendo esto en mente, usamos este modelo de diseño para que aprovechar ese tiempo de espera y evitar a la vez una espera activa por parte del lector a que el módulo de persistencia haya guardado los datos para mandar más (evitando el llenado de la memoria) en la que se gastaría tontamente recursos de CPU.

Explicación del modelo

El modelo pretende aprovechar las ventajas de aplicaciones multitarea y el acceso compartido de varios hilos (aplicaciones) a un mismo objeto y sus recursos que se ejecuta en su propia hebra de ejecución.

Mediante el modelo varios hilos pueden tener acceso a un objeto compartido en un hilo diferente sin que el resto de los hilos deban estar esperando a que termine el servicio a otro hilo para seguir ejecutando. De este modo estas aplicaciones paralelas pueden dejar su petición y seguir su ejecución normalmente.

Para conseguirlo tendremos una cola de acciones en las que los hilos que se ejecutan en paralelo irán introduciendo los mensajes que desean que el objeto activo ejecute por ellos, junto con un "futuro", donde estos programas buscarán el resultado de la ejecución del proceso cuando sea avisado de que está listo. El "objeto activo" (servant) le pedirá al controlador de la cola (scheduler) la siguiente petición a de los clientes y la ejecutará en su propio hilo



- **Proxy:** es la única clase que es visible al servidor. Mediante esta clase los distintos programas (hebras de ejecución) mandan los mensajes (peticiones de ejecución) al Scheduler que controla la cola. También le pasará los "Futuros" en donde quedarán guardados los resultados de las peticiones de ejecución.

- **Scheduler:** es el encargado de la cola. El Proxy ejecutara su función *enqueue()* para meter una nueva petición de ejecución en la cola de activación. El servant tras acabar la ejecución de una petición le pedirá (función *dispatch()* **)** el siguiente mensaje o código a ser ejecutado.

- **Servant:** es el que ejecutará en su propio hilo las peticiones (Method Request) que le pase el scheduler.

- **Activation Queue:** Es la cola donde se almacenan las peticiones de ejecución según orden de llegada.

- **Method Request:** contiene el código que se quiere que sea ejecutado o guardado por el Servant.

Implementación del modelo

Variación del modelo respecto al inicial

Hemos cambiado la implementación del modelo a una variación a la que ya se hace mención en el propio paper que hace la explicación de este. Ésta se refiere a un pool de hebras que a serás las que vayan realizando el trabajo a medida haya información que procesar.

En nuestra implementación el lector irá colocando "mensajes" en una cola, a la espera a que alguien los coja y los procese. Para que esta acción no provoque un desbordamiento de memoria ante una entrada masiva de datos, el lector parará cuando alcance un nivel de llenado en la cola, quedando a la espera (espera no activa) hasta que sea informado de que puede seguir insertando valores.

Varios hilos de ejecución irán entrando en esa cola con la información y cogiendo los mensajes para que sean procesados (guardados en la base de datos). Así en el caso de disponer de un multiprocesador reduciremos costes en tiempo de un modo notable con la ejecución en paralelo de estos hilos que traducen el mensaje transformándolo a un objeto que luego sea introducido en la base de datos.

Cuando la cola de mensajes baje de otro cierto tope de carga el lector será informado de que puede seguir insertando valores si lo desea, reiniciando su ejecución en el caso de que parara por llenado de la cola. En el momento que el lector acabe de insertar todos los datos informa de que ha acabado al controlador (scheduler) de la cola.

Clases de nuestro paquete reading_module

La disposición del paquete reading_module es la que podemos ver en la siguiente figura (Figura 3.1), cuya explicación de clases detallamos a continuación.

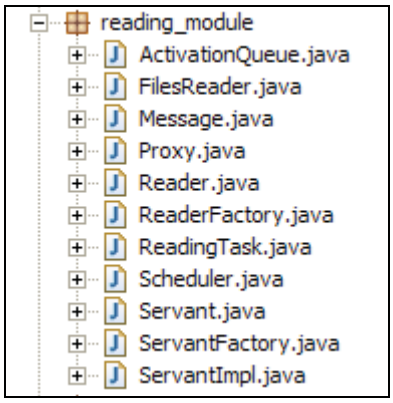


Figura 3.1.1

Interfaz Reader

Es la interfaz que debe cumplir la clase que haga la lectura desde la fuente que sea (seguimos intentando hacer que el framework sea lo mas genérico y configurable posible).

Intentando hacer aún mas independiente cualquier tipo de implementación del uso de esta clase, la creación de una instancia de esta clase se realiza mediante el método *static public Reader newReader(Project project)* de la clase *ReaderFactory*. Por lo tanto es en esta factoría donde debemos cambiar la llamada para incluir el nuevo lector que queramos insertar. Así la implementación del lector queda aislada del resto de la aplicación.

Esta interfaz sólo posee dos funciones imprescindibles para que pueda ser llamada desde el Proxy, que son:

- *public void readData():* esta es la función que será invocada por el Proxy para informar al Reader que empiece a mandarle datos (mensajes) para introducirlos en la cola de mensajes. El modo de mandar los mensajes es a través de su Proxy y de su función *public void interpretateData(String string)* pasando como parámetro una linea de información.

- *public void setProxy(Proxy proxy):* con esta función el Reader guarda el Proxy al cual tiene que mandar los mensajes, para que sean incluidos en la cola, y también al que informará una vez haya introducido todos los mensajes, diciéndole que su trabajo ya ha terminado.

Una última cosa, que la implementación que el usuario haga de esta interfaz debe cumplir, es el informar a su Proxy cuando haya terminado de mandarle toda la información existente en la fuente de datos. Este aviso lo realiza mediante la llamada a la *función public void isDone()* con la que ya el Proxy se encargará de avisar al resto de las clases.

En la distribución por defecto que hemos creado, incluimos la clase *FilesReader* que lee de un fichero de texto que se le indica mediante el "proyecto" (insertado al lanzar la aplicación y guardado en un XML) que se ha insertado.

Clase Message

Es la clase que contiene la línea de información (muestra en el caso de funciones estadísticas) que será guardada en la base de datos. Estos mensajes son los que el lector dejará, a través del Proxy, en la cola de mensajes (Activation Queue) para que sean cogidos y guardados.

Si elegimos implementar una nueva clase *Message* sólo tenemos que extender ésta clase redefiniendo la función *public void Call(Servant servant)* del modo necesario. En esta clase se define como se ejecutará (en este caso guardar) el mensaje que incluye el lector en la cola. Es la función que los servant ejecutarán cuando cojan el mensaje de la cola.

Los mensajes son creados a través de un String y pasados al scheduler.

Clase Proxy

El proxy sirve como nexo entre el Reader y el Scheduler que controla la cola de mensajes, creando la instancia de ambas clases que será usada por la aplicación y pasándole al Scheduler el número de hilos que deben usarse en la interpretación de los datos.

Es el encargado de lanzar el lector para que comience a mandarle información. El lector ejecutará sus funciones:

- *public void interpretateData(String string)* para pasándole un String de datos se cree la clase *Message* que le pasa al scheduler para que la añada a la cola de mensajes y sea ejecutado.

- *public void isDone():* Ésta es la función que invoca el lector cuando ha acabado de mandarle información y con la que el Proxy le dice al scheduler que no se le mandarán más mensajes.

Contiene un puntero al Scheduler que ejecutará los mensajes que le pase el lector y otro a la tarea de lectura (ReadingTask) de la que forma parte y a la que tras acabar informará que ha terminado toda la lectura.

Clase ActivationQueue

La clase Activation Queue implementa la cola de mensajes que serán ejecutados con lo que los métodos que implementarán seran los de añadir y sacar elementos de la cola:

- *public synchronized void add(Message message):* mediante este método incluimos una nueva instancia de la clase Message en nuestra cola. Esta función controla el tamaño de la cola, durmiendo el hilo en el que se está ejecutando cuando el tamaño de ésta supera los 50000 mensajes. Esta acción tendrá como consecuencia que el hilo en el que se están ejecutando el Reader y el Proxy quede a la espera de ser despertado, cosa que debe hacer el Scheduler como luego veremos.

- *public synchronized Message getNext():* ésta sera la función que deberá ser llamada (en nuestro caso por el Scheduler) para obtener el siguiente elemento de la cola de mensajes a ser guardado en la base de datos. Tras sacar el elemento de la cola de mensajes decrementa en uno el tamaño de la cola.

  Es un método sincronizado para evitar que dos hilos (hebras de ejecución) puedan entrar a la cola en el mismo instante y obtener el mismo elemento u otros problemas provenientes de la concurrencia.

  Por último también se encarga de despertar (*notify*) el hilo que fue obligado a dormirse (*wait*) para evitar que la cola se sobrecargase (mediante la función anterior) cuando la carga de la cola baja de una cantidad (en este caso 25000 mensajes) haciendo que se vuelva a activar y mande más mensajes si fuera necesario.

Clase Scheduler

La clase scheduler (despachador) es la encargada de el acceso a la cola de mensajes, tanto para insertar un nuevo mensaje en ella como para retirar el primero y dárselo a uno de los sirvientes (clase Servant) para que sean guardados en la base de datos.

Al crearse la instancia de la clase Scheduler mediante su contructor *Scheduler(int numThread, Proxy myProxy)* le pasamos el número de threads o sirvientes, que se ejecutarán en hilos diferentes procesando la información.

También le pasamos la instancia de la clase Proxy que lo ha creado y a quien tendremos que informar tras acabar de procesar todos los mensajes de la cola.

Funciones a señalar de la clase Scheduler:

- *Scheduler(int numThread, Proxy myProxy)*: en el constructor de la clase, asignará myProxy a la referencia que contiene, creará numThread servants mediante sucesivas llamadas a ServantFactory pasándose a sí mismo como parámetro de la función de la factoría.
- También creará una cola de mensajes (ActivacionQueue) vacía en la que luego añadirá los mensajes a ser guardados.

- *public synchronized void threadEnd():* una vez que un servant acaba de ejecutar (guardar) un mensaje, pedirá al Scheduler otro mensaje que procesar. Si el scheduler le da un objeto vacío, indicando que la cola esta vacía, mirará si tiene que llegar mas información o no.

  Al chequear que no van a llegar mas mensajes (condicion=trae), el servant llamará a esta función para que el Scheduler disminuya en 1 el número de threads (servants) activos y terminará. Si el número de servants es 0, el server informará al Proxy que ya ha acabado con su trabajo y todo se ha guardado.

- *private void finish():* el la función con la cual el Scheduler indica al Proxy que ya ha acabado su trabajo y puede dar por concluida la tarea de lectura de datos.

- *public void add(Message message):* mediante esta función y pasando como parámetro una instancia de la clase *Message* , éste será añadido a la cola de mensajes (ActivationQueue) para su posterior uso.

- *public synchronized Message getNext():* obtiene de la cola de mensajes el primer mensaje listo para ser ejecutado.

  Es un método sincronizado para evitar problemas derivados de la concurrencia ante la posibilidad de que dos o más servants puedan intentar acceder al siguiente elemento de la cola en el mismo instante de tiempo.

- *public void setCondition():* cuando el lector informe al Proxy (ver clases Proxy y Reader) de que ha terminado de introducir datos (Messages) en la cola de activación, el Proxy llamará a esta función indicando este hecho. Como consecuencia el atributo **condicion** del Scheduler se cambia a trae. De este modo los servants pueden conocer llamando a la función *public boolean condition()* que si no hay mas mensajes en la cola ya pueden acabar e indicarselo al Scheduler.

Interfaz Servant

El Servant es la clase encargada de guardar o hacer todas las acciones que sean necesarias con los datos de la cola. Aunque no es imprescindible para seguir los requisitos que impone esta interfaz, lo ideal es que la clase que implemente esta interfaz corra en su propio hilo de ejecución.

Mediante la ejecución de un servant en un hilo propio podemos conseguir evitar parte de los retrasos debidos a espera de entrada salida y escritura en la base de datos, con inserción de varios datos paralelamente (sistemas gestores de bases de datos como MySQL admiten varios hilos de inserción trabajando concurrentemente).

Continuando con nuestra intención de hacer unas clases lo más independientes unas de otras. Declaramos la interfaz Servant que será el tipo de objetos que serán creados por el Scheduler y que guardarán o harán lo que quiera que el programador de su implementación desee con los datos.

Para evitar el conocimiento por parte del Scheduler de los pormenores de esta clase (su contructor o parámetros necesarios) hemos incluido una factoría que será la encargada de crear objetos de este tipo *ServantFactory* mediante llamadas a su única función *public static Servant newServant(Scheduler scheduler)* y pasándole como parámetro el Scheduler donde deberá ir a buscar mas Mensajes que guardar o con los que trabajar. Por lo tanto será en l factoría donde el programador que desee usar su propio Servant debe ir a cambiar la implementación.

La interfaz cuenta con una función:

- *public void saveData(String string):* esta función es la que llamará la clase Message cuando se invoque su método *public void Call(Servant servant)*, en la implementación dada por defecto. Si se cambiara la clase Message y la implementación de su método call, éste método dejaría de tener utilidad y podría obviarse su implementación dejándolo simplemente vacío.

En la distribución por defecto que se da, se incluye una clase que implementa ya este interfaz y así mismo sirve como ejemplo para observar como debería implementarse un objeto que siguiera esta interfaz ejecutando en su propio hilo, es decir extendiendo de la clase *Thread* . Esta clase es **ServerImpl** y su implementación se detalla a continuación:

Clase ServantImpl

Esta es la implementación de la interfaz Servant dada en esta distribución como implementación por defecto. También será útil para el programador como ejemplo de cómo implementar un Servant que corra en su propio hilo

aprovechando las ventajas que nos ofrecen los multiprocesadores. Para ello extenderá la clase Thread.

Como atributos contiene:

- Un atributo de tipo Scheduler, que será a quien le pida constantemente y hasta que no haya mas trabajo por hacer, los mensajes de la cola de datos (ActiveQueue)

- PersistenceModuleProject. Un puntero o referencia a un objeto del módulo de persistencia que será el que guardará en la base de datos o destino implementado en el módulo de persistencia, la información que le pasemos.

Sus métodos son:

- *ServantImpl (Scheduler myScheduler):* es el constructor para la clase. Se le pasa una referencia al Scheduler al que le pedirá las instancias de la clase *Message* para ser tratados por el.

  Tras guardar el Scheduler hace una llamada al método *start()*, perteneciente a la clase Thread que estamos extendiendo. A partir de esta llamada la ejecución de esa clase se ejecutará en un hilo diferente. El código que se ejecutará en ese hilo será el que esté en el método *public void run()*.

- *public void run():* es el método que se ejecuta en un hilo diferente tras la llamada a *run()* en la clase Thread o en este caso en una extensión de esta clase.

  En este método nuestra implementación del sirviente tiene un bucle en el que constantemente pide un nuevo mensaje (clase Message) al Scheduler y lo ejecuta llamando al método *call()* de Message y éste a *saveData()*.

  Si el mensaje proporcionado por el Scheduler es igual a null y la condición del Scheduler (metodo *boolean condition()*) es cierta, quiere decir que no hay más mensajes en la cola y no llegarán mas. Tras esta circunstancia avisa al Scheduler de que acabó su trabajo y sale del bucle terminando la ejecución del hilo.

- *public void saveData(String string):* éste método es el que llamará la clase Message para ser guardado. Le pasará como parámetro el String de datos que contiene para que lo inserte en base de datos o aquel sistema que esté implementado.

En este método el Servant creará una instancia a un objeto de tipo *PersistenceModuleProject* y llamará a su método *void storeData(String data)*
pasando como parámetro la información suministrada por Message.

3.2. MODULO RUN-TIME

Introducción

El módulo de Run-time es el hilo de ejecución principal del programa, esto es, el motor principal de la aplicación que lanzará el resto de los módulos para que realicen las tareas.

La idea del módulo Run-Time es la del despachador de tareas de un procesador normal y corriente en cualquier ordenador. Tendremos una cola te tareas (más tarde veremos que esta función la tomara la clase abstracta *Task*) esperando su turno para ser ejecutadas. Estas tareas pueden ser de diversos tipos: tareas de lectura, de ejecución de medias, de obtención de datos de diversos tipos o de creación de gráficas que más tarde serán guardadas en archivos de texto, pdf's etc.

Los nombres y ubicación de las clases que implementan las tareas serán proporcionadas por la interfaz del programa.

Después de que una tarea se ejecute dejará su resultado para que el resto de las tareas que puedan necesitarla o bien el módulo de presentación recoja esta información. También informará de su finalización correcta para que otras tareas que esperan por ésta puedan comenzar su ejecución cuando le toque su turno.

Tras la ejecución de todas las tareas que fueron cargadas para este "proyecto", el módulo run-time llamará al módulo de presentación (*presentationModule*).

Como extras a lo que es el módulo de presentación y su paquete, hemos añadido un subpaquete llamado *utilities* donde tendremos diferentes clases de utilidad para el usuario a la hora de tratar la información que obtenga del tratamiento de los datos. Estas clases permites la generación de diferentes tipos de gráficos mediante FreeChart (diagramas de barras, diagramas de puntos o diagramas de tarta).

Explicación detallada del paquete run-time

Explicaremos a continuación el funcionamiento del módulo de un modo más detallado y usando las clases del paquete que podemos ver en la Figura 3.2.1:

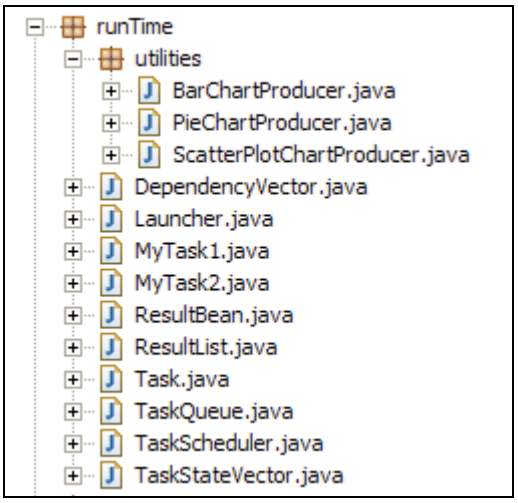


Figura 3.2.1

La primera clase en ser invocada será la clase *Launcher*. Tras guardar la configuración se cargarán las clases que implementarán las tareas que ejecutarán en el proyecto.

Se pasará un vector estático con las direcciones completas de las tareas, se compilarán y cargará una instancia de cada una de ellas en la cola de tareas (*TaskQueue*), incluyendo en la lista de dependencias de cada una las tareas de quien depende. Todas estas clases deben extender la clase abstracta *Task* para ser ejecutadas. Al ser cagada cada clase, se incluirá un elemento en el vector *TaskStateVector* indicando el estado en que se encuentra la tarea (inicialmente comenzará a finalizado=false).

Una vez cargadas todas las clases se indicará al planificador de tareas (*TaskScheduler*) que puede comenzar a lanzar las tareas en orden, siempre y cuando cumplan con sus dependencias. El planificador preguntará a la primera tarea en la cola si puede ser ejecutada. Si la respuesta es negativa se pasará al siguiente elemento, sino se ejecutará la tarea que se ha indicado.

En las tareas que se han implementado para ser ejecutadas se podrá, a parte de cargar información, realizar operaciones de cualquier tipo sobre los datos insertados y generar resultados finales o parciales; usar clases del paquete de utilidades que se incluye con el run-time. Con éstas podran generarse

diferentes tipos de gráficos para ser impresos como parte de los resultados generados en el proyecto.

Una vez una tarea termine su ejecución, si la tarea ha realizado una acción cuyo resultado vaya a ser guardado o impreso, o bien se necesite por otra tarea, creará un elemento de resultado *ResultBean* y lo incluirá en la cola de resultados, con su identificador de tarea para que quien necesite ese resultado sepa que él lo generó. Tras esto cambiará su estado en el vector de estados y avisará al planificador que un cambio se ha producido.

Cuando la última de las tareas haya finalizado, el planificador mismo será quien invoque al módulo de presentación para que tome el vector de resultados *ResultList* y realice las acciones que sean necesarias para mostrar los resultados que deban ser mostrados.

Al terminar todo, se informará al usuario mediante un mensaje de la finalización de todas las tareas (proyecto). Con esto terminará la ejecución del programa.

Clases del paquete RunTime

Clase Launcher

La clase Launcher es la que podría llamarse la clase principal del sistema (lanzador como su propio nombre indica). Desde ésta lanzaremos los otros módulos del sistema ya sean de lectura (dentro de una tarea de lectura), ejecución de las operaciones en las tareas correspondientes o presentación. Creará un planificador de tareas, cargará las tareas que sean ejecutadas y lanzará todas las tareas.

Atributos:

- *private TaskScheduler taskScheduler*: referencia al despachador que ejecutará las tareas.
- *private TaskQueue queue*: cola donde serán guardadas las tareas hasta su lanzamiento.
- *static Category logger*: referencia del logger del sistema donde indicaremos el inicio de las tareas.
- *private LogsProperties logsProperties*: referencia a los properties que contienen la información para introducir en los logs (base para multilingüe).

Funciones de la clase Launcher:

- *Launcher():* es el constructor de la clase. En él se crea una nueva cola de tareas TaskQueue y un planificador (TaskScheduler) asociado a esa cola de tareas.

- *public void loadTasks():* carga las tareas que se le pasan en un vector estático por parte de la interfaz. Las compila y crea "on the fly" una instancia de esta clase insertándola en la cola de tareas como elemento de la clase Task.

- *public void start():* llama a la funcion *loadTasks()* cargando las clases y tras esto indica al scheduler que ya puede comenzar a lanzarlas mediante la llamada a la función *execute()* del scheduler. Por último avisa al usuario de la finalización de la ejecución mandándole un mesaje.

Clase TaskScheduler

El TaskScheduler o despachador es el encargado de dar paso a las tareas de la cola de tareas TaskQueue a ejecución.

Después de que el lanzador (*Launcher*) llame a su método *execute()*, se encargará, mientras la cola no esté vacía, de ir cogiendo el primer elemento de la cola mediante la función *Task getNext()* de la cola de tareas.

Si éste es nulo (al no estar la cola vacía) querrá decir que los elementos de la cola están todos esperando a que otras tareas terminen primero así que se dormirá hasta que sea notificado (mediante su función *warning()*) de que algo ha cambiado.

Si el elemento no es null llamará a la función *start()* de la tarea para que empiece su ejecución en un nuevo hilo, volviendo a por el siguiente elemento de la cola. Al quedar la cola vacía volverá a quedarse esperando a que le avise el último elemento de que ha terminado. Cuando esto ocurra llamará al modulo de persistencia pasándole como parámetro el vector de resultados.

Las funciones más importantes de la clase TaskScheduler son:

- *TaskScheduler(TaskQueue _queue):* el constructor de la clase TaskScheduler. Crea una nueva instancia de la clase y le asocia la cola de tareas (TaskQueue) _queue, donde irá a buscar las tareas a ejecutar.

- *public synchronized void warning():* cuando una tarea acaba su ejecución en un hilo paralelo, llama a esta función avisando al scheduler de que ha terminado, ha puesto su resultado en el vector de resultados y ha cambiado su estado en el vector de estados a "terminado". Mediante el

notify() despierta al scheduler si es que estaba dormido para que continúe su ejecución.

Si llegados a este punto la cola de estados indica que todas las tareas han terminado, en esta misma función se lanza el módulo de presentación.

```
if(TaskStateVector.getInstance().isDone()){
   presentation=PresentationFactory.getPresentationInstance();
this.presentation.developePresentation(ResultList.getInstance());
}
```

- *public synchronized void execute():* el lanzador llama a esta función para que el planificador comience su trabajo. Tras ser invocado entra en un bucle en el que si la cola de tareas no está vacía toma el primer elemento, si es igual a nulo dormirá su hilo de ejecución (*wait()*) evitando una espera activa que consuma recursos del procesador hasta que sea despertado (*notify()*) por algún hilo o tareaa llamando a *warning()*.

  Si el elemento tomado de la cola no es nulo llama a su función *start(this)* para que comience su ejecución en un hilo adicional, pasándose a si mismo como parámetro para ser notificado de su terminación.

  Tras el vaciado de la cola de tareas, el scheduler volverá a quedar esperando hasta que todas las tareas hayan terminado su ejecución. Momento en que acabará su hilo de ejecución y devolverá el control al *Launcher*.

Clase TaskQueue

Es la cola de tareas que se ejecutarán en la aplicación. Será un objeto estático, es decir implementa el patron ***singleton***, de modo que pueda ser accesible por todos los objetos del paquete mediante su la función *static TaskQueue getInstance()* y sólo pudiendo existir una instancia de esta clase en el sistema que será compartida por los objetos que quieran acceder a ella.

Cuando el scheduler pida el primer elemento de la cola, se comprobará que esta tarea esta lista para ejecutarse (las tareas de las que depende han terminado) si no está lista se la sacará de la cabeza de la cola y la insertará en la última posición de la cola.

La TaskQueue irá chequeando todos los elementos de la cola, si cuando los chequee todos no hay ninguna tarea disponible para su ejecución devolverá un null, con el que el scheduler sabrá que no hay ninguna tarea disponible.

La cola por tanto implementa una algoritmo de planificación Round Robin poniendo a en la última posición de la cola el elemento de la cabeza que no se pudo ejecutar.

Las funciones más importantes de esta clase son:

- *private TaskQueue():* constructor privado de la clase. Sólo podrá ser invocado por la función getInstance() obligando de éste modo a que sólo puede haber una instancia de esta clase en el sistema.

- *static TaskQueue getInstance():* llamando a este método obtenemos la única instancia de la cola de tareas. Si se había creado una antes se devuelve esa, sino, se crea una nueva instancia, se asigna al atributo estático de tipo TaskQueue que contiene y se la devuelve.

- *public void addTask(Task task):* añade la tarea task al final de la cola de tareas.

- *public Task getNext():* va chequeando todos los elementos de la cola, si el elemento actualmente en la cabeza puede ser ejecutado (las tareas de las que depende han terminado) lo devuelve y lo saca de la cola. Si al chequear el elemento no puede ser ejecutado, se sacará de la cabeza y se insertará al final de la cola de tareas. Si se recorren todas las tareas de la cola sin éxito se devuelve un null como resultado.

- *public boolean idDone():* devuelve *true* si la cola de tareas está vacía y no hay elementos que ejecutar. Devuelve false en caso contrario

Clase TaskStateVector

Es el vector de estado de las tareas. En este vector se guarda el estado de cada una de las tareas, con lo que tanto el scheduler como el resto de las tareas saben si todas las tareas han acabado o si una tarea en concreto ha terminado su ejecución.

Implemente el patrón ***Singleton*** con lo que el constructor de esta clase es privado y sólo se podrá obtener la única instancia de esta clase que habrá en el sistema mediante una llamada a la función *static TastStateVector getInstance()*.

Las funciones más importantes de la clase son:

- *static TaskStateVector getInstance():* mediante esta función se obtiene la única instancia de esta clase que habrá en el sistema y que compartirán todas las tareas.

- *public void SetState(int ID, boolean state):* guarda en el vector el estado de la tarea con identificado ID. Si el estado es terminado state=trae, en caso contrario es false.

- *public boolean getState(int ID):* devuelve el estado de la tarea con identificador ID.

- *public boolean isDone():* devuelve true cuando todas las tareas del vector de estado han terminado su ejecución y tienen por tanto un true en su casilla. Devuelve false en caso contrario.

Clase ResultBean

Es la clase que implementa el lugar donde serán guardados los resultados de las tareas que quieran dejar los resultados en la lista de resultados (ResultList), bien porque deban ser usados por otras tareas o porque quieran que sean tratados y mostrados por el módulo de persistencia.

Hay dos tipos de resultados, los que contienen un gráfico (Chart) o un valor o lista de valores normal y corriente.

Estos *Beans* serán los que se guardarán en la lista de resultados, con una clave (*Key*) que lo identificará cuando otra tarea quiera buscarlo en la lista de resultados.

Atributos de la clase ResultBean:

- *private List list:* lista de elementos en caso de que sean varios los que formen parte del resultado.

- *private int type:* tipo del resultado, los valores posibles son los que vemos mas abajo. Indicará al módulo de persistencia como tiene que usar estos resultados para imprimirlos o guardarlos del modo que tenga programado.
- *private String title:* título del resultado si queremos que se escriba en la salida. Este valor no identifica el valor sino que es útil sólo para su presentación.

- *private String key:* este String es el que identificará el bean entre todos a la hora de que otra tarea quiera encontrar este resultado para usarlo en sus propios cálculos.

- *private boolean isFinal:* indica si el valor va a ser utilizado por el módulo de presentación o únicamente es útil para que otra tarea use este resultado en sus propios cálculos. Si es false el módulo de persistencia no hará caso de este valor, si es true lo mostrará.

- *private String description:* descripción del valor o del gráfico obtenido. Sólo útil para presentación.

- *private String comments:* comentarios o aclaraciones de los posibles resultados. Sólo útil en la presentación.

- *public static final int CHART = 1;*

- *public static final int VALUE = 2;*

Todas los atributos excepto los dos últimos cuentan con getters y setters para obtener y asignar sus valores. Las funciones destacables son:

- *public ResultBean():* constructor de la clase. Crea una lista de resultados, asigna a null tanto los comentarios como la descripción y el título…etc. Asigna también isFinal a false, con lo que si quiere ser mostrado tendrá que cambiarse este valor a true.

- *public void addResult(Object object):* Añade un nuevo objeto a la lista de resultados en caso de resultados compuestos.

Clase ResultList

Implementa la lista de resultados donde las tareas dejarán los resulsBeans que creen para que o bien los usen otras tareas o los use el módulo de presentación.

Implementa el patrón ***Singleton*** con lo que sólo habrá una instancia en el sistema que será compartido por todas las tareas. Para obtener una instancia de esta clase habrá que llamar a la función estática *static ResultList getInstance()*.

Funciones más importantes:

- *static ResultList getInstance():* devuelve la única instancia de la clase ResultList que habrá en el sistema, creando una si es la primera vez o devolviendo la creada anteriormente.
- *public void addElement(ResultBean resultBean):* añade un nuevo elemento de tipo ResultBean a la lista de resultados.
- *public int size():* devuelve el tamaño de la lista de resultados.
- *public ResultBean get(int i):* devuelve el ResultBean que ocupa la posición i de la lista de resultados.
- *public ResultBean getKey(String key):* devuelve el resultado (ResultBean) de la lista de resultados con clave key.

Clase abstracta Task

La clase task es la que implementa lo que queremos que la aplicación obtenga de los datos introducidos. El usuario tendrá que implementar sus propias tareas y hacerlas extender de esta clase para que sean ejecutadas, implementando el único método abstacto de esta clase que es donde debemos introducir toda nuestra funcionalidad.

Esta clase extiende la clase Thread ejecutando por tanto en un hilo diferente lo que quiera que escribamos en el método run() una vez sea llamado el método start() que hereda de Thread.

Task es una clase abstracta para forzar al usuario a realizar ciertas acciones antes y después de ejecutar lo que pongamos en ese método abstracto que debe implementar el usuario en su propia clase. También se le dan funciones para guardar los resultados que obtenga, del modo adecuado y estandarizado para ser tratados más tarde por el módulo de persistencia.

Al terminar la ejecución de lo que se introduzca en la función execute(), la clase abstracta hace que, sin que el usuario se tenga que preocupar por ello, se introduzcan los resultados en la cola, se ponga su estado en la lista de estados a "terminado" y por fín notifique al TaskScheduler que ha terminado su ejecución y por tanto si una tarea dependía de ésta podrá ejecutarse.

Atributos de la clase abstracta Task:

- *private int ID:* identificador de la tarea. Este identificador debería ser único.
- *private DependencyVector vector:* vector de dependencies donde quedan guardados los identificadores de las tareas de la que depende esta tarea.
- *private TaskScheduler scheduler:* planificador al que avisará la tarea una vez acabada su ejecución.
- *protected ResultBean result:* resultado obtenido por esta tarea para ser guardado en la cola de resultados si es que elegimos introducirlo en ella.
- *private boolean save:* indica si queremos introducir el valor en la lista de resultados. Si true introduciremos el valor en la cola al acabar la tarea, si false se desechará el resultado.

Funciones más importantes de la clase abstracta Task:

- *public Task(int iD):* constructor de la clase Task, crea una tarea con un identificador iD. Éste identificador debe ser único. Crea una instancia de ResultBean y le asigna como clave su propio nombre. Pone su estado en el vector de estados a false (aún no finalizada).
- *public void run():* método que se ejecutará, cuando se llame al método start() heredado de Thread, en un hilo diferente.

  Incluye la llamada a *execute()* que el usuario deberá implementar en su clase. Después de que acabe esta llamada, directamente sin que el usuario se tenga que preocupar, inserta el resultado en el vector de resultados si save=true y avisa al Taskcheduler de la terminación de la tarea.
- *abstract public void execute():* es el método que queremos que la tarea que implementemos reescriba. En el pondremos el cuerpo de la tarea con todas las operaciones que queremos que se ejecuten, como carga de datos, medias, creación de gráficos, etc.
- *public void addDependency(int id):* inserta una nueva dependencia de la tarea con identificador id en la lista de dependencias.
- *public void start(TaskScheduler _scheduler):* asigna _scheduler a la referencia a TaskScheduler que contiene con lo que ahora tiene a quién avisar cuando acabe la tarea y llama a Thread.start() creando un nuevo hilo y empezando la ejecución de lo que quiera que haya en el método run() de la clase que extiende Thread.

- *public boolean isReady():* indica si la tarea está lista para ejecutar. Mira su lista de dependencias y las contrasta con la TaskStateVector para ver si todas las tareas de las que depende han acabado. Si true, todas las tareas de las que depende han terminado su ejecución y por lo tanto está lista para comenzar.

- *public void setResultTitle(String title):* asigna un título al resultado que va a producir.

- *public void saveResultFinal(String key, String title):* asigna la clave key y el título title al resultado que contiene y lo deja listo para grabar como resultado final, es decir, será leido y usado por el módulo de presentación.

- *public void saveResultPartial(String key):* asigna la clave key al resultado que contiene y lo deja listo para grabar como resultado parcial, es decir, no será leido ni usado por el módulo de presentación. Sólo servirá para que otras tareas lo usen en sus cálculos.

- *public void storeResult(Object object, int type):* guarda el objeto object en la lista de su resultBean indicando qué tipo de resultado es (gráfico o valor).

Uso de la clase Task

En la implementación hemos dejado ejemplos de cómo extender y hacer uso de la clase Task. Uno de estos ejemplos es MyTask

```
package es.ucm.fdi.statistics4j.runTime;

import java.util.LinkedList;
import java.util.List;

import es.ucm.fdi.statistics4j.runTime.utilities.PieChartProducer;

public class MyTask extends Task {
        public MyTask(int iD) {
                super(iD);
        }

        public void execute(){

                List listValues=new LinkedList();
                List listNames=new LinkedList();
```

```
                listValues.add(new Double(27.200000000000003D));
                listNames.add("14 mm ");

                listValues.add(new Double(33.200000000000003D));
                listNames.add("10 mm ");

                listValues.add(new Double(26.200000000000003D));
                listNames.add("12 mm ");

                listValues.add(new Double(2.200000000000003D));
                listNames.add("18 mm ");

                listValues.add(new Double(18.200000000000003D));
                listNames.add("8 mm ");

                PieChartProducer pieProducer=new
                PieChartProducer(listValues,listNames,"Lenght of legs");
                this.result=pieProducer.getResult();

                storeResult("temp\\Protozooz.jpg", ResultBean.CHART);
                saveResultFinal(new String("myKey"),"ProtozoosPerCm2");

        }
}
```

En este ejemplo creamos un gráfico de tarta con las utilidades también suministradas en el paquete *utilities* y lo asignamos a nuestro resultado.

Además guardamos la dirección de otra imagen que queremos que también se muestre indicando que es un Chart y lo dejamos para guardar como dato final con clave “myKey” y título “ProtozoosPerCm2”.

Subpaquete Utilities

Se incluye un paquete con unas cuantas utilidades de creación de gráficos mediante la librería itext.

Para hacer más sencillo su uso y no tener que obligar al usuario a aprender a usar esta librería hemos creado unas clases con procedimientos simples para crear imágenes con gráficos que tras ser guardados en una localidad temporal, serán insertados en la presentación.

Para añadir estas imágenes (Charts) a la presentación sólo debemos guardar en nuestro ResultBean la dirección de donde se guardó la imagen en nuestro disco duro.

Las tres clases que hemos creado son:

En todas las clases obtenemos directamente un elemento de la clase ResultBean al llamar a su función *ResultBean getResult()*

BarChartProducer

Mediante esta clase obtenemos un diagrama de barras. El constructor de la clase es el siguiente.

BarChartProducer(List listValues, String category, List names, String _title, String value)

Donde **listValues** es la lista de los valores en orden que van a ser mostrados,
category es el nombre de las categories que se comparan,
names los nombres de esas categorías,
_title el título del gráfico,
value el nombre de la medida de los valores.

PieChartProducer

Con esta clase se obtiene un gráfico de tarta como el mostrado en la figura 3.2.2. El constructor es el siguiente:

PieChartProducer(List values, List names, String _title)

Donde **values**, es la lista ordenada de los valores a mostrar (en porcentajes). Deben ser introducidos en clases Double.

Y **names** son la lista de los nombres correspondientes a esos valores. Por lo tanto deber ser insertados en ese preciso orden.

ScatterPlotChartProducer

Con esta utilidad obtenemos gráficas como la de la figura 3.2.3. En la que podemos ver agrupaciones de valores o como se comportan los valores en una muestra grande.

ScatterPlotChartProducer(List valuesX, List valuesY, String _title)

Donde **valuesX** es la lista de valores para la X y lo mismo **valuesY**. Ambas listas de elementos deben seguir el mismo orden de inserción en sus listas y ser insertados con valores tipo Double.

Ejemplos:

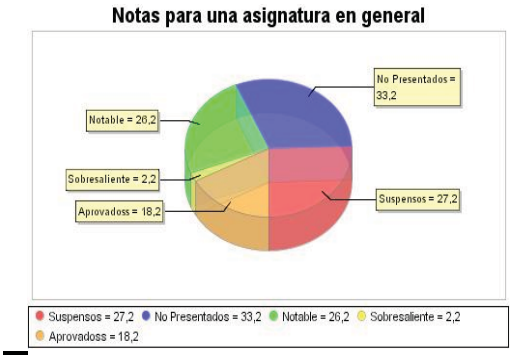


Figura 3.2.2

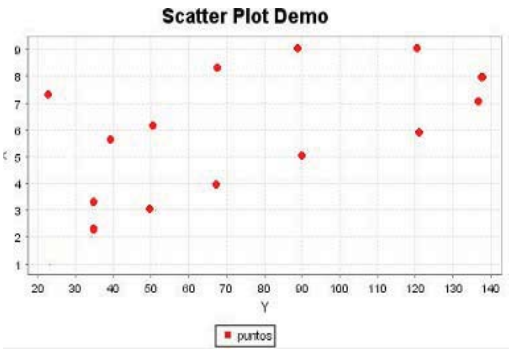


Figura 3.2.3

3.3. MÓDULO DE INTERFAZ GRÁFICA

Introducción

A continuación se mostrarán el conjunto de pantallas que componen la aplicación, así como una explicación de las mismas. Se ha intentado reducir el número de páginas lo más posible para conseguir una interfaz gráfica lo más sencilla posible, en la que el usuario, con un número mínimo de pasos sea capaz de ejecutar la aplicación.

Inicio

En la pantalla inicial que se muestra en figura 3.3.1, se disponen de cuatro opciones:

- **NEW:** Opción para dar de alta un nuevo proyecto en el sistema.
- **PROJECTS:** Opción para seleccionar uno de los proyectos ya introducidos en el sistema para posteriormente realizar las tareas deseadas sobre el mismo.
- **¿?:** Ayuda de la aplicación
- **CLOSE:** Cerrar la aplicación

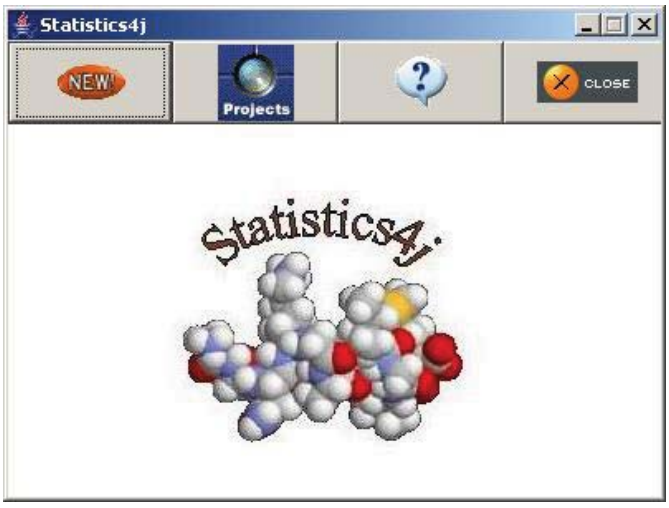


Figura 3.3.1

Nuevo Proyecto

La pantalla de inserción de un nuevo proyecto se divide en dos etapas: por una parte los datos del proyecto y los datos genéricos de las muestras que se representan y por otro lado la definición de las variables que contienen.

En la primera imagen (figura 3.3.2) se muestra la pantalla de inserción de los datos del proyecto. Al usuario se le pide que introduzca el nombre del proyecto, así como una breve descripción del mismo. También que seleccione la ruta del archivo que quiere introducir. Por último debe introducir el separador entre cada una de las muestras.

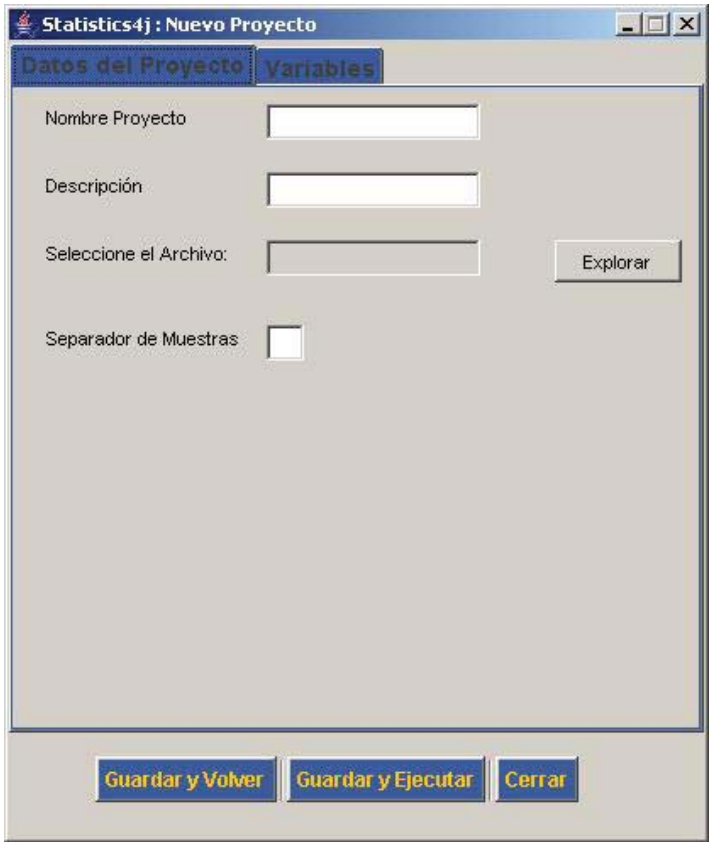


Figura 3.3.2

Y en la siguiente imagen (figura 3.3.3) se muestra la pantalla para la inserción de las diferentes variables que contienen las muestras del proyecto, así como datos específicos de la colocación de las mismas. Es decir, se le pide al usuario definir el carácter o caracteres que separan cada una de las variables, así como introducir la colocación de las variables dentro del proyecto.

Figura 3.3.3

Una vez introducidos todos los datos del proyecto el usuario tiene tres opciones:

- Guardar el proyecto en la base de datos y volver a la pantalla inicial.
- Guardar el proyecto en la base de datos y pasar a las pantallas de ejecución de tareas.
- Cancelar la operación y volver a la página inicial.

Selección de Proyecto

En esta pantalla (figura 3.3.4) al usuario se le da la opción de seleccionar uno de los proyectos previamente almacenados, para posteriormente realizar las tareas que el usuario desee sobre el mismo.

Figura 3.3.4

En el caso de pulsar continuar se cargarán todos los datos del proyecto seleccionado y se pasará a la siguiente pantalla, la de seleccionar las tareas que se quieren realizar sobre el proyecto.

En el caso de pulsar el botón cerrar, la pantalla se cerrará y se volverá a la página inicial.

Selección de tareas y nuevas tareas

Una vez que el usuario selecciona un proyecto, se cargan todos sus datos en el sistema y es el momento de seleccionar las tareas que se quieren ejecutar. Para ello se le muestra al usuario la siguiente pantalla (figura 3.3.5):

Figura 3.3.5

El usuario dispone de las siguientes opciones:

- Ir seleccionando cada una de las tareas que quiere ejecutar sobre el proyecto e ir añadiéndolas mediante la flecha dirección derecha. Del mismo modo puede rehacerse seleccionando una de las tareas seleccionadas y pulsando la flecha dirección izquierda.

- También dispone de la opción de añadir una nueva tarea para su proyecto (figura 3.3.6). Para ello pulsará sobre el botón +, momento en el que le aparecerá la siguiente pantalla:

Figura 3.3.6

En ella el usuario debe introducir el nombre de la tarea nueva que va a introducir, y posteriormente seleccionar la ruta en la que esta el archivo .java donde está implementada la función que quiere ejecutar.

Una vez pulsado el botón "continuar" el sistema compilará la clase y la tratará a partir de ese momento como una tarea más del sistema. Aquí es donde se refleja el dinamismo de la aplicación y se refleja su definición de framework.

Pulsando el botón de cerrar, no se realizará ninguna operación y se volverá a la pantalla anterior.

Dependencias entre tareas

Una vez seleccionadas las tareas que se quieren ejecutar sobre el proyecto, es el momento de introducir las dependencias entre las tareas.

Esto es debido a que se ha realizado una implementación multi-hilo para la ejecución de las tareas, de tal manera que todas las tareas se lanzarán a ejecución en el mismo momento, provocando una mejora del rendimiento de la aplicación, salvo que el usuario decida poner dependencias entre unas tareas y otras, lo que supondrá que una tarea no se lance a ejecución mientras haya otra tarea de la que dependa que no haya acabado.

La pantalla para seleccionar las dependencias se representa en la figura 3.3.7:

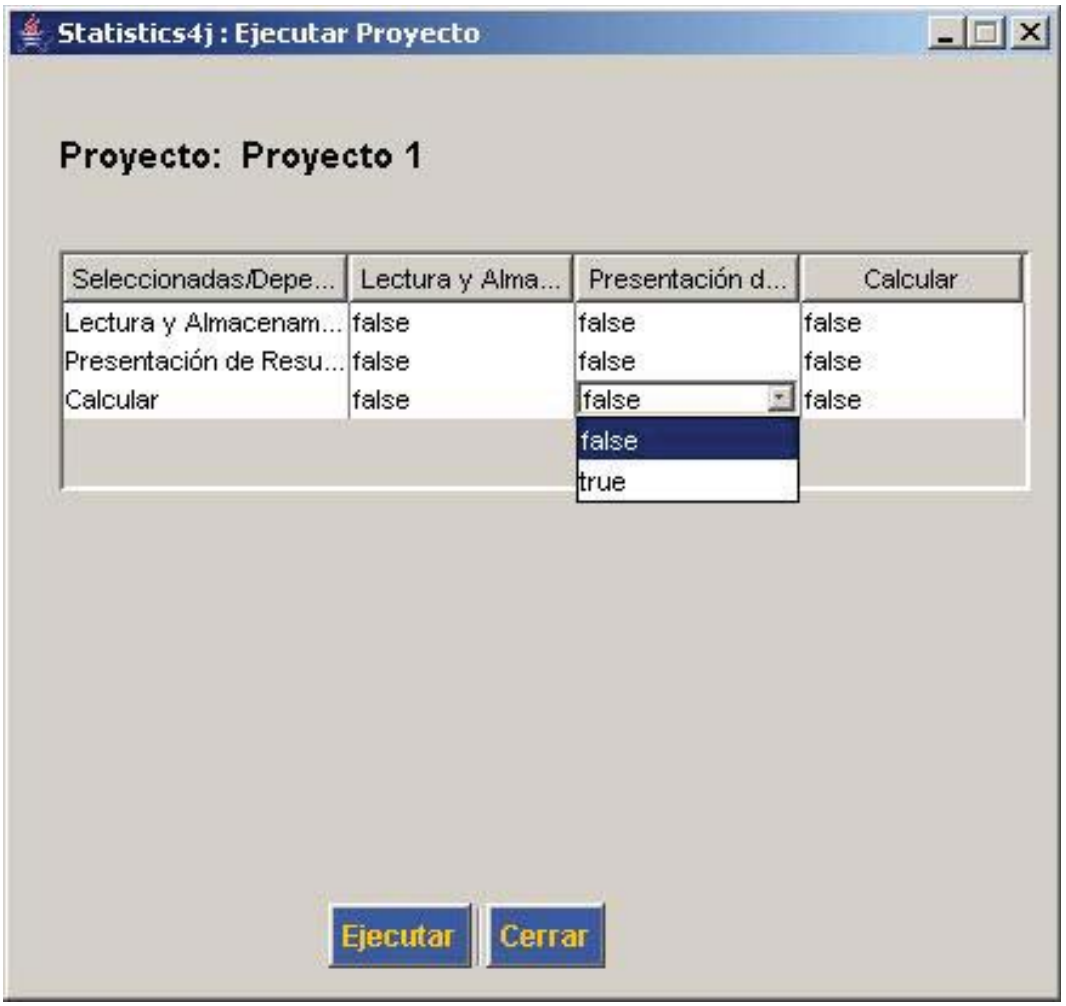


Figura 3.3.7

Una vez seleccionadas las dependencias entre tareas, se lanzará la ejecución de las mismas en el sistema.

Una vez concluidas todas las tareas se le mostrará al usuario los resultados obtenidos mediante un archivo pdf. A continuación se muestra parte de dicho pdf (figura 3.3.8). Posteriormente en este documento, en el apartado de generación de informes perteneciente al módulo de presentación se explica con más detalle como se generan los resultados y se muestran más imágenes.

En el caso en que el usuario cancele la ejecución durante la misma, las tareas que se estén realizando en ese momento dejaran de hacerlo, al igual que sus posteriores que nunca se lanzarán a ejecución.

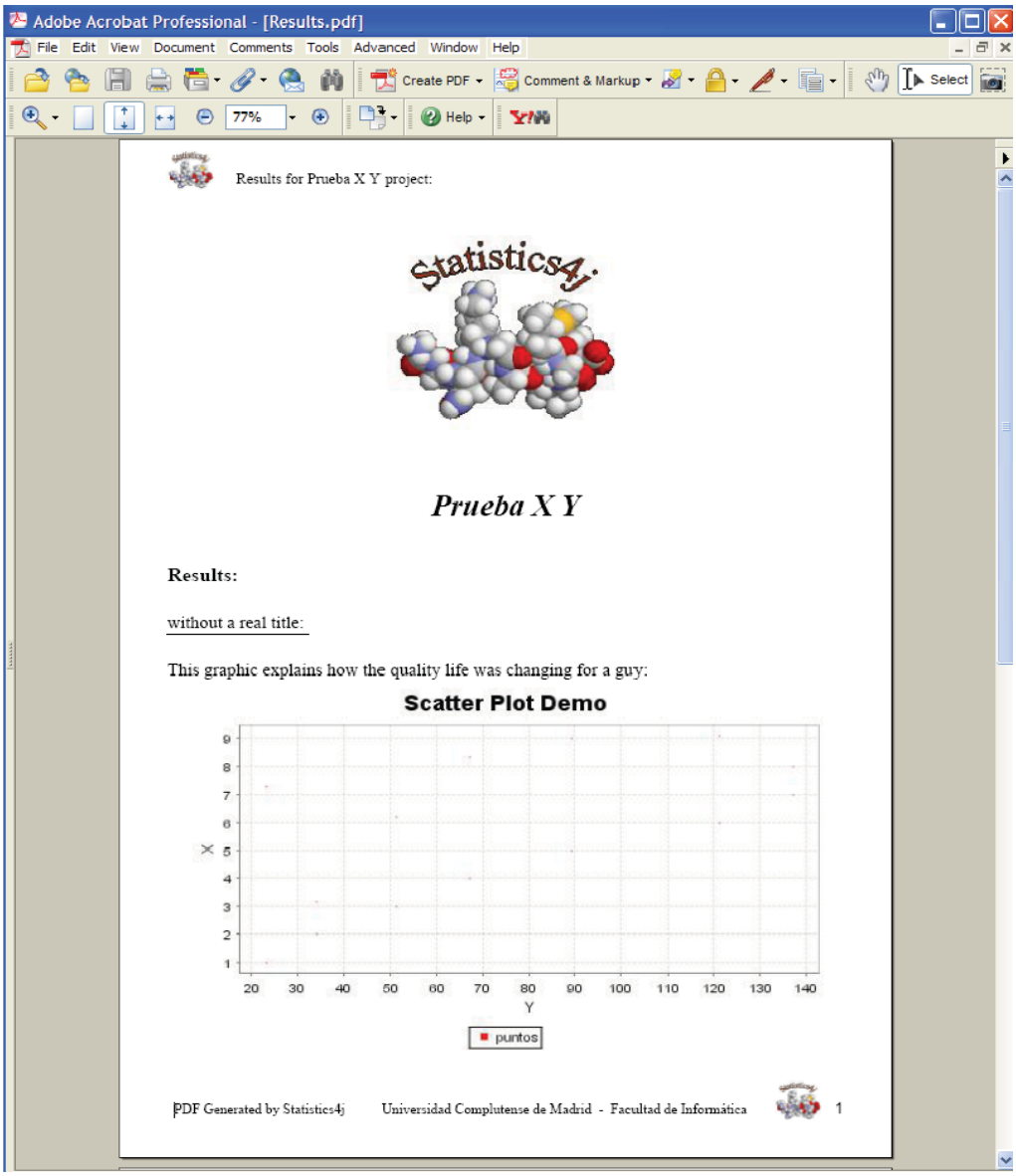


Figura 3.3.8

3.4 MÓDULO DE PERSISTENCIA

Introducción

El módulo de persistencia es el encargado del almacenamiento, tratamiento y lectura de datos en la base de datos.

Siguiendo la filosofía del proyecto, el módulo de persistencia también se ha tratado de la forma más genérica posible, dándole la oportunidad al usuario de adecuarla a sus necesidades. Este dinamismo se divide en dos grandes apartados:

- Base de datos: el usuario tiene la posibilidad de instalar statistics4j en cualquier base de datos: Oracle, mysql, access...

- Modelo de datos: el usuario dispone de dos posibilidades en cuanto al almacenamiento de los datos, dependiendo de sus gustos, necesidades, permisos que tenga en la base de datos... Estas dos posibilidades son las del modelo de datos estático y el modelo de datos dinámico.

En cuanto al primer apartado, a la hora de la instalación de la aplicación, al usuario se le da la posibilidad de seleccionar la base de datos sobre la que va a trabajar. Esta elección es irrevocable, de tal manera que para realizar un cambio del tipo de base de datos, el usuario deberá reinstalar la aplicación.

Sobre la segunda elección posible del usuario, se explica mas detalladamente en los siguientes apartados.

Modelo de datos Dinámico

Introducción

El módulo de datos dinámico es la mejor opción en cuanto a rendimiento del sistema, pero para ello se requiere que el usuario tenga todos los permisos en la base de datos. El esquema general de este modelo de datos es muy sencillo, ya que únicamente se necesitan tres tablas en la base de datos para almacenar todos los valores requeridos de cualquier proyecto. Por una parte existen dos tablas genéricas para cualquier proyecto y que almacenan, por una parte el listado de todos los proyectos que existen en el sistema y por otro las tareas que se pueden ejecutar sobre los mismos. Por último existe una tercera tabla que se genera dinámicamente

según las características del proyecto, que son introducidas por el usuario. A continuación se detallarán cada una de las tablas que se van a necesitar, explicando su significado y el de sus atributos. También se explicarán sus características, como son la clave primaria y las restricciones de integridad entre tablas.

A continuación se muestra el diagrama entidad – relación de la base de datos, así como la misma en el modelo relacional.

Tablas

Definición del conjunto de tablas que componen la base de datos dinámica:

TB_TASK

Definición

Tabla que almacena las diferentes tareas que se le ofrecen al usuario predefinidas en el sistema. En un principio estos estados serán:

- Lectura de datos.
- Cálculos básicos.
- Presentación de resultados.

Atributos

- **idTask**: índice de la tarea, identifica unívocamente a la tarea. (obligatorio)
- **name**: nombre de la tarea. (obligatorio)
- **definition**: definición de la tarea. (opcional)

Clave primaria y restricciones de integridad

- primary key: idTask

TB_PROJECT

Definición

Tabla que almacena los diferentes proyectos que están o han utilizado el sistema y el estado de su última ejecución.

Atributos

- **idProject**: índice del proyecto, identifica unívocamente al proyecto. (obligatorio)

- **name**: nombre del proyecto. (obligatorio)
- **idTask**: identificador de la última tarea que se ha efectuado sobre el proyecto. (obligatorio)
- **definition**: definición del proyecto. (opcional)

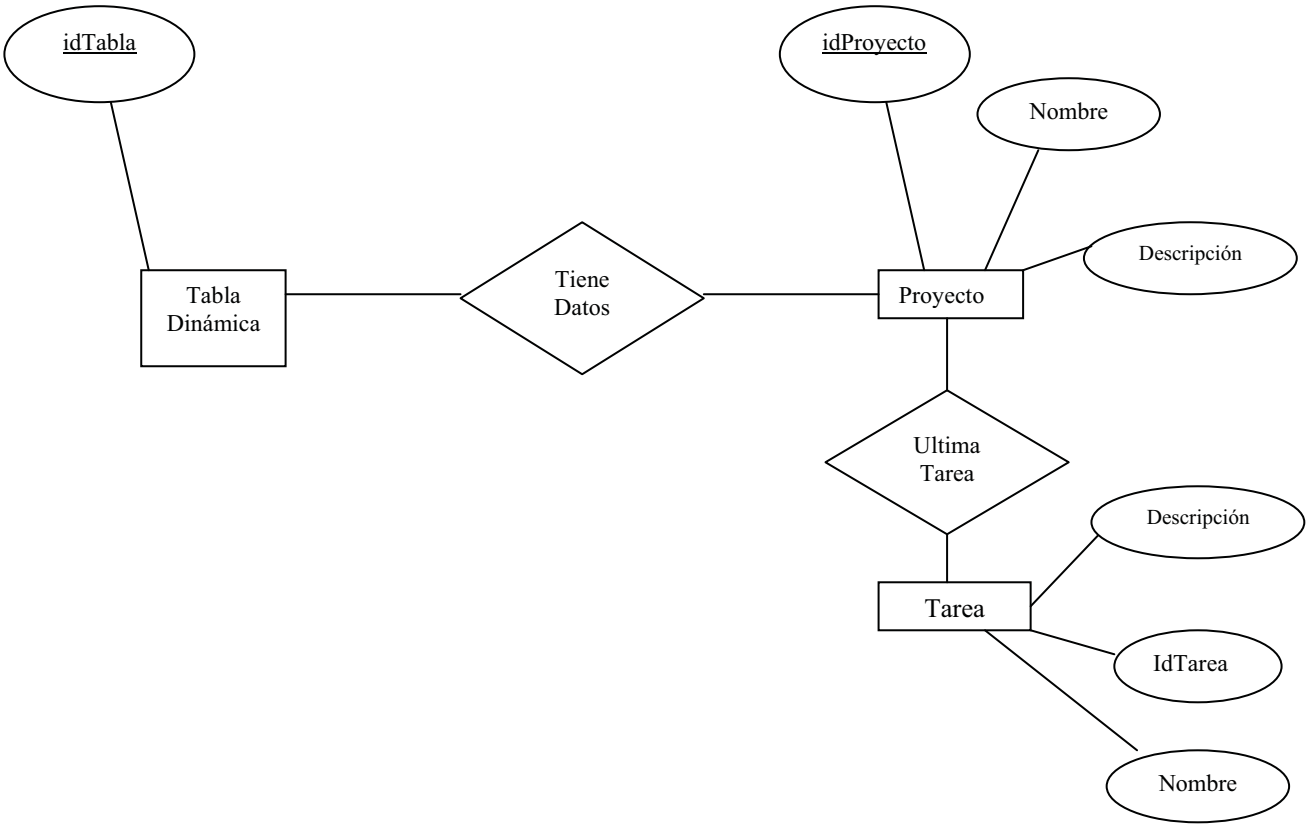
Clave primaria y restricciones de integridad

- primary key: idProject
- foreign key (idTask) references tb_task(idTask) .

TABLA DINÁMICA

Esta tabla se generará dinámicamente por cada proyecto y tendrá tantas columnas como variables, con su tipo correspondiente, y tantas filas como muestras contenga el proyecto. Su nombre será: tb_'idProject'_Sample

Diagrama Entidad - Relación



Modelo Relacional

Definición del modelo relacional de la base de datos dinámica de Statistics4j:

- Task (idTask, name, definition).
- Project (idProject, name, idTask, definition).

Modelo de datos Estático

Introducción

El modelo de datos estático es un modelo que tiene la misma funcionalidad que el modelo dinámico, pero con por eficiencia puesto que contiene un numero mayor de tablas. Este modelo esta indicado únicamente para usuarios que no tienen permisos sobre la base de datos y que necesitan un modelo que una vez instalado no haya que modificarlo. De la misma manera que el modelo de datos dinámico en primer lugar se detallarán cada una de las tablas que se van a necesitar, explicando su significado y el de sus atributos. También se explicarán sus características, como son la clave primaria y las restricciones de integridad entre tablas.

A continuación se muestra el diagrama entidad – relación de la base de datos, así como la misma en el modelo relacional.

Tablas

Definición del conjunto de tablas que componen la base de datos estática:

TB_TASK

Definición

Tabla que almacena las diferentes tareas que se le ofrecen al usuario predefinidas en el sistema. En un principio estos estados serán:

- Lectura de datos.
- Cálculos básicos.
- Presentación de resultados.

Atributos

- **idTask**: índice de la tarea, identifica unívocamente a la tarea. (obligatorio)
- **name**: nombre de la tarea. (obligatorio)
- **definition**: definición de la tarea. (opcional)

Clave primaria y restricciones de integridad

- primary key: idTask

TB_PROJECT

Definición

Tabla que almacena los diferentes proyectos que están o han utilizado el sistema y el estado de su última ejecución.

Atributos

- **idProject**: índice del proyecto, identifica unívocamente al proyecto. (obligatorio)
- **name**: nombre del proyecto. (obligatorio)
- **idTask**: identificador de la última tarea que se ha efectuado sobre el proyecto. (obligatorio)
- **definition**: definición del proyecto. (opcional)

Clave primaria y restricciones de integridad

- primary key: idProject
- foreign key (idTask) references tb_task(idTask) .

TB_SAMPLE

Definición

Tabla que almacena las diferentes muestras que aparecen en cada uno de los proyectos que están o han utilizado el sistema.

Atributos

- **idSample**: índice de la muestra, identifica unívocamente a la muestra. (obligatorio)
- **idProject**: identificador del proyecto al que pertenece. (obligatorio)

Clave primaria y restricciones de integridad

- primary key: idSample
- Foreign key (idProject) references tb_project(idProject) .

TB_VARIABLE

Definición

Tabla que almacena la instancia de las diferentes variables que van apareciendo en los diferentes proyectos.

Atributos

- **idVariable**: índice del estado, identifica unívocamente a la variable. (obligatorio)
- **name**: nombre de la variable. (obligatorio)
- **type**: tipo de la variable. (obligatorio)
- **idProject**: identificador del proyecto al que pertenece. (obligatorio)

Clave primaria y restricciones de integridad

- primary key: idVariable
- foreign key (idProject) references tb_project(idProject) .

TB_SAMPLE_VARIABLE

Definición

Tabla que almacena la relación entre las muestras y las variables de un mismo proyecto.

Atributos

- **idSampleVariable**: índice de la tabla, identifica unívocamente la relación entre la muestra y la variable. (obligatorio)
- **idSample**: identificador de la muestra donde aparece la variable. (obligatorio)
- **idVariable**: identificador de la variable. (opcional)

Clave primaria y restricciones de integridad

- primary key: idSampleVariable
- foreign key (idSample) references tb_sample(idSample)
- foreign key (idVariable) references tb_variable(idVariable)

TB_VALUE_INTEGER

Definición

Tabla que almacena los valores de las variables que tienen como tipo: Integer.

Atributos

- **idValue**: índice de la tabla, identifica unívocamente el valor. (obligatorio)
- **idSampleVariable**: identificador de la variable y del proyecto al que pertenece. (obligatorio)
- **value**: value de la variable. (obligatorio)

Clave primaria y restricciones de integridad

- primary key: idValue
- foreign key (idSampleVariable) references tb_sample_variable(idSampleVariable)

TB_VALUE_BIT

Definición

Tabla que almacena los valores de las variables que tienen como tipo: bit.

Atributos

- **idValue**: índice de la tabla, identifica unívocamente el valor. (obligatorio)
- **idSampleVariable**: identificador de la variable y del proyecto al que pertenece. (obligatorio)
- **value**: valor de la variable. (obligatorio)

Clave primaria y restricciones de integridad

- primary key: idValue
- foreign key (idSampleVariable) references tb_sample_variable(idSampleVariable)

TB_VALUE_BOOL

Definición

Tabla que almacena los valores de las variables que tienen como tipo: Bool.

Atributos

- **idValue**: índice de la tabla, identifica unívocamente el valor. (obligatorio)
- **idSampleVariable**: identificador de la variable y del proyecto al que pertenece. (obligatorio)
- **value**: valor de la variable. (obligatorio)

Clave primaria y restricciones de integridad

- primary key: idValue
- foreign key (idSampleVariable) references tb_sample_variable(idSampleVariable)

TB_VALUE_INT

Definición

Tabla que almacena los valores de las variables que tienen como tipo: int.

Atributos

- **idValue**: índice de la tabla, identifica unívocamente el valor. (obligatorio)
- **idSampleVariable**: identificador de la variable y del proyecto al que pertenece. (obligatorio)

- **value**: valor de la variable. (obligatorio)

Clave primaria y restricciones de integridad

- primary key: idValue
- foreign key (idSampleVariable) references tb_sample_variable(idSampleVariable)

TB_VALUE_REAL

Definición

Tabla que almacena los valores de las variables que tienen como tipo: real.

Atributos

- **idValue**: índice de la tabla, identifica unívocamente el valor. (obligatorio)
- **idSampleVariable**: identificador de la variable y del proyecto al que pertenece. (obligatorio)
- **value**: valor de la variable. (obligatorio)

Clave primaria y restricciones de integridad

- primary key: idValue
- foreign key (idSampleVariable) references tb_sample_variable(idSampleVariable)

TB_VALUE_DOUBLE

Definición

Tabla que almacena los valores de las variables que tienen como tipo: double.

Atributos

- **idValue**: índice de la tabla, identifica unívocamente el valor. (obligatorio)

- **idSampleVariable**: identificador de la variable y del proyecto al que pertenece. (obligatorio)

- **value**: valor de la variable. (obligatorio)

Clave primaria y restricciones de integridad

- primary key: idValue

- foreign key (idSampleVariable) references tb_sample_variable(idSampleVariable)

TB_VALUE_FLOAT

Definición

Tabla que almacena los valores de las variables que tienen como tipo: float.

Atributos

- **idValue**: índice de la tabla, identifica unívocamente el valor. (obligatorio)

- **idSampleVariable**: identificador de la variable y del proyecto al que pertenece. (obligatorio)

- **value**: valor de la variable. (obligatorio)

Clave primaria y restricciones de integridad

- primary key: idValue

- foreign key (idSampleVariable) references tb_sample_variable(idSampleVariable)

TB_VALUE_DATE

Definición

Tabla que almacena los valores de las variables que tienen como tipo: date.

Atributos

- **idValue**: índice de la tabla, identifica unívocamente el valor. (obligatorio)
- **idSampleVariable**: identificador de la variable y del proyecto al que pertenece. (obligatorio)
- **value**: valor de la variable. (obligatorio)

Clave primaria y restricciones de integridad

- primary key: idValue
- foreign key (idSampleVariable) references tb_sample_variable(idSampleVariable)

TB_VALUE_TIME

Definición

Tabla que almacena los valores de las variables que tienen como tipo: time.

Atributos

- **idValue**: índice de la tabla, identifica unívocamente el valor. (obligatorio)
- **idSampleVariable**: identificador de la variable y del proyecto al que pertenece. (obligatorio)
- **value**: valor de la variable. (obligatorio)

Clave primaria y restricciones de integridad

- primary key: idValue
- foreign key (idSampleVariable) references tb_sample_variable(idSampleVariable)

TB_VALUE_VARCHAR

Definición

Tabla que almacena los valores de las variables que tienen como tipo: varchar.

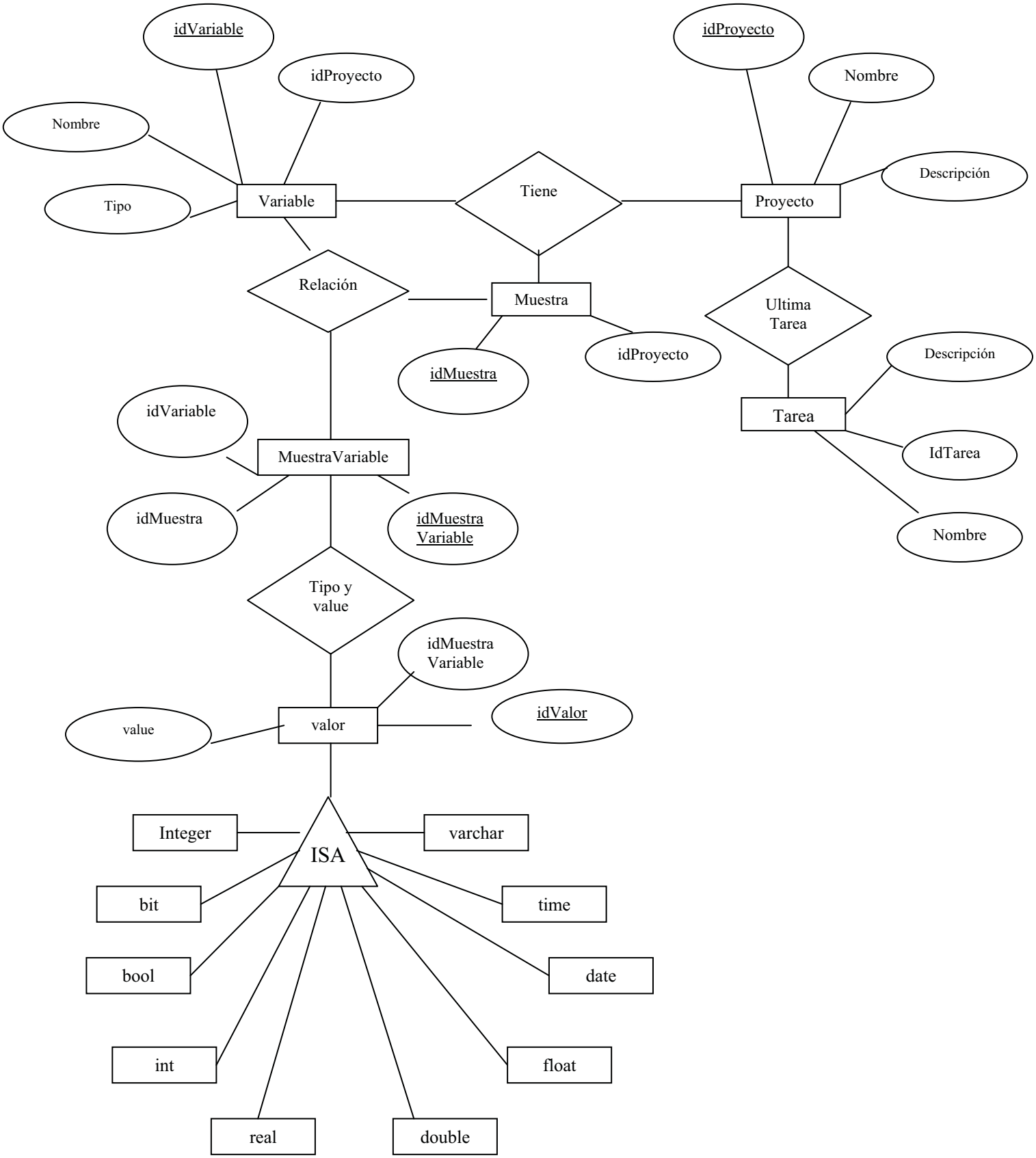
Atributos

- **idValue**: índice de la tabla, identifica unívocamente el valor. (obligatorio)
- **idSampleVariable**: identificador de la variable y del proyecto al que pertenece. (obligatorio)
- **value**: valor de la variable. (obligatorio)

Clave primaria y restricciones de integridad

- primary key: idValue
- foreign key (idSampleVariable) references tb_sample_variable(idSampleVariable)

Diagrama Entidad - Relación



Modelo Relacional

Definición del modelo relacional de la base de datos estática de Statistics4j:

- Task (idTask, name, definition).
- Project (idProject, name, idTask, definition).
- Sample (idSample, idProject).
- Variable (idVariable, name, tipo, idProject).
- Sample_Variable(idSampleVariable, idSample, idVariable).
- Value_Integer(idValue, idSampleVariable, value).
- Value_bit(idValue, idSampleVariable, value).
- Value_bool(idValue, idSampleVariable, value).
- Value_int(idValue, idSampleVariable, value).
- Value_real(idValue, idSampleVariable, value).
- Value_double(idValue, idSampleVariable, value).
- Value_float(idValue, idSampleVariable, value).
- Value_date(idValue, idSampleVariable, value).
- Value_time(idValue, idSampleVariable, value).
- Value_varchar(idValue, idSampleVariable, value).

Implementación

La implementación del módulo de persistencia se ha dividido en dos paquetes, por un lado las clases necesarias para implementar las funciones correspondientes al apartado de almacenamiento de datos sobre el proyecto que se esta ejecutando sobre la aplicación y por otro lado las clases necesarias que realizan las funciones del sistema.

El paquete relacionado con el proyecto dejó de tener funcionalidad en la última versión de la aplicación puesto que para la comunicación con la base de datos por parte del sistema, es decir, lectura de datos, almacenamiento, modificación... se utilizo el mapeador relacional: hibernate.

Dentro de cada uno de los paquetes, las clases se dividen en dos grupos: por un lado las clases que implementan los métodos sobre el modelo de datos estático y otro sobre el modelo de datos dinámico.

A continuación se muestra una imagen (figura 3.4.1) del directorio de clases correspondiente al módulo de persistencia:

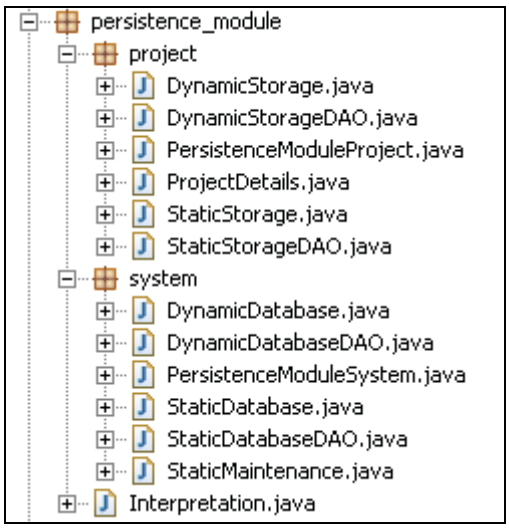


Figura 3.4.1

La clase **Interpretation.java** se encarga de, dada una frase, y un separador de palabras, devuelve un vector con todas las palabras que componen esa frase. Esta clase es útil cuando debemos separar todas las variables dentro de una muestra.

Puesto que el paquete project y sus clases se quedaron obsoletas al introducir hibernate, únicamente se detallaran las clases del paquete system:

- PersistenceModuleSystem.java: Se encarga de distinguir con que base de datos estamos trabajando, es decir, si con la estática o con la dinámica y enviar el trabajo a la clase correspondiente: Dynamicdatabase.java o StaticDatabase.java

- DynamicDatabase.java: Se encarga de llevar a cabo las funciones correspondientes a la base de datos dinámica que necesita el sistema, llamando a las funciones necesarias que conectan con la base de datos y que están implementadas en la clase DynamicDatabaseDAO.java

- StaticDatabase.java: Se encarga de llevar a cabo las funciones correspondientes a la base de datos estática que necesita el sistema, llamando a las funciones necesarias que conectan con la base de datos y que están implementadas en la clase StaticDatabaseDAO.java

Pruebas de Carga

A continuación mostramos en una tabla los resultados de tiempos de ejecución de la tarea de almacenamiento de datos a partir de un proyecto dado:

Las pruebas se han realizado sobre un Pentium IV 2.6 y con 1 GB de memoria RAM:

Número de muestras	Número de variables por muestra	Tiempo en modelo datos Dinámico	Tiempo en modelo datos Estático
200	2	3’ 17’’	26’ 47’’
10.000	2	15’ 33’’	1h 52’
1.000.000	2	18 h apx	No evaluado

3.5. MODULO DE PRESENTACION

Introducción

El módulo de presentación como su propio nombre indica es el encargado de hacer las operaciones que sean necesarias para mostrar o guardar (dependiendo de la implementación del módulo) los resultados obtenidos después de el lanzamiento del módulo de lectura y la finalización de las tareas que se realizaron sobre los datos.

Al módulo de presentación que se haya implementado se le pasará desde el módulo de ejecución, el Run-Time un vector de elementos de tipo ResultBean con los que trabajar.

Hay diferentes modos de hacer la presentación:

- Simplemente dejando archivos de gráficas
- Añadiendo los valores a una pagina Excel
- Sacándolos por consola
- Guardándolos a base de datos
- ……

Nosotros en nuestra implementación por defecto hemos incluido un módulo de presentación que guarda los resultados produciendo un archivo PDF dinámicamente mediante la librería gratuita IText.

¿Por qué usar IText?

Sería bastante sencillo guardar todos los datos obtenidos en un archivo normal y corriente o incluso en base de datos. En cambio nos resultaba mucho más interesante la generación dinámica de PDF's que por otro lado es el modo más común en los últimos tiempos para el archivo e intercambio de informes.

La herramienta IText es una librería gratuita, aunque debido a que es relativamente nueva la información sobre su uso se limita a la suministrada por sus creadores y algunas opciones no están bien documentadas del todo o aún esa documentación no está disponible.

Clases del paquete Presentation

Como se puede observar en la Figura 3.5.1 en la distribución que hemos hecho contamos con 3 clases dentro del paquete **presentation** que en seguida pasamos a describir:

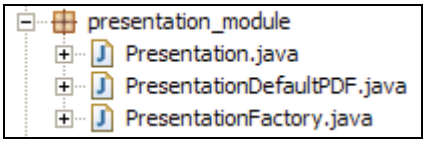


Figura 3.5.1

Interfaz Presentation:

Es la interfaz que deben cumplir las diferentes versiones del paquete de presentación que el usuario final o el programador que quiera un nuevo módulo con una funcionalidad distinta debe implementar.

Sólo contiene la siguiente función:

public void developePresentation(ResultList resultList);

Éste es el método que será invocado por el módulo de ejecución para lanzar la presentación.

Como parámetro se le pasa un ResultList cuya explicación y detalles podemos encontrar en la parte dedicada al RunTime en esta memoria pero que en resumen es una lista de elementos del tipo ResultBean que son creados como resultado de distintas operaciones (Task) realizadas por la aplicación. Estos elementos del tipo ResultBean (ver módulo run-time para mayor información) contienen la información a ser mostrada por éste módulo, por lo que debemos tener en cuenta su implementación.

Factoria PresentationFactory:

En nuestro intento por hacer todo lo más genérico posible queremos que no haya ningún vínculo entre el módulo de ejecución y el de presentación. El módulo de ejecución sólo tendrá acceso al método de esta factoría para crear una instancia que implemente la interfaz Presentation quedando al margen de su constructor o su implementación.

El único método de esta factoría será:

static public Presentation getPresentationInstance()

que devolverá un objeto que implementa la interfaz Presentation. El usuario que quiera cambiar el módulo de presentación que se lance, debe pues, cambiar la implementación de la función de la factoría, apuntando a la clase que él haya desarrollado.

Clase PresentationDefaultPDF:

Esta es nuestra propia implementación para la presentación de la aplicación. Crea un archivo PDF dinámicamente con los datos obtenidos de ResultBeanList.

Usa la librería de generación de PDF's iText, que permite la creación de PDF asi como la unión o modificación de campos de formularios en PDF's ya creados. En nuestro PDF introducimos gráficas (Charts) como imágenes, introducimos valores con sus explicaciones y todo va insertado en un PDF con nuestro logotipo, pie y cabecera.

El método más fácil para observar su funcionamiento no es otro que ver el código directamente aunque sería muy largo el copiar aquí la clase, por lo que sólo veremos un trozo del código.

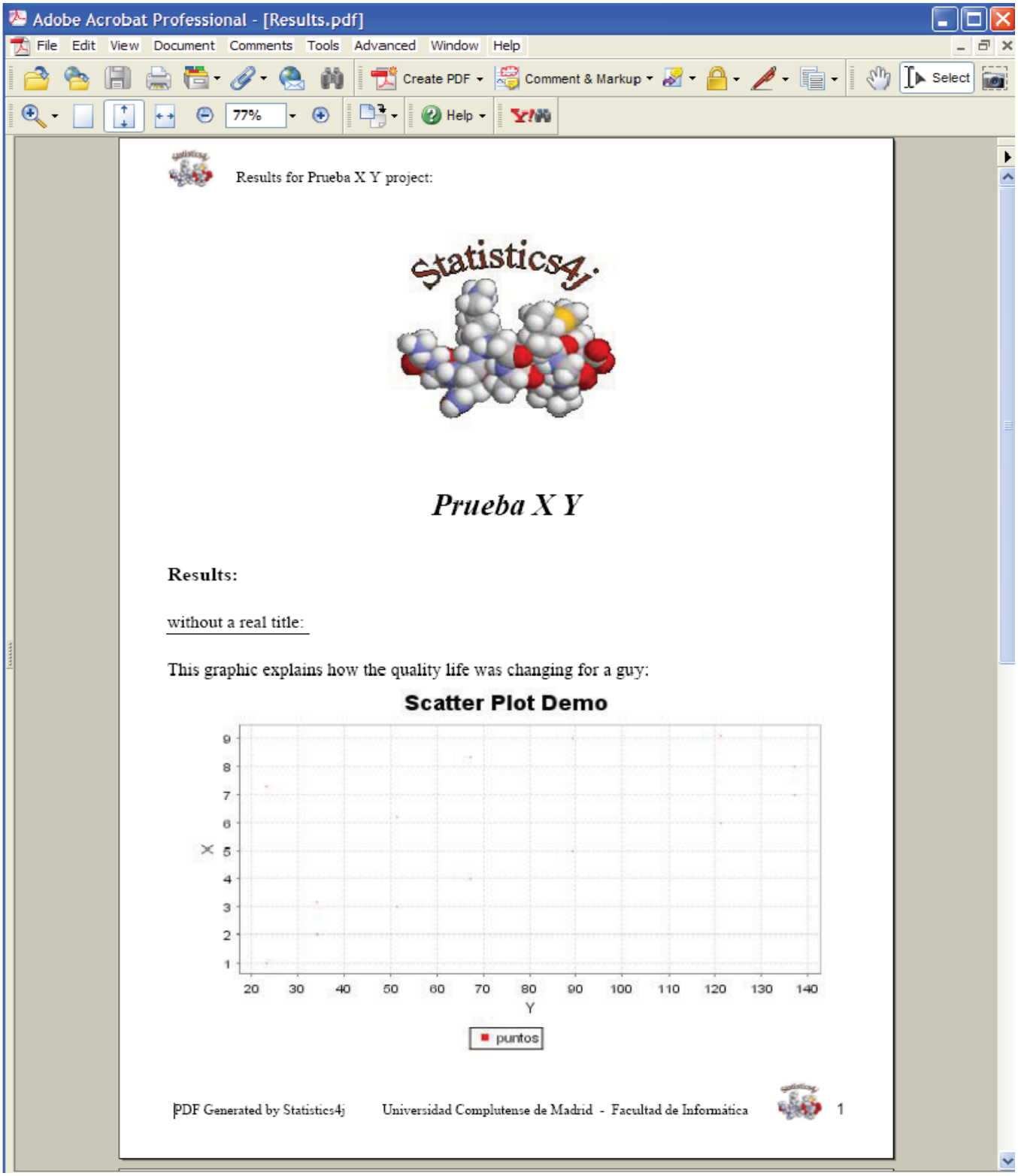
```
public PresentationDefaultPDF() {
      this.document = new Document();

       try {
          File finle = new File("statistics_respository\\" +
Project.getInstance().getName());
          finle.mkdirs();
                              Creo el archivo donde será guardado
          PdfWriter.getInstance(this.document, new FileOutputStream(
                "statistics_respository\\" + Project.getInstance().getName()
                      + "\\Results.pdf"));

                                    Cargo el logotipo que usaré tanto al principio
como en la cabecera y pie.

          this.imagen = Image.getInstance("img\\logotipo.jpg");
          this.imagen.scalePercent(20);

                              Creo la cabecera que estará en la parte de arriba de
cada página

          Chunk ck = new Chunk(this.imagen, 0, -5);
```

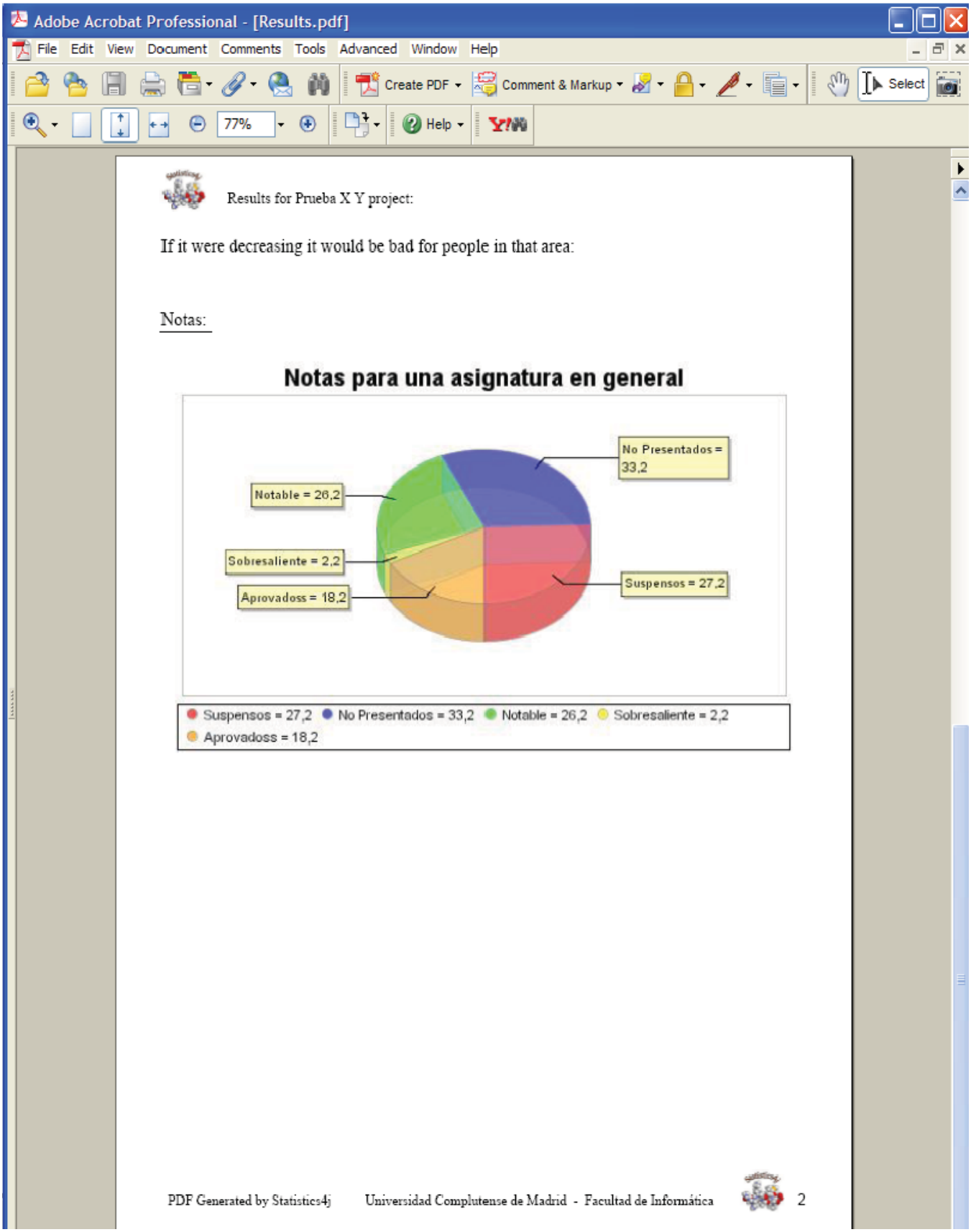
```
        this.document.addTitle("Document generated by Statistics4j");

        Phrase phraseHeader = new Phrase();
        Phrase phraseFooter = new Phrase();
        phraseHeader.add(ck);
        phraseHeader.add("     ");
        Paragraph paragr=new Paragraph("Results for " +
Project.getInstance().getName()
              + " project: ",new Font(Font.TIMES_ROMAN,12 ));
        phraseHeader.add(paragr);
        HeaderFooter headf = new HeaderFooter(phraseHeader, false);
        headf.setBorderWidthBottom(0);
        headf.setBorderWidth(0);
        headf.setAlignment(Element.ALIGN_LEFT);
        this.document.setHeader(headf)
;
```

Creo el pie de página

```
       Chunk frase2Chunk=new Chunk("Universidad Complutense de Madrid  -
Facultad de       Informática   ",new Font(Font.TIMES_ROMAN,11));
        phraseFooter.add(new Paragraph("PDF Generated by Statistics4j
",new Font(Font.TIMES_ROMAN,11)));

        phraseFooter.add(frase2Chunk);
        phraseFooter.add("    ");
        phraseFooter.add(ck);
        phraseFooter.add("   ");

        HeaderFooter head = new HeaderFooter(phraseFooter, true);
        head.setBorderWidthBottom(0);
        head.setBorderWidth(0);
        head.setAlignment(Element.ALIGN_RIGHT);
        this.document.setFooter(head);

     } catch (Exception ex) {
        System.err.println(ex.getMessage());
     }
```

Ejemplo de archivo .pdf generado por la aplicación:




Adobe Acrobat Professional - [Results.pdf]
File Edit View Document Comments Tools Advanced Window Help
Create PDF
Comment & Markup
Select
77%
Help
Results for Prueba X Y project:
If it were decreasing it would be bad for people in that area:
Notas:
Notas para una asignatura en general
No Presentados = 33,2
Notable = 26,2
Sobresaliente = 2,2
Aprovadoss = 18,2
Suspensos = 27,2
Suspensos = 27,2
No Presentados = 33,2
Notable = 26,2
Sobresaliente = 2,2
Aprovadoss = 18,2
PDF Generated by Statistics4j
Universidad Complutense de Madrid - Facultad de Informática
2

4. LOGS Y MULTILENGUAJE

4.1 Logs

Se ha utilizado en la aplicación un sistema de logs para poder reflejar los diferentes hechos que ocurren tanto durante su desarrollo como durante su ejecución. Estos mensajes pueden ser de varios tipos, según el mensaje que quieran expresar: pueden ser simplemente mensajes informativos, indicando las funciones que se han ejecutado o como ayuda para el desarrollo de la aplicación plasmando las diferentes advertencias o errores que se generan.

El formato de los mensajes que queremos que se muestren los decidimos nosotros, mediante un xml que indica como queremos que se muestren los mensajes, donde queremos que se almacenen los mensajes, así como indicar los tipos de mensaje que queremos que se generen. Dicho xml es el siguiente:

```
<appender name="Fichero" class="org.apache.log4j.DailyRollingFileAppender">
        <param name="File"   value="logs/statistics4j.log" />
        <param name="Append" value="true" />
        <param name="DatePattern" value="'.'yyyy-MM-dd-HH" />
        <layout class="org.apache.log4j.PatternLayout">
                <param name="ConversionPattern" value="%-5p [%d{ISO8601}]
(%M [%C:%L]) - %m%n"/>
        </layout>
    </appender>

    <appender name="Consola" class="org.apache.log4j.ConsoleAppender">
          <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern"
                value="%-5p [%d{ISO8601}] (%M [%C:%L]) - %m%n"/>
          </layout>
    </appender>

    <root>
      <priority value ="debug" />
      <appender-ref ref="Fichero" />
    </root>

</log4j:configuration>
```

El formato que hemos elegido es el de mensajes que se guardan en archivos de texto, de los cuales se crea una copia de seguridad cada día, de tal manera que el nombre de los ficheros es: statistics4j.log.(fecha).txt.

Cada vez que se quiere generar un mensaje, en el punto del código correspondiente es necesario poner las siguientes instrucciónes:

Por una parte declarar en la cabecera de la clase la variable estática:

```
//Se inicializan los Logs
static Category log =category.getInstance(Interpretation.class.getName());
```

Y para generar un mensaje se utiliza la variable **log.tipoMensaje ("mensaje")**

Ejemplos:

- log.info ("esto es un ejemplo de log de información");
- log.error ("esto es un ejemplo de log de error");

A continuación se muestra un fragmento de uno de los archivos de logs que se generan:

```
INFO  [2005-06-12 18:27:25,937] (<clinit>
[net.sf.hibernate.cfg.Environment:478]) - Hibernate 2.1.7
INFO  [2005-06-12 18:27:25,937] (<clinit>
[net.sf.hibernate.cfg.Environment:507]) - hibernate.properties not found
INFO  [2005-06-12 18:27:25,953] (<clinit>
[net.sf.hibernate.cfg.Environment:538]) - using CGLIB reflection optimizer
INFO  [2005-06-12 18:27:25,953] (<clinit>
[net.sf.hibernate.cfg.Environment:567]) - using JDK 1.4 java.sql.Timestamp
handling
INFO  [2005-06-12 18:27:25,953] (configure
[net.sf.hibernate.cfg.Configuration:934]) - configuring from file:
hibernate.cfg.xml
DEBUG [2005-06-12 18:27:26,015] (resolveEntity
[net.sf.hibernate.util.DTDEntityResolver:20]) - trying to locate
http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd in classpath
under net/sf/hibernate/
DEBUG [2005-06-12 18:27:26,031] (resolveEntity
[net.sf.hibernate.util.DTDEntityResolver:29]) - found
http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd in classpath
DEBUG [2005-06-12 18:27:26,062] (addProperties
[net.sf.hibernate.cfg.Configuration:858]) -
hibernate.connection.url=jdbc:mysql://localhost/statistics4j
DEBUG [2005-06-12 18:27:26,062] (addProperties
[net.sf.hibernate.cfg.Configuration:858]) -
hibernate.connection.driver_class=org.gjt.mm.mysql.Driver
```

```
DEBUG [2005-06-12 18:27:26,062] (addProperties
[net.sf.hibernate.cfg.Configuration:858]) - hibernate.connection.username=root
DEBUG [2005-06-12 18:27:26,062] (addProperties
[net.sf.hibernate.cfg.Configuration:858]) - hibernate.connection.password=root
DEBUG [2005-06-12 18:27:26,078] (addProperties
[net.sf.hibernate.cfg.Configuration:858]) -
dialect=net.sf.hibernate.dialect.MySQLDialect
DEBUG [2005-06-12 18:27:26,078] (addProperties
[net.sf.hibernate.cfg.Configuration:858]) - hibernate.show_sql=false
DEBUG [2005-06-12 18:27:26,078] (addProperties
[net.sf.hibernate.cfg.Configuration:858]) - hibernate.use_outer_join=true
DEBUG [2005-06-12 18:27:26,078] (addProperties
[net.sf.hibernate.cfg.Configuration:858]) -
hibernate.transaction.factory_class=net.sf.hibernate.transaction.JDBCTransactio
nFactory
DEBUG [2005-06-12 18:27:26,078] (doConfigure
[net.sf.hibernate.cfg.Configuration:1017]) - null<-
org.dom4j.tree.DefaultAttribute@1408a92 [Attribute: name resource value
"es/ucm/fdi/statistics4j/hibernate/classes/model/beans/dynamics/Sample1.hbm"]
```

4.2. Multilenguaje

El sistema está preparado para ser multilenguaje. Para todos los mensajes que se generan en la aplicación como pueden ser logs, excepciones... así como todos los textos que aparecen en la interfaz gráfica se utiliza un sistema de properties. De tal manera que todos los textos vienen reflejados en diversos archivos de texto, cuyo nombre es “nombreArchivoIdioma.properties”. De tal manera que cuando se quiere escribir un cualquier mensaje se le solicita al archivo correspondiente, según el idioma, que devuelva el mensaje.

A continuación se muestra un ejemplo, en el que se escribe un log según el idioma elegido por el usuario:

- Dispondremos de archivos properties, tantos como idiomas soporte la aplicación. Es decir, suponiendo que solo se admitan español e ingles, deberemos disponer de dos archivos de texto: logsES.properties y logsEN.properties. Estos archivos de texto son de la forma:

logsEN.properties.txt:

```
#Logs of the persistence module

logs.statistics4j.createDBStart = Start of the Data Base
creation.
logs.statistics4j.createDBEnd    =  End of the Data Base
Creation.
```

Donde la parte izquierda del símbolo = representa el código del texto y la parte derecha el texto que se quiera mostrar. El archivo en español seria:

logsES.properties.txt:

```
#Logs del modulo de persistencia

logs.statistics4j.createDBStart = Comienzo de creacion de la
base datos
logs.statistics4j.createDBEnd    =  Fin de creacion de la base
de datos
```

- Antes de generar un mensaje, debemos inicializar el sistema de properties. Esto es unicamente necesario al arrancar la aplicación o cuando el usuario modifica el idioma en el que quiere que aparezcan los textos:

```
try {
        FileInputStream propFile = new FileInputStream(
 Dirección donde estan los ficheros properties + logs + LANGUAGE);
        this.properties = new Properties();
        this.properties.load(propFile);
        propFile.close();
     } catch (Exception ex) {
 }
```

La constante **LANGUAGE** almacena el String correspondiente al idioma predefinido por el usuario.

Para solo realizar esta inicialización al comienzo de la aplicación introducimos este código en el constructor de una clase que contenga un atributo estatico de ella misma, de esa manera solo necesitaremos obtener una instacia de esa clase y utilizarla.

El ejemplo de creación de log seria:

```
log.info(this.logsProperties
        .getProperty("logs.statistics4j.createDBStart"));
```

5. SAX y JDOM

5.1. Introducción

El lenguaje *XML* es una parte muy importante de nuestro proyecto, aporta una gran facilidad a la hora de almacenar configuraciones de manera persistente, para poder leerlas posteriormente. Si recordamos como funciona *statistics4j*, nos daremos cuenta de que tanto la configuración del sistema como los datos de entrada, están recogidos en un fichero *XML*. Además este tipo de ficheros también realizan una labor importante en la generación dinámica de clases y ficheros de otros tipos (en particular para ficheros de configuración y mapeo de *hibernate*, como ya veremos más adelante), ya que la herramienta de generación de código por plantillas que utilizamos (*velocity*), necesita del API *SAX* para leer los datos básicos a partir de los cuales generar los archivos pertinentes. Vamos a explicar más detalladamente que papel juegan estas dos herramientas de manipulación de archivos *xml*, y su papel dentro de *statistics4j*.

5.2. El API SAX

SAX define un API para un analizador de archivos *XML* basado en eventos. Estar "basado en eventos" significa que el analizador lee un documento *XML* desde el principio hasta el final, y cada vez que reconoce una sintaxis de construcción, se lo notifica a la aplicación que lo está ejecutando. SAX notifica a la aplicación llamando a los métodos del interface `ContentHandler`. Por ejemplo, cuando el analizador encuentra un símbolo ("<"), llama al método `startElement`; cuando encuentra caracteres de datos, llama al método `characters`; y cuando encuentra un símbolo ("</"), llama al método `endElement`, etc.

¿Por qué SAX?

Cuando comenzamos a pensar en nuestro proyecto, se nos planteo la duda de que API utilizar para la lectura de *XML's*, teníamos dos opciones: SAX o DOM. DOM es un conjunto de interfaces para construir una representación de objeto, en forma de árbol, de un documento XML analizado, al ser más complicado que SAX, y debido a que los documentos de configuración que íbamos a crear eran sencillos y no muy extensos, nos decidimos por este último.

SAX en Statistics4j

Nuestra aplicación permite realizar todo el proceso de lectura y análisis de datos y presentación del resultado, tanto a la vez como por separado. Para esta segunda opción contamos con la división del proceso en tareas. Estas tareas las define el usuario cuando instala la aplicación, se almacenan en el documento de configuración de sistema, y para poder recuperarlas se utiliza *SAX*. Recordemos que también la información de la configuración de la base de datos está en este documento.

A continuación mostramos una imagen del paquete en el que está implementada la lectura del documento de configuración del sistema mediante *SAX*:

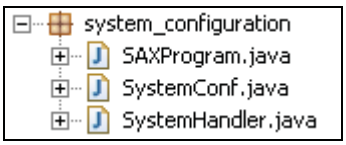


Figura 5.2.1

Ahora explicamos el contenido de cada clase y para que lo utilizamos:

- **SystemConf:** Es una clase auxiliar en la que almacenaremos todos los datos que saquemos de la lectura del *xml*. La creamos simplemente para no tener que declarar en la clase correspondiente al manejador (*SystemHandler*), todos los atributos, y sus métodos de acceso y modificación, ya que esto aumentaría el número de líneas de código de dicha clase y dificultaría su manipulación.

- **SystemHandler:** Esta clase es la correspondiente al manejador de SAX, es decir que hereda de la clase *ContentHandler*. Contiene un atributo de tipo *SystemConf* en el cual almacenamos los valores que vamos recogiendo del documento de configuración. Aquí se implementan las acciones a realizar cada vez que se llama a los eventos de la clase *ContentHandler*. En particular implementa los métodos:

  - **startElement**: Se le llama cuando encuentra la etiqueta de inicio de un elemento. Aquí tenemos el código:

```
public void startElement(String space, String localName,
String completeName, Attributes atrs) {
        if (localName.equals("dataBase")) {
           this.element="dataBase";
        }
```

```
            else if (localName.equals("taskName")) {
               this.element="taskName";
            } else if (localName.equals("taskDescription")){
               this.element="taskDescription";
            }
            for (int i=0; i<atrs.getLength(); i++) {
              if (atrs.getLocalName(i).equals("isDynamic")) {
                  Boolean b=new Boolean(atrs.getValue(i));
                       this.system.setIsDynamic(b.booleanValue());
              } else if(atrs.getLocalName(i).equals("name")){
                    this.system.setName(atrs.getValue(i));
              } else if(atrs.getLocalName(i).equals("user")){
                    this.system.setUser(atrs.getValue(i));
              } else if (atrs.getLocalName(i).equals("ip")) {
                    this.system.setIp(atrs.getValue(i));
              } else if (atrs.getLocalName(i).equals("pwd")){
                     this.system.setPassword(atrs.getValue(i));
        }
    }
```

Lo que hacemos en este método es almacenar en el atributo *element* que elemento nos hemos encontrado (valor que está en *localName*). Después se recorren todos los atributos y se leen sus valores (se accede a estos valores con el método *getLocalName( )*).

- **endElement**: Se le llama cuando encuentra la etiqueta de fin de un elemento. Aquí tenemos el código:

```
public void endElement(String space, String localName,
                       String completeName) {
        if (localName.equals("task")) {
for (int i=0; this.names.size(); i++) {
this.system.addList(this.names.get(i));
this.system.addList(this.descriptions.
get(i));
            }
        }
}
```

Lo que hacemos es simplemente controlar el caso de las tareas, que cada vez que se cierre un elemento tarea, éste se almacene en la lista de tareas que tiene el atributo system.

- **characters**: Se le llama cuando se encuentra el texto de un elemento (contenido). Aquí tenemos el código:

```
public void characters(char[] ch, int begin, int end) {
  String s = new String(ch, begin, end);
     if (this.element.equals("stateName")) {
    this.system.addName(s);
} else if(this.element.equals("stateDescription")) {
  this.system.addDescription(s);
     }
}
```

En este método almacenamos el texto del elemento en que nos encontramos en el lugar correspondiente. Sabemos de quien es el texto en el que nos encontramos por el valor que hemos asignado a *element* en el método *startElement.* En nuestro caso particular, sólo los elementos correspondientes a los nombres y descripciones de las tareas tienen texto, por eso observamos en el código que los valores de estos textos se almacenan en sendas listas, una de descripciones y otra de nombres, ambas atributos a su vez del atributo *system.*

- **SAXProgram:** En esta clase está el método que realiza la operación completa de lectura del archivo de configuración. A continuación explicamos las líneas de código importantes de este método, quitando el tratamiento de excepciones y logs:

```
String sistFile = Statistics4jConstants.XML_SYSTEM_PATH;
SystemHandler systemHandler = null;
SAXParser parser;
SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setNamespaceAware(true);
factory.setValidating(true);
parser = factory.newSAXParser();
systemHandler = new SystemHandler();
parser.parse(sistFile, systemHandler);
```

Se crea un *parser* utilizando la factoría de *SAX*. Se habilitan las opciones de respetar los espacios entre palabras y de utilizar validación con *dtd*, se crea un objeto de la clase que hemos visto antes *SystemHandler*, y con esta clase y la ruta donde está el archivo a analizar hacemos una llamada a *parse*. Esta llamada almacena en el objeto *system* de *systemHandler* los datos del documento de configuración del sistema.

Con esto hemos finalizado la explicación de la lectura con el API *SAX* del documento de configuración de sistema, pero recordemos que existe otro documento *xml* de configuración de proyecto, que se genera dinámicamente dependiendo del archivo de datos introducido. Para este documento se realiza exactamente el mismo proceso, son las mismas clases con nombres parecidos y que realizan las mismas acciones, sólo cambia ligeramente el código, debido principalmente a que el *xml* es de estructura diferente. Mostramos el paquete con las clases, xo no detallamos el contenido de cada una de las clases porque es prácticamente el mismo que el de las clases el paquete de la figura 5.2.2.

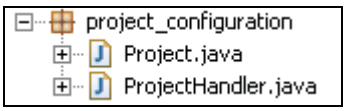


Figura 5.2.2

Observamos que falta la clases *SAXProgram,* eso es debido a que no vimos la necesidad de crear dos clases, así que simplemente añadimos el método de análisis del documento de configuración del proyecto al mismo *SAXProgram* del paquete *system_configuration,* así como un objeto de la clase *Project* (ver figura 8.2), que equivaldría a la clase *SystemConf* del caso de la configuración del sistema.

5.3. El API JDOM

JDOM es un API para leer, crear y manipular documentos XML de una manera sencilla y muy intuitiva para cualquier programador en Java, en contra de otras APIs tales como DOM y SAX, las cuales se idearon sin pensar en ningún lenguaje en concreto, de ahí que resulte un poco incomoda su utilización. Precisamente esta comodidad que ofrece JDOM frente a SAX o DOM, es la que nos ha llevado a utilizarlo en nuestro proyecto.

El *xml* se forma con objetos *Element*, que representan a los elementos de un *xml* (*cada tag* que se abre y se cierra es un elemento). Para añadir un elemento dentro de otro elemento se utiliza el método *addContent(Element element)*. Para añadir un atributo a un elemento se utiliza el método *setAttribute(Attribute atributte)*, de donde podemos deducir que la clase *Attribute* representa a un atributo

de un elemento del *xml*. Por último tenemos el método *setText(String text)*, que está en la clase *Element* y sirve para añadir un texto al elemento. Estos son los métodos y clases básicas, con ellos vamos creando una especie de árbol de elementos que representa el documento, y que se convierte fácilmente en un Archivo *xml.*

En *statistics4j* utilizamos este API sólo para generar el XML de configuración del proyecto. El código que realiza esta acción lo mostramos a continuación, así servirá para entender mejor como funciona JDOM:

```
public static String createXML() {


String param = "";
try {
    CurrentProject project = CurrentProject.getInstance();
    Element rootElem = new Element("project");
    DocType docType = new DocType("project", "XMLConf.dtd");
    Document doc = new Document(rootElem, docType);
    Element root = new Element("root");
    root.setText(project.getRoot());
    rootElem.addContent(root);
    Element name = new Element("name");
    Attribute att = new Attribute("isNew", project.getIsNew()
                    .toString());
    name.setAttribute(att);
    name.setText(project.getName());
    rootElem.addContent(name);
    Element desc = new Element("description");
    desc.setText(project.getDescription());
    rootElem.addContent(desc);
    Element samp = new Element("samplesSeparator");
    samp.setText(project.getSamplesSep());
    rootElem.addContent(samp);
    Element varList = new Element("variableList");
            Attribute row = new Attribute("row", project.getRow().toString());
    varList.setAttribute(row);
```

```
        Attribute separator = new Attribute("separator", project
                                        .getVarSep().toString());
        varList.setAttribute(separator);
                Attribute number = new Attribute("number",
                                project.getVarNum().toString());
        varList.setAttribute(number);
                for (int i = 0; i < project.getVariableList().size(); i++) {
        Element var = new Element("variable");
                Element varName = new Element("variableName");
                        varName.setText((String) (projet.getVariableList().get(i)));
                var.addContent(varName);
                Element varType = new Element("type");
                        varType.setText((String) (project.getTypeList().get(i)));
                var.addContent(varType);
                varList.addContent(var);
        }
        rootElem.addContent(varList);
                XMLOutputter out = new
                                XMLOutputter(Format.getPrettyFormat());
        FileOutputStream file = new FileOutputStream(
                                Statistics4jConstants.XML_PROJECT_PATH +
                                        "XML"+
                                        projectNameXML(project.getName()) +
                                        ".xml");
        out.output(doc, file);
        param = out.outputString(doc);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return param;
    }
```

Lo primero que hace el método es coger los datos que se han cargado de la interfaz en un objeto de tipo *CurrentProyect*, que es común a toda las clases (hemos usado el patrón de diseño *singleton*). A continuación crea un objeto de tipo *DocType*, que representa en el árbol JDOM al DOCTYPE del documento *xml,* donde declara el nombre del elemento raíz y el *dtd* correspondiente. A continuación empieza ya la generación del árbol de elementos, primero se crea el elemento raíz, y luego se van añadiendo sus hijos y los atributos de estos. Utilizamos un *for* para la generación de los elementos "variable", cuyo número varía dependiendo del proyecto.

Cuando el árbol está completo, se crea un objeto de tipo *XMLOutputter*. Esta clase ayuda a la generación del archivo a partir del árbol JDOM. Metiendo como parámetro del constructor *Format.getPrettyFormat()*, tabulamos el *xml* de manera que se vean bien los elementos, y no todo el texto seguido. Ahora se crea un archivo en la ruta que deseamos que esté el documento de configuración, se añade como texto el *xml*, correspondiente al *XMLOutputter* creado, y se devuelve también como *String*.

6. HIBERNATE

6.1. ¿Qué es hibernate?

Hibernate ofrece *Persistencia Relacional para Java*, lo que quiere decir que mapea automáticamente las tablas de una base de datos relacional a objetos del lenguaje. Esto proporciona muchas facilidades a la hora de realizar operaciones sobre la base de datos subyacente a tu aplicación. Hibernate se preocupa del SQL y de que las cosas terminen en la tabla correcta.

Puedes cargar muy fácilmente los datos de un registro de una tabla en un objeto de una clase Java, así como también puedes almacenar en dicha tabla un registro cuyos campos correspondan con el contenido de un objeto cuyos datos se han rellenado previamente por código.

6.2. ¿Por qué utilizamos hibernate?

Nuestro proyecto tiene una gran parte relacionada con bases de datos relacionales, de hecho uno de sus módulos está dedicado enteramente a comunicación con este tipo de persistencia.

Al tratarse de un framework, teníamos que proporcionar a los futuros usuarios, un sistema que soportara grandes volúmenes de datos (lo que se traduce en un también grande número de tablas en la base de datos, o como mínimo, tablas que soportan un gran número de registros). Hibernate reduce el número de líneas en lenguaje SQL, es decir, si utilizáramos únicamente JDBC, tendríamos que insertar en el código Java complejas sentencias SQL para realizar las diferentes operaciones con las tablas de la base de datos, mientras que con hibernate, muchas de estas operaciones quedan simplificadas en métodos java (incluidos en el API de hibernate) que ocupan a lo sumo un par de líneas.

Alguien que conozca el funcionamiento de esta herramienta, sabrá que por cada tabla de la base de datos se ha de crear un archivo con extensión *hbm* (archivo de mapeo) y una clase java. En la mayoría de las aplicaciones estos ficheros se crean a mano, o utilizando un *plugging* para eclipse que lo hace automáticamente, el *hibernate synchronizer* (desconocemos si existe algún *plugging* para otra herramienta de desarrollo). Nuestra aplicación debe soportar muchas bases de datos diferentes, con diferente número de tablas y diferentes campos, por esta razón no podemos crear los archivos de mapeo y las clases java en tiempo de compilación, sino en tiempo de ejecución. Estos ficheros se crean dinámicamente a partir de unas plantillas de código elaboradas con *velocity,* una herramienta cuyo funcionamiento explicaremos mas adelante. El *hibernate synchronizer* fue utilizado como herramienta auxiliar, para crear los archivos de mapeo y las clases java de las tablas estáticas, las que son comunes para cualquier implementación del framework, y para

crear las bases para las plantillas utilizadas para la generación dinámica de código con *velocity*.

6.3. ¿Por qué hibernate 2?

En el momento en el que comenzamos a montar la aplicación, acababa de salir la nueva versión de hibernate, la 3.0. En un principio se pensó en utilizar esta nueva *release,* pero debido a que casi toda la documentación disponible en Internet era sobre la versión 2.0, y también debido a que el *hibernate synchronizer* sólo funciona para dicha versión, optamos por utilizar hibernate 2.0.

6.4. Hibernate en Statistics4j

Nuestro proyecto java contiene un paquete con el nombre de hibernate, que contiene todo el código relacionado con el manejo de esta librería. Detallamos el contenido de dicho paquete a continuación:

Esquema del paquete:

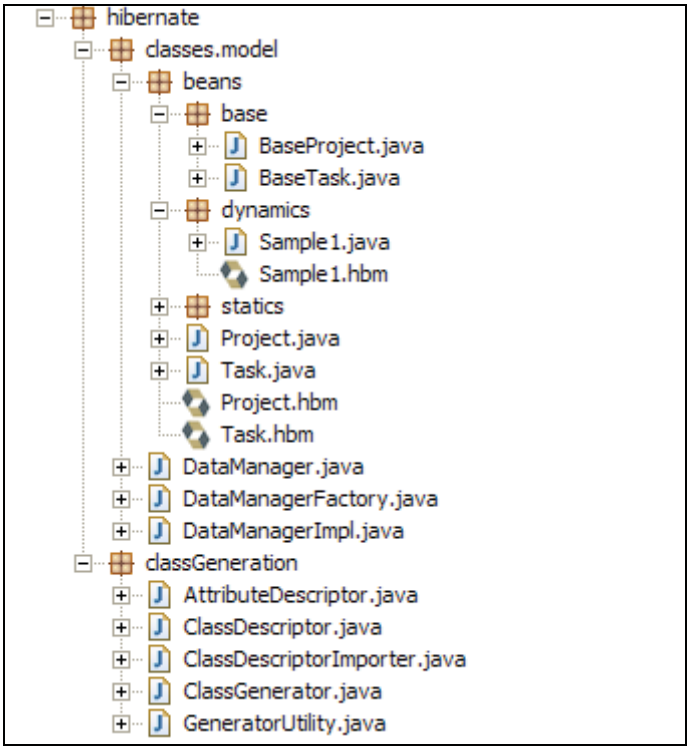


Figura 6.4.1

Descripción de subpaquetes y clases:

- **classes.model**: En la raíz de este paquete encontramos las clases que implementan las operaciones básicas de acceso mediante hibernate a la base de datos. Explicamos su funcionamiento dentro de este capítulo más adelante.

- **classes.model.beans**: Este paquete es el que contiene las clases y los archivos de mapeo que, como explicamos unas líneas más abajo, están separados en dos subpaquetes según correspondan a la base de datos estática o dinámica.

  En la raíz del paquete están los correspondientes a las tablas comunes para cualquier base de datos, *Project* y *Task*. *Project* representa cada proyecto que se introduce en la aplicación para analizar (recordemos que se introducen los datos del proyecto en un fichero de datos), y *Task* representa cada tarea que se puede realizar sobre el proyecto.

- **classes.model.beans.base**: Aquí están *BaseProject* y *BaseTask*, clases auxiliares de las que heredan *Project* y *Task*.

- **classes.model.beans.dynamics**: En este paquete están los archivos de hibernate que se generan cuando se elige que la base datos sea dinámica (que se cree en tiempo de ejecución).

- **classes.model.beans.statics**: Aquí se encuentran los archivos de mapeo y clases java de hibernate para la base de datos cuando se elige que ésta sea estática (que se cree cuando se instale la aplicación). Recordemos que cuando se elige este tipo de base de datos, ésta tiene unas tablas fijas comunes a cualquier implementación de statistics4j. Estos archivos puede que no sean utilizados por la aplicación en ningún momento (si se elige que la base de datos sea dinámica), pero debido a que no ocupan mucho espacio de memoria, decidimos tenerlos creados en tiempo de compilación.

- **classes.model.classGeneration**: En este paquete encontramos las clases que implementan la generación dinámica de los ficheros hibernate mencionados mediante *velocity*. Detallaremos su contenido y su utilidad en el apartado correspondiente a dicha herramienta.

Por último nos quedaría hacer mención del fichero de configuración de hibernate, el *hibernate.cfg.xml.* Este documento *xml* tiene que ir obligatoriamente en la raíz del paquete *src* (source), y contiene la configuración de la base de datos, así como la lista de todos los archivos de mapeo.

Como en el caso de la generación dinámica de la base de datos no sabemos el número de archivos de mapeo que vamos a necesitar, ni su contenido, necesitamos que este archivo de configuración se genere en tiempo de ejecución al igual que los *hbm* y las clases.

Las clases de acceso

Como ya adelantamos en el apartado anterior, estas clases ofrecen una implementación de las operaciones básicas que realizamos con la base de datos, y se encuentran en el subpaquete *classes.model* del paquete hibernate (ver figura 6.4.1).

Se basan en el patrón de diseño *factory,* ahora explicamos su contenido y como funcionan:

- **DataManager**: Este es el interfaz que contiene las operaciones básicas de comunicación con la base de datos. Cualquier clase que creemos que queramos que implemente la interacción con la base de datos, tiene que heredar de este interfaz, e implementar las operaciones básicas:

  - **public Object load(Class cl,Integer id)throws Exception:** Este método carga el registro de la tabla de la base de datos correspondiente a la clase *cl* y con clave primaria igual a *id,* en un objeto de la clase *Object*, que devuelve como resultado. Devolver un objeto de esta clase nos permite utilizar el método para cargar objetos de cualquier tabla de la bases de datos en cualquiera de las clases de hibernate del sistema. Equivaldría en *sql* a la consulta a:

    SELECT * FROM cl WHERE cl.id = *id*

    (donde cl es el nombre de la tabla, y su clave primaria es el campo id)

  - **public void store(Object object)throws HibernateException:** Este método carga el un registro de la tabla de la base de datos correspondiente, el objeto *object*. Al pasar como parámetro un objeto de la clase *Object*, nos aseguramos de que este método lo podemos utilizar para todas las clases de hibernate, y podemos insertar datos en cualquier tabla del sistema.

- **DataManagerFactory**: Esta clase es la factoría de las clases que implementan el interfaz que hemos explicado en el párrafo anterior. Para la creación de la única clase que tenemos en nuestra aplicación que implemente este interfaz, tenemos el método:

```
static public DataManager newDataManager() {
                    return new DataManagerImpl();
        }
```

- **DataManagerImpl**: Esta clase implementa el interfaz DataManager para nuestro caso particular. Además de los métodos descritos en dicho interfaz, están los métodos de manejo de las sesiones de hibernate.
  En esta clase o en otras que quisiéramos crear como implementaciones de DataManager, podemos añadir métodos que realizaran operaciones con la base de datos más complejas, tales como búsquedas por varios campos, o *joins*, de manera mucho más sencilla que con sentencias *sql.* Ej:

  Supongamos que en una tabla de la base de datos almacenamos coches, con los siguientes campos: coches(matricula, color, marca, año). La clase de hibernate correspondiente sería la clase *Coche.java*, cuyos atributos serían los mismos que los campos de la tabla coches. A continuación tenemos el código que, mediante operaciones de hibernate, buscaría en la base de datos coches de color rojo:

```
Object p = dataManagerImpl.load(Coche.class, new Integer(1));
list = dataManagerImpl.search("from Coche task where  color=’rojo’");
```

  La consulta devolvería en la lista list, una colección de objetos de la clase coche con todos sus atributos correspondiente, y por supuesto con el atributo color igual a “rojo”.

7. VELOCITY

7.1. ¿Qué es velocity?

Podemos decir que velocity es un lenguaje de plantillas. A partir de plantillas de código muy sencillas, podemos construir todo tipo de archivos con un esquema determinado. Para la generación del contenido de los ficheros se hace uso de unas sencillas clases, en las cuales, además de objetos propios de la librería velocity, se utiliza la herramienta de manipulación de archivos *xml* SAX, explicada con detalle en el apartado que se le dedicamos anteriormente. Las plantillas de velocity son simples archivos de texto plano cuya extensión tiene que ser *.vm*.

7.2. ¿Por qué utilizamos velocity?

La razón de que hayamos preferido velocity al resto de los lenguajes de plantillas, es únicamente que es una librería de apache, con lo cual además de la garantía que esto supone, es una librería gratuita y la documentación es bastante completa.

7.3.Velocity en Statistics4j

En el caso de nuestro proyecto, utilizamos velocity para crear en tiempo de ejecución las clases java, los archivos de mapeo y el archivo de configuración (hibernate.cfg) que utilizamos para el mapeo relacional de la base de datos con hibernate, cuando en la instalación de la aplicación, el usuario ha elegido utilizar una base de datos generada dinámicamente.

A continuación mostramos un ejemplo de una de las plantillas de nuestro sistema, la que se utiliza para la generación de las clases java:

```
package
es.ucm.fdi.statistics4j.hibernate.classes.model.beans.dynamics;


public class $class.Name {

#foreach( $att in $class.Attributes)
      // $att.Name
      private $att.Type $att.Name;
#end
#foreach( $att in $class.Attributes)
      // $att.Name
      public $att.Type get$utility.firstToUpperCase($att.Name)() {
            return this.$att.Name;
```

```
        }
        public void
set$utility.firstToUpperCase($att.Name)($att.Type $att.Name) {
                this.$att.Name = $att.Name;
        }
#end
        }
```

Las palabras que no llevan un símbolo '$' ni '#' delante son texto normal, lo que quiere decir que en cualquier generación de una clase saldrá ese texto escrito tal cual. Las palabras precedidas del símbolo '$' son las variables de las que *velocity* saca el valor pertinente para cada ejecución. En este caso los valores necesarios están en la variable *class*, que es de tipo *ClassDescriptor* como veremos más adelante. Podemos fijarnos en que a los atributos de *class* accedemos mediante un punto, y que dichos atributos coinciden con los de la clase mencionada.

Las palabras precedidas por el símbolo '#' son palabras reservadas del lenguaje de plantillas. En este caso particular, "#foreach" crea una especie de bucle *for* que recorre los elementos de "$class.attributes" (recordemos que el atributo attributes de la clase *ClassDescriptor* es una lista de objetos de la clase *AttributeDescriptor*).

Todas las clases involucradas en la generación dinámica de archivos están en el paquete *hibernate.classGeneration*.

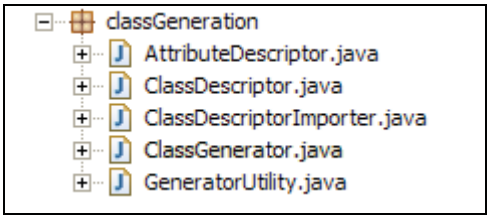


Figura 7.3.1

A continuación detallamos los paquetes y las clases que utilizamos para la implementación de la generación dinámica de los ficheros mencionados:

- **AttributeDescriptor**: Clase auxiliar para la generación de archivos *.java* que representa a un atributo de la clase a generar. Tiene dos atributos:

· *name*: De tipo String, representa el nombre del atributo.
· *type*: También de tipo String, representa el tipo del atributo.

- **ClassDescriptor**: Clase auxiliar que representa la clase a generar dinámicamente. Contiene tres atributos que son:

· *name*: De tipo String, representa el nombre de la clase. No incluye la extensión *.java*.
· *attributtes*: De tipo ArrayList, contiene la lista de atributos de la clase.

- **ClassDescriptorImporter**: Esta es la clase encargada de leer el xml en donde está la información necesaria para la generación de las clases dinámicas. Este *xml* coindcide con el *xml* de configuración de proyecto que hemos mencionado y explicado al detalle en el manual del programa. Para leer dicho *xml* utilizamos la herramienta SAX, cuyo funcionamiento ya hemos explicado en el apartado correspondiente a SAX y JDOM.

- **GeneratorUtility**: Esta clase contiene métodos auxiliares para la creación dinámica del código, sólo relacionados con la sintaxis. En nuestro caso sólo contiene un par de métodos:

· **public static String firstToUpperCase(String string)**: Pone la primera letra de la cadena de caracteres *string* en mayúsculas.

· **public static String firstToLowerCase(String string)**: Pone la primera letra de la cadena de caracteres *string* en minúsculas.

- **ClassGenerator**: Esta clase contiene los métodos que realizan todo el proceso de generación de los archivos dinámicamente. Por cada tipo de archivo hay un método. Hay un método para generar clases *java*, otro para generar el archivo de configuración de *hibernate*, y otro para los archivos de mapeo. Todos estos métodos tienen la misma estructura, explicamos detalladamente uno de ellos como ejemplo, el que se encarga de la creación de clases *java*:

```
public static void startJava(String modelFile, String
templateFile)throws Exception {

        // Load the model
        FileInputStream input = new FileInputStream(modelFile);
        xmlReader.parse(new InputSource(input));
        input.close();
        classes = cdImporter.getClasses(); // ClassDescriptor Array

        //Generate classes
        GeneratorUtility utility = new GeneratorUtility();
        for (int i = 0; i < classes.size(); i++) {
                VelocityContext context = new VelocityContext();
                ClassDescriptor cl

        =(ClassDescriptor).classes.get(i);
```

```
                    context.put("class", cl);
                    context.put("utility", utility);
                    Template
        template=Velocity.getTemplate(templateFile);
                    BufferedWriter writer = new BufferedWriter(new
                    FileWriter(Statistics4jConstants.
                            GENERATED_CLASSES_PATH+cl.getNa
                    me()+".java"));
                    template.merge(context, writer);
                    writer.flush();
                    writer.close();
                }

        }
```

La secuencia de acciones que realiza este método sería:

· Se carga el xml del cual cargamos los datos necesarios para generar la clase.

· Se almacena en el array *classes* las clases que leemos desde el archivo xml cargado en el paso anterior. Cada clase viene representada en el array por un objeto del tipo *ClassDescriptor.*

· Por cada *ClassDescriptor* se crea una clase java. Para crear una clase, se crea un contexto de Velocity, se carga la plantilla correspondiente, se obtiene la ruta en la que se quiere generar el archivo, y con todo esto y la operación *merge* de velocity, se crea la clase.

Observamos que en el contexto cargas los datos mediante el método *put*. Por ejemplo cuando escribimos context.put("class", cl), asociamos la variable "class" a los datos almacenados en el objeto cl (de tipo *ClassDescriptor*). De esta manera cuando en la plantilla pongamos $class.name, querrá decir que en esa posición queremos que se escriba el valor del atributo name del objeto cl.

Además, esta clase también contiene dos métodos auxiliares en el proceso de generación de clases:

· **public static void loadClass(String arg, String root):** Esta función compila un archivo java creado dinámicamente, es decir, crea el archivo *.class* correspondiente. El parámetro arg contiene el nombre de la clase (sin la extensión *.java*), y el parámetro root contiene la ruta absoluta en la que se encuentra este fichero. Es importante tener en cuenta que la ruta debe ser absoluta, con lo cual cualquier

implementación de nuestro framework necesitará modificarla. Esto supondrá que el valor de la ruta esté en un archivo properties, para que pueda ser modificada con facilidad. El código es éste:

```
public static void loadClass(String arg, String root) {

        Properties p = new Properties();
        p.setProperty("file.resource.loader.cache", "false");
        p.setProperty("velocimacro.library.autoreload", "true");
        String[] argsComp = new String[] { "-d",

        "c:\\eclipse\\workspace\\statistics4j\\target\\classes\\",
                                root + arg + ".java" };
        int status = Main.compile(argsComp);

}
```

Para la compilación de la clase utilizamos la clase *com.sun.tools.javac.Main.*

· **public static Object getInstance(String arg):** Con este método se crean las instancias de las clases generadas dinámicamente, ya que no podemos crearlas de la manera habitual (*new* [nombre del objeto](*params*)), porque en tiempote compilación no conocemos el nombre de estas clases, y aunque lo supiéramos, al no tener su *.class,* la aplicación no compilaría. El código del método es el siguiente:

```
public static Object getInstance(String arg) {
                Object instance = null;
                try {
ClassLoader loader = ClassLoader.getSystemClassLoader();

                        Class clazz = loader.loadClass(arg);
                        instance = clazz.newInstance();
                } catch (Exception e) {
                        e.printStackTrace();
                }
                return instance;
        }
```

Como vemos, utilizamos la clase *ClassLoader* para cargar la nueva clase creada y compilada con el método anterior.

8. LIBRERÍAS UTILIZADAS

A continuación mostramos una tabla con las librerías utilizadas para la implementación de la aplicación, con una breve descripción de la misma y la utilización dentro de statistics4j.

Nombre	***Versión***	***Descripción***	***Módulo de Uso***
cglib-full	*2.0.2*	Librería utilizada para la conexión entre hibernate y j-boss	Módulo de Persistencia
commons-logging	*2.0*	Librería de Apache para la generación de logs	Todos
dom4j	*1.4*	Librería que ofrece soporte para API's de procesamiento de XML, cn nuestro caso SAX.	Módulo de Runtime
hibernate2	*2.1.7c*	Librería de hibernate	Módulo de Persistencia
hibernatesynchronizer	*1.0*	Librería para la creación automática de ficheros de mapeo de hibernate con eclipse	Módulo de Persistencia
itext	*1.3*	Librería para la creación de pdf's	Módulo de presentación
jcommon-1.0.0.pre2	*1.0*	Librería para la generación de gráficos	Módulo de presentación
jfreechart-.1.0.0-pre2	*1.0*	Librería para la generación de gráficos	Módulo de presentación
jta	*1.0.1*	Libería de la Java Tansaction API	Todos
JUnit	3.8.1	Librería utilizada para la generación de clases de pruebas de los métodos de la aplicación	Todos
log4j	1.2.8	Librería para la generación de logs	Todos
mysql-connector-java	*3.0.15-ga-bin*	Driver de conexión con la base de datos	Módulo de Persistencia
sax2	*2.0.2*	Librería del api SAX	Módulo de Runtime
tools	*1.4.0*	Librería para la compilación de clases	Módulo de Runtime
velocity	*1.4*	Librería para la generación de código	Módulo de Runtime
velocity-dep	*1.4*	Librería para la generación de código	Módulo de Runtime
xercesImpl	*2.6.2*	Librería para la manipulación de XML	Módulo de Runtime

9. APÉNDICE

9.1 GESTIÓN DE CONFIGURACIÓN

Planificación del Proyecto

A continuación se detallan los diferentes puntos que describen la planificación que se va a llevar del proyecto.

Condiciones generales

El grupo se compone de cinco miembros, tres alumnos que desarrollarán el proyecto más dos profesores-directores, que guiarán a los alumnos.

Al tratarse de un grupo de tres personas, no habrá una división jerárquica entre los miembros del grupo.

El proyecto consiste en la realización de un framework orientado al desarrollo de aplicaciones estadísticas en el campo de la biología.

Seguimiento y reuniones

El seguimiento y las reuniones entre los tres alumnos y los directores profesores se llevarán a cabo los martes y jueves, a las 19:30, en la sala de reuniones de la facultad de informática, situada en la primera planta del edificio.

Si alguno de los miembros no pudiese asistir, lo comunicará mediante mail al resto de integrantes.

Comunicación entre el grupo

La comunicación entre los tres alumnos del grupo y los profesores directores se llevará a cabo mediante correos electrónicos.

Gestión de archivos

Mediante CVS

Documentación

Listado de documentos

La documentación necesaria que hay que generar y la tecnología necesaria para ello es la siguiente:

Documento	**Modo de desarrollo**
Plantillas	Word
Gestor de Configuración	Word
Especificación de Requisitos	Enterprise Architect → Word
Modelo de casos de uso	Enterprise Architect → Word
Diagramas de casos de uso	Enterprise Architect → Word
Diagramas de secuencia	Enterprise Architect → Word
Modelo Vista Usuario	Enterprise Architect → Word
Especificación módulo de persistencia	Enterprise Architect → Word
Diagramas módulo de persistencia	Enterprise Architect → Word
Especificación de librerías utilizadas	Word

Normas de documentación

A continuación se describen los modos de desarrollo de la documentación, dependiendo de la tecnología utilizada.

Word

Tomar la plantilla general de documento, una plantilla específica o una versión ya existente del mismo y generar la documentación siguiendo las normas descritos en el apartado correspondiente a estándar de documentación de este documento.

Enterprise Architect → Word

Para generar los documentos con el Enterprise Architect hay que seguir los siguientes pasos una vez generados los datos:

- Exportar el paquete como documento rtf.
- Seleccionar las características necesarias según el tipo de documento.

En el caso de incluir alguna imagen, se exportará en formato jpg.
Si la imagen no es lo suficientemente clara, se ampliará una vez generado el documento.

- Para terminar se escogerá la plantilla general de documento, una plantilla específica o una versión ya existente y se pegará el código generado por el Enterprise Architect manteniendo el formato especificado en el apartado de estándar de documentación de este documento. A su vez se completarán los datos relativos al autor e historial de versiones del documento.

Estándar de documentación

El formato de cualquier documento del proyecto será el siguiente:

Títulos

El título principal se escribirá en mayúsculas con un tamaño de 16 en negrita.

Los subtítulos deben escribirse en letras minúsculas con un tamaño de 14 y en negrita.

El estilo de los títulos debe ser el de un título predefinido de Word. El estilo del título principal será el título1, los subtítulos del primer nivel de la jerarquía tendrán como estilo el título 2, los subtítulos del segundo nivel de la jerarquía tendrán como estilo el título 3, y así sucesivamente. Si alguno de los títulos predefinidos por Word no se ajusta al formato de los títulos explicado más arriba, se modificará el formato de los títulos para que lo cumplan. Es necesario que los títulos tengan el estilo de uno de los títulos predefinidos por Word para que se genere correctamente el índice.

Los títulos aparecerán numerados de manera jerárquica. La numeración tendrá una sangría mayor cuanto más se descienda en la jerarquía. El esquema de numerado de Word escogido debe ser este:

En la opción de numeración de listas debe seleccionarse la opción de continuar la lista anterior.

Índices

Se incluirá un índice al principio de cada documento. Dicho índice se realiza del siguiente modo:

En la opción "Índice y tablas" del menú "Insertar" de Word se escogerá la pestaña de "Tabla de contenido". Se marcarán las opciones de mostrar los números de página y alinear los números de página a la derecha. El número de niveles de los títulos será de 3. La opción de "formatos" debe ser "elegante". Finalmente se debe pulsar el botón aceptar.

Si en algún momento es necesario actualizar el índice, puede hacerse situando el cursor con el teclado sobre el mismo, y seleccionando la opción de "actualizar campos", la cual aparece cuando se pincha con el botón derecho del ratón.

Fuente, interlineado y sangrías

- El tipo de fuente utilizado para escribir los documentos será "Times New Roman" con un tamaño de 12.
- El documento se encontrará justificado y con un interlineado sencillo.
- La primera línea de cada párrafo se situará a la misma altura del comienzo del subtítulo.
- Las siguientes líneas comenzarán a la altura de la numeración del subtítulo.
- Cada párrafo deberá separarse por una línea en blanco.
- Cada subtítulo debe estar separado por una línea en blanco.
- Cada título principal debe estar separado por un salto de página, salvo que el contenido del mismo sea lo suficientemente pequeño.

Viñetas

El formato para las viñetas será el siguiente:

- Nivel 1
  - Nivel 2
    - Nivel 3

Tablas

El formato de las tablas será el siguiente:

Portadas, encabezado y pie de páginas

Cada documento deberá incluir una portada con el logotipo del proyecto, el nombre del documento, la fecha de creación y la versión.

En el encabezado se incluirá el nombre del documento y el logotipo del proyecto.

El pie de página debe incluir la versión del documento y la numeración de las páginas en el margen inferior derecho.

Nombrado de versiones

Para nombrar los documentos deben seguirse estas pautas:

NombreDocumento.V.X.Y

Para evitar problemas, no se permitirá poner espacios en blanco en el nombre de los documentos. Si el nombre de un documento está formado por más de una palabra, se escribirá la letra inicial de cada palabra con mayúsculas y el resto de la palabra en minúsculas. Por ejemplo: NombreDocumentoConVariasPalabras.

Tampoco deben ponerse tildes en los nombres de los documentos.

Estándar de código

Para los programas que hayan sido implementados en Java, se tomará como estándar el proporcionado por **SUN**. Dicho estándar está disponible en la dirección web **http://java.sun.com/docs/codeconv/** .

Hardware necesario

Para el desarrollo de la aplicación se requiere como configuración mínima un Pentium III a 800 Mhz.

Software necesario

El software requerido para el desarrollo del proyecto es:

- Eclipse 2.0 o superior.
- JDK 1.4.2
- Enterprise Architect 4.1 o Together
- Microsoft Office
- JUnit 3.8.1
- Hibernate 2.0.3
- MySql
- Log4j

10. BIBLIOGRAFÍA

- **Professional XML**, Peer Information Inc.; 1ª edición (Enero, 2000), by Mark Birbeck, Michael Kay, stev Livingstone, Stephen F. Mohr, Jonathan Pinnock, Brian Loesgen, Steven Livingston, Didier Martin, Nikola Ozu, Mark Seabourne, David Baliles.

- **Active Object.An Object Behavioral Pattern for Concurrent Programming** by R. Greg Lavender Douglas, C. Schmidt G. , ISODE Consortium Inc. Department of Computer Science Austin, TX Washington University, St. Louis

- **Design Patterns**, Addison-Wesley Professional; 1ª edición (Enero 15, 1995) by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.

- **Reference Manual for Hibernate 2.1.7 (English)** de http://www.hibernate.org.

- **Página oficial de Velocity,** http://jakarta.apache.org/velocity/

- **Página O'Reilly onJava.com,** http://www.onjava.com/pub/a/onjava/2004/06/02/cg-vel-2.html

- **Página oficial de Sun,** http://www.sun.com

- **Página oficial de log4j,** http://logging.apache.org/log4j/docs/

- **Manual de log4j,** http://www.vimeworks.com/~mauricio/manualLog4J.html

- **Thinking in JAVA,** by Bruce Eckel, versión disponible en internet.

- **Manual de iText,** (herramienta para la generación de pdf's) www.lowagie.com/iText/

- **Manual de jfreeChart,** (herramienta para la generación de gráficos) http://www.jfree.org/jfreechart/

- **Documentación del API de JAVA**

11. PALABRAS CLAVE

- **active-object**
- **Framework**
- **hibernate**
- **jdom**
- **modelo de datos**
  - ***dinámico***
  - ***estático***
- **multi-hilo**
- **properties**
- **SAX**
- **velocity**
- **xml de configuración**
  - ***proyecto***
  - ***sistema***

Autorizamos a la Universidad Complutense de Madrid a la difusión y utilización con fines académicos, no comerciales y, siempre y cuando se mencione a los autores, tanto esta memoria, como el código, la documentación y/o el prototipo desarrollado.

Los autores:

Ruth María Charro Henche:

Pablo Lavín Mera:

Arturo Mengotti Arribas: