Universitat Autònoma
de Barcelona

Componente Editor & Analizador de SPL-SQL de *karat*

Memoria del proyecto

de Ingeniería Técnica en

Informática de Gestión

realizado por

Daniel Dueñas Ruiz

y dirigido por

Jordi Pons Aróztegui

Escuela Universitaria de Informática

Sabadell, Junio de 2009

El abajo firmante, Jordi Pons Aróztegui,

professor de la Escuela Universitaria de Informática de la UAB,

CERTIFICA:

Que el trabajo al que corresponde la presente memoria ha
sido realizado bajo su dirección
por Daniel Dueñas Ruiz

Y para que conste firma este escrito.
Sabadell, Junio de 2009

Firmado: Jordi Pons Aróztegui

El abajo firmante, Ezequiel Parra Mestre de CCSAgresso

CERTIFICA:

Que el trabajo al que corresponde la presente memoria ha sido realizado bajo su supervisión por Daniel Dueñas Ruiz

Y para que conste firma este escrito.
Sabadell, Junio de 2009

Firmado: Ezequiel Parra Mestre

Prólogo

Esta memoria corresponde al Proyecto Final de Carrera de la titulación *Ingeniería Técnica en Informática de Gestión*. El proyecto ha sido realizado mediante la firma de un convenio entre la *UAB* y la empresa *CCS Agresso*.

Este proyecto ha sido realizado en el departamento de *Desarrollo* de dicha empresa y el objetivo principal es la creación de un componente "Editor y Analizador para el lenguaje *SPL-SQL*" del producto *karat*. Este editor y analizador tiene como objetivo poder ayudar a los desarrolladores en la programación de código de este lenguaje implementando diferentes funcionalidades que ayuden a esta tarea.

El proyecto ha sido desarrollado con el lenguaje *Java*, entre los meses de noviembre de 2008 y junio de 2009.

Índice

Capítulo 1: Introducción……………………………………………………………………..1

1.1. Presentación……………………………………………………………………2

1.2. Empresa…………………………………………………………………………...3

1.2.1. CCS Agresso……………………………………………………………....3

1.2.2. Entorno de trabajo………………………………………………………....4

1.3. Objetivos………………………………………………………………………….5

1.4. Motivaciones……………………………………………………………………...6

1.5. Contenido de la memoria…………………………………………………………6

Capítulo 2: Estudio de viabilidad y definición de requerimientos…………………………...8

2.1. Objeto…………………………………………………………………………….9

2.1.1. Descripción de la situación actual………………………………………..9

2.1.2. Perfil de usuario…………………………………………………………11

2.1.3. Objetivos………………………………………………………………...11

2.2. Sistema a realizar………………………………………………………………..12

2.2.1. Descripción……………………………………………………………...12

2.2.2. Definición de requerimientos…………………………………………...13

2.2.3. Recursos…………………………………………………………………14

2.2.4. Análisis coste – beneficio……………………………………………….15

2.2.5. Evaluación de riesgos…………………………………………………...15

2.2.6. Alternativas……………………………………………………………...16

2.3. Modelo de desarrollo……………………………………………………………16

2.4. Planificación……………………………………………………………………..18

Capítulo 3: Análisis y diseño……………………………………………………………21

3.1. Visión general…………………………………………………………………...22

3.1.1. Diagrama de contexto……………………………………………...22

3.1.2. Descripción de los componentes…………………………………..22

3.2. Modelo de objetos……………………………………………………………26

3.2.1. Diagrama de clases………………………………………………...26

3.2.2. Descripción de componentes………………………………………27

3.3. Especificaciones de los procesos……………………………………………..36

3.3.1. Syntax coloring…………………………………………………….36

3.3.2. Content assist………………………………………………………39

3.3.3. SPL Preferences……………………………………………………40

3.3.4. SPL Editor Options………………………………………………...43

Capítulo 4: Codificación y pruebas………………………………………………...45

4.1. Lenguaje……………………………………………………………………...46

4.2. Herramientas para el diseño………………………………………………….47

4.3. Entornos de desarrollo………………………………………………………..47

4.4. Estilo de codificación………………………………………………………...48

4.5. Pruebas………………………………………………………………………..51

Capítulo 5: Conclusiones…………………………………………………………...52

5.1. Objetivos cumplidos………………………………………………………….53

5.2. Desviaciones respecto la planificación temporal inicial

y cambios en el desarrollo………………………………………………….53

5.3. Ampliaciones…………………………………………………………………56

5.4. Valoración personal…………………………………………………………..57

Bibliografía…………………………………………………………………………..58

Anexo A – Glosario…………………………………………………………………59

Anexo B – Contenido del CD-Rom………………………………………………...62

Índice de ilustraciones

Ilustración 1. Empresa CCS Agresso 1

Ilustración 2. Organización de *karat* 1

Ilustración 3. Editor de SPL 1

Ilustración 4. Lista de SPL 1

Ilustración 5. Ejemplo de un posible editor SPL 1

Ilustración 6. Modelo en espiral 1

Ilustración 7. Planificación y diagrama de Gantt 20

Ilustración 8. Diagrama de contexto 1

Ilustración 9. Diagrama de clases 1

Ilustración 10. Editor SPL 1

Ilustración 11. Particiones y familias de la funcionalidad syntax coloring 1

Ilustración 12. Ejemplo de syntax coloring en un fichero spl 1

Ilustración 13. Content assist con palabras clave 1

Ilustración 14. Content assist con funciones y su ayuda correspondiente 1

Ilustración 15. Content assist con plantillas 1

Ilustración 16. Selección de la plantilla “CREATE PROCEDURE” del content assist ... 1

Ilustración 17. Página de SPL Preferences 1

Ilustración 18. Página de Code Templates 41

Ilustración 19. Página de Formatting/Translation 1

Ilustración 20. Menú SPL Editor Options 43

Ilustración 21. Ejemplo de una spl traducida a diferentes lenguajes de bases de datos ... 1

CAPÍTULO 1: INTRODUCCIÓN

En este apartado se introduce qué es y cómo nació el lenguaje SPL. Se realiza una descripción de la empresa y del departamento en el que se desarrolla el proyecto, además de hablar del convenio con la empresa. También se presenta el proyecto que se quiere realizar y se exponen las motivaciones para realizarlo. Por último se comenta el contenido de la memoria.

1.1. Presentación

El proyecto que se presenta consiste en un desarrollo de software para la empresa *CCS Agresso*, el cual será realizado, mediante un convenio de colaboración con la *Universitat Autònoma de Barcelona (UAB)*, por un alumno de la *Escola Universitària d'Informàtica (EUI)* en su Proyecto Final de Carrera.

La duración prevista en el convenio para el proyecto, en principio, es de 560 horas desglosadas según las siguientes etapas:

- Formación (30H)
- Investigación (60H)
- Requerimientos (30H)
- Análisis (60H)
- Diseño y programación (310H)
- Pruebas (40H)
- Documentación (30H)

El coste del proyecto para *CCS* sería aproximadamente el siguiente:

- Horas que el tutor debería dedicar a formación, gestión del proyecto y evaluación del mismo. Aproximadamente 60h.
- Coste económico según lo especificado en el convenio que recibe el alumno en concepto de ayuda al estudio.

La empresa utiliza una plataforma tecnológica para la gestión de las empresas que se llama *karat* y que aporta un nuevo concepto de soluciones basado en la independencia total y real de entornos.

El reto que se propone en este proyecto es el desarrollo de un nuevo elemento destacado en la plataforma *karat*, el componente *Editor y Analizador de lenguajes de programación de base de datos SPL y SQL*. Se trata pues de realizar una herramienta para editar y desarrollar fácilmente código *SPL (Stored Procedure Language)*.

Un *SP* es un procedimiento almacenado en la base de datos, escrito en un lenguaje procedural de alto nivel que proporciona la base de datos y que permite la interrogación y manipulación de datos de una forma fácil y rápida.

Una desventaja que tienen los procedimientos almacenados, es que son dependientes de la base de datos. Cada fabricante ofrece unas prestaciones y características ligadas y optimizadas para su base de datos, por lo que existen sintaxis muy dispares entre distintas bases de datos.

Para solucionar este inconveniente *karat* ha diseñado un lenguaje, el lenguaje *SPL* estándar *karat*, que visto a grosso modo es el máximo común divisor de los tres lenguajes de SP de las bases de datos soportadas originalmente por *karat*, a saber: *Informix, Oracle y SQL Server*. Actualmente las bases de datos que se soportan son *Caché, Oracle, DB2, SQL Server, PostgreSQL y MySQL.*

El componente a desarrollar será utilizado desde distintos sitios de *karat*, por ejemplo en edición de procedimientos almacenados y disparadores en lenguaje *SPL*, Vistas, expresiones *SQL*, etc.

La nueva herramienta debería tener el formato de un *Plug-in IDE Eclipse* a añadir a los actualmente disponibles. Un *plug-in* es una aplicación que se relaciona con otra, en este caso con *IDE Eclipse*, para aportarle una función nueva y generalmente muy específica.

1.2. Empresa

1.2.1. *CCS Agresso*

Ilustración 1. Empresa CCS Agresso

Centro de Cálculo de Sabadell (*CCS Agresso*) es la filial española del grupo *Unit 4 Agresso*, uno de los fabricantes internacionales de software de gestión. Este centro de desarrollo se ubica en la población de Barberà del Vallès (Barcelona).

CCS Agresso es interlocutor global para empresas y organismos públicos, y cuenta con un extenso conjunto de partners repartidos por toda la Península Ibérica que ofrecen al cliente soluciones y servicios de alto nivel.

El objetivo de la empresa es, mediante su oferta de soluciones y servicios, proporcionar una auténtica ventaja competitiva y diferenciación a sus clientes mediante la implantación de eficientes soluciones informáticas de gestión.

Para conseguir su objetivo, la empresa se divide en las siguientes actividades:

- Atención al cliente
- Comercialización
- Diseño
- Investigación
- Desarrollo
- Implantación
- Soporte
- Mantenimiento

La realización de este proyecto se engloba en la actividad de "Desarrollo".

1.2.2. Entorno de trabajo

El proyecto consiste en desarrollar una herramienta para el producto *karat* y este producto se desarrolla íntegramente en el departamento de *Desarrollo* (Fábrica) de la organización *CCS Agresso*.

El equipo de proyecto *karat* está formado por un grupo formado por 16 investigadores y desarrolladores que son los principales integradores de conocimiento técnico. Es un equipo multi-disciplinario en distintas áreas de conocimiento. El alumno, como desarrollador de este proyecto, se integra dentro de este equipo.

El equipo de dirección de proyecto consta de un *Gerente karat* y de un par de *Directores de Proyecto*. El *Gerente karat* depende funcionalmente del *Director de Desarrollo* (Fábrica) de *CCS Agresso*.

La estructura de trabajo que sigue este equipo está organizada por fases de la siguiente manera:

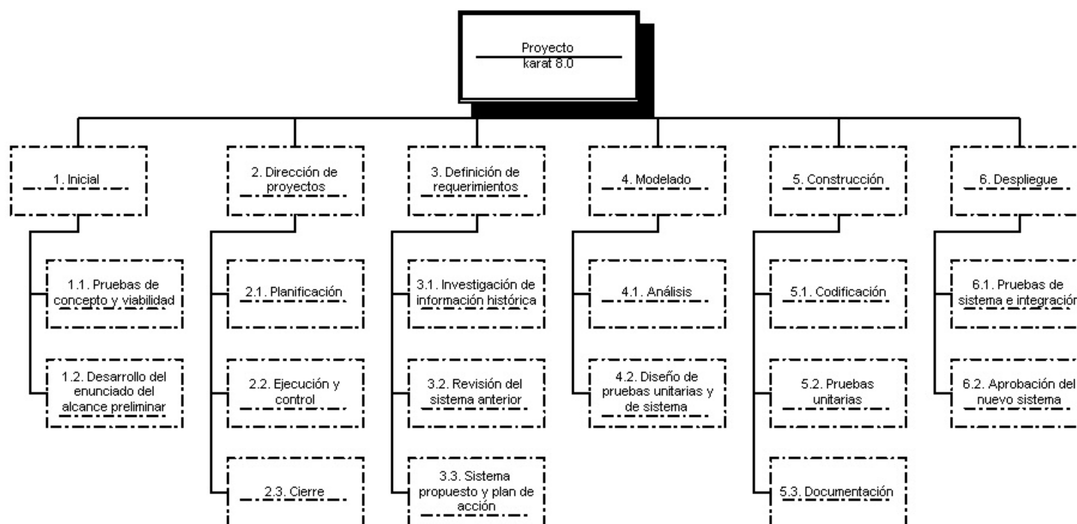


Ilustración 2. Organización de *karat*

1.3. Objetivos

El objetivo principal del proyecto es conseguir una herramienta que facilite el desarrollo de código *SPL* y *SQL* para mejorar las herramientas actuales de edición. Esta dotará de utilidad a otras herramientas relacionadas en la plataforma *karat*.

El nuevo componente se desea integrar dentro de la versión 8.0 del producto *karat*, versión que ha cambiado su estructura, ya que las anteriores se encuentran implementadas en lenguaje *Visual Basic* y la nueva versión en lenguaje *Java*. Por lo tanto para que la integración sea correcta, el objetivo es, que el componente se realice mediante este lenguaje y se encuentre dentro del entorno *IDE Eclipse*, entorno utilizado en la empresa para el lenguaje especificado.

Otro objetivo será dotar de valor a la herramienta para conseguir que la tarea de programación de código para el desarrollador sea más rápida, eficiente y segura.

1.4. Motivaciones

Cuando decidí realizar el proyecto final de carrera en una empresa, la primera idea que me llevó a tomar esa decisión fue la posibilidad de obtener experiencia laboral en el ámbito de la informática, ya que pienso que acabar los estudios y no tener nada de experiencia no abre las mismas puertas en el mundo laboral que teniendo una iniciación. Este convenio me permite aplicar conocimientos teóricos que he ido obteniendo a lo largo de mis estudios y complementarlos con la práctica real.

Trabajar en una empresa me ayudará a conocer como se trabaja en el mundo de la informática, siguiendo unas directrices y un modelo de desarrollo para aprender todas las etapas y fases de un proyecto de un tamaño considerable. Además, saber que el proyecto que se solicita será de gran utilidad para el desarrollo del producto de la empresa me motiva a realizarlo lo mejor posible para que los desarrolladores que lo utilicen se lleven una buena impresión de la herramienta.

Por último, el hecho de desarrollar el proyecto en uno de los lenguajes más importantes del mundo de la informática como es *Java* y aprender el funcionamiento de un editor son objetivos muy interesantes.

1.5. Contenido de la memoria

En este apartado se realiza una pequeña descripción de los diferentes capítulos que forman parte de esta memoria:

- **Introducción:** En este apartado se introduce qué es y cómo nació el lenguaje SPL. Se realiza una descripción de la empresa y del departamento en el que se desarrolla el proyecto, además de hablar del convenio con la empresa. También se presenta el proyecto que se quiere realizar y se exponen las motivaciones para realizarlo.
- **Estudio de viabilidad y definición de requerimientos:** Se hace una descripción de la situación actual y de las herramientas con las que trabajan. A continuación se habla de los diferentes objetivos que se quieren conseguir con la realización de este proyecto. También se describe el sistema a realizar, se

definen los diferentes requerimientos y se explican las diferentes alternativas que existen para la realización del proyecto. Por último se presenta el modelo de desarrollo a seguir y una planificación de las fases del proyecto.

- **Análisis y diseño:** Se exponen los diagramas de contexto y de clases, con una explicación de cada uno de los componentes que aparecen para comprender el funcionamiento del sistema. También se especifica el funcionamiento de las diferentes operaciones definidas en las fases.

- **Codificación y pruebas:** Se explica el lenguaje *Java*, la herramienta *IDE Eclipse* y los entornos en los que se desarrolla el proyecto, como son un *plug-in IDE Eclipse* y *karat*. También se habla del estilo de codificación de la empresa y su forma de trabajar. Por último se explican todas las pruebas que se realizan durante las distintas fases.

- **Conclusiones:** Se comentan las posibles ampliaciones que se pueden realizar al proyecto, es decir, la inclusión de más funcionalidades. Se comenta la planificación real y los cambios con la planificación inicial. Acabamos con conclusiones finales, para hablar de los objetivos conseguidos y hacemos una valoración personal sobre el proyecto y la estancia en la empresa.

CAPÍTULO 2: ESTUDIO DE VIABILIDAD Y DEFINICIÓN DE REQUERIMIENTOS

Se hace una descripción de la situación actual y de las herramientas con las que trabajan. A continuación se habla de los diferentes objetivos que se quieren conseguir con la realización de este proyecto. También se describe el sistema a realizar, se definen los diferentes requerimientos y se explican las diferentes alternativas que existen para la realización del proyecto. Por último se presenta el modelo de desarrollo a seguir y una planificación de las fases del proyecto.

2.1. Objeto

2.1.1. Descripción de la situación actual

Actualmente desde la plataforma *karat* y desde distintas herramientas se accede a la posibilidad de edición de expresiones SQL y SPL, ofreciéndose interfaces gráficas dispares y con distinta funcionalidad. En cada una de las diferentes interfaces la forma de desarrollar el código es diferente, por lo que la programación de estos lenguajes es bastante compleja.

Estas distintas herramientas son las siguientes:

- *Editor de SPL*
- *Editor de Disparadores*
- *Editor de Vistas*
- Expresiones en consultas base
- Seguridad de Datos en Tabla

Principalmente la herramienta más utilizada es el *Editor de SPL* (véase la Ilustración 3). Este editor contiene una lista de los diferentes archivos *SPL* creados (véase la Ilustración 4), desde la que se puede acceder a cada uno de ellos. Las funcionalidades que ofrece la herramienta son limitadas. Contiene funcionalidades estándar de edición, como pueden ser copiar, pegar, cortar, etc. Otra funcionalidad atractiva es la traducción de este código *SPL* a código de la base de datos conectada, pero toda la programación de código *SPL* es totalmente manual.

A continuación les presento un cuadro resumen describiendo las funcionalidades que contienen o no las diferentes herramientas que pueden editar expresiones SQL y SPL:

Herramienta	Edición	Ayuda a la programación	Determinación de errores	Traducción
Editor de SPL	SI	NO	NO	SI
Editor de Disparadores	SI	NO	NO	NO
Editor de Vistas	SI	NO	NO	NO
Expresiones en consultas base	SI	NO	NO	NO
Seguridad de Datos en Tabla	SI	NO	NO	NO

También es difícil la programación, ya que estas interfaces son muy simples, hablando visualmente y no ayudan a diferenciar palabras claves en sentencias, simples nombres de variables, comentarios, etc. Facilidad para cometer errores en expresiones, ya que no detecta muchos y los que detecta no los especifica para ayudar a corregirlos, simplemente los marca. No ofrece ayudas al desarrollo de procedimientos.

El objetivo que se propone es disponer de una única herramienta que facilite el desarrollo de código de una única forma y que ayude visualmente y técnicamente a la programación.

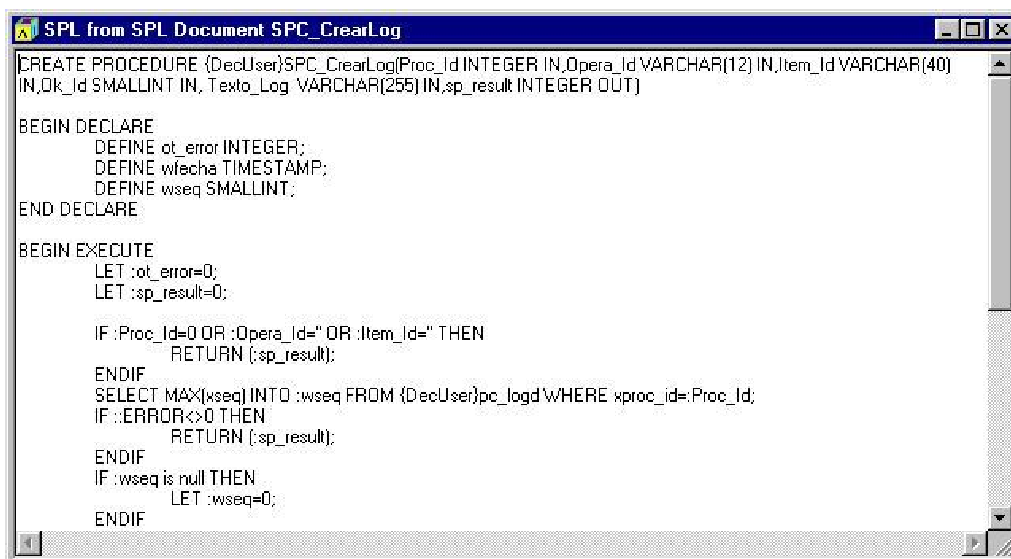


Ilustración 3. Editor de SPL

Lista de SPL

Elemento	Descripción	Propietario	Ver	Autor	Fecha	Modificado	Fecha	Producto asociado
fi_Baja_Inform	Baja informática en Fiscal	data55	1	DBA	04/02/2002	DBA	04/02/2002	Finanzas
SPC_CrearLog	Monitorización de procesos	data55	1	DBA	04/02/2002	DBA	04/02/2002	Común
SPC_DivisaCalcular	Calculo de importes en divisa	data55	1	DBA	04/02/2002	DBA	04/02/2002	Común
SPF_AFBienesAmort	Listado de bienes amortizados por bien y ...	data55	1	DBA	04/02/2002	DBA	04/02/2002	Finanzas
SPF_AFDepBienAmort	Elimina los bienes seleccionados que ya e...	data55	1	DBA	04/02/2002	DBA	04/02/2002	Finanzas
SPF_AFDiarioAmort	Listado del diario de amortizaciones	data55	1	DBA	04/02/2002	DBA	04/02/2002	Finanzas
SPF_AFFichaBiAmort	Ficha de bienes amortizados	data55	1	DBA	04/02/2002	DBA	04/02/2002	Finanzas
SPF_AFGenAmort	Generar movimientos de amortizaciones	data55	1	DBA	04/02/2002	DBA	04/02/2002	Finanzas
SPF_AFMovAmortAbono	Movimientos de amortización por cuenta ...	data55	1	DBA	04/02/2002	DBA	04/02/2002	Finanzas
SPF_AFMovAmortCargo	Movimientos de amortización por cuentas ...	data55	1	DBA	04/02/2002	DBA	04/02/2002	Finanzas
SPF_AFMovAmortEjer	Amortizaciones del ejercicio	data55	1	DBA	04/02/2002	DBA	04/02/2002	Finanzas
SPF_AFMovAmortFicha	Ficha de amortizaciones	data55	1	DBA	04/02/2002	DBA	04/02/2002	Finanzas
SPF_AjustesCalculo	Gestiona el cálculo de las diferencias par...	data55	1	DBA	04/02/2002	DBA	04/02/2002	Finanzas
SPF_AjustesCartera	Gestiona la búsqueda y el cálculo de los ...	data55	1	DBA	04/02/2002	DBA	04/02/2002	Finanzas
SPF_AjustesTesoreria	Gestiona la búsqueda y el cálculo de los ...	data55	1	DBA	04/02/2002	DBA	04/02/2002	Finanzas
SPF_AsientoAjustar2Moneda	Ajustar importes en segunda moneda de u...	data55	1	DBA	04/02/2002	DBA	04/02/2002	Finanzas
SPF_AutorizacionesUsuario	Autorizaciones Usuario	data55	1	DBA	04/02/2002	DBA	04/02/2002	Finanzas
SPF_BalanceGenSaldo	Calcular el saldo de las cuentas	data55	1	DBA	04/02/2002	DBA	04/02/2002	Finanzas
SPF_BalancePlanAsoAcum	Cálculo de los acumulados de deb y haber	data55	1	DBA	04/02/2002	DBA	04/02/2002	Finanzas
SPF_BalancePlanAsoc	Procedimiento encargado de generar los ...	data55	1	DBA	04/02/2002	DBA	04/02/2002	Finanzas
SPF_BalancePlanAsoIN	Generación de registros de la tabla de tra...	data55	1	DBA	04/02/2002	DBA	04/02/2002	Finanzas
SPF_BalanceSS	Ejecución del Balance de Sumas y Saldos	data55	1	DBA	04/02/2002	DBA	04/02/2002	Finanzas
SPF_BalanceSSFec	Acumulados de líneas por fecha para el B...	data55	1	DBA	04/02/2002	DBA	04/02/2002	Finanzas
SPF_BalanceSSFecAcum	Grabación de acumulados por fecha para...	data55	1	DBA	04/02/2002	DBA	04/02/2002	Finanzas
SPF_BalanceSSIns	Grabación de las líneas de trabajo para el...	data55	1	DBA	04/02/2002	DBA	04/02/2002	Finanzas
SPF_BalanceSSPer	Acumulados de líneas a partir de la tabla ...	data55	1	DBA	04/02/2002	DBA	04/02/2002	Finanzas
SPF_BalanceSSPerAcum	Grabación de acumulados a partir de la ta...	data55	1	DBA	04/02/2002	DBA	04/02/2002	Finanzas
SPF_BalanceSSPerDiv	Acumulados de líneas a partir de la tabla ...	data55	1	DBA	04/02/2002	DBA	04/02/2002	Finanzas

Ilustración 4. Lista de SPL

2.1.2. Perfil de usuario

La nueva herramienta será utilizada por técnicos informáticos. Estos técnicos deberán tener conocimientos en gestión de bases de datos, además de conocer los lenguajes de programación SPL y SQL.

En principio no tienen porque tener conocimientos del funcionamiento de la plataforma *karat*.

2.1.3. Objetivos

Los principales objetivos que deberá cumplir la nueva herramienta serán:

- Mejorar las herramientas de edición actuales.
- Conseguir una herramienta única.
- Poder interaccionar con una interfaz sencilla.
- Mejorar la visualización gráfica para el usuario.
- Interaccionar con diferentes base de datos.
- Facilitar y ayudar en la programación.
- Automatizar la programación.

2.2. Sistema a realizar

2.2.1. Descripción

La herramienta que se propone realizar consiste en un editor de un lenguaje de programación específico. Este editor tendrá el formato de un *Plug-in IDE Eclipse*.

Como ya habíamos comentado anteriormente, un plug-in es una aplicación que se relaciona con otra, en este caso con *IDE Eclipse*, para aportarle una función nueva y generalmente muy especifica. Esta aplicación adicional es ejecutada por la aplicación principal e interactúa con ella por medio de una API.

Eclipse contiene plantillas de diferentes plug-ins que se pueden realizar. Entre estas plantillas existe una que consiste en un editor. Por lo tanto esta plantilla será la estructura general del editor solicitado. A partir de aquí, persiguiendo los objetivos comentados anteriormente, obtendremos un editor que ayudará a la programación SPL y SQL, haciendo esta tarea más fácil de realizar.

Este editor, al estar integrado dentro de *Eclipse*, visualmente será muy parecido al editor del *Eclipse* de archivos *Java* (véase un ejemplo del posible editor en la Ilustración 5), requisito definido por la empresa. La empresa desea que *IDE Eclipse* sea una herramienta interna para la programación en *Java*, y para programación en su propio lenguaje.

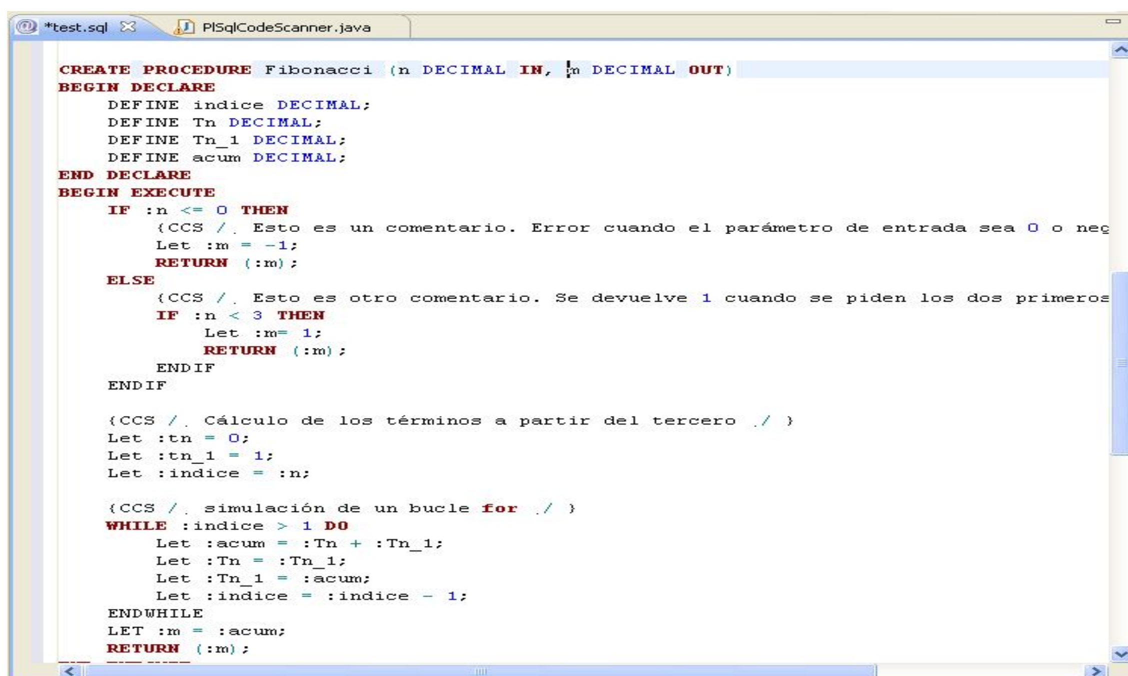


Ilustración 5. Ejemplo de un posible editor SPL

2.2.2. Definición de requerimientos

2.2.2.1. Requerimientos funcionales

El reto que se propone es el desarrollo de un nuevo elemento destacado en la plataforma *karat*; el *karat Editor y Analizador de lenguaje SPL y SQL*. Este componente será utilizado desde distintos sitios, por ejemplo en edición de procedimientos almacenados y disparadores en lenguaje *SPL*, Vistas, expresiones *SQL*, etc.

Esta herramienta debe ofrecer las siguientes funcionalidades:

- Aceleradores estándar de edición y búsqueda por código. Consistirá en las funcionalidades generales de edición, como copiar, cortar, pegar y buscar.
- El editor codificará con colores la sintaxis *SPL* y *SQL*. Consistirá en detectar palabras clave, comentarios, sentencias, etc. diferenciando los dos lenguajes, sabiendo en todo momento cual se está utilizando.
- Posibilidad de añadir plantillas o aceleradores para la creación de plantillas de *SPL*, *Disparadores*, etc. (del estilo del asistente de contenidos de *IDE Eclipse*, *Ctrl + Space)*. Consistirá en poder automatizar la programación añadiendo código, además de poder obtener ayudas al desarrollo e información adicional.
- Análisis de errores en las sentencias. Consistirá en la detección y señalización de errores para tener conocimiento de ellos y poder intentar corregirlos.
- Verificación de sintaxis *SPL* y *SQL* basada en los propios traductores *SPL* ya disponibles en la plataforma *karat*. Para el usuario de la herramienta la verificación de código será cuestión de pulsar el botón *Verificar*. Consistirá en traducir todas las sentencias del fichero *SPL* a código de la base de datos que se elija o que esté conectada.
- Posibilidad de enviar la consulta a una base de datos para su procesamiento a modo de *Analizador de consultas*. Consistirá en poder ejecutar las sentencias *SQL* para su análisis.
- Posibilidad de cambiar la configuración del editor. Consistirá en tener la opción de cambiar configuraciones del editor, como pueden ser los colores de señalización.
- Posibilidad de conexión del editor con el *Outline*. Consistirá en mostrar las diferentes funciones y variables que contengan los ficheros *SPL*.

2.2.3. Recursos

Software:

- Pc usuario:
    - Sistema operativo: Microsoft Windows XP Profesional o superior pero siempre un entorno profesional.
    - Navegador: Microsoft Internet Explorer.
- Entornos de programación:
    - Java, IDE Eclipse 3.4.0
    - Lenguajes de Base de datos: SPL, SQL.
- Generación de la documentación:
    - Microsoft Office 2007.

Hardware:

Recursos mínimos pc:

- Memoria RAM: 1 GB
- Procesador: Intel Core 2 Duo
- Disco duro: 120 GB
- DVD-ROM
- Monitor SVGA
- Tarjeta de red
- Teclado y ratón

Humanos:

En el proyecto participarán de forma distinta:

- Gerente de proyectos: Ezequiel Parra.
- Director de proyecto: Eduard Compte
- Analista: Montserrat Santacana
- Técnico de sistemas: Josep Miquel García
- Técnico programador: Daniel Dueñas
- Tutor en la universidad: Jordi Pons

2.2.4. Análisis coste-beneficio

El principal beneficio de este proyecto es la obtención de una nueva herramienta que conlleva una mejora considerable de los actuales editores de SPL y SQL, la cual facilitará y hará más cómoda su programación. Al mismo tiempo ayudará a mejorar la plataforma *karat*.

El coste de la realización de este proyecto consiste principalmente en el sueldo del técnico programador, en este caso el alumno de la universidad. Los recursos comentados anteriormente ya están disponibles en la empresa.

Los costes de personal, referente al alumno, es un convenio entre empresa y universidad, el cual consiste en 560 horas trabajadas por el alumno a un coste de aproximadamente 2.200 € Referente al personal comentado en recursos humanos, la analista conlleva un coste de aproximadamente 60 horas, horas que debería dedicar a formación, gestión del proyecto y evaluación del mismo. El resto no añaden ningún coste al proyecto, ya que este personal simplemente hace de apoyo al alumno. Los costes materiales vendrían dados por la utilización de un ordenador, características comentadas anteriormente, valorado en aproximadamente 1.000 €con una conexión a la red de Internet, conexión de la cual desconocemos su coste. Añadiríamos el coste de la licencia del programa utilizado, el *IDE Eclipse*, pero al tratarse de un programa de código libre no implica coste alguno. Para la realización de la documentación utilizamos Microsoft Office 2007, por lo tanto debemos de añadir como coste la licencia de este programa, pero desconocemos cuál es su coste. Otros costes a añadir serían los costes indirectos, como podrían ser agua, luz, etc. pero son costes de empresa que no se pueden averiguar.

2.2.5. Evaluación de riesgos

Este proyecto no tiene un elevado porcentaje de riesgos, ya que se trata de realizar una aplicación totalmente nueva e independiente. Los mayores riesgos pueden venir a la hora de no cumplir los plazos previstos para las fases planificadas y que esto provoque un ligero retraso del proyecto. Otro de los riesgos es la posibilidad de no tener el conocimiento suficiente para seguir con la aplicación y no poder finalizarla.

2.2.6. Alternativas

Las alternativas que pueden existir para este proyecto son las siguientes:

- Uso de algún componente externo que contenga lo que se quiere, este componente podría ser el editor PL/SQL, informado en el punto fuentes de información. Este editor contiene muchas de las funcionalidades que se quieren para el proyecto, pero es de otro lenguaje de programación. Al no existir ningún editor del lenguaje SPL, esta alternativa se descarta.

- Desarrollo del editor completamente independiente de cualquier plataforma y de cualquier entorno, desarrollando las diferentes funcionalidades para conseguir el objetivo. Esta alternativa se descarta debido a que implicaría un mayor trabajo y un mayor tiempo, ya que no se podría aprovechar la estructura *plug-in* de *Eclipse*, donde las funcionalidades más básicas ya se encuentran implementadas.

2.3. Modelo de desarrollo

Debido a la naturaleza específica de este proyecto se sugiere un modelo de proceso incremental o en espiral con el cual se pretende, mediante la consecución de hitos progresivos, el objetivo final. El modelo en espiral es un modelo de proceso de software evolutivo e iterativo.

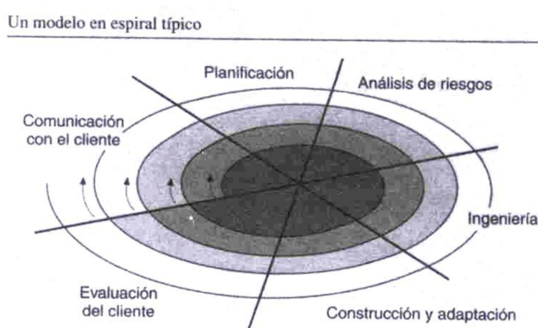


Ilustración 6. Modelo en espiral

Se proponen diversas iteraciones o fases alrededor de la espiral con entregas evolutivas. En cada vuelta el producto gana en madurez hasta que en una vuelta la evaluación lo apruebe y el bucle pueda abandonarse.

Antes de comenzar con este modelo, primeramente se realizarán las fases de:

- Formación
- Investigación
- Definición de requisitos

Realizadas estas fases pasamos a dividir el proyecto en fases más específicas:

- Una primera fase consistiría en la creación del *Plug-in Editor* en el *IDE Eclipse*, donde se crea la estructura general de un editor. Dentro de esta estructura encontramos un archivo llamado *Manifest*, en el cual se definen las características y las configuraciones del editor, las cuales vemos instanciadas en otro archivo llamado plugin.xml. La última parte de esta fase sería la construcción de la clase principal del editor.
- Una segunda fase consistiría en la implementación total de la funcionalidad colores. Realizar particiones del editor, particiones que traten zonas diferentes, como puede ser detección de comentarios, de *strings*, etc. Añadir palabras claves, sentencias, nombre de funciones que se quieran diferenciar de lo demás. Definir todas las reglas del lenguaje *SPL* para añadir a las particiones.
- Una tercera fase consistiría en la implementación de la funcionalidad *Content Assist (Ctrl + Space).* Crear las plantillas de código necesarias y más comunes en uso para poder seleccionarlas al presionar las teclas. Definir palabras claves, sentencias y nombres de funciones para ofrecer la ayuda a la programación como también la información de las funciones, parámetros de entrada, de salida, etc.
- Una cuarta fase consistiría en la implementación de las páginas de configuración, el botón de verificación de sentencias *SPL* y el envío de consultas a modo *Analizador de Consultas*.
- Una quinta fase consistiría en la implementación de la funcionalidad búsqueda avanzada y conexión con el *Outline*.
- Una sexta fase consistiría en la implementación de análisis de errores. Implementar la gramática *BNF* del lenguaje *SPL* creando un *parser* que analice el código desarrollado para detectar errores.

La división del proyecto en fases queda abierta para posibles cambios, como puede ser la suma de alguna funcionalidad más.

Por último acabaríamos el proyecto con la última fase, la redacción de la documentación.

2.4. Planificación

A continuación se muestra un desglose del proyecto por tareas a realizar:

1. Formación (30h)

1.1 Montaje del equipo informático e instalación y explicación de programas a usar (10h)

1.2 Definición inicial del proyecto y presentación de karat (10h)

1.3 Formación sobre SPL y plug-in Editor (10h)

2. Investigación (60h)

2.1 Sistema actual (10 h)

2.2 Investigación sobre el ejemplo Plug-in del Java Editor de Eclipse (10 h)

2.3 Investigación sobre el ejemplo Plug-in del XML Editor de Eclipse (10 h)

2.4 Investigación sobre el ejemplo PL/SQL Editor (20 h)

2.5 Investigación sobre el ejemplo Log4j Editor (10h)

3. Definición de requerimientos (30h)

3.1 Análisis del problema (10h)

3.2 Especificación de requerimientos funcionales y no funcionales (10h)

3.3 Planificación del proyecto (5h)

3.4 Corrección del documento (5h)

4. Fase 1: Creación del Plug-in Editor

4.1 Análisis (6h)

4.2 Codificación (31h)

4.3 Pruebas (4h)

5. Fase 2: Syntax coloring

5.1 Análisis (12h)

5.2 Codificación (62h)

5.3 Pruebas (8h)

6. Fase 3: Content Assist

6.1 Análisis (12h)

6.2 Codificación (62h)

6.3 Pruebas (8h)

7. Fase 4: Páginas de configuración, verificación y envíos de consulta

7.1 Análisis (6h)

7.2 Codificación (31h)

7.3 Pruebas (4h)

8. Fase 5: Búsqueda avanzada y Outline

8.1 Análisis (6h)

8.2 Codificación (31h)

8.3 Pruebas (4h)

9. Fase 6: Análisis de errores

9.1 Análisis (18h)

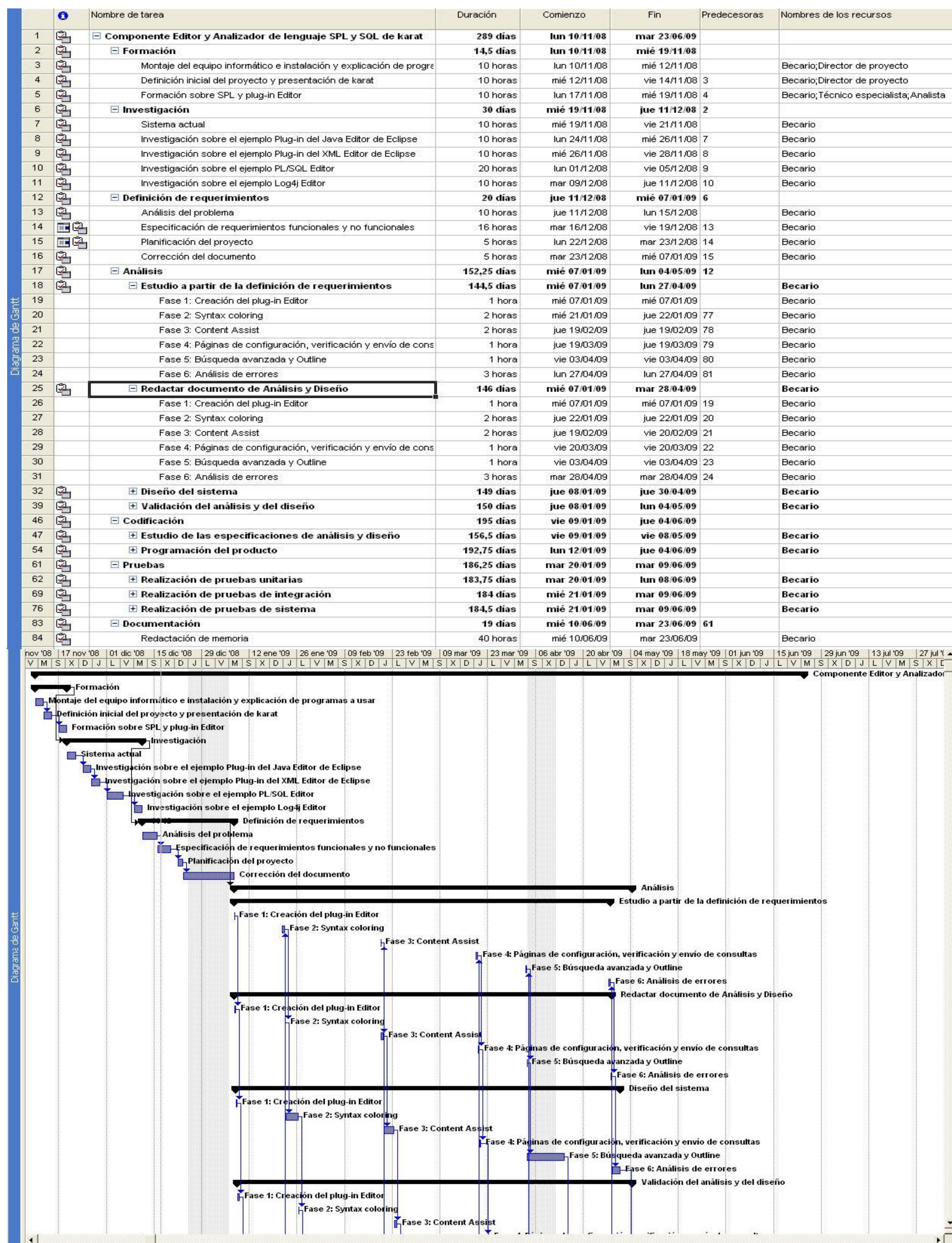
9.2 Codificación (93h)

9.3 Pruebas (12h)

10. Documentación (30h)

Fase de desarrollo	Tiempo estimado de realización(en horas)
Formación	30
Investigación	60
Definición de requerimientos	30
Análisis	60
Codificación	310
Pruebas	40
Documentación	30
Total	**560**

A continuación vemos el diagrama de Gantt, donde se especifican las fechas previstas de inicio y finalización, y los recursos necesarios para el desarrollo de las distintas etapas.



	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras	Nombres de los recursos
1	**Componente Editor y Analizador de lenguaje SPL y SQL de karat**	**289 días**	**lun 10/11/08**	**mar 23/06/09**		
2	**Formación**	**14,5 días**	**lun 10/11/08**	**mié 19/11/08**		
3	Montaje del equipo informático e instalación y explicación de progra	10 horas	lun 10/11/08	mié 12/11/08		Becario;Director de proyecto
4	Definición inicial del proyecto y presentación de karat	10 horas	mié 12/11/08	vie 14/11/08	3	Becario;Director de proyecto
5	Formación sobre SPL y plug-in Editor	10 horas	lun 17/11/08	mié 19/11/08	4	Becario;Técnico especialista;Analista
6	**Investigación**	**30 días**	**mié 19/11/08**	**jue 11/12/08**	**2**	
7	Sistema actual	10 horas	mié 19/11/08	vie 21/11/08		Becario
8	Investigación sobre el ejemplo Plug-in del Java Editor de Eclipse	10 horas	lun 24/11/08	mié 26/11/08	7	Becario
9	Investigación sobre el ejemplo Plug-in del XML Editor de Eclipse	10 horas	mié 26/11/08	vie 28/11/08	8	Becario
10	Investigación sobre el ejemplo PL/SQL Editor	20 horas	lun 01/12/08	vie 05/12/08	9	Becario
11	Investigación sobre el ejemplo Log4j Editor	10 horas	mar 09/12/08	jue 11/12/08	10	Becario
12	**Definición de requerimientos**	**20 días**	**jue 11/12/08**	**mié 07/01/09**	**6**	
13	Análisis del problema	10 horas	jue 11/12/08	lun 15/12/08		Becario
14	Especificación de requerimientos funcionales y no funcionales	16 horas	mar 16/12/08	vie 19/12/08	13	Becario
15	Planificación del proyecto	5 horas	lun 22/12/08	mar 23/12/08	14	Becario
16	Corrección del documento	5 horas	mar 23/12/08	mié 07/01/09	15	Becario
17	**Análisis**	**152,25 días**	**mié 07/01/09**	**lun 04/05/09**	**12**	
18	**Estudio a partir de la definición de requerimientos**	**144,5 días**	**mié 07/01/09**	**lun 27/04/09**		**Becario**
19	Fase 1: Creación del plug-in Editor	1 hora	mié 07/01/09	mié 07/01/09		Becario
20	Fase 2: Syntax coloring	2 horas	mié 21/01/09	jue 22/01/09	77	Becario
21	Fase 3: Content Assist	2 horas	jue 19/02/09	jue 19/02/09	78	Becario
22	Fase 4: Páginas de configuración, verificación y envío de cons	1 hora	jue 19/03/09	jue 19/03/09	79	Becario
23	Fase 5: Búsqueda avanzada y Outline	1 hora	vie 03/04/09	vie 03/04/09	80	Becario
24	Fase 6: Análisis de errores	3 horas	lun 27/04/09	lun 27/04/09	81	Becario
25	**Redactar documento de Análisis y Diseño**	**146 días**	**mié 07/01/09**	**mar 28/04/09**		**Becario**
26	Fase 1: Creación del plug-in Editor	1 hora	mié 07/01/09	mié 07/01/09	19	Becario
27	Fase 2: Syntax coloring	2 horas	jue 22/01/09	jue 22/01/09	20	Becario
28	Fase 3: Content Assist	2 horas	jue 19/02/09	vie 20/02/09	21	Becario
29	Fase 4: Páginas de configuración, verificación y envío de cons	1 hora	vie 20/03/09	vie 20/03/09	22	Becario
30	Fase 5: Búsqueda avanzada y Outline	1 hora	vie 03/04/09	vie 03/04/09	23	Becario
31	Fase 6: Análisis de errores	3 horas	mar 28/04/09	mar 28/04/09	24	Becario
32	**Diseño del sistema**	**149 días**	**jue 08/01/09**	**jue 30/04/09**		**Becario**
39	**Validación del análisis y del diseño**	**150 días**	**jue 08/01/09**	**lun 04/05/09**		**Becario**
46	**Codificación**	**195 días**	**vie 09/01/09**	**jue 04/06/09**		
47	**Estudio de las especificaciones de análisis y diseño**	**156,5 días**	**vie 09/01/09**	**vie 08/05/09**		**Becario**
54	**Programación del producto**	**192,75 días**	**lun 12/01/09**	**jue 04/06/09**		**Becario**
61	**Pruebas**	**186,25 días**	**mar 20/01/09**	**mar 09/06/09**		
62	**Realización de pruebas unitarias**	**183,75 días**	**mar 20/01/09**	**lun 08/06/09**		**Becario**
69	**Realización de pruebas de integración**	**184 días**	**mié 21/01/09**	**mar 09/06/09**		**Becario**
76	**Realización de pruebas de sistema**	**184,5 días**	**mié 21/01/09**	**mar 09/06/09**		**Becario**
83	**Documentación**	**19 días**	**mié 10/06/09**	**mar 23/06/09**	**61**	
84	Redactación de memoria	40 horas	mié 10/06/09	mar 23/06/09		Becario

Ilustración 7. Planificación y diagrama de Gantt

CAPÍTULO 3: ANÁLISIS Y DISEÑO

Se exponen los diagramas de contexto y de clases, con una explicación de cada uno de los componentes que aparecen para comprender el funcionamiento del sistema. También se especifica el funcionamiento de las diferentes operaciones definidas en las fases.

3.1. Visión general

3.1.1. Diagrama de contexto

En los dos siguientes apartados veremos un diagrama de alto nivel del proyecto, el cual debe reflejar claramente mediante componentes la arquitectura tecnológica, la dependencia entre módulos y las formas de interacción que los une.

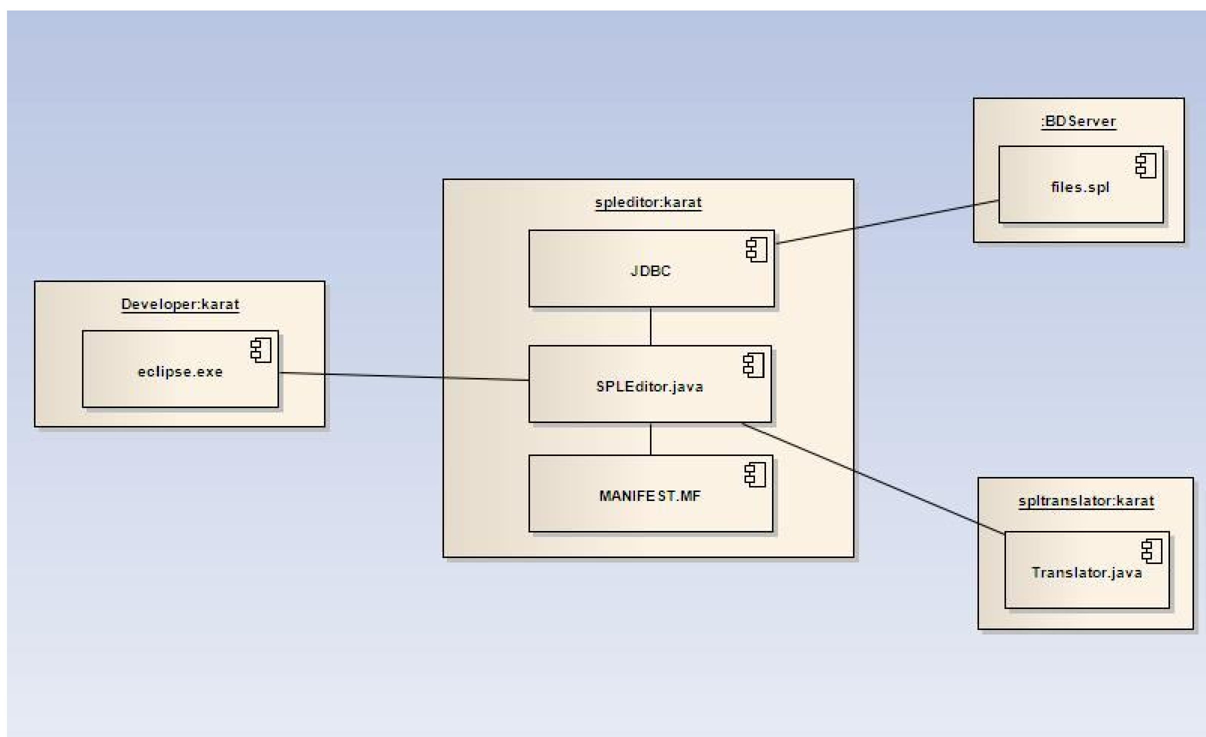


Ilustración 8. Diagrama de contexto

3.1.2. Descripción de los componentes

Developer:karat: Como su propio nombre indica, este componente hace referencia a los ordenadores de la empresa, los cuales contendrán la herramienta de trabajo *IDE Eclipse* para que los técnicos puedan trabajar con el sistema realizado.

spleditor:karat: El sistema a realizar estará integrado dentro del producto *karat* y como podemos ver en el diagrama, *karat* contiene diferentes *plug-ins* que son conectados con *IDE Eclipse*. Uno de ellos será nuestro editor, por lo que podremos acceder a el desde *IDE Eclipse*.

Nuestro editor, para su correcto funcionamiento, necesita el archivo *MANIFEST.MF*, un archivo que describe la información general del *plug-in*, donde se informa de las conexiones con otros *plug-ins* y con extensiones. Estos tipos de archivo son los primeros que analiza el *IDE Eclipse* a la hora de comenzar la aplicación.

El producto *karat*, mediante *IDE Eclipse*, puede realizar conexiones a sus servidores. Para nuestro editor se necesita una conexión a la base de datos que contiene los archivos spl, para que los técnicos tengan la posibilidad de consultar o modificar dichos archivos.

:BDServer: Este componente es la base de datos a la que se accede para conseguir los diferentes archivos spl existentes.

spltranslator:karat: Este componente es una herramienta integrada dentro del producto *karat* cuya función es la de traducir código *SPL* a código de diferentes bases de datos, por lo que nuestro editor tendrá una conexión directa para poder traducir el código editado.

3.1.2.1. SPL (Stored Procedure Language)

SPL es el acrónimo de *Stored Procedure Language* (en castellano, lenguaje de procedimientos almacenados). Un *stored procedure* es un procedimiento almacenado en la base de datos, escrito en un lenguaje procedural de alto nivel que proporciona la misma base de datos y que permite la interrogación y manipulación de datos de una forma fácil y rápida.

Este lenguaje nace a partir de los lenguajes de procedimientos de las diferentes bases de datos. La empresa trabaja con varias bases de datos y, por lo tanto, realizar procedimientos almacenados en cada una de ellas se convierte en una tarea complicada para los desarrolladores. Por ello se optó por un lenguaje que englobara a todas las bases de datos y así construir los procedimientos almacenados en un solo lenguaje, con la posibilidad de traducir este código al lenguaje de la base de datos deseada en cualquier momento.

3.1.2.2. IDE Eclipse

Se trata de un Entorno Integrado de Desarrollo (*IDE*) abierto y extensible, para cualquier cosa y nada en particular. Su uso más popular es como un *IDE* para *Java*, pero *Eclipse* es adaptable a cualquier tipo de lenguaje. La característica clave de *Eclipse* es la extensibilidad. *IDE Eclipse* es una gran estructura formada por un núcleo y muchos plug-ins que van conformando la funcionalidad final.

El *IDE Eclipse* emplea módulos (en inglés *plug-in*) para proporcionar toda su funcionalidad al frente de la plataforma, a diferencia de otros entornos monolíticos donde la funcionalidad está toda incluida, la necesite el usuario o no. Este mecanismo de módulos es una plataforma ligera para componentes de software.

3.1.2.3. Editor SPL

El proyecto consiste en realizar, para el producto *karat,* un componente editor y analizador de los lenguajes de programación *SPL* y *SQL* y que deberá tener el formato de un *Plug-in IDE Eclipse*.

Dentro de *Eclipse* existen diferentes plantillas para la implementación de un plug-in y entre ellas encontramos una que consiste en un editor, concretamente un editor del lenguaje *XML*. Este editor tiene implementado las funciones básicas. (cortar, copiar, pegar, buscar, etc.) Además tiene la funcionalidad de codificación de sintaxis en colores. Por lo tanto este editor será la estructura base a seguir para conseguir la estructura general del editor que se pretende construir.

El uso de esta plantilla nos proporciona un gran valor para poder comenzar, ya que nos da el esqueleto de un editor, con las clases más importantes e imprescindibles para su funcionamiento. Estas clases nos permiten poder abrir el editor, es decir, poder ver la ventana en la cual editar código además de darnos funciones básicas y típicas de edición.

Con el esqueleto del sistema, podemos centrarnos en las diferentes funcionalidades a implementar, funcionalidades que se pueden leer en el documento de requerimientos. Como hemos comentado en el párrafo anterior la plantilla ya nos proporciona las funcionalidades básicas de un editor, por lo tanto debemos pensar y analizar el resto.

La primera será la codificación con colores de la sintaxis *SPL* y *SQL*, que consiste en detectar las distintas palabras claves de los lenguajes y darles un color específico y exclusivo para diferenciar distintas familias de palabras claves.

La siguiente funcionalidad a analizar sería la del asistente de contenidos (*Content Assist*), que mediante la pulsación de las teclas *Ctrl+Space* podamos obtener ayuda en una ventana automática sobre los lenguajes mencionados. La ayuda proporcionada trataría sobre plantillas de código, funciones y palabras claves, además de posibles informaciones con una ventana adicional.

Otra de las funcionalidades a realizar son las diferentes páginas de preferencias para poder configurar el editor, como sería el cambio de colores, la creación de plantillas o modificación de las ya creadas o la posibilidad de activar alguna opción. También se debería poder enviar el código editado al *Traductor de SPL*, para poder obtener el código de la base de datos que deseemos y mostrarlo.

3.2. Modelo de objetos

3.2.1. Diagrama de clases

En este apartado veremos un diagrama de clases, diagrama que se va ampliando en el comienzo de cada una de las fases que vimos en el documento de requerimientos. Este diagrama pretende mostrar las clases relevantes que forman el conjunto de la aplicación y las relaciones entre ellas.

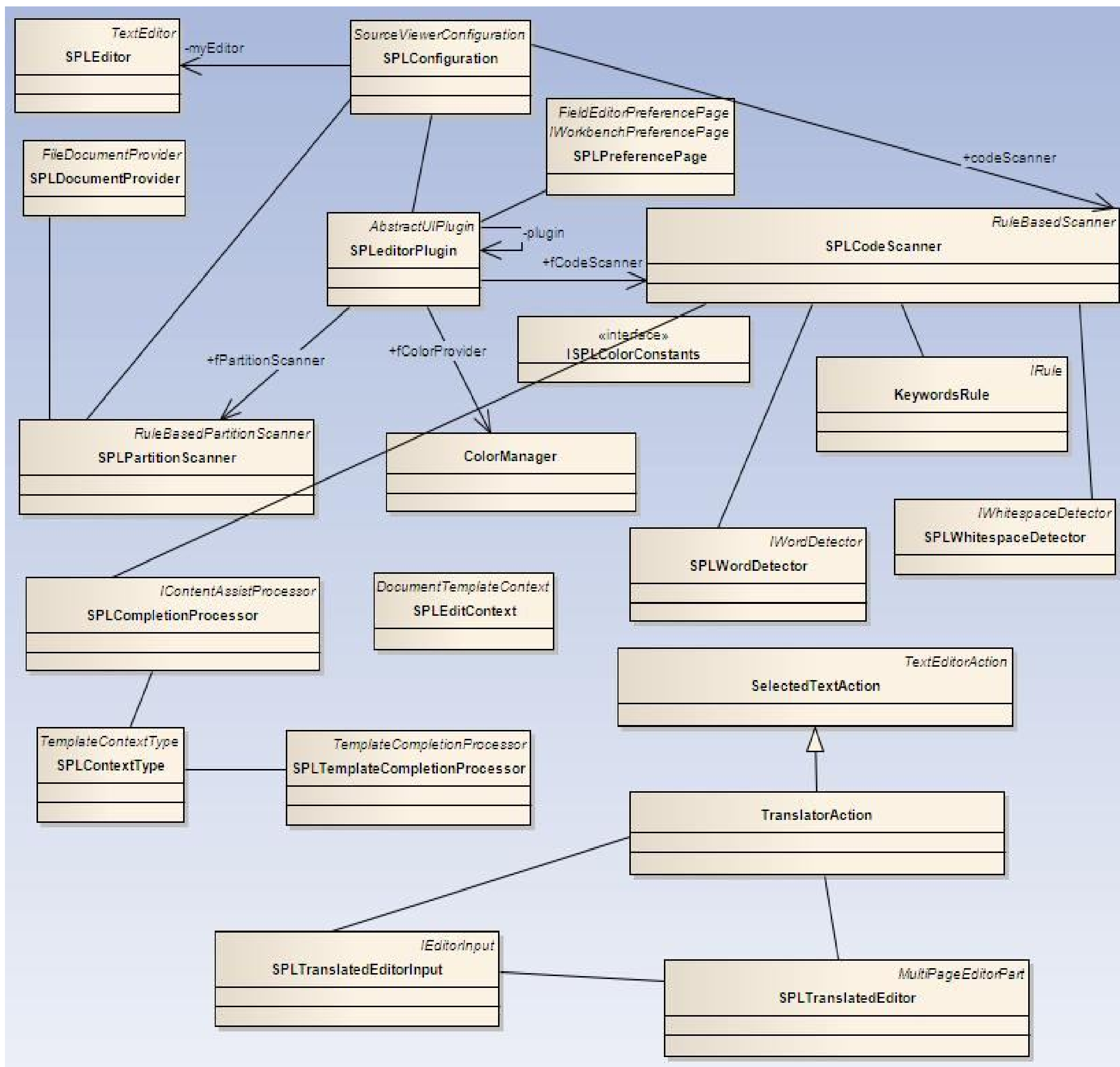


Ilustración 9. Diagrama de clases

3.2.2. Descripción de componentes

3.2.2.1. Creación del Plug-in Editor

La creación del *plug-in* editor corresponde a la primera fase del proyecto. El objetivo de esta primera fase es conseguir la estructura inicial y los aceleradores estándar de edición para poder comenzar a funcionar el editor.

Con la implementación de las cuatro clases que describimos en los siguientes párrafos junto a la construcción del archivo *MANIFEST.MF*, un archivo característico de los *plug-ins* de *IDE Eclipse*, podremos conseguir el objetivo.

La implementación del archivo *MANIFEST.MF* consiste en realizar la configuración del *plug-in* editor para la integración con *IDE Eclipse*, donde por ejemplo podemos darle un nombre al editor. Se trata de relacionar este *plug-in* editor con otros de *Eclipse* para conseguir una cohesión y un funcionamiento correcto además de poder utilizar clases e interfaces que las clases de nuestro editor extienden o implementan para conseguir las funcionalidades comentadas.

Con la implementación de las clases, descritas a continuación, buscamos realizar la configuración del editor y así poder comenzar a trabajar sobre él. Estas clases se encuentran implementadas dentro de la plantilla *XML Editor* que hemos comentado en el capítulo anterior. Nuestra misión es construir nuestras clases haciendo servir como modelo las del *XML Editor*. Con esta implementación básica podemos utilizar los aceleradores y conforme se van consiguiendo fases, estas cuatro clases se irán modificando, configurando y conectando con otras clases para que las funcionalidades desarrolladas puedan funcionar.

SPLEditor: Esta clase extiende de la clase TextEditor, clase que nos crea la interfaz del editor y nos permite su visualización, su edición y el uso de los aceleradores estándar de edición. La clase SPLEditor es la principal de nuestro editor, es donde se inicializa el editor y donde podemos crear las diferentes acciones que pueda contener el menú del editor.

<u>SPLConfiguration</u>: Esta clase extiende de la clase SourceViewerConfiguration, clase que configura la vista del editor, es decir, configura la clase SPLEditor. En la clase SPLConfiguration configuramos el editor declarando el Content Assist para poder utilizarlo y las diferentes particiones de texto que debe detectar para más adelante poder configurar estas particiones con colores. Estas particiones están definidas en la clase SPLPartitionScanner.

<u>SPLDocumentProvider</u>: Esta clase extiende de la clase FileDocumentProvider, clase que tiene la misión de crear el editor como documento para poder editar código. En la clase SPLDocumentProvider creamos el documento indicándole las diferentes particiones de texto.

<u>SPLeditorPlugin</u>: Esta clase extiende de la clase AbstractUIPlugin, clase base abstracta para los plug-ins que se integran con la plataforma de interfaz de usuario de Eclipse. En la clase SPLeditorPlugin establecemos conexión con la interfaz del editor mediante las clases que son configuradas por la clase SPLConfiguration.

3.2.2.2. Syntax coloring

Syntax coloring es una de las funcionalidades principales del Editor *SPL* y corresponde a la segunda fase del proyecto. El objetivo de esta fase es la de implementar una codificación con colores de la sintaxis *SPL* y *SQL*.

Syntax coloring consiste en la detección de palabras clave, comentarios de código, sentencias, strings, etc. y darles un color específico para diferenciar los dos lenguajes, sabiendo en todo momento cual se está utilizando.

Para conseguir esta funcionalidad se deben detectar diferentes particiones en el editor. Estas particiones consisten en detectar zonas de código que se diferencian del resto. Debemos de distinguir cuatro particiones distintas que son: comentarios de código, strings, sentencias y la partición por defecto. Estas particiones tienen un color asignado cada una, excepto la de defecto, donde debemos señalar las diferentes familias de palabras clave que se quieren detectar. Las familias que se quieren diferenciar son,

palabras del lenguaje *SPL*, palabras del lenguaje *SQL*, operadores, tipos de datos del lenguaje *SPL*, tipos de datos del lenguaje *SQL*, funciones y las constantes *false, true* y *null.* Cada una de estas familias también tiene un color específico asignado.

El proceso de detectar las diferentes particiones y familias para cambiar el color se basa en implementación de reglas para ello. A continuación se explican las clases principales para esta fase, donde se acabará de entender su funcionamiento.

ColorManager: Esta clase es la encargada de la manipulación de los colores definidos, es una clase gestora de colores.

ISPLColorConstants: Clase donde se definen los colores mediante sus códigos RGB.

SPLCodeScanner: Esta clase extiende de la clase RuleBasedScanner, clase que permite la programación de un escáner genérico mediante reglas o normas. En la clase SPLCodeScanner definimos las diferentes palabras claves que el escáner debe detectar. Además se realiza la programación del escáner con diferentes reglas para la detección de palabras y así poder dar color a éstas. Tres reglas serían las clases SPLWhiteSpaceDeteactor, SPLWordDetector y KeywordsRule.

SPLPartitionScanner: Esta clase extiende de la clase RuleBasedPartitionScanner, clase que permite la programación del escáner mediante particiones de edición. En la clase SPLPartitionScanner definimos las diferentes partes que puede contener el editor, es decir, las diferentes partes de edición. De esta manera programamos el escáner para tener más detalle de las reglas que debe seguir a la hora de la detección de palabras clave.

Ahora el escáner está programado con diferentes partes de edición y dentro de cada parte está programado con las reglas que hemos definido en la clase SPLCodeScanner.

SPLWhiteSpaceDetector: Esta clase implementa la interfaz IWhitespaceDetector, que determina si un carácter se considera en blanco en el contexto. Dentro de la clase SPLWhiteSpaceDetector definimos cuatro caracteres que se consideran en blanco.

SPLWordDetector: Esta clase implementa la interfaz IWordDetector, que determina si un carácter es válido como parte de una palabra en el contexto. Dentro de la clase SPLWordDetector se definen dos métodos para la detección del carácter leído como parte de una palabra o como primer carácter de una palabra.

KeywordsRule: Esta clase implementa la interfaz IRule, que define la interfaz para una norma utilizada en el escaneo de texto para el propósito del documento particionado o el estilo de texto. Dentro de la clase KeywordsRule se determina dicha interfaz, es decir, se determina la regla para la detección de palabras clave y su posterior cambio al color correspondiente. Esta regla es una con la que programamos el escáner que consiste en leer caracteres y comprobar que éstos forman parte de las palabras clave definidas en la clase SPLCodeScanner.

3.2.2.3. Content assist

Otra de las grandes funcionalidades del Editor *SPL* es el Content Assist, correspondiente a la tercera fase del proyecto. El objetivo de esta fase es ofrecer ayuda y automatizar la programación de código.

Content assist se activa mediante la pulsación de las teclas *Ctrl+Space*, apareciendo una ventana de contenidos para ayudar al desarrollador a la programación de código de una manera más eficiente, segura y rápida.

Dentro de los contenidos podemos encontrar las diferentes palabras clave definidas en la fase anterior, funciones con su correspondiente ayuda de información apareciendo en una ventana adicional y plantillas de código. A continuación se explican las clases principales para esta fase, donde se acabará de entender su funcionamiento.

SPLCompletionProcessor: Esta clase implementa la interfaz ICompletionProcessor, un procesador de contenido que propone ayudar con complementos e informar para un determinado tipo de contenido. Esta interfaz debe ser ejecutada por los usuarios. Dentro de la clase SPLCompletionProcessor se determinan las diferentes funciones, palabras clave y plantillas de código que se mostrarán en la ventana de ayuda para poder utilizar o informar.

SPLContextType: Esta clase extiende de la clase TemplateContextType, clase donde un contexto define un tipo de contexto en el que las plantillas se hayan resuelto. La clase SPLContextType crea el contexto para SPL.

SPLEditContext: Esta clase extiende de la clase DocumentTemplateContext, clase que describe el contexto de una plantilla como una región de un documento. La clase SPLEditContext crea un nuevo contexto de edición con un tipo y una ubicación en un documento.

SPLTemplateCompletionProcessor: Esta clase extiende de la clase TemplateCompletionProcessor, clase que define un procesador para completar plantillas. En la clase SPLTemplateCompletionProcessor se define el procesador para plantillas SPL, se crea la imagen y la información que se mostrará en la ventana del Content Assist.

3.2.2.4. Páginas de preferencias

Otra funcionalidad importante para el desarrollo del Editor *SPL* son las páginas de preferencias y pertenecen a la cuarta fase del proyecto. El objetivo que se busca con su implementación es la de poder realizar una configuración del editor al gusto del desarrollador.

Las páginas de preferencias ayudan a configurar el editor. En nuestro caso realizamos tres páginas distintas, que son:

- SPL Preferences: Esta página será la principal de la configuración del editor. Su función será la de configurar los colores que se visualizarán en los diferentes grupos de palabras clave a detectar. Se podrá seleccionar el color que se desee.

- Code Templates: Esta página contendrá las plantillas de código definidas en el Content Assist y se dará la oportunidad de modificarlas o eliminarlas. También se podrán crear nuevas plantillas, importarlas o exportarlas. La importación y exportación de plantillas se realizará en formato *XML*.

- Formatting/Translation: Esta página tendrá dos partes, la primera contendrá opciones que añaden más funcionalidad al editor y la segunda opciones sobre la traducción de código.

  En primera parte se tendrán dos opciones: la primera será sobre el Content Assist y consistirá en autocompletar una palabra clave si es la única opción a elegir. La segunda será poder elegir que el editor pase las palabras claves a minúsculas, ya que por defecto cuando se detecta una palabra clave, ésta se pasará a mayúscula.

  En la segunda parte se tendrán varias opciones a elegir:

  - *SQL PL - DB2 SQL Procedural Language*
  - *SQL PL - DB2 SQL (AS/400)*
  - *COS - Caché Object Script by InterSystems.*
  - *T-SQL - Microsoft Transact-SQL*
  - *PL/SQL - Oracle Procedural Language by Oracle*
  - *MySQL*
  - *PL/pgSQL - PostgreSQL Procedural Language de PostgreSQL*

  Estas opciones representan los diferentes lenguajes de bases de datos con los que trabaja la empresa. La selección de estas opciones permitirá saber a qué lenguajes se debe traducir el código *SPL* cuando se active la acción de traducción.

A continuación se explica la clase principal para esta fase, donde se acabará de entender el funcionamiento.

SPLPreferencePage: Esta clase extiende de la clase FieldEditorPreferencePage e implementa la interfaz IWorkbenchPreferencePage. Estas clases implementan la interfaz de la ventana de la página integrándola en *IDE Eclipse* y ayudan a su construcción. Estas clases ayudan a crear la clase SPLPreferencePage, la cual es una página de preferencias. En esta clase, por lo tanto se configura la página con texto y

parámetros que se deseen que aparezcan. Por lo tanto, por cada página a crear, se deberá crear una clase que extienda e implemente las clases nombradas.

Se implementarán dos páginas más de preferencias, pero no hemos puesto sus clases porque siguen el mismo modelo que la clase SPLPreferencePage y el contenido de las páginas que se crearán ya lo hemos explicado anteriormente.

3.2.2.5. Verificación de sentencias SPL

Una funcionalidad más, es la verificación de código en el traductor de *SPL*. Esta funcionalidad corresponde a la quinta fase del proyecto. El objetivo de esta fase es realizar una validación de la sintaxis y ofrecer la posibilidad de obtener el código nativo de la base de datos que se desee a partir del código *SPL* generado.

El Editor *SPL* tendrá la opción de la traducción, pero realmente la traducción no se realiza en este sistema, sino que se realiza en otra herramienta llamada *SPL Translator* del producto *karat*. Por lo tanto el proceso de traducción consiste en enviar el código del editor a esta herramienta y devolver el código nativo para poder mostrarlo en el Editor SPL en una nueva pestaña. Se realizarán diferentes traducciones, una por cada lenguaje de base de datos seleccionado en la página de preferencias, y se mostrará cada una en pestañas diferentes dentro del editor de traducción.

Esta herramienta también tiene la función de validar la sintaxis *SPL*. Por lo tanto si el código que se envía a traducir contiene errores, no se realiza la traducción y se indica que la sintaxis no es válida y el error o los errores que puede contener. Así que el Editor SPL mostrará también lo que indique la herramienta *Translator SPL*.

Para poder utilizar la herramienta *SPL Translator* se necesitan los archivos *karat-server.jar* y *karat-base.jar*. Estos archivos contienen las clases necesarias para el funcionamiento de la herramienta, pero también son necesarios para poder utilizar la herramienta desde el Editor *SPL*, ya que para poder enviar el código a traducir, se deberá de realizar una llamada a la clase *Translator.class* de estos archivos. A continuación se adjunta el código necesario para realizar la conexión con esta herramienta y una pequeña explicación para entender el funcionamiento:

```
    String translatedCode = "";

    String inCode = "";


    Translator TSpl;

    TSpl = new Translator();


    TSpl.setDataBaseType(DA.DA_DI_SQLSERVER);

    TSpl.setInCode(inCode);


    try {

        TSpl.translate();

    } catch (Exception e) {

        e.printStackTrace();

    }


  translatedCode = TSpl.getOutCode();
```

El string *translatedCode* es la variable que contendrá el código traducido y la variable *inCode* será el código del editor a traducir. Se declara una variable *TSpl* del objeto *Translator* y a continuación se inicializa.

Con el método *setDataBaseType* indicamos la base de datos para así traducir el código al lenguaje correspondiente. Con el método *setInCode* le pasamos el código *SPL* para traducir y el método *translate* es el encargado de realizar la traducción. Finalmente obtenemos el código nativo con el método *getOutCode*.

A continuación se explican las clases principales para esta fase, donde se acabará de entender su funcionamiento.

SelectedTextAction: Esta clase extiende de la clase TextEditorAction, clase que proporciona el esqueleto de un editor de textos estándar de acción. La acción es asociada inicialmente con un editor de texto a través del constructor. La clase SelectedTextAction nos permite poder realizar acciones sobre el texto seleccionado en el editor.

TranslatorAction: Esta clase extiende de la clase SelectedTextAction, que es la clase comentada anteriormente. En la clase TranslatorAction se define la acción completa de traducir código SPL al código de la base de datos que se desee. Por lo tanto en esta clase se realiza una llamada a la herramienta Translator de *karat* para pasarle el código SPL y que esta herramienta nos devuelva el código traducido y así poder mostrarlo.

SPLTranslatedEditor: Esta clase extiende de la clase MultiPageEditorPart, clase que consiste en un editor de varias páginas, cada una de las cuales puede contener un editor o un control arbitrario. En la clase SPLTranslatedEditor se define la creación de varias páginas, ya que la traducción de código se puede realizar en varios lenguajes de bases de datos y por lo tanto cada página podrá contener un lenguaje traducido diferente.

SPLTranslatedEditorInput: Esta clase implementa la interfaz IEditorInput. IEditorInput es un ligero editor de descripción de entrada, como un nombre de archivo, pero más abstracto. No se trata de un modelo, sino de una descripción de la fuente de un modelo. En la clase SPLTranslatedEditorInput se realiza el mantenimiento de cada uno de los lenguajes de las bases de datos.

3.3. Especificaciones de los procesos

En el siguiente apartado se puede ver el diseño y la especificación de cada una de las funcionalidades que contiene este editor: syntax coloring, content assist, *SPL* preferences y *SPL* editor options.

En la siguiente ilustración se puede ver la creación del *plug-in* con la interfaz del Editor *SPL*, totalmente integrado en *IDE Eclipse*.

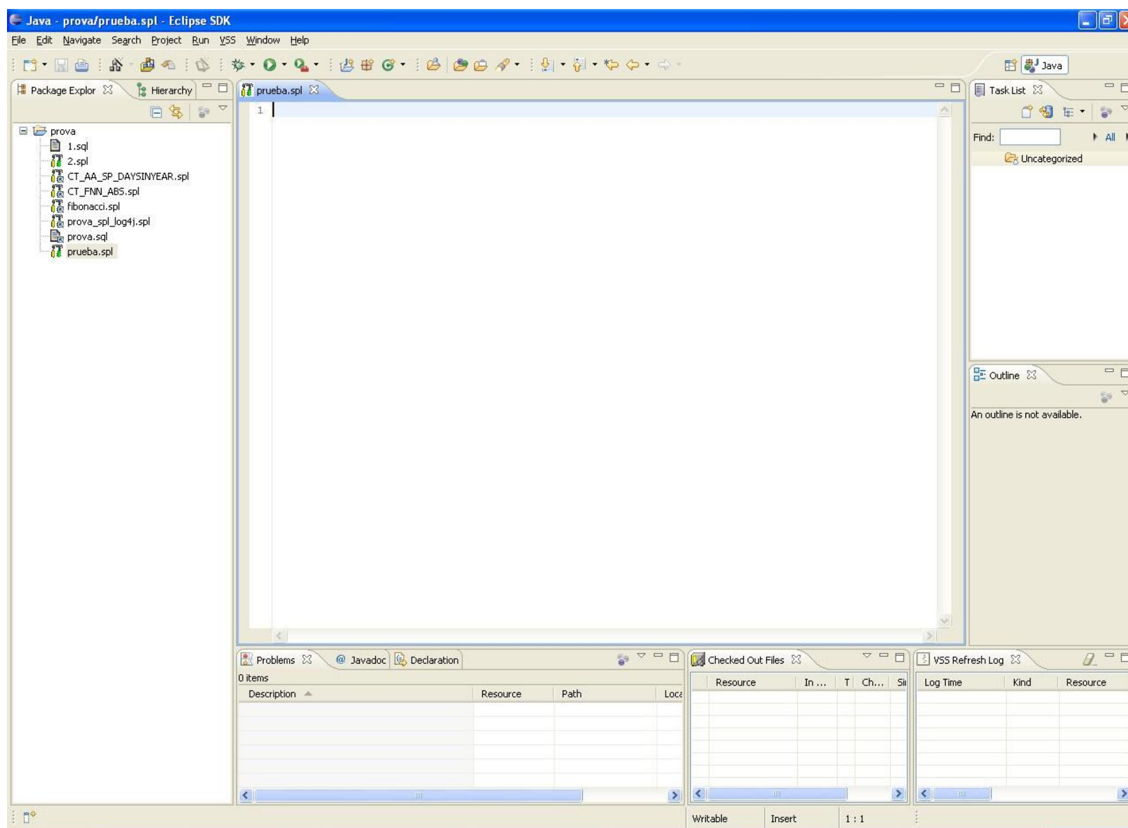


Ilustración 10. Editor SPL

3.3.1. Syntax coloring

El objetivo de esta funcionalidad es la detección de palabras clave, comentarios de código, sentencias, *strings*, etc. y darles a cada uno un color específico para diferenciarlos.

Para esta detección se han definido cuatro particiones: comentarios de código, strings, sentencias “LET” y la partición por defecto. En la partición por defecto se han definido siete familias con las siguientes palabras clave:

- Palabras del lenguaje *SPL*: *AFTER, ALL, AS, ASC, BEFORE, BEGIN, BLOCK, CLOSE, COMMIT, COMMITTED, CONNECT, CONTINUE, CREATE, CURRENT, DECLARE, DEFAULT, DEFINE, DESC, DO, EACH, ELSE, ELSIF, END, ENDIF, ENDFOREACH, ENDWHILE, EXECUTE, EXCEPTION, EXCLUSIVE, EXECPROC, EXIT, FETCH, FOR, FOREACH, FUNCTION, GOTO, IF, IN, INDEX, INNER, INTERSECT, IS, ISOLATION, LET, LEVEL, LOCK, LOOP, MINUS, MODE, NEW, NO_DATA_FOUND, NOT, OF, OLD, ON, OPEN, OTHERS, OUT, PROCEDURE, RAISE, READ, REFERENCING, RETURN, REVERSE, ROLLBACK, ROW, SAVEPOINT, SET, SHARE, START, TABLE, THEN, TO, TOP, TRANSACTION, TRIGGER, UNCOMMITTED, VALUES, WHEN, WHILE, WITH, WORK.*

- Palabras del lenguaje *SQL*: *AND, ANY, BETWEEN, BY, CASE, DELETE, DISTINCT, EXISTS, FROM, GROUP, HAVING, INTO, INSERT, JOIN, LIKE, OR, ORDER, OUTER, SELECT, UNION, UPDATE, WHERE.*

- Operadores: (,), !=, %, &, *, **, +, -, /, :=, <, <=, <>, =, >, >=, ||, \

- Tipos de datos del lenguaje *SPL*: *CHAR(n), DATE, DECIMAL, DECIMAL(p,s), FLOAT, INTEGER, REAL, SMALLINT, TIMESTAMP, VARCHAR(n).*

- Tipos de datos del lenguaje *SQL*: *SQL_BIGINT, SQL_BIT, SQL_CHAR, SQL_DATE, SQL_DECIMAL, SQL_DOUBLE, SQL_FLOAT, SQL_INTEGER, SQL_NUMERIC, SQL_REAL, SQL_SMALLINT, SQL_TIME, SQL_TIMESTAMP, SQL_TINYINT, SQL_VARCHAR, SQL_TSI_DAY, SQL_TSI_HOUR, SQL_TSI_MINUTE, SQL_TSI_FRAC_SECOND, SQL_TSI_SECOND, SQL_TSI_WEEK, SQL_TSI_MONTH, SQL_TSI_QUARTER, SQL_TSI_YEAR.*

- Funciones: *ABS, ACOS, ASCII, ASIN, ATAN, ATAN2, AVG, CALL, CAST, CEILING, CONCAT, CONVERT, COS, COT, COUNT, CURDATE, CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, CURTIME, DATABASE, DAYNAME, DAYOFMONTH, DAYOFWEEK, DAYOFYEAR, DEGREES, DIFFERENCE, EXP, FLOOR, FN, GETDATE, HOUR, IFNULL, INSERT, LCASE, LEFT, LENGTH, LOCATE, LOG, LOG10, LTRIM, MAX, MIN, MINUTE, MOD, MONTH, MONTHNAME, NOW, ODBC, OJ, PI, POWER, PRODUCT, QUARTER, RADIANS, RAND, REPEAT, REPLACE, RIGHT, ROUND, RTRIM, SECOND, SIGN, SIN, SOUNDEX, SPACE, SQRT, SUBSTRING, SUM, TAN, TIMESTAMPADD, TIMESTAMPDIFF, TRUNCATE, UCASE, USER, VENDOR, YEAR, WEEK.*

- Constantes: *false, true, null.*

Particiones y familias tienen colores asignados, color que se puede cambiar en las páginas de preferencias como se verá más adelante. En las siguientes ilustraciones se puede ver las diferentes familias codificadas en colores y un ejemplo de una *SPL*, para comprobar la detección de palabras clave y su cambio de color.

```
prueba.spl
1  Particiones:
2
3      Comentarios: {CCS /* Comment */}
4
5      Strings: 'hola' "hola"
6
7      Sentencias: LET :variable = aux;
8
9      Partición por defecto: hola
10
11 Familias:
12
13     Palabras SPL: EXECUTE
14
15     Palabras SQL: SELECT
16
17     Operadores: +, -, *, /
18
19     Tipos de datos SPL: INTEGER
20
21     Tipos de datos SQL: SQL_INTEGER
22
23     Funciones: INSERT
24
25     Constantes: TRUE
```

Ilustración 11. Particiones y familias de la funcionalidad syntax coloring

```
CT_FNN_ABS.spl
1  CREATE PROCEDURE (DecUser)CT_FNN_ABS (lParam1  float IN, lResult integer OUT, lCode integer OU
2  BEGIN DECLARE
3      DEFINE sAux1 varchar(30);
4      DEFINE sAux2 varchar(30);
5      DEFINE lAux1 integer;
6      DEFINE lAux2 integer;
7  END DECLARE
8
9  BEGIN EXECUTE
10     LET :lResult = 0;
11     LET :lCode = 0;
12
13     LET :lAux1 = {fn CONVERT({fn Abs(:lParam1)}, SQL_INTEGER)};
14     SELECT {fn CONVERT({fn Abs(:lParam1)}, SQL_INTEGER)} into :lAux2 FROM (DecUser)TARTICULO W
15
16     if :lAux1 <> :lAux2 then
17         LET :lCode = 1;
18     else
19         LET :lResult = :lAux1;
20     endif
21
22     RETURN (:lResult, :lCode);
23
24 END EXECUTE
25
26 END PROCEDURE
27
```

Ilustración 12. Ejemplo de syntax coloring en un fichero spl

3.3.2. Content assist

El objetivo de esta funcionalidad es ofrecer ayuda y automatizar la programación de código mediante la pulsación de las teclas *Ctrl+Space*, y la aparición de una nueva ventana de contenidos.

Esta ventana contiene las palabras clave definidas en *syntax coloring*, excepto la familia de operadores. Todas las palabras de la familia “funciones” aparecen con una ventana adicional mostrando información útil al desarrollador. También contiene plantillas de código para automatizar la programación.

En el CD-Rom de la memoria, se puede encontrar un archivo llamado “spl.xml” que contiene todas las plantillas de código definidas y que aparecen en el *content assist*.

En las siguientes ilustraciones se puede ver el content assist con algunas de las palabras clave, funciones y plantillas.

Ilustración 13. Content assist con palabras clave

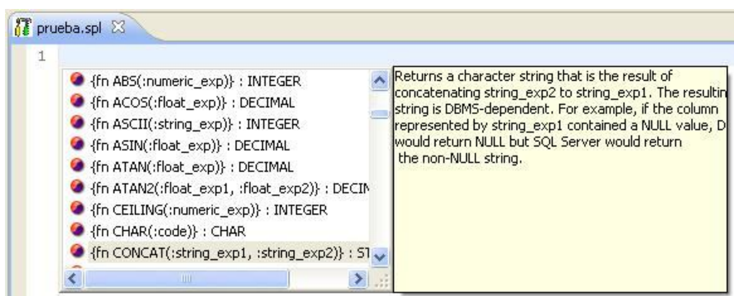


Ilustración 15. Content assist con funciones y su ayuda correspondiente

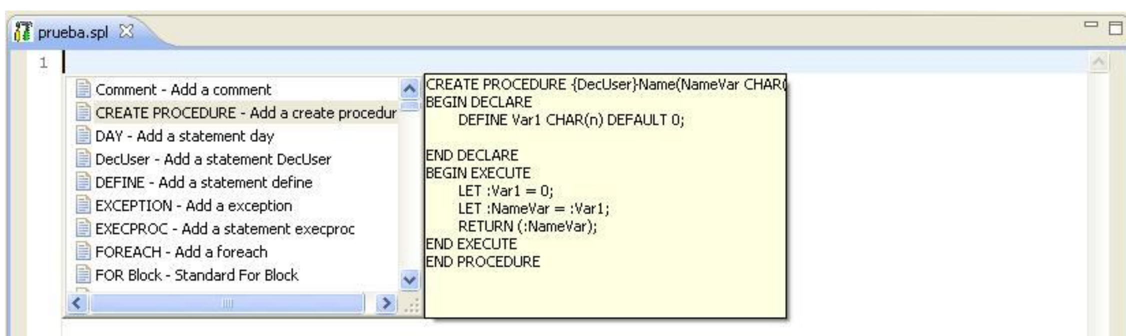


Ilustración 14. Content assist con plantillas

La siguiente ilustración muestra la automatización de la programación al haber seleccionado la plantilla “CREATE PROCEDURE” de la ilustración anterior. El *content assist* implementa automáticamente el bloque de código correspondiente.

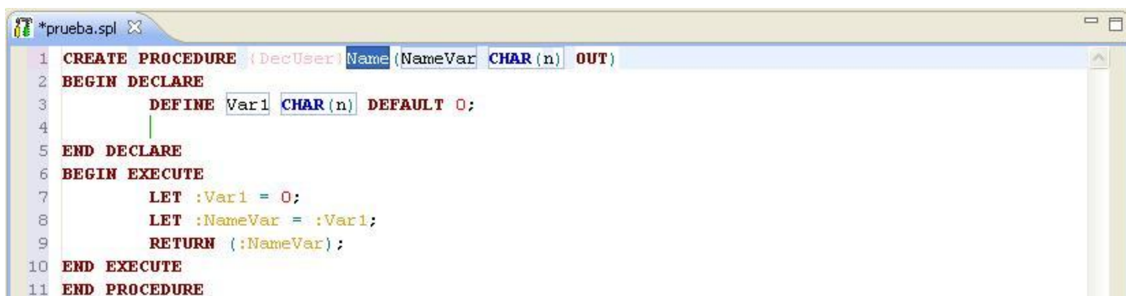


Ilustración 16. Selección de la plantilla “CREATE PROCEDURE” del content assist

3.3.3. SPL Preferences

El objetivo que se busca con la implementación de esta funcionalidad es la de poder realizar una configuración del editor al gusto del desarrollador.

Nuestro editor contiene tres páginas que veremos en las siguientes ilustraciones y se explicará su contenido.

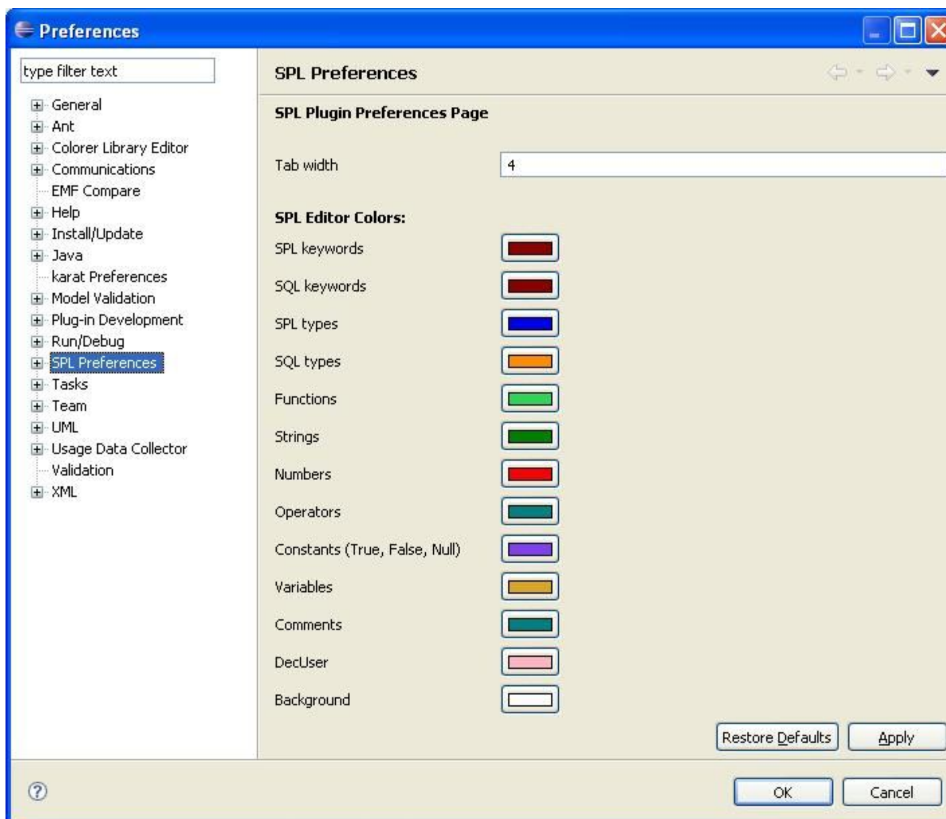


Ilustración 17. Página de SPL Preferences

Esta página es la principal de la configuración del editor. Su función es la de configurar los colores que se visualizan en los diferentes grupos de palabras clave a detectar. Se puede seleccionar el color que se desee.

Además contiene otra opción, la “Tab width”, que permite indicar el número de espacios en blanco que se desea dejar antes de posicionar el cursor de escritura cuando se tabula.

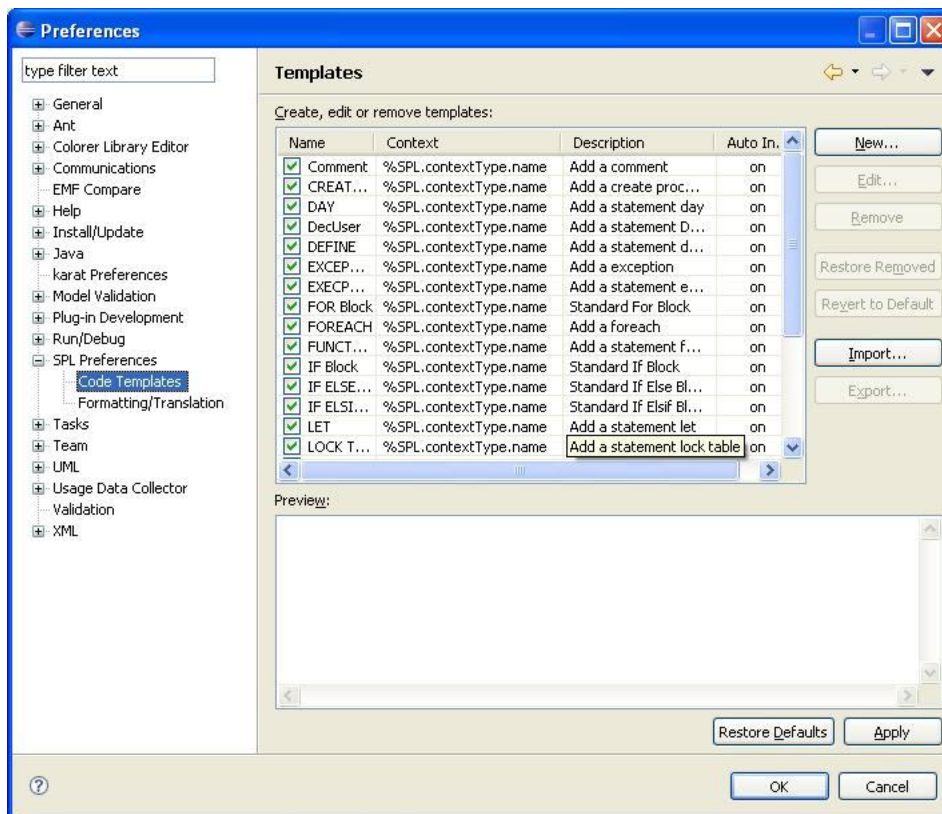


Ilustración 18. Página de Code Templates

Esta página contiene las plantillas de código definidas en el *content assist* y se da la oportunidad de modificarlas o eliminarlas. También se pueden crear nuevas plantillas, importarlas o exportarlas. La importación y exportación de plantillas se realiza en formato *XML*. A continuación se muestra el ejemplo de una plantilla en este formato:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<templates>
        <template autoinsert="true" context="SPLeditor.template.SPL"
        deleted="false" description="Standard If Block" enabled="true"
        id="SPLeditor.template.SPL.if" name="IF Block">
        IF ${Expression} THEN
                ${cursor}
        ENDIF
        </template>
</templates>
```

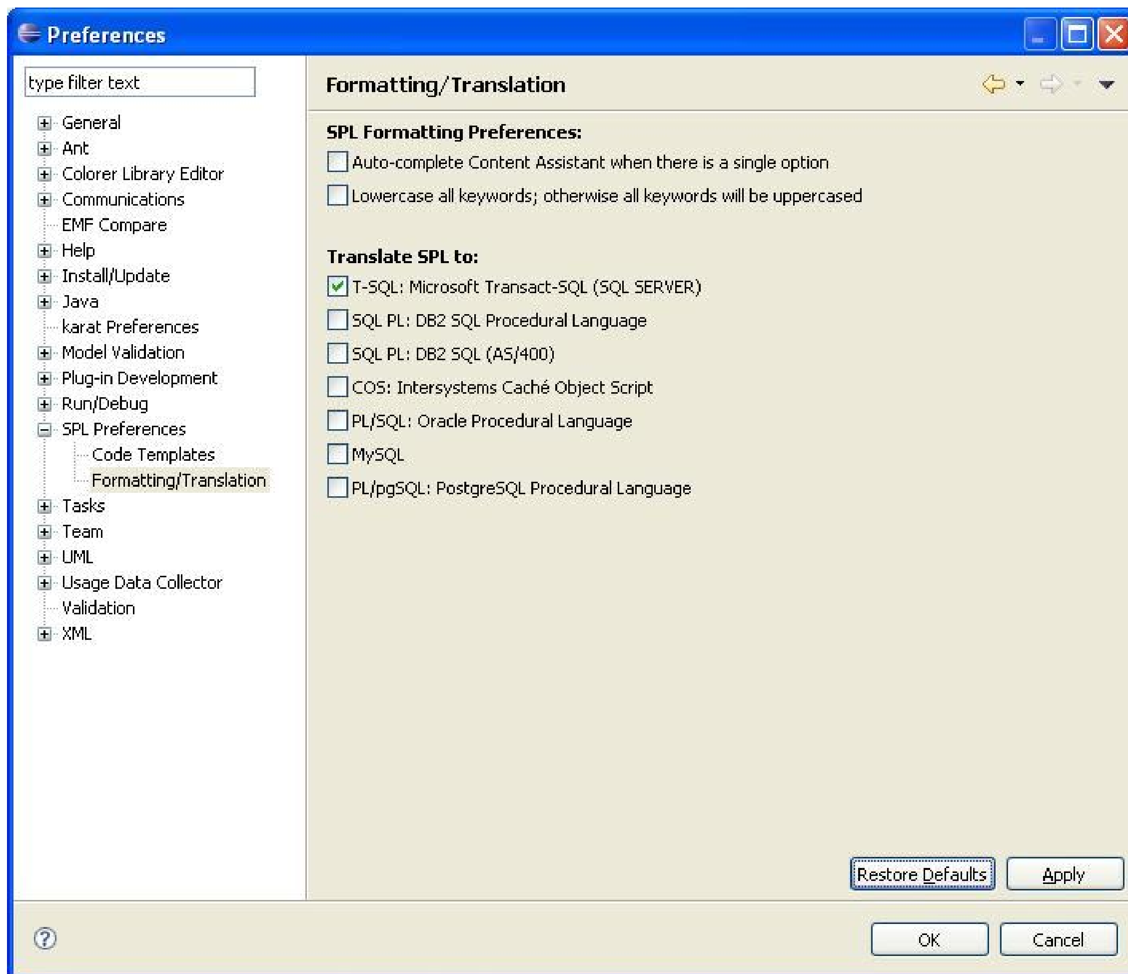


Ilustración 19. Página de Formatting/Translation

Esta página tiene dos partes: la primera contiene opciones que añaden más funcionalidad al editor y la segunda opciones sobre la traducción de código.

En primera parte se tienen dos opciones: la primera es sobre el *content assist* e indica si se desea autocompletar una palabra clave si es la única opción a elegir. La segunda es poder elegir que el editor pase las palabras clave a minúsculas, ya que por defecto cuando se detecta una palabra clave, ésta se pasa a mayúscula.

En la segunda parte se tienen una serie de opciones para seleccionar. Estas opciones representan los diferentes lenguajes de bases de datos con los que trabaja la empresa. La selección de estas opciones permite saber a qué lenguajes se debe traducir el código *SPL* cuando se active la acción de traducción.

3.3.4. SPL Editor Options

Es el menú del editor, están las opciones o acciones que pueden ser ejecutadas por el desarrollador.

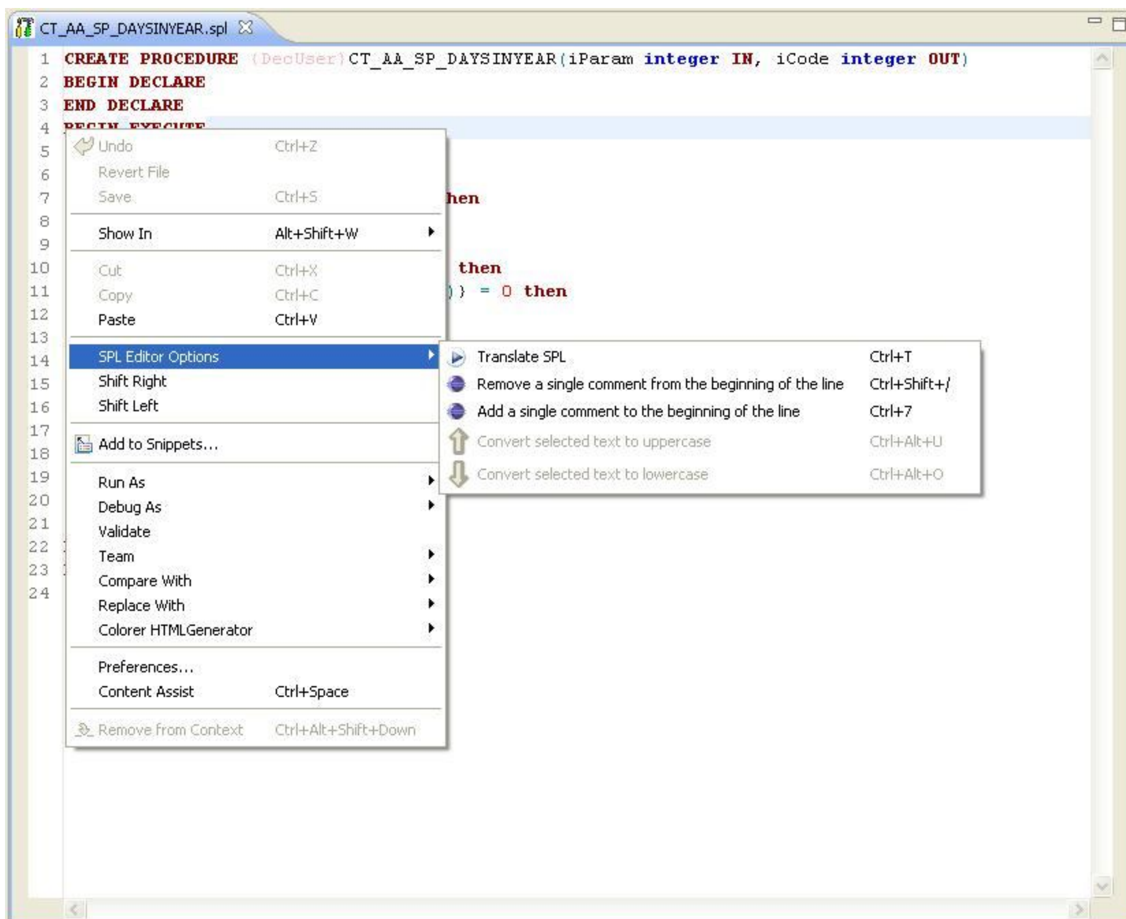


Ilustración 20. Menú SPL Editor Options

Este menú contiene cinco acciones:

- Translate SPL: Esta acción consiste en realizar una validación de la sintaxis y ofrecer la posibilidad de obtener el código nativo de la base de datos que se desee a partir del código *SPL* generado.

  La siguiente ilustración muestra el ejemplo de una *SPL* traducida a código de diferentes bases de datos. Se dispone de un editor con varias páginas y cada una de ellas contiene el código nativo de la base de datos.

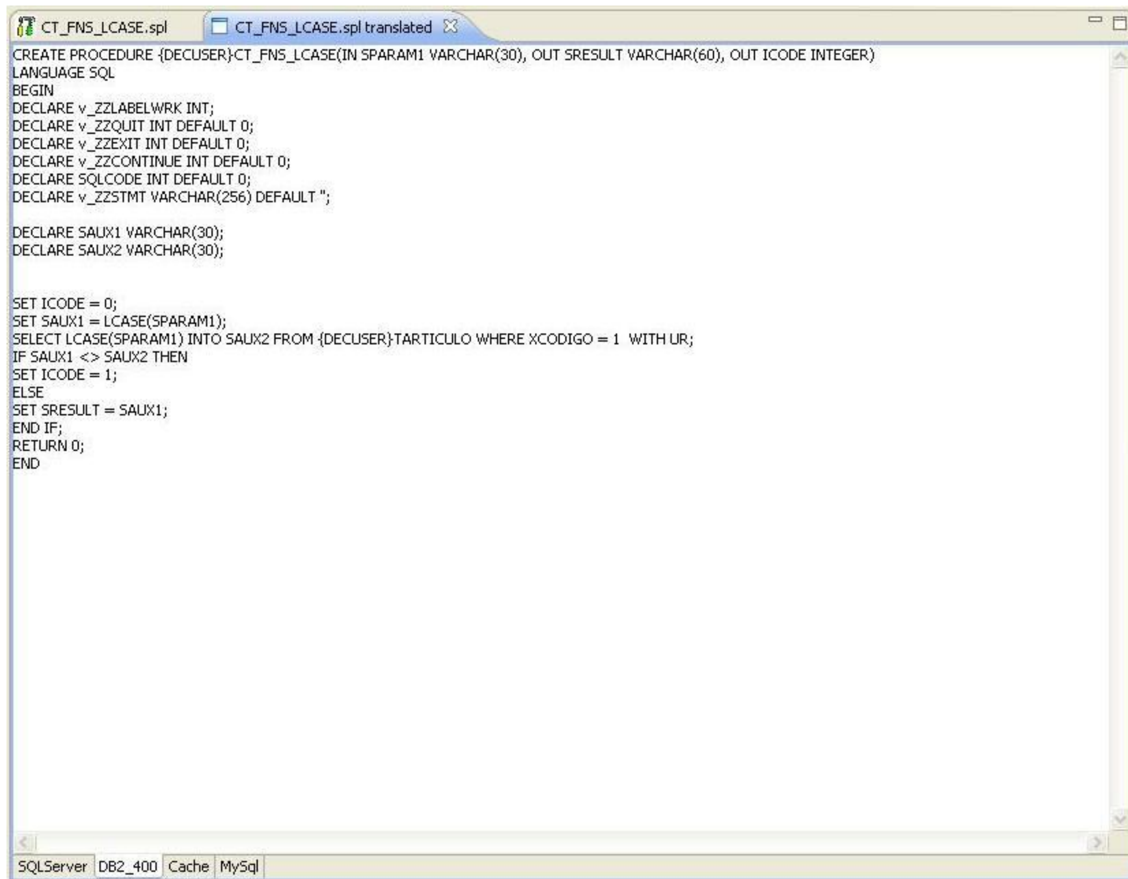


Ilustración 21. Ejemplo de una spl traducida a diferentes lenguajes de bases de datos

- Remove a single comment from the beginning of the line: Consiste en descomentar código comentado.
- Add a single comment to the beginning of the line: Consiste en comentar código.
- Convert selected text to uppercase: Consiste en convertir el código seleccionado a mayúsculas.
- Convert selected text to lowercase: Consiste en convertir el código seleccionado a minúsculas.

CAPÍTULO 4: CODIFICACIÓN Y PRUEBAS

Se explica el lenguaje Java, la herramienta IDE Eclipse y los entornos en los que se desarrolla el proyecto, como son un plug-in IDE Eclipse y karat. También se habla del estilo de codificación de la empresa y su forma de trabajar. Por último se explican todas las pruebas que se realizan durante las distintas fases.

4.1. Lenguaje

El nuevo componente construido estará integrado dentro de la nueva versión del producto de la empresa *karat*. Esta versión se encuentra implementada totalmente en lenguaje *Java*, por lo tanto este fue el lenguaje elegido para la realización del proyecto.

Java es un lenguaje de programación orientado a objetos desarrollado por *Sun Microsystems* a principios de los años 90. El lenguaje en sí mismo toma mucha de su sintaxis de *C* y *C++*, pero tiene un modelo de objetos más simple y elimina herramientas de bajo nivel, que suelen inducir a muchos errores, como la manipulación directa de punteros o memoria.

El lenguaje *Java* se creó con cinco características principales:

1. Usa la metodología de la programación orientada a objetos.
2. Permite la ejecución de un mismo programa en múltiples sistemas operativos.
3. Incluye por defecto soporte para trabajo en red.
4. Diseña para ejecutar código en sistemas remotos de forma segura.
5. Es fácil de usar y toma lo mejor de otros lenguajes orientados a objetos, como *C++*.

El diseño de *Java*, su robustez, el respaldo de la industria y su fácil portabilidad han hecho de *Java* uno de los lenguajes con un mayor crecimiento y amplitud de uso en distintos ámbitos de la industria de la informática. Algunos de estos entornos de funcionamiento se encuentran en dispositivos móviles y sistemas empotrados, en el navegador web, en sistemas de servidor y en aplicaciones de escritorio entre otros.

Existen distintos programas comerciales que permiten desarrollar código *Java*. La compañía *Sun*, creadora de *Java*, distribuye gratuitamente el *Java(tm) Development Kit* (*JDK*). Se trata de un conjunto de programas y librerías que permiten desarrollar, compilar y ejecutar programas en *Java*. Incorpora además la posibilidad de ejecutar parcialmente el programa, deteniendo la ejecución en el punto deseado y estudiando en cada momento el valor de cada una de las variables (con el denominado *Debugger*).

4.2. Herramientas para el diseño

Para poder realizar el desarrollo de código *Java* se ha utilizado un programa específico. Este programa consiste en un *IDE*, concretamente el *IDE Eclipse*.

Los *IDE*s (*Integrated Development Environment*), tal y como su nombre indica, son entornos de desarrollo integrados. En un mismo programa es posible escribir el código *Java*, compilarlo y ejecutarlo sin tener que cambiar de aplicación. Estos entornos integrados permiten desarrollar las aplicaciones de forma mucho más rápida, incorporando en muchos casos librerías con componentes ya desarrollados, los cuales se incorporan al proyecto o programa. Como inconvenientes se pueden señalar algunos fallos de compatibilidad entre plataformas, y ficheros resultantes de mayor tamaño que los basados en clases estándar.

Eclipse es un potente entorno de desarrollo de libre distribución, abierto y extensible. Está escrito en su mayor parte en *Java* y se ejecuta sobre una máquina virtual de ésta. Aunque su uso sea de un entorno de desarrollo para *Java*, este *IDE* es neutral y adaptable a cualquier tipo de lenguaje. La característica clave de este entorno es la extensibilidad. *IDE Eclipse* es una gran estructura formada por un núcleo y muchos módulos (*plug-ins*) que van conformando la funcionalidad final.

El *IDE Eclipse* emplea módulos para proporcionar toda su funcionalidad al frente de la plataforma, a diferencia de otros entornos monolíticos donde la funcionalidad está toda incluida, la necesite el usuario o no.

4.3. Entornos de desarrollo

Una característica del editor a construir en este proyecto, es su integración en *IDE Eclipse* para darle uso en forma de *plug-in* y también en el producto *karat*. El editor será un *plug-in* que se podrá integrar en distintos sitios de *karat* como en edición de procedimientos almacenados y disparadores en lenguaje *SPL*, vistas, expresiones *SQL*, etc.

Un *plug-in* es una aplicación que se relaciona con otra para aportarle una función nueva y generalmente muy específica. No se trata de un parche ni de una actualización, es un

módulo aparte que se incluye opcionalmente en una aplicación. Esta aplicación adicional es ejecutada por la aplicación principal e interactúan por medio de la *API*.

Karat es una completa plataforma tecnológica para la gestión de las empresas, que aporta un nuevo concepto de soluciones basado en la independencia total y real de entornos. El producto se encuentra desarrollado en *Java* y funciona para cualquier sistema operativo, trabaja en web sin la necesidad de explorador, puede usarse en cualquier dispositivo y soporta todo tipo de idiomas y alfabetos.

4.4. Estilo de codificación

A la hora de realizar el desarrollo de código se han seguido una serie de reglas de programación que cumplimentan los desarrolladores del equipo *karat* a la hora de trabajar con el lenguaje *Java*.

Las reglas generales y características que debe tener el código son:

- **Simplicidad**: escribir clases y métodos simples.
- **Claridad**: asegurarse de que cada elemento del código tiene una finalidad clara y documentar cómo usarlos.
- **Completitud**: proporcionar la funcionalidad esperada, y documentar todas las características y funcionalidad.
- **Consistencia**: las entidades similares deberían comportarse de forma similar. Aplicar estándares en la medida de lo posible.
- **Robustez**: proporcionar un comportamiento predecible y documentado en respuesta a los posibles errores y excepciones. No ocultar errores ni forzar al usuario a detectarlos.

Como reglas específicas:

- **Indentar el código anidado**
    - Usar TAB (4 espacios) por anidación.
    - Cerrar el bloque a la altura en que se abrió.

- **Separar líneas muy largas de código**
  - No juntar varias sentencias en una línea.
  - Separar expresiones muy complejas en subexpresiones.
- **Utilizar espacios en blanco para separar:**
  - Tras cada palabra reservada.
  - Entre operadores (+, *, -, /, etc.).
  - Tras } y).
- **Usar una línea en blanco para separar:**
  - Tras cada método de una clase.
  - Entre bloques lógicos de código (ej: declaraciones y código).
  - Entre clases de un mismo fichero.
- **Usar nombres con significado**
  - Utilizar nombres en inglés.
  - Evitar nombres de variables como “e” o “a”.
  - Evitar nombres abreviados como “crtf” en lugar de “certificate”.
  - Contadores de bucles: i, j, k
  - String: s
  - Caracteres: c, d, e
  - Coordenadas: x, y, z
  - Excepción: e
  - I/O Streams: in, out, inOut
- **Nombres de packages**
  - Usar el nombre del dominio internet de la organización, invertido, como cabecera de todos los packages de la organización.
  - Usar palabras sencillas y en minúsculas.

- **Nombres de tipos (clases e interfaces)**
    - Usar la mayúscula para la primera letra de cada nombre. (Ejemplo: PaintCanvas)
    - Usar nombres gramaticales para clases (no usar verbos, adjetivos, etc). (Ejemplo: ClientCount, Price)
    - Terminar con I las interfaces. (Ejemplo: ClientI, PriceI)
- **Nombres de métodos**
    - Comenzar en minúscula, y poner en mayúscula la primera letra de palabras compuestas.
    - Usar verbos para nombrar métodos. (Ejemplos: move, scale, paint, etc.)
    - Métodos get/set: String getName(); void setName(String s);
    - Métodos is/set (boolean): Boolean isUsed(); void setUsed(Boolean value);
- **Nombres de variables**
    - Comenzar en minúscula, y poner en mayúscula la primera letra de palabras compuestas.
    - Usar nombres gramaticales y no verbos o adjetivos. (Ejemplos: telephone, direction, codePrice)
    - Usar el plural para colecciones (arrays, vectores, etc). (Ejemplos: Client [] clients; java.util.Vector figures;)
- **Nombres de constantes**
    - Nombre en mayúsculas, separando las palabras compuestas con “_”. (Ejemplos: MAX_PISOS, LONGITUD_MINIMA)

4.5. Pruebas

Durante todo el proceso de desarrollo, conforme se iban consiguiendo las fases planificadas, se han ido realizando pruebas específicas para cada funcionalidad. Un buen plan de pruebas es esencial para corroborar su buen funcionamiento. Las pruebas realizadas han servido para encontrar errores y para mejorar la herramienta. Las pruebas nos han ayudado a conseguir los objetivos de una forma más segura y efectiva.

A continuación veremos una lista de las diferentes pruebas realizadas para comprobar el funcionamiento correcto del editor:

1. Pruebas básicas del Editor de *SPL*. Crear en el *IDE eclipse* un nuevo fichero con extensión .spl, y probar el nuevo editor de *SPL*.
2. Las palabras claves cambian de color y se distinguen diferentes familias.
3. Mediante la pulsación de las teclas *Ctrl + Space* se abre el content assist.
4. Navegar en el content assist y elegir plantillas, funciones o palabras clave con su correspondiente ventana de información si procede.
5. Si en el *IDE eclipse*, en *karat preferences* se ha informado los datos de conexión a una BBDD que contiene *SPLs*, visualizar el *karat Scripting view*. Abrir una *SPL* existente.
6. Pruebas básicas del Editor de *SPL* en una *SPL* ya existente en la BBDD.
7. Salvar *SPL* modificada en el repositorio.
8. Salvar *SPL* en *IDE eclipse*, y comparar con *SPL* del repositorio.
9. Crear un nuevo procedimiento *SPL* en la BBDD.
10. Eliminar *SPL* existente en la BBDD.
11. Realizar pruebas con las distintas opciones de *SPL preferences* del *IDE eclipse* (cambiar colores, modificar, eliminar, importar, exportar plantillas o crear una nueva).
12. En *SPL preferences* indicar a que BBDD se quieren traducir las *SPLs*. Y probar que se traduce a las BBDD indicadas seleccionando *Translate SPL*.
13. Comentar y descomentar código mediante las opciones del *SPL Editor Options*.
14. Pasar a mayúsculas y minúsculas mediante las opciones del *SPL Editor Options*.

CAPÍTULO 5: CONCLUSIONES

Se comentan las posibles ampliaciones que se pueden realizar al proyecto, es decir, la inclusión de más funcionalidades. Se comenta la planificación real y los cambios con la planificación inicial. Acabamos con conclusiones finales, para hablar de los objetivos conseguidos y hacemos una valoración personal sobre el proyecto y la estancia en la empresa.

5.1. Objetivos cumplidos

El proyecto consistía en la realización de un editor del lenguaje *SPL*. El primer día la empresa marcó unas pautas que se debían cumplir y expuso las características y objetivos para conseguir la herramienta deseada.

Una de las características del proyecto consistía en realizar el componente siguiendo un modelo de desarrollo en espiral. Seguir este modelo para planificar en fases la realización del proyecto ha sido uno de los objetivos cumplidos.

Otras características del componente eran que se debía implementar en lenguaje *Java* y desarrollarlo en un formato *plug-in Eclipse* con el objetivo de su integración posterior en el producto *karat*. La integración ha sido totalmente exitosa y los desarrolladores tienen la posibilidad de utilizar el nuevo editor, por lo tanto estos objetivos también se han conseguido.

Con la implementación de las diferentes funcionalidades definidas en capítulos anteriores, se ha conseguido construir una herramienta de gran valor para los desarrolladores de este lenguaje. Comparando el editor realizado con el editor anterior, se ven las muchas mejoras que aporta.

Por lo tanto, con todo lo que estamos comentando, se puede decir que el nuevo Editor de *SPL* es una herramienta que mejora todas las anteriores y que podrá substituirlas en los distintos entornos donde sea necesario. Está realizado con una interfaz fácil de interaccionar, mejorando la visualización gráfica, además de interactuar con varias bases de datos y facilitarle y automatizarle al desarrollador la programación, consiguiendo que esta tarea sea más rápida, eficiente y segura.

5.2. Desviaciones respecto la planificación temporal inicial y cambios en el desarrollo

Durante la realización del proyecto han surgido cambios que han obligado a modificar la planificación inicial estimada, por lo tanto hay tareas que han sufrido un cambio en el total de horas invertidas.

En la siguiente tabla vemos una relación de las horas estimadas y las horas utilizadas realmente:

Fase de desarrollo	Tiempo estimado de realización (en horas)	Tiempo real de realización (en horas)
Formación	30	25
Investigación	60	60
Definición de requerimientos	30	22
Análisis	60	60
Codificación	310	280
Pruebas	40	40
Documentación	30	60
Total	**560**	**547**

Como se puede ver, en las etapas de investigación, análisis y pruebas se han dedicado las mismas horas que las estimadas al principio. Estas etapas han transcurrido sin problemas.

En la etapa de formación se han utilizado menos horas a las previstas gracias a los conocimientos adquiridos en la universidad sobre el lenguaje *Java*. Estos conocimientos han ayudado a no dedicarle tantas horas al lenguaje y aprovecharlas para otros procesos como la construcción del *plug-in*.

En la etapa de definición de requerimientos también se han empleado menos horas, esto es debido a que el proyecto desde el principio estaba bien definido y sus funcionalidades eran claras, concisas y fáciles de entender.

Otra de las etapas en las que se han invertido menos horas, ha sido la de codificación. La rebaja de horas es debido a que finalmente una fase fue anulada, ya que desde la

empresa comunicaron que por el momento no le iban a dar uso a la funcionalidad correspondiente. Esto se explica más adelante.

La única etapa en la que ha habido un incremento de horas ha sido en la de documentación. Esto es debido a que se debe escribir con tranquilidad y revisar varias veces para crear un documento entendible y estructurado.

A continuación se muestra el desglose del proyecto por tareas y fases, con las horas estimadas y reales:

1. **Formación (30 h → 25 h)**
   - 1.1 Montaje del equipo informático e instalación y explicación de programas a usar (10 h → 5 h)
   - 1.2 Definición inicial del proyecto y presentación de *karat* (10 h → 10 h)
   - 1.3 Formación sobre *SPL* y *plug-in Editor* (10 h → 10 h)

2. **Investigación (60 h → 60 h)**
   - 2.1 Sistema actual (10 h → 10 h)
   - 2.2 Investigación del ejemplo *Plug-in* del *Java Editor de Eclipse* (10 h → 10 h)
   - 2.3 Investigación del ejemplo *Plug-in* del *XML Editor de Eclipse* (10 h → 10 h)
   - 2.4 Investigación sobre el ejemplo *PL/SQL Editor* (20 h → 20 h)
   - 2.5 Investigación sobre el ejemplo *Log4j Editor* (10 h → 10 h)

3. **Definición de requerimientos (30 h → 22 h)**
   - 3.1 Análisis del problema (10 h → 8 h)
   - 3.2 Especificación de requerimientos funcionales y no funcionales (10 h → 8 h)
   - 3.3 Planificación del proyecto (5 h → 4 h)
   - 3.4 Corrección del documento (5 h → 2 h)

4. **Fase 1: Creación del *Plug-in Editor***
   - 4.1 Análisis (6 h → 6 h)
   - 4.2 Codificación (31 h → 41 h)
   - 4.3 Pruebas (4 h → 4 h)

5. **Fase 2: Syntax coloring**
   - 5.1 Análisis (12 h → 12 h)
   - 5.2 Codificación (62 h → 85 h)
   - 5.3 Pruebas (8 h → 8 h)

6. **Fase 3: Content Assist**
   - 6.1 Análisis (12 h → 12 h)
   - 6.2 Codificación (62 h → 100 h)
   - 6.3 Pruebas (8 h → 8 h)

7. **Fase 4: Páginas de configuración, verificación y envíos de consulta**
   - 7.1 Análisis (6 h → 6 h)
   - 7.2 Codificación (31 h → 54 h)
   - 7.3 Pruebas (4 h → 4 h)

8. Fase 5: Búsqueda avanzada y Outline
8.1 Análisis (6 h → No procede)
8.2 Codificación (31 h → No procede)
8.3 Pruebas (4 h → No procede)

9. Fase 6: Análisis de errores
9.1 Análisis (18 h → No procede)
9.2 Codificación (93 h → No procede)
9.3 Pruebas (12 h → No procede)

10. Documentación (30 h → 60 h)

Las fases se dividen en tres subtareas: análisis, codificación y pruebas. Las subtareas de análisis y pruebas han transcurrido sin problemas y según lo previsto. La única alteración en las horas la ha sufrido la subtarea de codificación en cada una de las fases, esta tarea siempre es la más difícil de predecir, ya que a la hora del desarrollo de código suelen surgir problemas imprevistos que hacen retardar la implantación.

La fase 5 fue anulada por la empresa, ya que consideraron que no le iban a dar uso por el momento y preferían emplear más tiempo en el desarrollo de las demás fases. La fase 6 fue mal planificada inicialmente porque esta fase se podría tratar como un proyecto, es una funcionalidad muy complicada de implementar y que requiere muchas horas de investigación y codificación, por lo que el tiempo del proyecto no permitía su creación.

5.3. Ampliaciones

Todos los proyectos de software son susceptibles de ampliaciones y actualizaciones a lo largo de su vida útil. Por eso, se plantean una serie de funcionalidades que posiblemente se tengan que implantar en un futuro no muy lejano.

- Una de las funcionalidades sería el análisis de errores en las sentencias del código desarrollado. Consistiría en realizar un análisis en tiempo real de desarrollo y así ayudar a escribir un código sin errores gramaticales. Esta funcionalidad ya se había planificado desde el primer día (fase 6), pero como se explica en el siguiente apartado, por cuestión de variación en la planificación no se ha podido lograr, por lo que queda pendiente como una posible mejora.

- Otra funcionalidad sería la implementación del *Outline*. Consistiría en conectar el editor desarrollado con esta herramienta para mostrar funciones y variables

del código y así poder acceder a ellas de una manera más fácil y rápida. Como ocurre en la anterior funcionalidad, también se había planificado inicialmente (fase 5).

- Por último, para ayudar aún más a automatizar la programación, la funcionalidad que podría implementarse sería la de detectar zonas de código en el editor, es decir, detectar bloques para restringir opciones. Un ejemplo sería, que en un bloque “begin declare” y “end declare”, restringir plantillas y palabras clave que nunca se podrían utilizar en este bloque, ya que en este bloque solo se permiten definir variables.

5.4. Valoración personal

La realización de este proyecto me ha ayudado a conocer la planificación y el desarrollo desde el primer día hasta el último de un proyecto real y de un tamaño considerable.

He podido poner en práctica el modelo de desarrollo en espiral, un modelo del que había oído hablar, pero nunca había utilizado. Con este modelo he aprendido a realizar una planificación en fases y he comprobado que al finalizar cada fase, el proyecto se iba enriqueciendo tal y como dice el significado del modelo. Este modelo me ha sido de gran ayuda para la gestión del tiempo.

En cuanto al desarrollo de la herramienta en sí, puedo decir que he ampliado mis habilidades de programación en el lenguaje *Java* y he utilizado nuevos componentes del *IDE Eclipse*. He podido comprender el funcionamiento y la integración de un *plug-in*, además de ver y entender la estructura de un editor en general.

Y por lo que se refiere a la estancia en la empresa, puedo decir que ha sido una experiencia muy positiva. He trabajado totalmente integrado en un departamento y en un grupo de trabajo, siguiendo las pautas de una empresa informática, pudiendo así conocer como se trabaja en este ámbito. Después de esta estancia puedo decir que me veo con más capacidad para realizar, planificar y desarrollar proyectos en una empresa informática, aunque falta mucho por aprender. Por lo tanto como proyecto final de carrera, creo que ha sido un gran proyecto y recomiendo a los alumnos a realizarlo en una empresa, ya que la experiencia ha sido muy buena.

Bibliografía

Fuentes bibliográficas:

- David Gallardo, Ed Burnette, Robert McGovern, "Eclipse in Action, a guide for java developers". Editorial Manning, 2003. www.manning.com/gallardo
- Manuales propios de la empresa:
  - Introducción a *SPL*.
  - Acerca de los traductores SPL *karat*.
  - Tutorial sobre *Plug-ins*.
  - Acerca de componente *IDE Eclipse colorer plugin*.

Fuentes electrónicas:

- Help Contents del propio *IDE Eclipse*.
- Toby Zines, "Toby's PL/SQL Editor". http://plsqleditor.sourceforge.net
  Página que contiene un ejemplo de editor del lenguaje PL/SQL.
- Andrey Platov, Andrei Sobolev, Annika Karjakina, Wayne Beaton y otros, "eclipse WIKI: A guide to building a DLTK-based language IDE". http://wiki.eclipse.org/A_guide_to_building_a_DLTK-based_language_IDE
  Guía explicativa de la construcción de un editor para el lenguaje Python.
- Remy Chi Jian Suen, Nick Veys y Chris Laffra, "eclipse WIKI: FAQ How do I add Content Assist to my editor?". http://wiki.eclipse.org/FAQ_How_do_I_add_Content_Assist_to_my_editor%3F
  Ayuda sobre la implementación del Content Assist.
- "Los colores y sus códigos RGB". http://www.pagaelpato.com/tecno/colores.htm
  Web para consultar diferentes colores.
- Unit 4 Agresso, "Software de gestión empresarial Soluciones ERP para pymes CRM para pymes ERP en España". http://www.ccsagresso.com
  Página de la empresa CCS Agresso.
- Jimmy Wales y Larry Sanger, "Wikipedia, la enciclopedia libre". http://es.wikipedia.org/wiki/Wikipedia:Portada
  Enciclopedia

Anexo A – Glosario

API

Una API (del inglés Application Programming Interface - Interfaz de Programación de Aplicaciones) es el conjunto de funciones y procedimientos (o métodos si se refiere a programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.

IDE Eclipse

Entorno Integrado de Desarrollo (IDE) abierto y extensible, para cualquier cosa y nada en particular. Pese a que eclipse esté escrito en su mayor parte en Java (salvo el núcleo), se ejecuta sobre una máquina virtual de ésta y su uso más popular es como un IDE para Java, Eclipse es neutral y adaptable a cualquier tipo de lenguaje. La característica clave de Eclipse es la extensibilidad. IDE Eclipse es una gran estructura formada por un núcleo y muchos plug-ins que van conformando la funcionalidad final.

El IDE Eclipse emplea módulos (en inglés plug-in) para proporcionar toda su funcionalidad, a diferencia de otros entornos monolíticos donde la funcionalidad está toda incluida, la necesite el usuario o no. Este mecanismo de módulos es una plataforma ligera para componentes de software.

Java

Java es un lenguaje de programación orientado a objetos desarrollado por Sun Microsystems a principios de los años 90. El lenguaje en sí mismo toma mucha de su sintaxis de C y C++, pero tiene un modelo de objetos más simple y elimina herramientas de bajo nivel, que suelen inducir a muchos errores, como la manipulación directa de punteros o memoria.

Las aplicaciones Java están típicamente compiladas en un *bytecode*, aunque la compilación en código máquina nativo también es posible. En el tiempo de ejecución, el *bytecode* es normalmente interpretado o compilado a código nativo para la ejecución, aunque la ejecución directa por hardware del *bytecode* por un procesador Java también es posible.

La implementación original y de referencia del compilador, la máquina virtual y las librerías de clases de Java fueron desarrollados por Sun Microsystems en 1995. Desde entonces, Sun ha controlado las especificaciones, el desarrollo y evolución del lenguaje a través del Java Community Process, si bien otros han desarrollado también implementaciones alternativas de estas tecnologías de Sun, algunas incluso bajo licencias de software libre.

Entre noviembre de 2006 y mayo de 2007, Sun Microsystems liberó la mayor parte de sus tecnologías Java bajo la licencia GNU GPL, de acuerdo con las especificaciones del Java Community Process, de tal forma que prácticamente todo el Java de Sun es ahora software libre (aunque la biblioteca de clases de Sun que se requiere para ejecutar los programas Java todavía no es software libre).

Plug-in

Un complemento (o plug-in en inglés) es una aplicación que se relaciona con otra para aportarle una función nueva y generalmente muy específica. Esta aplicación adicional es ejecutada por la aplicación principal e interactúan por medio de una API.

SPL

SPL es el acrónimo de *Stored Procedure Language* (en castellano, lenguaje de procedimientos almacenados). Un *stored procedure* es un procedimiento almacenado en la base de datos, escrito en un lenguaje procedural de alto nivel que proporciona la misma base de datos y que permite la interrogación y manipulación de datos de una forma fácil y rápida.

SQL

El lenguaje de consulta estructurado (SQL en inglés Structured Query Language) es un lenguaje declarativo de acceso a bases de datos relacionales que permite especificar diversos tipos de operaciones sobre las mismas. Una de sus características es el manejo del álgebra y el cálculo relacional permitiendo lanzar consultas con el fin de recuperar -de una forma sencilla- información de interés de una base de datos, así como también hacer cambios sobre la misma.

XML

XML, siglas en inglés de *Extensible Markup Language* (lenguaje de marcas), es un metalenguaje extensible de etiquetas desarrollado por el World Wide Web Consortium (W3C). Es una simplificación y adaptación del SGML que permite definir la gramática de lenguajes específicos (de la misma manera que HTML es a su vez un lenguaje definido por SGML). Por lo tanto XML no es realmente un lenguaje en particular, sino una manera de definir lenguajes para diferentes necesidades. Algunos de estos lenguajes que usan XML para su definición son XHTML, SVG, MathML.

XML no ha nacido sólo para su aplicación en Internet, sino que se propone como un estándar para el intercambio de información estructurada entre diferentes plataformas. Se puede usar en bases de datos, editores de texto, hojas de cálculo y casi cualquier cosa imaginable.

XML es una tecnología sencilla que tiene a su alrededor otras que la complementan y la hacen mucho más grande y con unas posibilidades mucho mayores. Tiene un papel muy importante en la actualidad ya que permite la compatibilidad entre sistemas para compartir la información de una manera segura, fiable y fácil.

Anexo B – Contenido del CD-Rom

- **Memoria**

  Documento que contiene toda la información correspondiente al proyecto.

- **Código: spleditor**

  Carpeta que contiene todas las clases con el código que forman el proyecto.

- **Plug-in: com.ccsagresso.karat.spl_1.0.0.v20090611205321**

  Complemento que contiene el "Componente Editor & Analizador de SPL-SQL" para *IDE Eclipse*.

- **Ayuda realizada para la empresa: kdev0020**

  Carpeta que contiene una pequeña ayuda online, explicando el funcionamiento del componente realizado.

- **CCS**

  Carpeta que contiene las dos presentaciones realizadas en la empresa, una primera después de 3 meses en la empresa y la presentación final. Además también contiene dos archivos que han sido de gran ayuda para la realización del proyecto: Introducción a SPL y Tutorial sobre plug-ins.

Daniel Dueñas Ruiz

Junio 2009