FRAMEWORK DE SOPORTE PARA MIGRACIÓN DE SISTEMAS POWERBUILDER

UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

INGENIERÍA

EN INFORMÁTICA

PROYECTO FIN DE CARRERA

Framework de soporte para migración de sistemas PowerBuilder

Darío Ureña García

Agosto, 2016

UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

Departamento de Tecnologías y Sistemas de Información

PROYECTO FIN DE CARRERA

Framework de soporte para migración de sistemas PowerBuilder

Autor: Darío Ureña García

Director: Dr. Ignacio García Rodríguez de Guzman

Agosto, 2016

Darío Ureña García

Ciudad Real – Spain

E-mail: darioaxel@gmail.com
Web site: http://github.com/darioaxel

TRIBUNAL:

Presidente:

Secretario:

Vocal:

FECHA DE DEFENSA:

CALIFICACIÓN:

PRESIDENTE	**SECRETARIO**	**VOCAL**
Fdo.:	Fdo.:	Fdo.:

Resumen

Este proyecto fin de carrera surge como respuesta al estudio de una herramienta software, dentro de una estructura empresarial real, cuyo ciclo de vida se encuentra en un momento crítico debido a la necesidad de su revisión para poder abordar las exigencias del mercado actual.

Gracias a la aplicación de las técnicas de reingeniería y la utilización de una aproximación basada en modelos, se busca recuperar la mayor cantidad de información posible del sistema en estudio. El resultado generado habrá de servir de base para afrontar el desarrollo de una nueva herramienta que permita sustituir el producto obsoleto con el menor coste para la empresa.

La poca difusión del lenguaje de programación y el framework del sistema heredado, hacen obligatoria la creación desde cero de los objetos que permitan el análisis del mismo. Todo el desarrollo que se ha efectuado, estará abierto a la comunidad de desarrolladores permitiendo su interacción, aprovechamiento y futuro uso.

Índice general

Índice de cuadros

Índice de figuras

Índice de listados

Agradecimientos

A mi familia por toda la presión infligida y por hacer de recordatorio continuo de mi deber.

A Virginia por aguantar un año falto de escapadas los fines de semana, vacaciones y viajes planeados.

A los amigos y a tantas grandes amigas que aún sin casi haberme visto durante demasiado tiempo, han seguido llamándome y azuzándome con sus punzantes comentarios jocosos para que finalizara este proyecto y poder así cumplir con mis deberes sociales.

A Nacho, por no mandarme «lejos» cuando no tenía noticias mías por largos periodos y contestar amablemente mis dudas.

Al equipo de desarrollo de Savia, por darme esos permisos y privilegios que han permitido que este proyecto se haga realidad.

A mi madre, mi padre y mi tía-abuela

Capítulo 1

Introducción

UNO de los mayores retos que pueden afrontar las empresas de desarrollo de software es la actualización de sistemas obsoletos, pero funcionales, para su adaptación a las nuevas arquitecturas de software y estándares actuales de calidad. Con ello se conseguirá mantenerlos en explotación, ofreciendo su funcionalidad, ya que muchos de estos sistemas heredados son de gran valor para la organización y casi insustituibles por las pérdidas económicas que de su desaparición podrían derivarse.

Dentro del catálogos de herramientas de muchas empresas es fácil encontrar sistemas que se desarrollaron tiempo atrás y que en muchos casos, ya sea por desconocimiento o por falta de experiencia, no cumplen estándares que permitan su mantenimiento y mejora. Estas herramientas, que pueden ser aún funcionales y que contienen la gran base de conocimiento de la lógica de negocio de la empresa que la desarrollo, terminan convirtiéndose en un problema enorme para el futuro de la propia empresa. El aprovechamiento del conocimiento encerrado en el código, es vital para evitar perdidas monetarias y de competitividad, que podrían incluso a la desaparición del desarrollador.

Cuando se afronta una situación como la descrita anteriormente, es necesario realizar un estudio y valoración pormenorizado de la estrategia a seguir para conseguir superar el problema con el menor coste, pero obteniendo un sistema final que evite tener enfrentarse a la misma situación en el futuro. Estos estudios sobre la cartera de aplicaciones de los sistemas de las organizaciones dan como resultado qué sistemas merece la pena mantener y cuáles no. Aquellos sistemas, que a pesar de su obsolescencia deben mantenerse activos son .ªctualizados"mediante las herramientas adecuadas durante la fase de mantenimiento.

El mantenimiento del software se define como la *modificación de un producto software después de su implantación para corregir fallos, para mejorar su funcionamiento u otros atributos, o para adaptar el producto a un entorno cambiante* [CC90] es uno de los procesos que en toda factoría de software se ha de llevar a cabo según los modelos de desarrollo establecidos. En concreto, y dada su alta popularidad y acoplamiento a las necesidades empresariales actuales, los modelos Ágiles y en Espiral reservan en sus iteraciones cíclicas una parte para este proceso dentro de los pasos de Refactorización y Análisis de nuevos requisitos, respectivamente.

La documentación de los programas es un aspecto sumamente importante, tanto en el desarrollo de la aplicación como en el mantenimiento de la misma. El proceso de documentación debe realizarse de manera exhaustiva y en cada paso del desarrollo como recomiendan las técnicas ágiles. Todo aquel producto software que no cuente con la debida documentación, impide a futuros desarrolladores conocer detalladamente el funcionamiento del programa así como su mantenimiento de manera eficaz.

A la hora de abordar la compleja tarea de mantener sistemas en los que ni se dispone de acceso a documentación precisa, ni a los desarrolladores que crearon el proyecto, la IEEE-1219a[1] recomienda la utilización de la ingeniería inversa como herramienta en el desarrollo del proceso.

Sin embargo la ingeniería inversa por si sola no implica la modificación del propio sistema, ya que se trata de [CP07] *"un proceso de estudio y descripción sin cambios"* y habrá de utilizarse un proceso mas amplio como la reingeniería de software para el mantenimiento y refactorización sistema objetivo. Así pues se certifica que la reingeniería de software, en el estricto sentido de su definición como *"El proceso de análisis de un sistema sujeto, para identificar los componentes del sistema, sus relaciones y crear una representación del sistema en otra forma o nivel superior de abstracción"* [CC90] y los distintos subprocesos que se suelen incluir en el, es el proceso adecuado al propósito de actualización de las aplicaciones software que la industria demanda.

La ventaja competitiva para las empresas que por su escalabilidad y de ahorro de costos en licencias supone el surgimiento de los nuevos conceptos de *Software como Servicio* y *Computación en la nube*, está forzando a la industria del software a rediseñar sus productos.

La situación descrita en el presente apartado y el conocimiento de los procesos necesarios para cubrir las necesidades que se formulan, sugieren la creación de una herramienta que recogiendo la información del sistema objetivo por medio de las técnicas disponibles actualmente, represente de la manera mas precisa y estandarizada tanto la arquitectura como los procesos propios del software a estudio. Esta representación ha de permitir su posterior adaptación a los paradigmas actuales de manera manual o con la ayuda de herramientas CASE.

1.1 Motivación

La motivación para la realización del presente proyecto surge dentro de una de las reuniones periódicas que realiza el equipo de I+D+I de la empresa Savia[Sav]. Uno de los productos que la empresa desarrolla está escrito utilizando la solución integrada de desarrollo PowerBuilder® y su lenguaje de programación Powerscript. Esta decisión ha tenido efectos

[1] The IEEE SA-1219-1988 IEEE Standard for Software Maintenance

positivos y negativos desde su toma en la década de los 90.

Como parte positiva se puede destacar que el desarrollo permitió la creación rápida de una aplicación visualmente atractiva para el usuario y con una arquitectura preestablecida que evita al desarrollador de la toma de decisiones complejas. Por otro lado, también existen diversos puntos negativos derivados de dicha decisión, y entre los que destacan los siguientes:

- Alto acoplamiento entre las tres capas definidas por el patrón Modelo Vista Controlador (MVC).
- Imposibilidad de la creación de sistemas de pruebas que permita proporcionar información objetiva e independiente sobre la calidad del producto.
- Capa de servicios muy limitada y compleja.
- Excesiva dependencia de un producto propietario costoso y que no se actualiza con la velocidad necesaria.

Además y por la no aplicación de una metodología de desarrollo de software adecuada, el fabricante se enfrenta a problemas como; la no existencia de una documentación precisa que recoja los procesos de negocio y las modificaciones implementadas en el devenir del mantenimiento del software, la utilización de consultas SQL dentro del código fuente de manera no controlada, código fuente repleto de comentarios que ofuscan su comprensión, etc.

Valorado el estado actual del software y las necesidades que se prevén en un futuro no lejano, se toma la decisión de establecer un plan de recuperación del conocimiento existente en el producto para usarlo como apoyo dentro del desarrollo de una nueva aplicación que sustituya a la existente en la actualidad.

[Nombre del proyecto] nace como necesidad básica para poder llevar a cabo la recopilación de información del software obsoleto.

1.2 Estructura del documento

Pueden incluirse aquí una sección con algunos consejos para la lectura del documento dependiendo de la motivación o conocimientos del lector. También puede ser útil incluir una lista con el nombre y finalidad de cada uno de los capítulos restantes.

Capítulo 1: Introducción
Capítulo presente.

Capítulo 2: Objetivos
Donde se definen las metas que mediante el presente proyecto se alcanzarán.

Capítulo 3: Estado del arte
Descripción de las bases teóricas y de las técnicas que se aplicarán en el proyecto.

Capítulo 4: Método y fases de trabajo

Definición de la metodología que se utilizará para el desarrollo realizado.

Capítulo 5: Resultados

Explicación detallada de todo aquello que se ha realizado.

Capítulo 6: Conclusiones y propuestas

Conclusiones de la realización del PFC y se plantean distintas futuras líneas de trabajo.

Capítulo 2

Objetivos

En este capítulo se describen los objetivos que se marcan para la consecución del Proyecto Fin de Carrera, así como los medios técnicos que se emplean para su óptima consecución.

2.1 Objetivo principal

El objetivo principal de este PFC es:

Desarrollo de un sistema que ofrezca soporte a las tarea de migración de sistemas heredados en explotación desarrollados en PowerBuilder y en PowerScript hacia plataformas y arquitecturas actuales

De esta forma se posibilitará el aprovechamiento del conocimiento acumulado durante la fase de explotación se han implementado y su utilización como base en una futura aplicación informática.

2.1.1 Objetivos parciales

Para lograr este objetivo principal se plantean los siguientes objetivos parciales:

Obj.1 Análisis de la arquitectura de las aplicaciones PowerBuilder, del lenguaje de desarrollo con el que se escriben y estudio en detalle del código fuente disponible que generan. Con ello se pretende obtener un conocimiento pormenorizado de cada uno de los objetos, funciones, tipos de datos y estructuras del lenguaje.

Obj.2 Generación de una gramática que permita reconocer el lenguaje PowerScript mediante el uso del analizador sintáctico ANTLR. Este gramática será liberado para su posterior uso por parte de la comunidad de desarrolladores y su inclusión dentro de la colección de gramáticas de la herramienta de análisis en el repositorio Github de la comunidad.

Obj.3 Diseño y Desarrollo de un módulo para la generación de modelos basados en KDM a partir de la información extraída del analizador PowerBuilder.

2.1.2 Objetivos docentes

Para la consecución del objetivo principal será necesario llevar a cabo los siguientes objetivos docentes:

- Estudiar las técnicas de reingeniería e ingeniería inversa.
- Análisis de la sintaxis de los lenguajes de programación.
- Aprender a crear analizadores sintácticos mediante el diseño de gramáticas.
- Perfeccionar los conocimientos y mejorar la experiencia actual en técnicas de testing y de los procesos de desarrollo basados en tests.

2.1.3 Recursos tecnológicos

- Ordenador portátil Dell Inspiron 14z con procesador CoreI5 y 6Gb de memoria RAM. El sistema operativo sobre el que se implementa es Linux Mint 17
- Como herramienta de diseño para los distintos diagramas se han utilizado tanto la herramienta DIA como la aplicación online Cacoo.
- Para la redacción de la documentación se ha empleado el lenguaje LaTeX y la herramienta gráfica Kile v2.13 de KDE en combinación con la clase **arco-pfc**

Capítulo 3

Estado del arte

En en capítulo que se presenta a continuación, se definirán primero los conceptos teóricos en los que se basa el desarrollo de proyecto, así como se introducen los principios de algunas de las herramientas y lenguajes de programación que han sido utilizados. Los primeros tres puntos desarrollan los conceptos de ingeniería inversa, transformación de modelos y la modernización del software, son básicos para entender el planteamiento que se ha llevado a cabo para cumplir los objetivos propuestos en el capítulo siguiente. Además en la sección Powerbuilder y Powerscript se introduce el lenguaje usado en la aplicación a la que aplicaremos las técnicas explicadas. En las siguientes secciones se numeraran las diferencias y mejoras mas significativas de las versiones actuales de las herramientas utilizadas durante el desarrollo llevado a cabo.

3.1 Ingeniería inversa

De acuerdo con la definición de ingeniería inversa propuesta por Chikofsky y Cross en el artículo [CC90], establecemos que en el contexto de un proceso de ingeniería basado en modelos, el resultado de este análisis es un modelo. Para llegar a este modelo, se proponen como posibles soluciones, el análisis estático del código fuente o el seguimiento de la ejecución de la aplicación mediante herramientas diseñadas a tal efecto. En el caso de no disponer del código fuente por no estar disponible, el proceso de ingeniería debe también llevar a cabo la descompilación o desensamblado de los archivos binarios del sistema, para obtener un código válido para la tarea que se intenta realizar.

3.2 Transformación de modelos

Una transformación basada en modelos toma como entrada un sistema, que está basado en determinado meta-modelo y generar como resultado otro modelo que se determina por un metamodelo distinto. Existen diferentes aproximaciones a la trasformación basada en modelos como la manipulación directa, relacional, operacional, guiada por estructuras, etc [czarnecki and helsen 2003]. Todas estas aproximaciones implementan actividades ampliamente utilizadas dentro del desarrollo del software tales como son la refactorización, ingeniería inversa y el uso de patrones.

3.3 Modernización del software

La modernización del software es un proceso de apliación de técnicas de reingeniería a una aplicación obsoleta, para conseguir que cumpla una serie de nuevos requerimientos e incrementar la calidad del sistema. El proceso de modernización de un sistema puede ser visualizado como una herradura, donde el lado izquierdo se compone de la extracción de información e ingeniería inversa, el lado derecho el proceso de desarrollo e ingeniería y la conexión entre ambas partes la transformación al sistema antiguo para llegar al sistema objetivo.

3.4 PowerBuilder y Powerscript

PowerBuilder es un entorno completo para el desarrollo de aplicaciones de negocio, propiedad de la empresa Sysbase. Este sistema está compuesto por frameworks de desarrollo, herramientas de conexión, gestión y tratamiento de bases de datos, así como de un lenguaje propio de programación PowerScript. Actualmente la versión de trabajo para el entorno de desarrollo y la aplicación a estudio es la 11.5., mientras que la versión más actual del software de PowerBuilder es la 12.5.

Para la creación de aplicaciones con PowerBuilder se dispone de un Integrated Development Enviroment (IDE) de desarrollo propio. Este IDE permite realizar la implementación de programas de un modo visual, abstrayendo al desarrollador de la codificación de los comandos.

Powerscript es el lenguaje usado para especificar el comportamiento de la aplicación en respuesta a eventos del sistema o del usuario, tal como cerrar una ventana o presionar un botón. Las aplicaciones desarrolladas con PowerBuilder se ejecutan exclusivamente en el sistema operativo Microsoft Windows. [Inc08]

Objetos PowerBuilder

Un programa Powerbuilder está construido como una colección de objetos proveídos por el propio entorno de desarrollo y los objetos hijos que el desarrollador crea mediante su extensión.

Objeto	Uso	Prefijo	Extensión
Proyecto	Punto de entrada a la aplicación		.pbt
Window	Interfaz primaria entre el usuario y la aplicación PowerBuilder	w_	.srw
Menú	Lista de comandos u opciones que un usuario puede seleccionar en la ventana activa	m_	.srm
DataWindow	Recupera y manipula datos desde una fuente.	d_	.srd
Funciones globales	Realiza procesos de propósito general	n_	.sru
Estructuras	Colecciones de variables relacionadas bajo un mismo nombre	s_	.srs
Query	Ficheros de texto con consultas auxiliares		.txt
Librerías	Objetos de apoyo a la compilación que indican los objetos que serán compilados conjuntamente		.pbg

Cuadro 3.1: Objetos de PowerBuilder

Objetos de tipo Aplicación Estos objetos representan el punto de entrada a una aplicación. Se trata de un objeto que lista el conjunto de librerías que conforman la aplicación en su totalidad además de identificarla. Al igual que el resto de objetos (DataWindow, Estructura, etc) se guarda en una librería (archivo PBL) que generará el compilador.

Objetos de tipo Ventanas Las ventanas son la interfaz principal de comunicación entre el usuario y la aplicación PowerBuilder. En ellas se muestra información, y se recupera de la entrada que el usuario utilice, ya sea respondiendo a eventos lanzados por clicks de ratón o texto introducido por teclado. Una ventana consiste en:

- Propiedades de definición de la apariencia y comportamiento, tales como nombre de la barra de titulo, botón de minimizado/maximizado, tamaño, etc.
- Eventos lanzados por las acciones del usuario.
- Controles establecidos en la ventana.

Objetos DataWindow Un DataWindow es un objeto que el desarrollador utiliza para recuperar y manipular datos desde una fuente externa. Así pues este tipo de objetos se comunicarán utilizando sentencias del lenguaje SQL con bases de datos o hojas de cálculo Microsoft Excel. Normalmente se utilizaran integrados dentro de objetos ventana para mostrar los datos recuperados al realizar una acción de esta.

Objetos de tipo menú Los objetos de tipo menú contienen listas de items que el usuario puede seleccionar desde la barra de menú de la ventana activa. Normalmente se trata de agrupaciones de elementos relacionados, y cada uno de ellos permite al usuario lanzar una orden al sistema como puede ser la apertura de una ventana, la ejecución de un proceso o la edición del estilo de un campo de texto.

Objetos de funciones PowerBuilder permite al desarrollador definir dos tipos de clases de funciones:

- Funciones a nivel de objetos definidas para un menu o ventana particular. A su vez se pueden subdividir en funciones de sistema (disponibles siempre para objetos de una cierta clase) y funciones definidas por el usuario.
- Funciones globales que no están asociadas a un objeto en particular y que se encuentran ubicadas en un objeto independiente. Al contrario que las funciones a nivel de objetos realizan procesos de propósito general y pueden ser utilizadas en cualquier tipo de objeto. Un ejemplo de este tipo de objetos serían funciones de cálculos matemáticos o manejo de cadenas.

Objetos de estructuras Una estructura es una colección de uno o mas variables relacionadas del mismo tipo o de diferentes tipos, que se encuentran definidas bajo un único nombre que las identifica. Por ejemplo, una estructura llamada s-user-struct que contiene las variables que identifican al usuario: Identificador, dirección, nombre, una imagen, etc. Al igual que en los objetos de funciones disponemos de dos tipos de estructuras:

- Estructuras a nivel de objeto que se asocian a un determinado objeto tal como una ventana o menu. Estas estructuras se utilizaran en los scripts definidos para el propio objeto que lo contiene.
- Estructuras globales no asociadas a un objeto determinado y que pueden ser declaradas para su uso en cualquier script de la aplicación.

Objetos definidos por el usuario Normalmente las aplicaciones tienen características en común. Un ejemplo claro de ello son los botones existentes en la mayoría de ventanas que permiten al usuario cerrar, minimizar o maximizar un objeto. Al identificar este tipo de agrupación de se puede crear un objeto propio definido por el usuario una única vez y utilizarlo en los puntos de la aplicación que lo necesiten. Los objetos definidos por el usuario pueden agruparse en objetos de usuario estandarizados y objetos de usuario específicos.

La división establece aquellos que pueden ser exportados a otras aplicaciones PowerBuilder y los que son específicos para una en concreto. Además dentro de ellos podemos diferenciarlos entre los objetos que implican elementos visuales como pueden ser agrupaciones de botones, y los que contienen componentes no visuales como por ejemplo agrupaciones de funciones de cálculo que representan reglas de negocio y que pueden heredar eventos y propiedades de objetos definidos por el sistema.

Proyectos y librerías Los ficheros de extensión PBG y PBT definen estructuralmente la forma en la que el compilador generará la aplicación ejecutable resultante. Así pues, el fichero PBT contiene las relaciones establecidas entre las distintas librerías a generar y el nombre del producto final. Por otro lado los ficheros PBG definen las librerías resultantes de la compilación de objetos relacionados, que pueden encontrarse en el mismo directorio o en directorios separados. [Inc08]

3.5 Knowledge-Discovery Metamodel (KDM)

En Junio de 2003, Organization modeling group (OMG)[OMG] creó un equipo de trabajo para modelar artefactos software en el contexto de sistemas obsoletos. Inicialmente el grupo fue llamado *Equipo de trabajo para la transformación de sistemas obsoletos*, [1] aunque pronto se les renombró a *Equipo de trabajo para la modernización enfocada en la arquitectura* [2]

[1] traducción de «Legacy Transformation Task Force»
[2] traducción de «Architecture-Driven Modernization Task Force»

En Noviembre de 2003 la *ADMTF*[3] incorporó la solicitud de propuesta para la especificación KDM. La solicitud de propuesta establecía que el estándar del metamodelo KDM debía:

- Representar los artefactos de los sistemas obsoletos como entidades, relaciones y atributos.
- Incluir los artefactos externos con los que el interactúen los artefactos del software.
- Soportar diversos lenguajes y plataformas.
- Consistir en un núcleo independiente del lenguaje y la plataforma que pueda extenderse en caso necesario.
- Definir una terminología unificada para los artefactos de software obsoleto.
- Describir las estructuras lógicas y físicas de los sistemas obsoletos.
- La posibilidad de realizar agregaciones o modificaciones de la estructura física del sistema.
- Facilitar la identificación y trazabilidad de los artefactos desde la estructura lógica hacia la física.
- Representar el comportamiento de los artefactos hacia abajo, pero no por debajo, del nivel procedural.

[KDM]

3.5.1 El metamodelo KDM

El metamodelo KDM se divide en varias capas que representan tanto los artefactos físicos como los lógicos de un sistema obsoleto. Mas allá cada capa de abstracción diferente se separa el conocimiento sobre el sistema obsoleto en diversos estructuras de sofware conocidas como *vistas de la arquitectura*[4]. Las cuatro capas definidas en el estándar se describen a continuación:

Capa de infraestructura

La capa de infraestructura define el nivel mas bajo de las capas de abstracción y contiene una pequeña lista de conceptos utilizados a través de toda la especificación.

Core Define las abstracciones básicas de KDM, que son *KDMEntity* y *KDMRelationship*

KDM Proporciona el contexto compartido por todos los modelos KDM. Este paquete define los elementos que constituyen el «framework» de cada representación KDM. Por ejemplo,

[3] Siglas en inglés del nombre del equipo de trabajo
[4] Traducción de «Architecture views»

cada representación KDM consiste en uno o mas elementos de tipo *Segment* que contienen diversos modelos de KDM.

Source Define el conjunto de artefactos físicos del sistema de información heredado y permite referenciar partes del código fuente. Para ello se genera el *Inventory Model*, que enumera todos los artefactos físicos del sistema obsoleto (como ficheros de código, imágenes, ficheros de configuración, etc). Además este artefacto es la base que se utilizará para referenciar a los artefactos físicos desde los modelos KDM. Mas aún, los elementos del *Inventory Model* permiten identificar mediante el uso de *AbstractInventoryRelationships* relaciones de dependencia definidas en el sistema heredado del tipo *DependsOn* que declaran la interrelación de elementos durante pasos del proceso de reingeniería.

Capa de elementos del programa

Proporciona una representación intermedia, independiente del lenguaje de programación, para representar los constructores comunes a varios lenguajes de programación. Los dos paquetes que lo forman son:

Code Se trata de un paquete que define un conjunto de *CodeItems* que representan elementos comunes presentes en diversos lenguajes como pueden ser; métodos, clases, tipos de datos, funciones o interfaces. Los elementos *CodeItem* se especializan en 3 tipos base:

- **Module**: una unidad de programa discreta e identificable que contiene otros elementos y puede ser utilizada como componente lógico del software. Y a su vez se divide en *Package*, *CompilationUnit* ,*CodeAssembly*, etc.
- **ComputationalObject**: representación de métodos, funciones, etc.
- **DataType**: que definen items nombrables del sistema obsoleto heredado como variables, parámetros de funciones, etc.

Action Define las acciones llevadas a cabo por los elementos del paquete code. Los elementos de ambos paquetes se representan dentro de un modelo de código *CodeModel*.

Capa de recursos

Permite representar conocimiento sobre el entorno y los recursos de ejecución utilizados por los sistemas de información heredados. Dispone de cuatro paquetes:

- **Data** Define los aspectos de los datos.
- **Event** Define el modelo de eventos, condiciones y acciones del sistema de información heredado.
- **UI** Define los aspectos de la interfaz de usuario del sistema de información heredado.

- **Platform** Define las características de la plataforma de ejecución.

Capa de Abstracción

Permite representar el conocimiento específico de dominio a la vez que da una visión de negocio de los sistemas de información heredados. Dispone de tres paquetes.

- **Conceptual** Define los elementos específicos de dominio del sistema de información heredado.
- **Structure** Define los componentes estructurales de los sistema de información heredados, es decir, los subsistemas, capas, paquetes, etc.
- **Build** Define los artefactos finales relativos al sistema de información heredado.

3.6 ANTLRv4

Un parser es un programa, normalmente parte de un compilador, que recibe entradas de forma secuencial como instrucciones de un fichero de código, etiquetas de marcado, o cualquier tipo de texto y lo divide en trozos que pueden ser utilizados por otro programa.

ANTLR es un generador de parsers que permite crear automáticamente árboles de representación de las estructuras que cumplen las gramáticas que el usuario diseña. También es capaz de generar objetos para recorrer el árbol generado nodo a nodo y a través de ellos realizar tareas programadas. En su desarrollo se ha utilizado el lenguaje de programación Java, y su utilización está ampliamente extendida, destacando por su popularidad entre el resto de competidores.

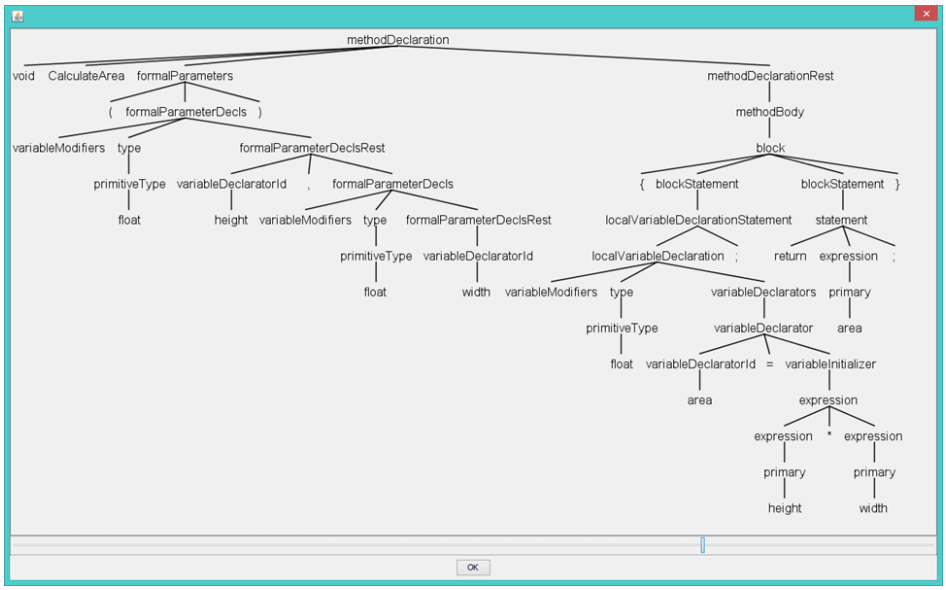

Figura 3.1: Vista de un árbol generado mediante ANTLR4

Mejoras de ANTLRv4 sobre sus versiones anteriores La versión 4 de ANTLR tiene una serie de nuevas capacidades que permiten reducir la curva de aprendizaje y hace el desarrollo de gramáticas y aplicaciones de reconocimiento de lenguajes mucho mas sencillas:

- La principal mejora de esta versión es la aceptación por defecto de toda gramática, con la excepción de la recursión indirecta a izquierda. ANTLRv4 no genera conflictos con la gramática o lanza avisos relacionados con la ambigüedad de las gramáticas.
- Esta última versión utiliza una nueva tecnología llamada *Adaptative LL(*) o ALL(*)*. Por ella se realiza un análisis dinámico en tiempo de ejecución en vez de el estático realizado en versiones anteriores, antes de la ejecución del parser generado. Como los parsers *ALL(*)* tienen acceso a las secuencias que se le introducen, pueden directamente encontrar la forma de reconocer las secuencias hilando la gramática. Hasta ahora, por el contrario, el análisis estático, tenía que considerar todas las posibles sentencias de entrada que pudieran existir.
- ANTLR v4 automáticamente reescribe las reglas con recursión a izquierda en sus equivalentes sin ella, donde las reglas se referencian inmediatamente así mismas. La única restricción que se pone es que existe es que las reglas no pueden referencias a otra regla en la parte izquierda de una alternativa que pueda volver a referenciar a la regla inicial sin emparejarse a un token.

```
1   expr : expr '*' expr
2     | expr '+' expr
3     | INT
4     ;
```

Listado 3.1: ALL(*)

- El mayor cambio de la nueva versión v4 es que se des-enfatiza la inclusión de código embebido dentro de la gramática, en favor de objetos *listener y visitor*. Este nuevo mecanismo de desarrollo permite desacoplar la gramática del código. Sin acciones embebidas, la reutilización de gramáticas es mucho mas sencilla sin ni siquiera tener que recompilar los parsers generados.

[Par]

3.7 MoDisco

MoDisco es un proyecto «open source» que forma parte de manera oficial de la *Eclipse Foundation*(EF)[Ecl] y está integrado en el proyecto base de modelado de dicha fundación, promocionando las técnicas de *Ingeniería dirigida a modelos*(MDE) [5] dentro de la comunidad de la comunidad de Eclipse. Además está reconocida por la OMG como proveedor de referencia para la implementación de diversos estándares como:

[5] Traducción de *Model Driven Engineering*

- *Knowledge Discovery Metamodel* (KDM)
- *Software Measurement Metamodel* (SMM)
- *Generic Abstract Syntax Tree Metamodel* (GASTM)

MoDisco provee de una serie de componentes que permiten elaborar soluciones de ingeniería inversa para la transformación, con independencia del lenguaje en el que esté desarrollado, de sistemas obsoletos utilizando meta modelos. De manera nativa se ofrecen soluciones para Java, pero gracias a la API que proporciona se puede representar cualquier otro lenguaje. El soporte al proceso de reingeniería comienza con la especificación de los metamodelos, cuyo detalle puede variar en función de la tecnología con la que se trabaje. Con objeto de obtener el modelo, se han de usar los llamados Discoverers. Todos los Discoverers pueden ser acoplados al «framework», y usados a través de él, únicamente con su registro en el *Discoverer Manager*.

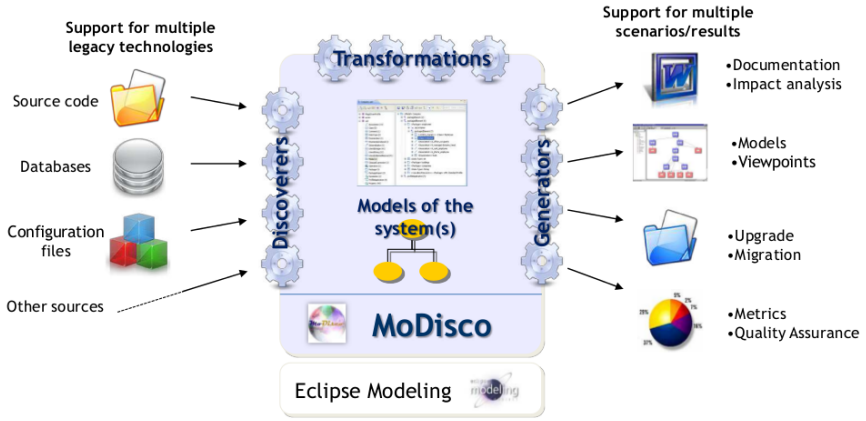

Figura 3.2: Estructura MoDisco

De manera similar a los *Discoverers*, MoDisco proporciona también *Generators* que se corresponden con los metamodelos soportados. Estos *Generators* pueden igualmente ser acoplados al «framework» y habilitan la transformación de los modelos generados en otros tipos de artefactos. Por ejemplo, para Java se disponen de *Generators* que permiten transformar los sistemas en estudio en estándares como KDM y su representación estándar basada en XMI (XML Metadata Interchange) o UML (Unified Modeling Language).

3.8 Aproximación a la transformación eficiente de lenguajes de programación a KDM en 3 pasos

La propuesta de aproximación a la transformación de sistemas heredados obsoletos en 3 pasos o fases hecha por Christian Wulf [Wul12], define una estructuración del proceso que permite la generación de un modelo en KDM desde el sistema inicial mediante los siguientes pasos:

- **Transformación de tipos internos**: En esta primera fase el sistema genera un fichero XMI con los artefactos KDM *Segment* que a su vez contienen el *InventoryModel* y los *CodeModel* de cada uno de los elementos contenedores del código fuente del sistema.
- **Transformación de miembros internos y métodos**: La segunda fase utiliza el artefacto base *Segment* generado en el paso anterior, y lo amplía añadiendo los *CodeElements* que representan tanto a los miembros instanciados dentro de cada contenedor de código, como los métodos de los mismos. Además recupera las relaciones entre *CodeElements* y añade el *LanguageUnit* de cada uno.
- **Transformación de sentencias**: La fase final del proceso es la encargada de mapear las sentencias del código y de generar los *ActionElements* definidos en él.

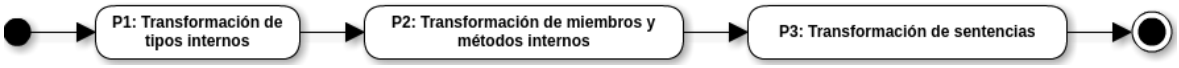

Figura 3.3: Esquema de los pasos llevados a cabo en la transformación

Capítulo 4

Método y fases de trabajo

Como metodología de trabajo se ha seleccionado, tras un primer estudio de la complejidad del proyecto y de los recursos disponibles, una versión simplificada del Proceso Unificado Rational (PUR)[Amb02] que desarrolló Scott Ambler, conocida como Proceso unificado ágil (PUA)[1]. Esta metodología describe un proceso mas sencillo que mezcla los principios del PUR con técnicas ágiles como el Desarrollo guiado por pruebas de software, Test Driven Development (TDD) [Bec03].

4.1 Proceso Unificado Ágil

Por tanto en el desarrollo del software se mantendrán los principios base del PUR:

- Implementación de un desarrollo dirigido por casos de uso, enfocado a dar valor al cliente y a cubrir las expectativas mediante la generación de documentación, casos de prueba, etc.
- Ampliando el punto anterior,la calidad y la estandarización del resultado serán un referente, cumpliendo el segundo principio del desarrollo al plantear un trabajo centrado en arquitectura.
- Un ciclo de vida iterativo, con un crecimiento incremental y en espiral permitirá mantener una gestión de riesgos de forma periódica y la retroalimentación típica de los procesos ágiles sobre la que basar la siguiente iteración. Dentro de cada iteración se implementará un ciclo del proceso de desarrollo basado en test.

Y de entre las prácticas del modelado ágil recomendadas se seguirán las siguientes :

- Una activa participación de los depositarios finales del software, haciéndolos copartícipes de las decisiones a tomar durante el proceso de desarrollo de una manera activa.
- Un modelado inicial de la arquitectura de alto nivel para identificar la estrategia a tomar en la creación del producto en desarrollo.
- Generación de documentación de manera continua en todos los pasos del desarrollo, y no como una tarea aislada, establecido en una única localización y utilizando para

[1] traducción del inglés Agile Unified Process

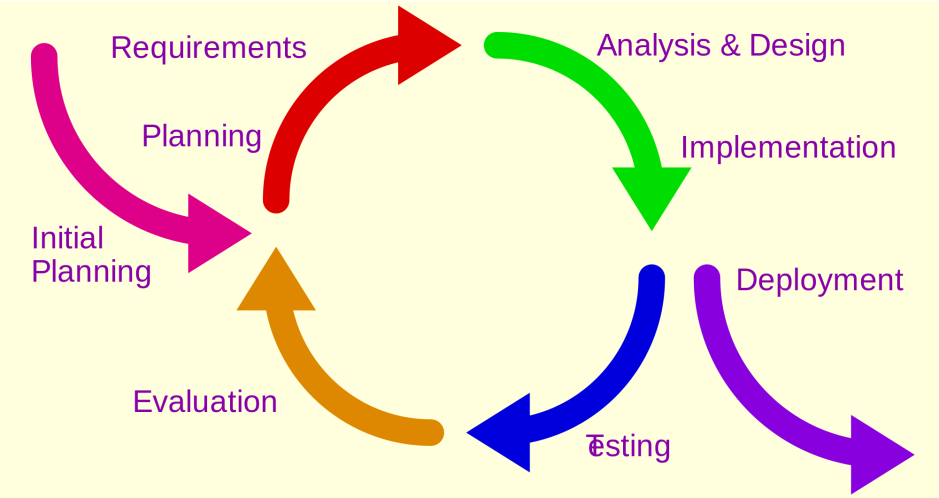

Figura 4.1: Ciclo de vida PUA

ello el estándar de calidad máximo. Para esta tarea usaremos el Lenguaje Unificado de Modelado UML[UML].

- Priorización en los requerimientos, ordenados y establecidos en colaboración con depositarios finales, que permitan proveer los mejores "resultados sobre la inversión"[2] posibles
- Realización de una lluvia de ideas sobre el modelo entre cada iteración. Con ello se exploraran los detalles de cada requerimiento y las cuestiones a plantear durante el diseño.
- Planteamiento de diversos modelos y selección del mejor para su implementación, teniendo en cuenta sus pros y contras.
- Estudio inicial de los requerimientos, invirtiendo al comienzo del proyecto un periodo de tiempo en establecer el alcance del mismo y creando una lista de prioridades en el desarrollo.

El desarrollo dirigido por test está definido por la creación de un sistema de pruebas, bien en el momento de la toma de requerimientos, bien al establecer el diseño, y posteriormente una codificación suficiente para permitir al software superar dichos tests. Esta metodología de trabajo permite reducir procesos posteriores de verificación así como los errores no controlados.

4.2 Estructura de las iteraciones del desarrollo

Una vez establecida la metodología base, procede detallar la estructura elegida para los ciclos de vida del desarrollo. Como se ha especificado en el punto anterior y siguiendo otra mas de las recomendaciones del modelado ágil, se utilizará una aproximación dirigida a test

[2] del inglés *ROI: return on investments*

o Test-driven development para cada una de las iteraciones de implementación del ciclo en espiral.

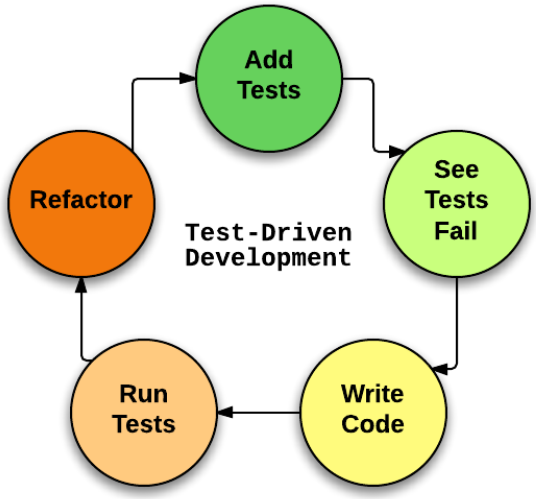

Figura 4.2: Flujo de las iteraciones del desarrollo basadas en TDD

Los pasos a realizar, en cada una de las iteraciones, serán los siguientes:

- **Generación de un test.** Toda iteración ha de iniciarse con la creación de un test, que inevitablemente fallará, para cubrir los requerimientos y especificaciones de los casos de uso identificados en el análisis del proyecto. Este test será la base sobre la cual se realizará la codificación del producto y cuyo objetivo ha de ser su superación con resultado positivo.
- **Ejecución de la batería de test acumulados y verificación del fallo del que se acaba de crear.** Una vez creado el test a superar, lanzaremos todos los test que se han ido generando en las anteriores iteraciones del desarrollo, para comprobar que se superan todos.
- **Escritura del código estrictamente necesario para cumplir con los requerimientos que permitan pasar el test.** Se toma como precepto aplicar la metáfora *"Déjelo simple"* [3].
- **Ejecución de los test.** Una vez creado el código, se ha de pasar de nuevo el conjunto de test acumulado para comprobar que no se ha interferido con el desarrollo existente de forma indeseada.
- **Refactorización del código.** Limpieza, reubicación y eliminación de posibles duplicidades. Además ha de una revisión de todo el código para que cumpla los estándares de calidad previstos.

[3] traducción de "Keep It Simple, Stupid"(KISS)

4.3 Planificación del Proyecto

Como consecuencia de la metodología seleccionada para la construcción del proyecto se mostrará a continuación el listado de iteraciones que se pretenden llevar a cabo para la obtención del producto final desarrollado. El proyecto se ha estructurado en 6 iteraciones + una iteración inicial de planificación:

4.3.1 Iteración 0: Planificación de las iteraciones

Esta primera iteración se utilizará para establecer la base tecnológica necesaria para el desarrollo del proyecto y se integrarán las librerías externas. Además se desarrollará una reunión con Savia para recabar información acerca del detalle de las necesidades específicas que tienen para obtener los procesos de negocio mas importantes del software a transformar. Una vez recabados los requerimientos y analizado el proyecto en conjunto, se establecerá una arquitectura base así como un plan de desarrollo para cada una de las iteraciones siguientes.

4.3.2 Iteración 1: Desarrollo de la gramática ANTLRv4 de Powerscript

En esta iteración se abordará el desarrollo de la gramática que permita reconocer el lenguaje Powerscript. Para ello se realizará un análisis pormenorizado de cada una de las estructuras y elementos, ordenándolos en complejidad creciente y estableciendo unos mínimos que servirán como test para los ciclos de desarrollo basados en TDD. Una vez finalizado el proceso de desarrollo, comprobaremos que todos los test establecidos han sido superados por el producto resultante de la iteración.

4.3.3 Iteración 2: Sistema de validación del software heredado

Generada una base que permita analizar el contenido de los ficheros de código del sistema heredado en la iteración anterior, el siguiente hito será establecer un sistema de comprobación para cerciorarse de que el proyecto que se analizará es correcto.Así se acotarán en mayor medida posibles errores de análisis producidos por una defectuosa importación del código fuente del sistema.

4.3.4 Iteración 3: Generación del *InventoryModel*

Como primera parte de la transformación del sistema, y siguiendo el planteamiento establecido tanto en el artículo referenciado en secciones anteriores de este documento (véase § 3.8), se establece como objetivo de la iteración el generar una primera aproximación al modelo de representación. En este primer modelo se incluirá el *InventoryModel* resultado del análisis de los artefactos que componen el sistema.

4.3.5 Iteración 4: Transformación de tipos internos

En la siguiente iteración, se aplicará la primera fase de la transformación. Se obtendrá como resultado una instancia del objeto *CodeModel* definido en el paquete Code de KDM, por cada unidad del *InventoryModel* analizada de los tipos *SourceFile* y *DescriptionResource*. Además se identificarán las clases y su identificación extrayendo la información desde el código fuente.

4.3.6 Iteración 5: Transformación de miembros internos y métodos

Una vez identificadas los objetos *ClassUnit*, en esta fase se identificarán los *LanguageUnit* que definen el lenguaje de programación de los archivos con código fuente. También se analizarán todos los tipos de funciones presentes, y la definición de miembros de clases que se utilicen. Por otra parte, y al ser Powerscript un lenguaje donde todos los objetos creados extienden de objetos propios, se establecerán estas relaciones entre *ClassUnit* padre e hijos.

4.3.7 Iteración 6: Transformación de sentencias

Como última iteración del desarrollo, se abordará la creación de objetos del paquete Action de los artefactos *BlockUnit*, siguiendo el método establecido por MoDisco y utilizando el API que proporciona libremente a la comunidad de desarrolladores.

Capítulo 5

Resultados

En este capítulo se presentan los resultados de las iteraciones descritas en la sección anterior.

5.1 Iteración 0: Planificación de las iteraciones

Como inicio del proyecto y siguiendo las tareas definidas en la metodología PAU, se realiza una primera iteración en la que se abordará la identificación del alcance del producto, usando como base una reunión con los depositarios finales del producto a desarrollar. A través de ella se recopilarán las necesidades y requisitos que planteen dichos depositarios, pudiendo plantearse una aproximación arquitectural.

Iteración 0	*«Planificación de las iteraciones y estructura técnica»*
Fase PUA	Iniciación, Elaboración
Actividades	Modelado de la arquitectura de alto nivel, Establecer conf. del entorno, Determinar factibilidad, Crear relaciones con los interesados
Entrada	Salida
Reunión con los depositarios finales	Listado de requisitos
Eclipse IDE y Librerías externas	Base arquitectural
	Sistema configurado de trabajo

Cuadro 5.1: Iteración 0

En la tabla 5.1 se pueden ver los artefactos de entrada y los resultados que se obtendrán como salida.

Reunión con los depositarios finales

Gracias a la reunión mantenida con los responsables de desarrollo de Savia[Sav], se recaban los siguientes aspectos a tener en cuenta:

- Se establece la necesidad de la realización del presente proyecto para evitar los posibles riesgos ya definidos en la introducción del documento (Ver 1).
- Se valida la librería y los ficheros de resultados producidos por MoDisco como solución final del proyecto y base para los siguientes procesos de ingeniería directa que se

deberán llevar a cabo.

- El depositario accede a proporcionar acceso a parte de su código fuente, sin poder ser liberado o reproducido públicamente, para su estudio y análisis.
- Se identifica el listado de requisitos que deberá cumplir el desarrollo final, como validación de los objetivos planteados en capítulos anteriores. (Ver 2)

Creación del entorno de trabajo

Para establecer el entorno de trabajo, se crea un proyecto nuevo asociado a la cuenta pública de Github del autor de este PFC:

https://github.com/darioaxel/PowerScriptToKDMTransformer.

Este repositorio de versiones será importado y usado como base para trabajar, permitiendo así que cualquier persona interesada pueda realizar aportaciones y/o usarlo en un futuro, respetando la licencia establecida.

También se procede a la instalación de las diversas librería que se utilizarán, así como a la descarga del entorno de desarrollo Eclipse.

Listado de requisitos

Arquitectura base

Como base arquitectural del sistema se utilizará una adaptación a las necesidades del proyecto, del diseño establecido por C.Wulf en [Wul12].

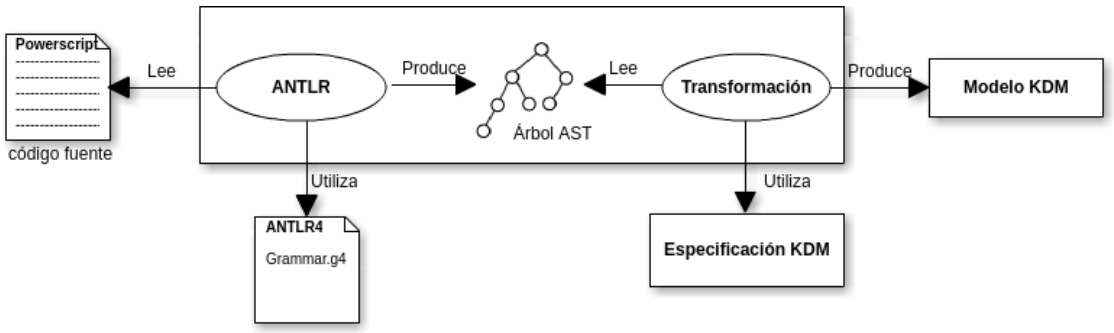

Figura 5.1: Base arquitectural del sistema

Se pueden identificar en la 5.1 los elementos base que se han de desarrollar en este PFC:

- La gramática completa para el análisis del lenguaje Powerscript.
- El sistema de transformación que permita generar el modelo KDM.

5.2 Iteración 1: Desarrollo de la gramática ANTLRv4 de PowerBuilder

En la primera iteración del proceso de desarrollo se cumplimentaran todas las fases de un ciclo de la metodología, además de incorporar varios ciclos de desarrollo basado en TDD. El objetivo principal de esta iteración tal y como se definió en la fase 0 del proyecto será: *Generar una gramática que permita procesar el lenguaje Powerscript*.

Iteración 1	*«Desarrollo de la gramática ANTLRv4 de PowerBuilder»*
Fase PUA	Modelado, Implementación
Actividades	Prototipado técnico, Construcción, Pruebas,
Entrada	Salida
Archivos de código fuente Powerscript	Gramática del lenguaje Powerscript
Documentación técnica Powerscript	

Cuadro 5.2: Iteración 1

En la tabla 5.2 se pueden ver los artefactos de entrada y los resultados que se obtendrán como salida.

5.2.1 Planificación y requerimientos

Para abordar esta parte de la iteración, inicialmente se realiza una búsqueda de gramáticas que pudiesen servir de base en la creación de la que se plantea como objetivo. Después de investigar acerca de las gramáticas ANTLRv4 se comprueba que no existe ninguna propia del lenguaje de programación Powerscript en el repositorio github del proyecto[ANT]. Por ello se pasa a realizar un análisis pormenorizado de una colección de ficheros de código extraídos del software heredado para crear la gramática desde cero. Como apoyo se utilizará tanto la guía de referencia del lenguaje [Inc08] como las estructuras de otras gramáticas del proyecto ANTLRv4 que puedan servir de orientación.

5.2.2 Análisis y diseño

Una vez analizados numerosos ficheros de código fuente se establecen dos conjuntos de artefactos a tratar:

- Estructuras comunes con otros lenguajes de programación conocidos.
- Estructuras propias de Powerscript.

A continuación se abordan ambos grupos de artefactos y en base a ellos se identifican los test mínimos que una vez resueltos en fases siguientes de la iteración permitan alcanzar el objetivo propuesto.

Estructuras comunes

Se analizarán ahora todas aquellas estructuras que se puedan relacionar con las más comunes de otros lenguajes de programación conocidos, como pueden ser las listadas a continuación:

Palabras reservadas del lenguaje, tipos de datos y comentarios Tanto las palabras reservadas como los tipos de datos, pueden encontrarse fácilmente en los primeros capítulos de la guía de desarrollo [Inc08]. Además los comentarios no difieren de los que se utilizan en los lenguajes de programación mas usuales. Por tanto, es sencillo establecer un primer test que además nos servirá para comprobar el entorno de desarrollo establecido en la iteración anterior.

El primer test, denominado «TestComments.txt» se crea y adjunta al proyecto en la carpeta: \resources\basics[1]

Declaración de variables y constantes La forma en la que se declaran las variables y las constantes, salvo por los tipos propios del lenguaje, también es idéntica a la que se utiliza en otros lenguajes de programación.

En este punto se ha de resaltar una característica propia del lenguaje a tener en cuenta y que va a condicionar todo el desarrollo de la gramática:

No existen identificadores terminales de las sentencias del lenguaje Powerscript.

```
1  type variables
2  boolean ib_Painting
3  integer width = 251, height = 72
4  fontcharset fontcharset = "ansi"
5  end variables
```

Listado 5.1: Definición de variables

Como se puede comprobar en el listado 5.1, la única forma de discernir cuándo termina una sentencia y comienza otra es mediante el objeto \n que marca el fin de la línea de texto.

Para validar esta parte de la gramática se establecen una serie de test que se adjuntan a la carpeta: \resources\literals y \resources\members\constants1[0].

Bucles Dentro de Powerscript encontramos los siguientes bucles de flujo:

- IF-THEN-ELSEIF-THEN

[1] Las rutas referenciadas se encuentran en DVD del proyecto adjunto a este documento

- FOR
- DO-WHILE
- TRY-CATCH
- CHOOSE

Para validar esta parte de la gramática se establecen una serie de test que se adjuntan a la carpeta $\backslash\text{resources}\backslash\text{statements}1^0$, con test definidos para los distintos bucles que se describen.

Sentencias SQL Dentro del código de un objeto PowerBuilder existe la capacidad de realizar directamente sentencias SQL y que son lanzadas a una conexión previamente establecida a una base de datos. Esta parte de la gramática es importante puesto que uno de los requisitos establecidos por la empresa Savia, es posibilitar la recuperación de las llamadas a base de datos desperdigadas por el sistema, que hacen de la tarea de modificación de cualquier campo o tabla de una base de datos un proceso muy tedioso.

- COMMIT
- CONNECT
- DISCONNECT
- ROLLBACK
- SELECT
- UPDATE
- INSERT
- DELETE
- OPENCURSOR
- CLOSECURSOR
- DECLARECURSOR
- FETCHCURSOR

Funciones La estructura de las cabeceras de las funciones es casi idéntica a la que existe en lenguajes como C o Java:

modificador de visibilidad + tipo del objeto retornado + identificador + parámetros

Se intentará por lo tanto reutilizar parte de alguna de las gramáticas oficiales para este punto.

Expresiones Este es uno de los puntos mas complejos a desarrollar por la gran variedad de expresiones que se pueden utilizar en un lenguaje de programación. Además y para facilitar la utilización de los métodos que ANTLRv4 genera como llamadas al objeto Listener, usaremos una de las novedades de la versión que permite identificar mediante alias cada una de las reglas derivadas de una regla padre, generando un método de entrada y otro de salida por cada alias establecido.

- Expresiones de comparación (<,>,==,>=, ...)
- Expresiones matemáticas (+,-,*,...)
- Expresiones de estructuras anidadas, por ejemplo: estructura.objeto.parametro
- Anidamiento de paréntesis y corchetes
- Expresión tricondicional

Estructuras propias

En esta sección se analizarán las estructuras propias del lenguaje y que no se pueden relacionar directamente con las utilizadas en otros lenguajes conocidos. Por tanto esta parte de la gramática deberá ser creada «ad hoc» para el proyecto. De cada uno de los tipos de estructuras se crearán uno o más test, dependiendo de la complejidad de la misma, incluyéndose en el directorio \resources\statements del proyecto y subdirectorios del mismo.

Estructura «forward» Se trata de la estructura que se encuentra al comienzo de cualquier fichero de código Powerscript. No se repite en el resto del código y dentro de ella se anidan estructuras de varios de los tipos que se analizan a continuación.

```
1  forward
2  global type w_trasp_dota from w_selection
3  end type
4  type cb_selectall from u_cb within w_trasp_dota
5  end type
6  type dw_input from datawindow within w_trasp_dota
7  end type
8  end forward
```

Listado 5.2: Estructura «forward» del fichero **w_trasp_dota.srw**

Estructura «type» Esta estructura, con o sin el modificador *global*, puede encontrarse tanto dentro de una estructura «forward» como se puede ver en la figura 5.2 o como estructura aislada dentro código.

Estructura «variables» Entre las etiquetas de inicio y fin de una estructura «variables», tal y como se aprecia en la figura 5.1 se realiza la definición e instanciación de las variables internas al objeto.

Estructuras «prototipos de funciones Al igual que el resto de estructuras propias abordadas hasta ahora, se compone de una etiqueta inicial y una final. En el bloque definido entre ambas etiquetas, se realiza el listado de cabeceras de funciones con un estilo similar al utilizado en ANSI C.

Estructuras «event» y «on» De forma parecida a las funciones se definen estas dos estructuras, de las que se observa gran similaridad gramatical y que se abordarán inicialmente como una regla única con distintas variaciones en función del tipo de la misma.

Comandos propios Además de las estructuras anteriormente citadas, el lenguaje presenta una serie de comandos propios que se listan a continuación:

- CALL
- SUPER
- EXIT
- RETURN

5.2.3 Desarrollo, test y validación

Durante el proceso de desarrollo se realizan 5 iteraciones dirigidas por los bloques de test establecidos en la fase anterior de la iteración:

Iteraciones de desarrollo mediante TDD	
Iteración 1	Comentarios, palabras reservadas y definición de literales
Iteración 2	Definición de variables y constantes
Iteración 3	Expresiones
Iteración 4	Funciones, bucles, SQL
Iteración 5	Estructuras propias de Powerscript

Cuadro 5.3: Iteraciones de desarrollo TDD

Como muestra de ello se adjunta el listado (ver 5.2) resultante del primer bloque de testing que incluye las «Estructuras comunes» (ver 5.2.2)

El ciclo de desarrollo, tal y como se estableció en la metodología, se realiza de manera iterativa y generando como resultado de cada iteración una gramática que permite pasar

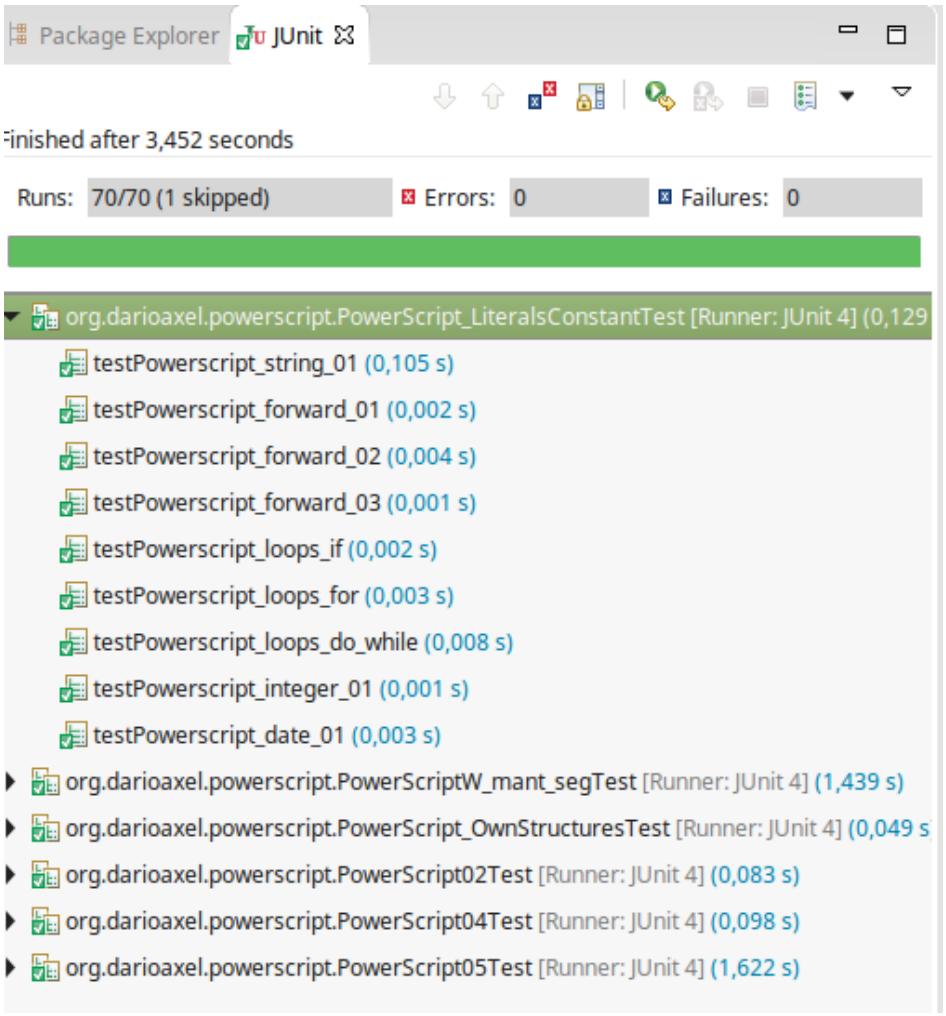

Figura 5.2: Resultados del testing de estructuras comunes

el/los test previamente creados en la fase de análisis. Además al ser utilizada como base de una iteración, la gramática resultante de la iteración anterior, al finalizar también se deben de pasar los test de dicha iteración previa.

5.3 Iteración 2: Sistema de validación del software heredado

Una vez desarrollado la gramática principal del proyecto, se plantea una nueva iteración dirigida a desarrollar la lógica necesaria para la validación de la estructura del proyecto.

La necesidad de realizar un validador del proyecto heredado, surge por la constatación de que los ficheros de PowerBuilder se encuentra altamente ligado a su IDE de trabajo y no se permite la obtención de ellos en formato texto directamente. Para poder recuperar los archivos fuente se necesita de la utilización como intermediario de un gestor de versiones autorizado por PowerBuilder.

Con este sistema de validación el sistema podrá cerciorarse de que la estructura de la aplicación y las librerías que se generan son consistentes, y que todos los objetos que el compilador tratará para generarlas están presentes.

Iteración 2	*«Sistema de validación del software heredado»*
Fase PUA	Modelado, Implementación
Actividades	Prototipado técnico, Construcción, Pruebas,
Entrada	Salida
Archivos de código fuente Powerscript	Gramática del archivos descriptores de proyecto
Documentación técnica Powerscript	Gramática del archivos descriptores de librerías
	Prototipo que permita validar una aplicación

Cuadro 5.4: Iteración 2

En la tabla 5.3 se establecen los objetos que se aportarán como salida una vez finalizada la iteración actual.

5.3.1 Planificación y requerimientos

En PowerBuilder, los objetos librería y aplicación, son dos archivos de código que no se encuentran escritos en el lenguaje de scripting. Por ello se necesitarán implementar dos nuevas gramáticas capaces de analizar su contenido y obtener las reglas que en su código se definen.

Una vez definidas ambas gramáticas, se procederá a recorrer el árbol de directorios del proyecto, recuperando los ficheros a procesar y validando cada uno de ellos.

Como test utilizaremos una pequeña aplicación creada utilizando PowerBuilder que utiliza la misma estructura que las aplicaciones reales propiedad de Savia. Esta aplicación se encuentra en la ruta $\backslash\text{resources}\backslash\text{advanced}\backslash\text{real}\backslash\text{myproject1}^0$

5.3.2 Análisis y diseño

En el análisis de la iteración se observa, además de la necesidad comentada en el punto anterior de crear dos gramáticas con ANTLRv4 para los objetos de tipo librería y proyecto de

PowerBuilder, que ha de realizarse un recorrido completo del árbol de directorios y tratar los archivos encontrados. Una de las aportaciones la ingeniería del software es el establecimiento de patrones de desarrollo, es decir, soluciones a problemas conocidos y que se repiten con frecuencia. Estos patrones permite generar soluciones muy sencillas de adaptar a problemas de diseño concretos.

Patrón Observador

El patrón «Observer» es un patrón de comportamiento que permite relacionar diferentes objetos entre si en torno a uno Principal, así cada vez que este ultimo cambie su estado, los demás también cambiaran de forma automática. Podemos decir entonces que dicho patrón se compone de un objeto «observable» y diversos objetos «observadores»

Un objeto «observable» puede tener uno o mas «observadores», a los que se informa mediante la llamada a un método común de interfaz que implementan todos ellos.

Este patrón es el utilizado por ANTLRv4 en una de las características presentes en su última versión y que se han descrito con anterioridad en este documento. (ver 3.6). En la imagen 5.3, se puede ver cómo es desarrollado.

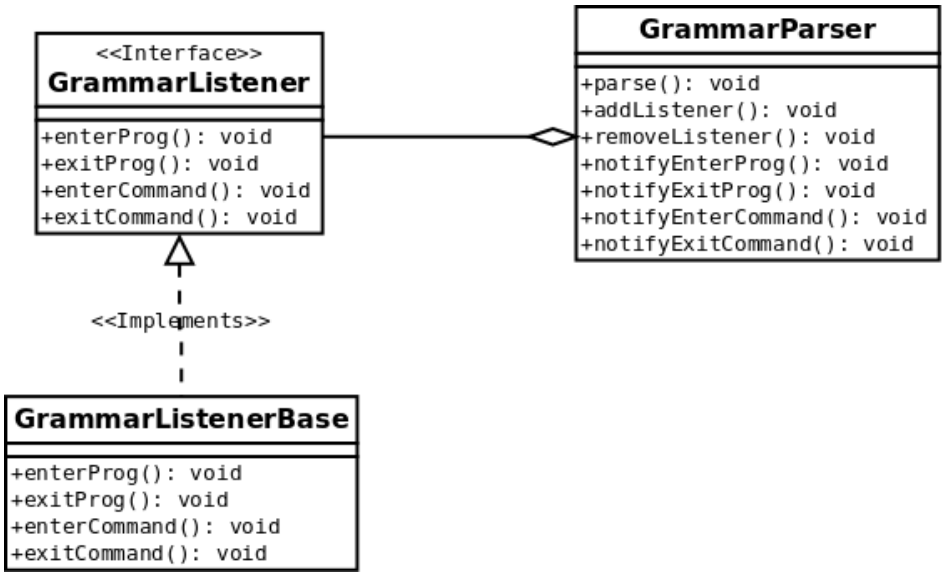

Figura 5.3: Patrón observador

5.3.3 Desarrollo, test y validación

La forma por la que se validará un proyecto será la siguiente (5.4):

1. Se valida la existencia de un objeto descriptivo del proyecto.
2. Se analiza el contenido del objeto descriptor y se extraen de él las librerías que lo componen
3. Se valida la existencia de cada uno de los objetos descriptores de librería en el sistema.

4. Se analiza el contenido de la librería y se extraen los objetos que el compilador utiliza para generar dicha librería.
5. Se valida la existencia de cada no de los objetos que componen la librería y se valida el lenguaje con la gramática de la iteración 5.2.

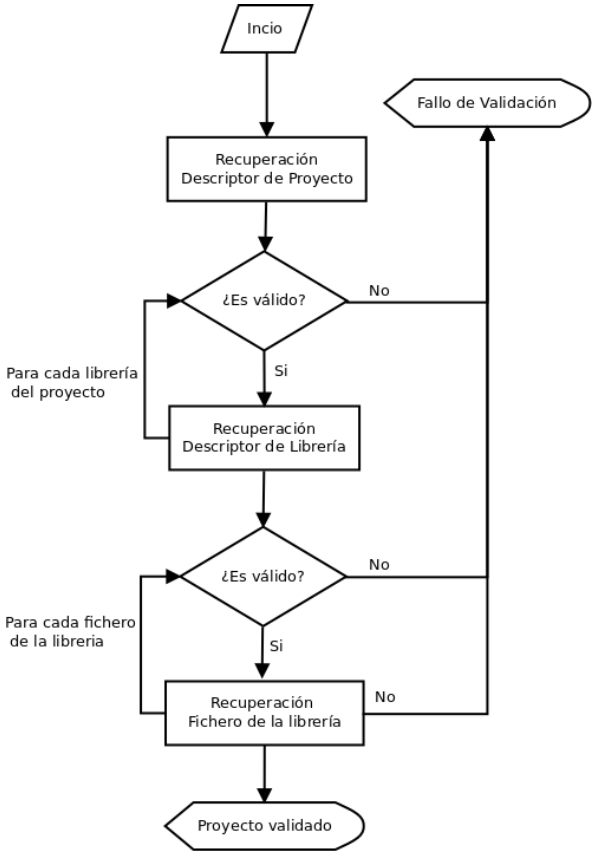

Figura 5.4: Secuencia del proceso de validación

Dentro del paso actual, y siguiendo la base TDD para el desarrollo establecida, se genera un conjunto de test unitarios bajo la clase PowerbuilderProjectValidatorTest.java que son necesariamente validados para poder dar por finalizada esta iteración.

5.4 Iteración 3: Generación del *InventoryModel*

Dentro del paquete *Source* de KDM, en el que se definen los meta-elementos que representan todos los artefactos físicos existentes en el sistema en estudio, tenemos el diagrama denominado *InventoryModel* (ver 5.5). Este diagrama identificará cada uno de los elementos del sistema, y los clasificará por la naturaleza de los mismos. Además también permite representar las relaciones establecidas entre elementos tales como herencia o interdependencia.

Iteración 3	*«Generación del InventoryModel»*
Fase PUA	Modelado, Implementación
Actividades	Prototipado técnico, Construcción, Pruebas,
Entrada	Salida
Archivos de código fuente Powerscript	Objetos de recorrido del árbol del sistema
Librería MoDisco	Modelo base de KDM con la representación de artefactos del sistema heredado

Cuadro 5.5: Iteración 3

5.4.1 Planificación y requerimientos

En esta iteración se realizará una implementación de una clase «walker» o «visitor» que permita recorrer el árbol de directorios del sistema de forma iterativa y que además pueda ser reutilizada como base para posteriores recorridos.

Por otro lado, se implementará una solución para la creación de las instancias de KDM utilizando la librería MoDisco, de la forma mas desacoplada posible y que permita no tener que importar artefactos de dicha librería en todas las clases a desarrollar.

Como resultado de la iteración obtendremos una primera versión válida del modelo KDM que represente lo mas fielmente que sea posible el contenido del conjunto de elementos que forman el proyecto de test que se creó en la iteración anterior.

5.4.2 Análisis y diseño

En la fase de análisis de esta iteración, se reconocen dos patrones que servirán de base para la implementación y se describen a continuación:

Patrón Visitador

El patrón visitador es descrito como: *La representación de una operación realizada sobre elementos presentes en una estructura. El objeto visitador permite definir una nueva operación sin necesidad de cambiar las clases de los elementos con los que opera*[Gam]

En el desarrollo del proyecto, utilizaremos este patrón para recorrer las diferentes estructuras (*InventoryModel*, árbol de directorio del sistema), realizando operaciones diferentes sobre cada uno de los elementos cuando son visitados.

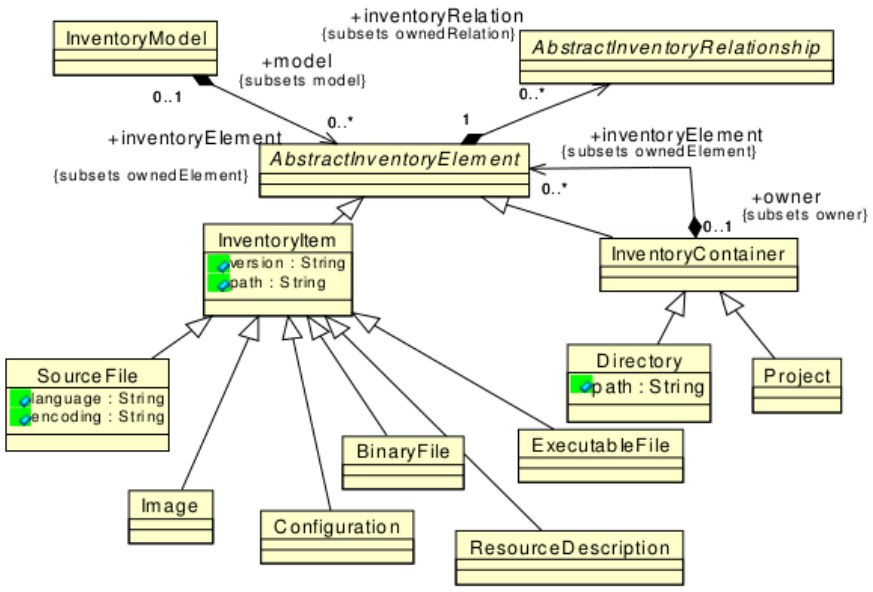

Figura 5.5: Diagrama InventoryModel [Gro]

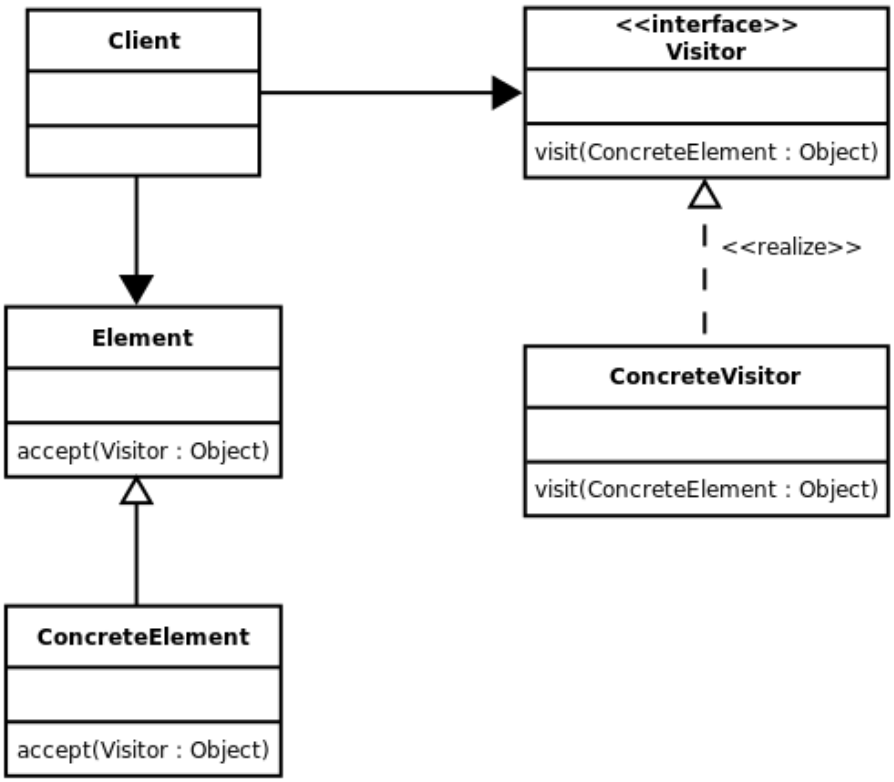

Figura 5.6: Patrón visitador

En la figura 5.6 vemos el diagrama que nos servirá como guía a la hora de desarrollar las clases.

Patrón Static Factory Method

En diseño de software, el patrón de diseño Factory Method consiste en utilizar una clase constructora (al estilo del Abstract Factory) abstracta con unos cuantos métodos definidos y otro(s) abstracto(s): el dedicado a la construcción de objetos de un subtipo de un tipo determinado. Es una simplificación del Abstract Factory, en la que la clase abstracta tiene métodos concretos que usan algunos de los abstractos; según usemos una u otra hija de esta clase abstracta, tendremos uno u otro comportamiento. (Ver figura 5.7)

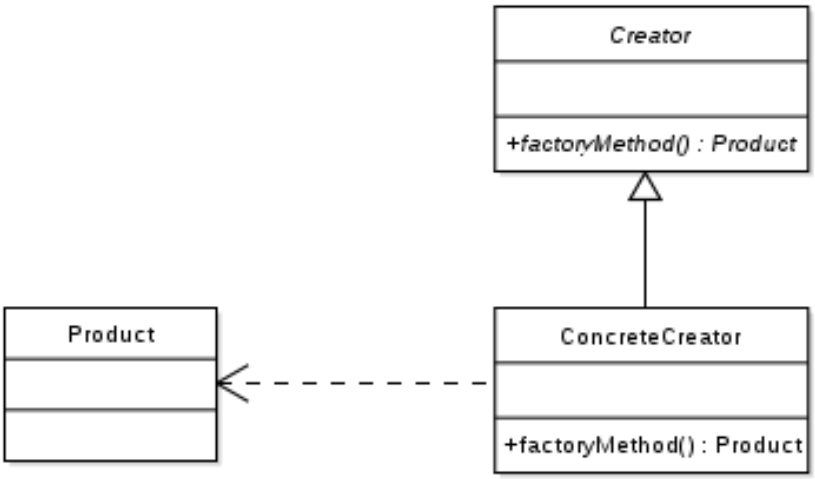

Figura 5.7: Patrón Factory Method

En el desarrollo de este PFC se implantará una variante del patrón Factory Method que utilizando clases estáticas como objetos constructores aporta las siguientes ventajas:

- No tiene que retornar necesariamente un objeto de la clase en la que fue llamada.
- El nombre del método puede dar una mejor descripción de cómo es el objeto que devuelve. Por ejemplo *Empleados::ObtenerEquipo(recursosHumanos)*
- Puede devolver un objeto ya creado mientras que un constructor siempre crea una nueva instancia del objeto.

Finalmente apuntar la ventaja que a la hora de utilizar un framework de Inyección de Dependencias tendrá el uso de los objetos basados en este patrón.

5.4.3 Desarrollo, test y validación

El desarrollo de esta iteración tiene como objetivo representar el modelo KDM que contenga el diagrama *InventoryModel* cuya estructura y elementos se pueden ver en 5.5.

En una primera iteración del proceso TDD, se realiza la implementación de las clases que se pueden ver en la figura 5.8 utilizando los patrones anteriormente analizados. Se puede comprobar como la clase estática *FileUtils.class*, realiza la visita de cada fichero del árbol de directorios, informando al *InventoryModelFileListener.class* de la recuperación de cada

elemento. Es esta segunda clase la que se encarga de inventariar y agregar al contenedor *InventoryModel* de KDM el artefacto correspondiente.

Como test se implementa la clase *createInventoryModelTest.class* que genera un primer modelo denominado *createInventoryModelTest.xmi*.

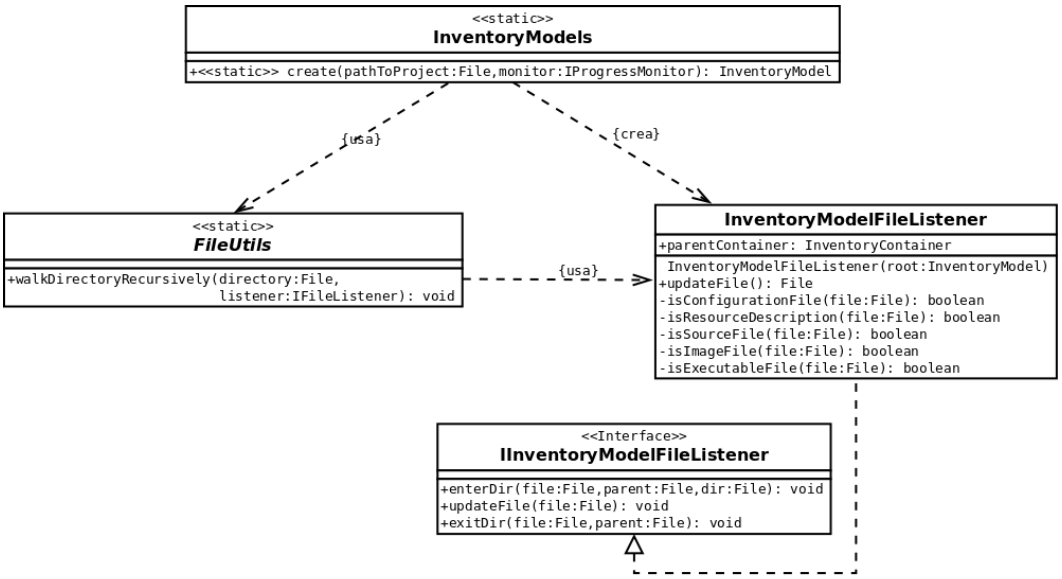

Figura 5.8: Diagrama de clases del InventoryModel

Una vez superada esta primera iteración TDD correctamente se aprecia que el meta modelo KDM no está recogiendo toda la información presente en el proyecto, ya que los artefactos *ResourceDescription* y *SourceFile* no contemplan una especialización de los mismos. Para aportar la información relativa al tipo de objeto que cada artefacto tiene dentro de PowerBuilder (Ver tabla 3.4, realizamos una nueva iteración. En ella se incluye un artefacto *Attribute* que informa del tipo de objeto de cada elemento.

Esta segunda iteración TDD genera el modelo *createInventoryModelWithAttributesTest.xmi*.

```
/*
* Original Author: Dar o Ure a
* E-Mail: darioaxel@gmail.com
*/

grammar powerscriptPBG;

@header {
package org.darioaxel.grammar.powerscript.pbg;
}

prog
 : header libraries objects EOF
 ;

header
 : 'Save Format v3.0(' NUMBER ')'
 ;

libraries
 : HEADER_BEGIN pathsFromTo+ ENDS SEMICOL
 ;

objects
 : OBJECTS_BEGIN pathsFromTo+ ENDS SEMICOL
 ;

pathsFromTo
 : quotedPath quotedPath SEMICOL
 | quotedPath QUOTE QUOTE SEMICOL
 ;

quotedPath
 : QUOTE path QUOTE
 ;

path
 : (ID DOUBLESLASH)*? file
 ;

file
 : ID DOT ID
 ;

HEADER_BEGIN : '@begin Libraries';
OBJECTS_BEGIN : '@begin Objects';
ENDS : '@end' ;

NUMBER : [0-9]+;
ID   :    [a-zA-Z0-9_]+ ;
QUOTE : '"';
DOT: '.';
DOUBLESLASH : '\\\\';
SEMICOL : ';' ;
WS: [ \t\n\r]+ -> skip;
```

Listado 5.3: Gramática generada para ficheros de librería

```
1 <model xsi:type="source:InventoryModel" name="source references">
2   <inventoryElement xsi:type="source:Directory" name="myproject" path="/
      home/darioaxel/git/PowerScriptGrammar/resources/advanced/real/
      myproject">
3     <inventoryElement xsi:type="source:SourceFile" name="m_login.srm"
        version="1467536741000" path="/home/darioaxel/git/
        PowerScriptGrammar/resources/advanced/real/myproject/m_login.srm"
        language="Powerscript" encoding="not checked"/>
4     <inventoryElement xsi:type="source:BinaryFile" name="euskera.lng"
        version="1403631220000" path="/home/darioaxel/git/
        PowerScriptGrammar/resources/advanced/real/myproject/euskera.lng"/
        >
```

Listado 5.4: Detalle resultado iteración 1 tdd

5.5 Iteración 4: Transformación de tipos internos

Esta iteración se alinea con la primera fase de transformación establecida en el documento [Wul12].

Iteración 4	*«Fase1 : Transformación de tipos internos»*
Fase PUA	Modelado, Implementación
Actividades	Prototipado técnico, Construcción, Pruebas,
Entrada	Salida
InventoryModel	Sistema base de fases de transformación
Librería MoDisco	Modelo KDM de la fase 1
Análisis de elementos del código Powerscript	

Cuadro 5.6: Iteración 4

5.5.1 Planificación y requerimientos

Utilizando como base el modelo obtenido en la iteración anterior, y establecida ya una primera aproximación estructural del proyecto, centraremos este paso del ciclo de vida en la recuperación de los artefactos que representan los *CompilationUnit* y *ClassUnit* del meta modelo KDM. El sistema base de fases de transformación resultante será utilizado en las siguientes iteraciones como estructura sobre la que se añadirán los desarrollos que permitan obtener el modelo resultante.

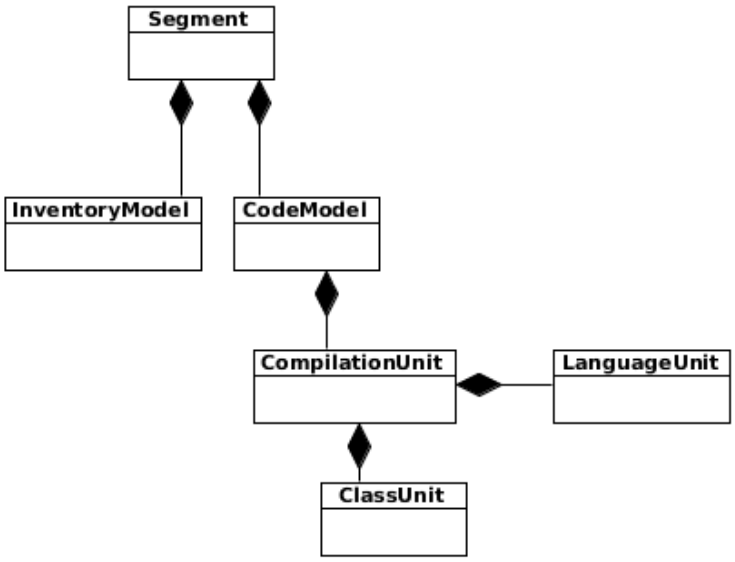

Figura 5.9: Visualización simplificada del modelo KDM generado en esta iteración

5.5.2 Análisis y diseño

En el diagrama de la figura 5.9 se establecen los artefactos a generar en este paso del desarrollo. Debido a la estructura de PowerBuilder, no se pueden establecer relaciones directas entre los artefactos identificados en KDM y los propios del lenguaje. Por ello, se realiza un análisis pormenorizado de los objetos presentes en Powerscript, para extraer el conocimiento necesario para poder establecer una interrelación. Todo este mapeo entre PowerBuilder y KDM se realiza desde cero y teniendo como guía el documento de definición del metalenguaje (ver [Gro]).

LanguageUnit

El *LanguageUnit* es una representación de los tipos de datos predefinidos del lenguaje en el que están escritos cada uno de los *CompilationUnit* que forman el modelo.

CompilationUnit

KDM define el artefacto *CompilationUnit* como el elemento capaz de representar un contenedor lógico para otros elementos del sistema, que es lo suficientemente completo para ser procesado por el entorno de desarrollo. Así pues se puede establecer una relación directa entre los ficheros objeto de PowerBuilder que tienen lógica (window, menu, etc) y el artefacto del metamodelo.

ClassUnit

Dado que los objetos de PowerBuilder no disponen de una definición de clases como tienen otros lenguajes, no se puede inferir directamente este artefacto. Sin embargo, y tomando la descripción del elemento que en la definición de [Gro] se da: *« [...] una colección ordenada de elementos identificados, cada uno de los cuales puede ser otro CodeItem del tipo MemberUnit o MethodUnit.»*

Permite establecer que, el propio objeto instanciado en el fichero de código debería de ser considerado como un *ClassUnit*.

Al analizar el código fuente de múltiples objetos de los tipos señalados, procedentes de los proyectos reales aportados por el depositario final, se puede extraer un patrón. Este patrón viene a definir la presencia de una estructura común inicial de tipo «forward». Esta estructura define un bloque de código en el que se anidan una o más subestructuras «type», que a su vez define miembros cuyo identificador se repite también en el nombre de este u otros objetos.

En el ejemplo de la figura 5.10 podemos ver como el identificador «w_clave» que se encuentra en la línea 2 y que instancia el elemento «type» con un definidor de ámbito «global» coincide con el nombre del objeto. Se puede inferir por tanto del estudio de este ejemplo que: *El CompilationUnit identificable por el nombre «w_clave.srw», contiene un ClassUnit*

w_clave.srw

```
1  forward
2  global type w_clave from window
3  end type
4  type st_2 from statictext within w_clave
5  end type
6  type st_1 from statictext within w_clave
7  end type
8  type cb_aceptar from commandbutton within w_clave
9  end type
10 type sle_clave from singlelineedit within w_clave
11 end type
12 end forward
13
14 global type w_clave from window
15 integer width = 1518
16 integer height = 580
17 boolean titlebar = true
18 string title = "Clave acceso"
19 boolean controlmenu = true
20 windowtype windowtype = "response"
21 long backcolor = 67108864
22 string icon = "UBD.ico"
23 boolean center = true
24 st_2 st_2
25 st_1 st_1
26 cb_aceptar cb_aceptar
26 sle_clave sle_clave
28 end type
29 global w_clave w_clave
```

Figura 5.10: Ejemplo de objeto PowerBuilder

de nombre «w_clave».

Patrón Strategy

Uno de los cinco principios básicos del desarrollo del software SOLID establecidos por C.Martin en la década del año 2000, conocido como «Principio de abierto-cerrado» establece que:

«las entidades de software . . . deben estar abiertas para su extensión, pero cerradas para su modificación»

El patrón de desarrollo de software que se extrae de este principio y que se muestra en la imagen 5.11, permite a través de la utilización de la interfaz común, en vez de la de una clase concreta, minimizar la acoplación del sistema. Así pues cualquier clase que implemente la interfaz, podrá ser utilizada sin que su sustitución futura por otra implique un impacto negativo en el sistema completo.

Este patrón implementado dentro del proyecto, habilitará la reutilización de la estructura

de transformaciones, teniendo que cambiar únicamente la implementación de los listeners asociados a cada una de las transformaciones y que implementan el objeto base proporcionado por ANTLRv4.

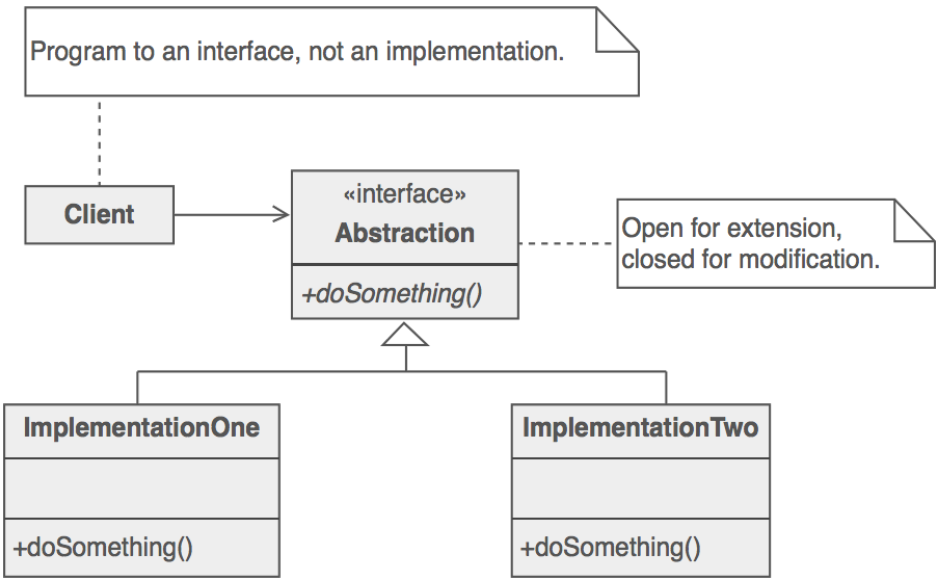

Figura 5.11: Patrón Strategy

5.5.3 Desarrollo, test y validación

A partir del análisis del punto anterior se realiza el desarrollo de la aplicación usando como test la aplicación referenciada en 5.3.1.

En la siguiente figura se muestra el diagrama de clases generado a partir de estado actual de la aplicación:

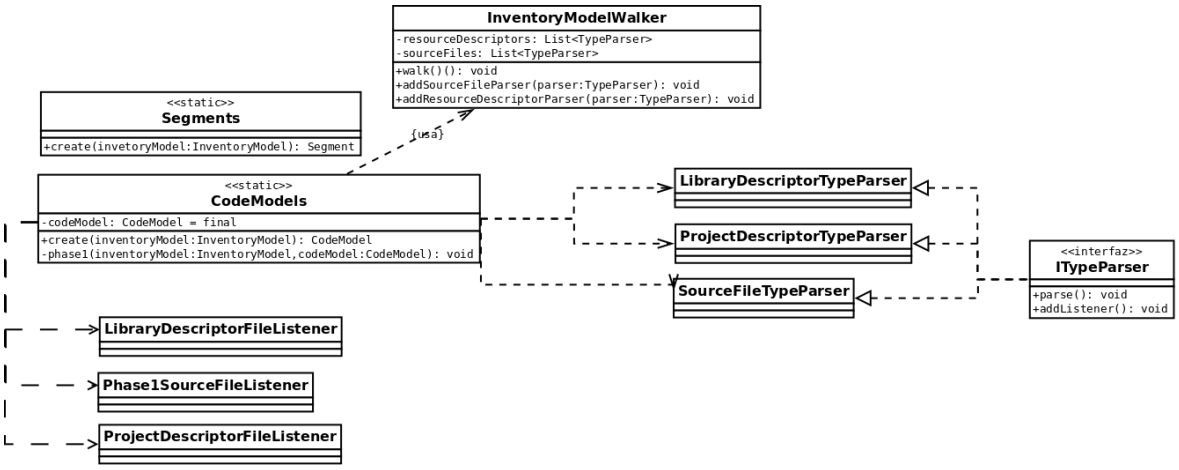

Figura 5.12: Diagrama de clases de la fase 1

El resultado del test propuesto para esta etapa de desarrollo queda generado en el modelo

createPhase1ModelTest.xmi. Se reproduce en la siguiente imagen un extracto del mismo donde se puede apreciar el detalle del modelo obtenido.

```
1 <model xsi:type="code:CodeModel" name="">
2   <codeElement xsi:type="code:CompilationUnit" name="m_login.srm">
3   <source>
4     <region file="//@model.0/@inventoryElement.0/@inventoryElement.0"
          path="/home/darioaxel/git/PowerScriptGrammar/resources/advanced/
          real/myproject/m_login.srm"/>
5   </source>
6   <codeElement xsi:type="code:ClassUnit" name="m_login">
7     <annotation text="Menu Type"/>
8   </codeElement>
9 </codeElement>
```

Listado 5.5: Detalle resultado fase 1 transformación

5.6 Iteración 5: Transformación de miembros internos y métodos

En esta segunda fase de las tres que componen el ciclo de transformación, se realizará la ampliación del modelo ya creado en la fase anterior añadiendo los artefactos que representan las definiciones de miembros y métodos.

Iteración 5	*«Fase2 : Transformación de la declaración de miembros y métodos»*
Fase PUA	Modelado, Implementación
Actividades	Prototipado técnico, Construcción, Pruebas,
Entrada	Salida
InventoryModel	Sistema resultante de la fase 2 de transformación
Librería MoDisco	Modelo KDM de la fase 2
Análisis de elementos del código Powerscript	
Sistema resultante de la fase 1	

Cuadro 5.7: Iteración 5

5.6.1 Planificación y requerimientos

Como se puede ver en la figura 5.13 que representa el diagrama del modelo resultante tras esta iteración, los artefactos que incluiremos en él serán:

- *MethodUnit* para representar los métodos y funciones, así como *ParameterUnit* para sus parámetros en caso de tenerlos.
- *MemberUnit* como forma de representar las declaraciones de miembros de tipos del sistema heredado.

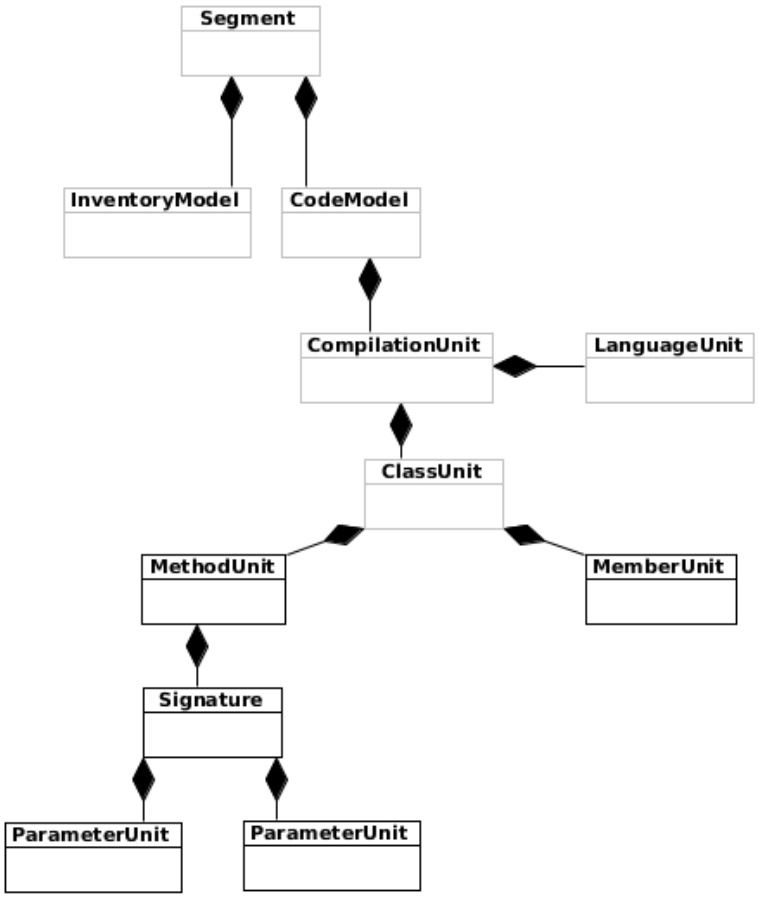

Figura 5.13: Visualización simplificada del modelo KDM generado en esta iteración

5.6.2 Análisis y diseño

Para establecer la relación entre los artefactos de KDM que se han de crear en esta fase y las estructuras recuperadas con la gramática del código fuente, se profundizará en el estudio de las estructuras ya conocidas «forward» y «type». Además también se abordará el análisis de las distintas formas de definición de funciones.

Análisis de la estructura «forward» y «type»

En este punto se continuará el análisis de la figura 5.10 y se seguirá lo establecido en el punto 5.5.2 sobre la relación entre cada uno de las estructuras «type» y las declaraciones de miembros del lenguaje.

Del análisis detallado de una estructura «type» se obtiene:

$$type\ \underbrace{cb\_aceptar}_{Identificador}\ from\ \underbrace{commandbutton}_{Id.padre}\ within\ \underbrace{w\_clave}_{Id.clase}\ endtype$$

Donde el «Id. padre» se refiere al objeto del sistema, predefinido por el lenguaje como en este caso o por el usuario, que indica el tipo del miembro. Y el «Id. clase» refiere a la clase/objeto en la cual será utilizado el miembro. Por tanto el resultado esperado y que nos servirá para testear que el desarrollo es correcto deberá generar una estructura similar a la siguiente:

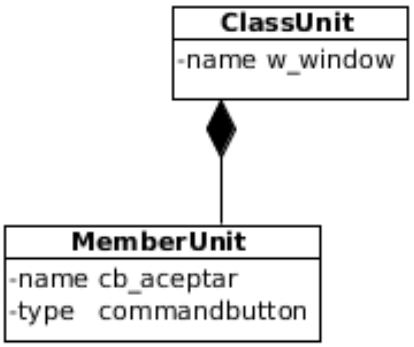

Figura 5.14: Esquema de la definición de un miembro

Análisis de las estructuras «function» y «on»

Del análisis del código de Powerbuilder se extrae que en él se definen dos tipos de métodos: «function» y «on».

«Function» Son estructuras similares a las que se pueden encontrar en C, Java y demás lenguajes de alto nivel. Sus características son:

- Permiten devolver una variable resultado de un tipo concreto que se declara al principio.
- Se identifican con un conjunto de caracteres que ha de comenzar con una letra.
- Obtienen valores de entrada desde una lista de parámetros de valores establecidos.
- En caso de error, permiten lanzar excepciones al sistema.

Además cuentan con una similitud al lenguaje C, y las cabeceras de todas las funciones que se definirán en un objeto han de ser previamente listadas en una estructura. En el la figura 5.6 se puede ver un ejemplo de la estructura «forward prototypes» donde se listan varias funciones.

```
1 forward prototypes
2 public function string of_read_employee (string s_dni)
3 public function string of_read_employee_payroll (string s_dni, integer
    i_month, integer i_year)
4 end prototypes
```

Listado 5.6: Ejemplo de estructuras «function»

Esta estructura proporciona la información suficiente para poder crear elementos *MethodUnit*.

«On» Las estructuras de este tipo, representan conjuntos de operaciones que se ejecutan en momentos puntuales del ciclo de vida del objeto. Como se puede ver en el ejemplo 5.7, los métodos de este tipo vienen asociados a identificadores del lenguaje que definen situaciones como la creación del objeto o la destrucción del mismo.

```
1  on m_marco.create
2  m_marco=this
3  call super::create
4  this.menustyle = "contemporarymenu"
5  this.menutextcolor = 134217735
6  this.menubackcolor = 67108864
7  this.menuhighlightcolor = 134217728
8  this.textsize = 8
9  this.weight = 400
10 end on

12 on m_marco.destroy
13 call super::destroy
14 destroy(this.m_espera)
15 end on
```

Listado 5.7: Ejemplo de estructuras «on»

Para añadir la información del tipo de método llamado, se utilizará el enumerador definido en el metamodelo KDM *MethodKind*, que tiene los siguientes valores:

- **method**
- **destructor**
- **constructor**
- **operator**
- **virtual**
- **abstract**
- **unknown**

5.6.3 Desarrollo, test y validación

Gracias a la utilización de los patrón Strategy en la iteración anterior, el desarrollo en la iteración actual se centra en la creación de una nueva implementación de las clases *FileListener* para agregar el modelo resultante una nueva capa de información. (Ver figura 5.12)

Una de las necesidades que se han de resolver para lograr generar el modelo correctamente es la creación de una clase que permita buscar dentro de la estructura interna del objeto *CodeModel* elementos de un tipo concreto. Para este propósito volveremos a aplicar el patrón «Static Factory Method» y se genera una clase estática fácilmente utilizable en posteriores implementaciones.

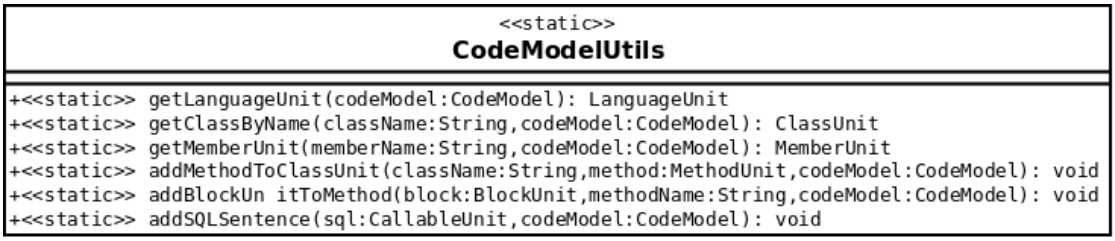

Figura 5.15: Esquema de la definición de un miembro

La clase *CodeModelUtil* que se muestra en la figura 5.15[2], es utilizada como base para la búsqueda y recuperación de elementos del *CodeModel*, así como para añadir artefactos.

El modelo resultante de la etapa de desarrollo se genera en el fichero *createPhase1ModelTest.xmi* y a continuación se adjunta una parte de él en la figura 5.8 que permite comprobar cómo se ha conseguido implementar la estructura descrita en la fase de planificación.

[2]En la imagen sólo se muestran unos pocos métodos representativos, puesto que el listado completo de los más de 25 métodos que posee haría la figura poco intuitiva.

```
<codeElement xsi:type="code:ClassUnit" name="n_param">
  <annotation text="Functions Collection Type"/>
  <codeElement xsi:type="code:MethodUnit" name="of_read_employee">
    <annotation text="Function method"/>
    <codeElement xsi:type="code:Signature">
      <parameterUnit name="s_dni" type="//@model.1/@codeElement.0/
          @codeElement.0/@codeElement.2"/>
      <parameterUnit name="" type="//@model.1/@codeElement.0/
          @codeElement.0/@codeElement.2" kind="return"/>
    </codeElement>
  </codeElement>
  <codeElement xsi:type="code:MethodUnit" name="of_read_employee_payroll
      ">
    <annotation text="Function method"/>
    <codeElement xsi:type="code:Signature">
      <parameterUnit name="s_dni" type="//@model.1/@codeElement.0/
          @codeElement.0/@codeElement.2"/>
      <parameterUnit name="i_month" type="//@model.1/@codeElement.0/
          @codeElement.0/@codeElement.5"/>
      <parameterUnit name="i_year" type="//@model.1/@codeElement.0/
          @codeElement.0/@codeElement.5"/>
      <parameterUnit name="" type="//@model.1/@codeElement.0/
          @codeElement.0/@codeElement.2" kind="return"/>
    </codeElement>
  </codeElement>
  <codeElement xsi:type="code:MethodUnit" kind="constructor">
    <attribute tag="onMethodObject" value="n_param"/>
    <annotation text="On method"/>
  </codeElement>
  <codeElement xsi:type="code:MethodUnit" kind="destructor">
    <attribute tag="onMethodObject" value="n_param"/>
    <annotation text="On method"/>
  </codeElement>
</codeElement>
```

Listado 5.8: Resultado de la fase 2: *MethodUnit*

Se comprueba efectivamente que el resultado representa tal y como se había previsto la definición de la función *of_read_employee_payroll* de la figura 5.6.

Las líneas del resultado 10 y 11 representan a través de un artefacto *MethodUnit* de KDM la definición de la función. Además se ha incluido un *AnnotationUnit* en el que se identifica el tipo para así distinguir este tipo de función de funciones otros como *On* y *Event*. Esta diferenciación se puede ver en la línea 25 de la figura 5.8 que describen un método del tipo *On*.

Para terminar el análisis del resultado de una transformación de un *MethodUnit*, en las líneas 13 a 16 de la figura 5.8, se representan los parámetros de la función en estudio, utilizando los valores de *type* previamente definidos en el lenguaje o *return* para describir el resultado devuelto por la función.

Finalmente se analiza el resultado de la transformación del otro elemento descrito en el análisis: *MemberUnit*. En el código de la figura 5.9 (línea 3) se define el objeto *m_espera* cuyo tipo es enlazado en *type*.

```
1  <codeElement xsi:type="code:ClassUnit" name="m_mydata">
2    <annotation text="Menu Type"/>
3    <codeElement xsi:type="code:MemberUnit" name="m_espera" type="//@model
         .1/@codeElement.1/@codeElement.0" export="protected"/>
4    <codeElement xsi:type="code:MethodUnit" kind="constructor">
5      <attribute tag="onMethodObject" value="m_mydata"/>
6      <annotation text="On method"/>
7    </codeElement>
8    <codeElement xsi:type="code:MethodUnit" kind="destructor">
9      <attribute tag="onMethodObject" value="m_mydata"/>
10     <annotation text="On method"/>
11   </codeElement>
12   <codeElement xsi:type="code:MethodUnit" kind="constructor">
13     <attribute tag="onMethodObject" value="m_espera"/>
14     <annotation text="On method"/>
15   </codeElement>
16   <codeElement xsi:type="code:MethodUnit" kind="destructor">
17     <attribute tag="onMethodObject" value="m_espera"/>
18     <annotation text="On method"/>
19   </codeElement>
20 </codeElement>
```

Listado 5.9: Resultado de la fase 2: *MemberUnit*

Todos los miembros, salvo que se especifique lo contrario, tendrán como modificador de acceso predefinido *protected*. Esta decisión se basa en la definición del lenguaje Powerscript, el cual permite que los objetos que heredan de una clase reutilicen las variables definidas en los objetos de los que heredan, pero no se permite su modificación desde otros objetos externos.[Inc08]

5.7 Iteración 6: Transformación de sentencias

En la tercera y última de las fases de transformación se aborda la representación de sentencias presentes en el cuerpo de las funciones y de la declaración de variables.

Iteración 6	*«Fase3 : Transformación de sentencias»*
Fase PUA	Modelado, Implementación
Actividades	Prototipado técnico, Construcción, Pruebas,
Entrada	Salida
InventoryModel	Modelo KDM de la fase 3
Librería MoDisco	
Análisis de elementos del código Powerscript	
Sistema resultante de la fase 2	

Cuadro 5.8: Iteración 6

5.7.1 Planificación y requerimientos

Los artefactos que en esta fase se añadirán al modelo recibido de la iteración anterior pertenecen al paquete *Action* y representan el comportamiento del sistema estudiado a nivel de implementación. Principalmente serán de los tipos *ActionElement*, *Calls* y *ActionRelations*.

5.7.2 Análisis y diseño

Para obtener un mejor conocimiento de la estructura de elementos del paquete *Action*, se puede ver a modo de ejemplo la figura 5.16. En ella se realiza la representación del resultado de mapear en KDM la llamada a una función que en la línea 2 de la figura 5.7 se muestra.

Análisis de la llamadas a métodos

Una de las sentencias de Powerscript a modelar en esta fase es *Call*. Como su definición indica (Ver [Inc08]) se trata de una llamada a una función de tipo *On*. En el ejemplo que se puede ver en el código de la figura 5.10, se realiza una llamada a la función *create* del objeto *super* que representa al antecesor del que hereda el objeto Powerscript desde el que se realiza la llamada.

```
1  call super::create
```

Listado 5.10: Ejemplo de función

El diagrama de la figura 5.16 basado en la especificación de KDM, aporta la información necesaria para conocer el resultado esperado de la implementación de este tipo de llamadas.

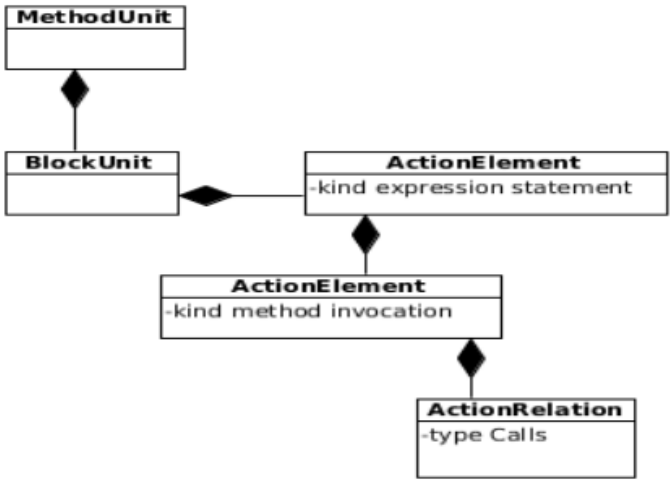

Figura 5.16: Visualización simplificada del modelo KDM de la llamada a una función

Análisis de la estructura «variables»

Las estructuras de tipo «variables», tal y como se puede comprobar en el listado 5.11, permiten delimitar una zona de código donde se realiza la instanciación de variables del propio objeto. Al igual que para los bloques internos a las funciones, y como se ha visto en el ejemplo del punto anterior, utilizaremos elementos *BlockUnit* para mapearlos en el metamodelo. A continuación se muestra en la figura 5.17 un diagrama de cómo debe de ser el resultado del desarrollo de la iteración actual.

```
1 type variables
2 boolean ib_Painting
3 end variables
```

Listado 5.11: Estructura de variables

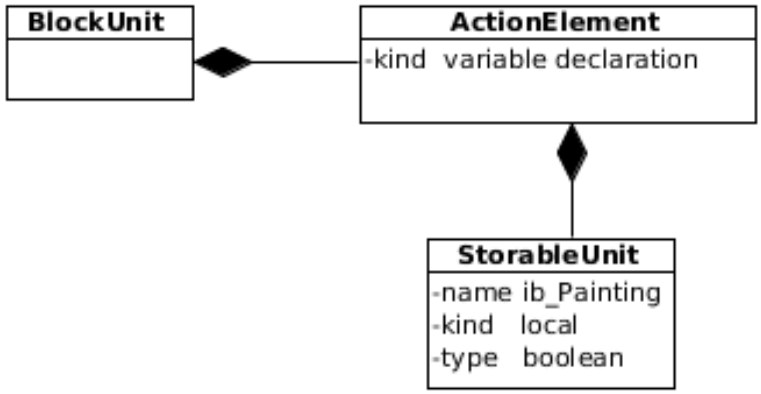

Figura 5.17: Ejemplo de mapeo de variable

Análisis de sentencias SQL

Uno de los puntos distintivos de Powerscript es la inclusión de sentencias SQL dentro del código mediante comandos propios del lenguaje.

```
1  public function string of_read_employee_payroll (string s_dni, integer
       i_month, integer i_year);
2  string ls_tabla

4  SELECT stb.tabname INTO :ls_tabla
5    FROM payroll data
6   WHERE data.dni = :s_dni
7   AND   data.month = :i_month
8   AND   data.year = :i_year
9   USING SQLCA;

11 IF SQLCA.SQLDBCode <> 0 THEN
12    ls_tabla = ""
13 END IF

15 RETURN ls_tabla
16 end function
```

Listado 5.12: Sentencia SQL dentro del código

Una estructura de código como la de la figura **??** es perfectamente viable en Powerscript.

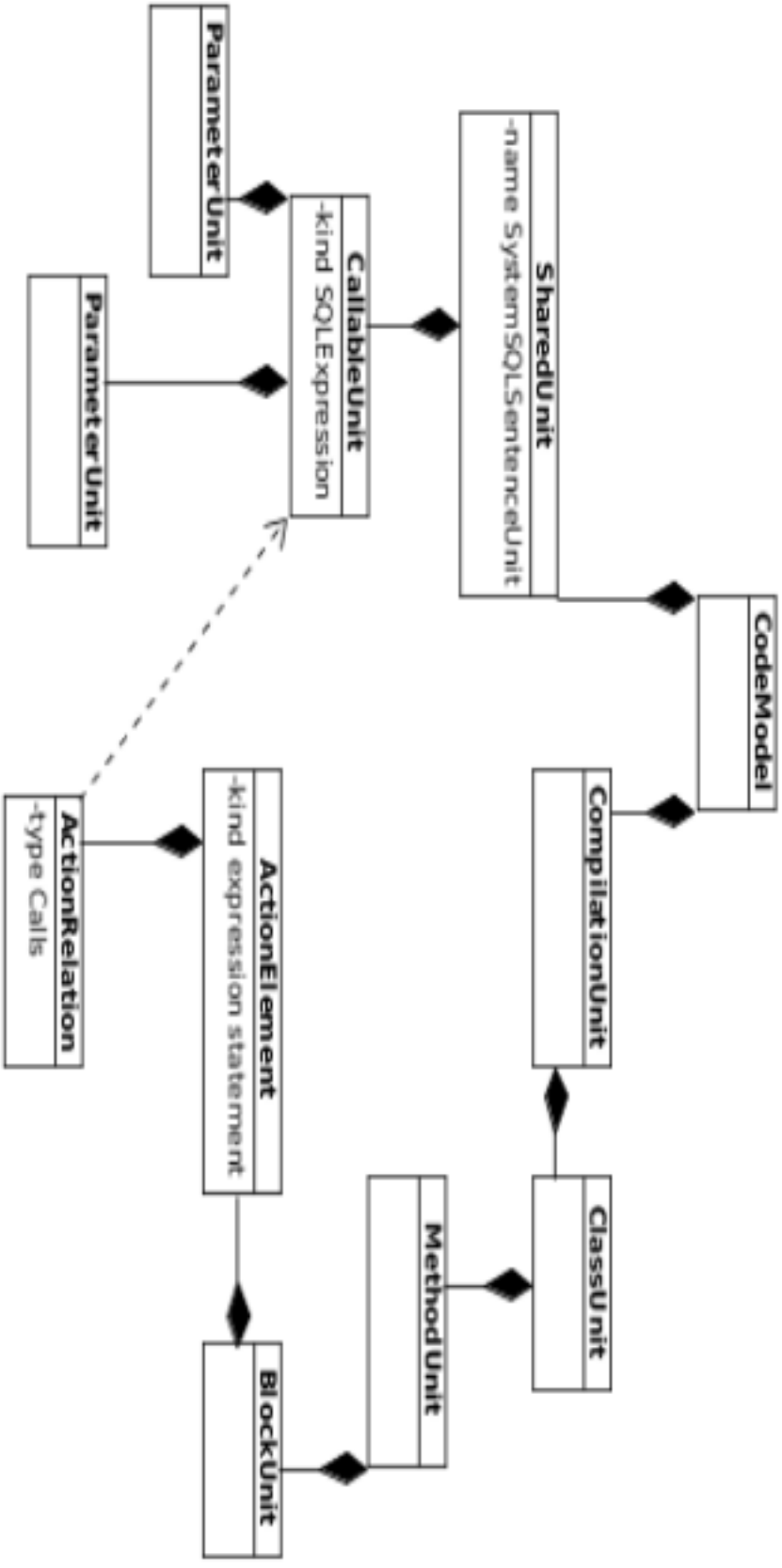

Figura 5.18: Ejemplo de mapeo de sentencia SQL

La inclusión de sentencias SQL repartidas dentro del código de manera discrecional no se encuentra soportada directamente por KDM en el paquete *Model*, sino que se encuentra definida en el paquete *Data*. Para modelar esta parte del lenguaje y que el resultado mantenga la coherencia a la vez que permita recuperar la máxima información posible se asumirán las siguientes premisas:

- La utilización del paquete *Data* únicamente, impide obtener la información completa del sistema al no conocer en qué punto del código heredado se realizan las llamadas y cual es su relación con los demás elementos del punto donde se hace la consulta.
- Dado que las sentencias SQL en Powerscript se localizan dentro de los bloques de sentencias de funciones, por definición del metalenguaje, no pueden ser representadas como un objeto *Callable* dentro de una función.

Como solución al problema que aquí se encuentra, se propone la siguiente estructura de modelado, representada en el esquema 5.18.

1. Todas las sentencias SQL del sistema se modelaran como elementos de un único *SharedUnit* del sistema. Con ello se facilita que un futuro desarrollo realice la comprobación y eliminación de sentencias duplicadas,
2. Se creará un objeto *CallableUnit* del metalenguaje por cada sentencia SQL dentro del *SharedUnit* del sistema.
3. Las resultados y variables externas de las sentencias se modelarán como *ParameterUnit* del *CallableUnit*.
4. En el punto del código donde se encuentre una sentencia SQL se modelará una llamada al objeto *CallableUnit* manteniendo así la coherencia entre código fuente y modelo.

Cada uno de los códigos expuestos en este punto de la iteración serán utilizados como test base del proceso de desarrollo que se realizará a continuación.

5.7.3 Desarrollo, test y validación

En la etapa de desarrollo, test y validación de esta iteración se obtienen los siguientes resultados, basados en los análisis descritos en la fase anterior.

Análisis de la llamadas a métodos

En el código de la figura 5.13 se muestra el resultado obtenido una vez implementado el código que permite mapear las sentencias *Call* de Powerscript.

Se puede apreciar como la llamada realizada a la función *On*, es modelada como una invocación al método en la línea 2 de 5.13. Para mejorar la comprensión y diferenciar a qué tipo de función se ha realizado la llamada, se ha incluido un *AnnotationUnit* que especifica el tipo de método llamado.

```
1  <codeElement xsi:type="action:ActionElement" name="expression statement"
       kind="expression statement">
2    <codeElement xsi:type="action:ActionElement" name="method invocation"
         kind="method invocation">
3    <annotation text="On Method Called"/>
4      <actionRelation xsi:type="action:Calls" to="//@model.1/@codeElement
           .1/@codeElement.0/@codeElement.0" from="//@model.1/@codeElement
           .0/@codeElement.1/@codeElement.2/@codeElement.0/@codeElement.0/
           @codeElement.0"/>
5    </codeElement>
6  </codeElement>
```

Listado 5.13: Modelo resultante de una llamada a función

Por último la acción *Calls*, se establece entre el objeto de la expresión y la referencia al objeto padre del objeto en el que está la llamada.

Análisis de la estructura «variables»

Para modelar la declaración de una variable, ha de tenerse en cuenta que el tipo de la misma ha estar definido dentro del *LanguageUnit* y debe enlazarse con él.

```
1  <codeElement xsi:type="action:ActionElement" name="variable declaration"
       kind="variable declaration">
2    <codeElement xsi:type="code:StorableUnit" name="newValue" type="/0/
         @model.0/@codeElement.1/@codeElement.0" kind="local">
3    </codeElement>
4  </codeElement>
```

Listado 5.14: Modelo de la declaración de una variable

El código de la figura 5.14 muestra cómo es el resultado obtenido y que permite superar el test de esta parte del desarrollo.

Análisis de sentencias SQL

Como se ha definido en el análisis de esta iteración y mas concretamente en la referente a sentencias SQL, el resultado del desarrollo genera un objeto *SharedUnit* (línea 1 del código de la figura 5.15) dentro del cual se modela cada una de las sentencias como *CallableUnit*.

Una vez resueltos todos los casos de test de esta iteracción podemos dar el proceso de desarrollo por finalizado.

```
languagelanguage
<codeElement xsi:type="code:SharedUnit" name="SystemSQLSentenceUnit">
  <codeElement xsi:type="code:CallableUnit" name="SELECTdata.tabnameINTO:
      ls_tablaFROMpersonaldatadataWHEREdata.dni=:s_dniUSINGSQLCA;" kind="
      stored">
    <annotation text="Select"/>
    <annotation text="of_read_employee"/>
    <codeElement xsi:type="code:ParameterUnit" name="ls_tabla" kind="
        return"/>
    <codeElement xsi:type="code:ParameterUnit" name="s_dni" kind="unknown
        "/>
  </codeElement>
  <codeElement xsi:type="code:CallableUnit" name="SELECTdata.tabnameINTO:
      ls_tablaFROMpayrolldataWHEREdata.dni=:s_dniANDdata.month=:
      i_monthANDdata.year=:i_yearUSINGSQLCA;" kind="stored">
    <annotation text="Select"/>
    <annotation text="of_read_employee_payroll"/>
    <codeElement xsi:type="code:ParameterUnit" name="ls_tabla" kind="
        return"/>
    <codeElement xsi:type="code:ParameterUnit" name="s_dni" kind="unknown
        "/>
    <codeElement xsi:type="code:ParameterUnit" name="i_month" kind="
        unknown"/>
    <codeElement xsi:type="code:ParameterUnit" name="i_year" kind="
        unknown"/>
  </codeElement>
</codeElement>
```

Listado 5.15: Sentencia SQL como *CallableUnit*

Capítulo 6

Conclusiones y propuestas

Una vez finalizado el desarrollo del proyecto fin de carrera, en el presente capítulo se recopilan las conclusiones obtenidas del mismo, así como las futuras líneas de trabajo.

6.1 Conclusiones

Al finalizar este proyecto se ha obtenido un sistema capaz de realizar una transformación que genera el metamodelo KDM de un proyecto creado en PowerBuilder. Este modelo permite recuperar la información subyacente que se encuentra en el código fuente y los archivos descriptores de un sistema heredado. Además se ha creado un proceso de validación, que sirve como filtro de errores e incrementa la solidez de la herramienta.

Todo el proceso de desarrollo ha sido realizado aplicando el mayor número de patrones de diseño posible y con la intención de ser, no un producto final sino un sistema base, que permita ampliarse y ser reutilizado por los depositarios finales en el futuro.

A continuación se valorará el grado de conclusión de cada uno de los objetivos parciales establecidos en el apartado 2.1.1, y finalmente como consecuencia de ello el grado de conclusión del objetivo principal.

Objetivo	**Descripción**
1	Análisis de la arquitectura de las aplicaciones PowerBuilder, el lenguaje de desarrollo con el que se escriben y estudio en detalle del código fuente disponible que generan.
Validación	**Justificación**
OK	En todas y cada una de las etapas de análisis de las iteraciones llevadas a cabo, se ha procedido a realizar un estudio pormenorizado de PowerBuilder y su lenguaje de programación Powerscript.

Objetivo	**Descripción**
2	Generación de una gramática que permita reconocer el lenguaje Powerscript mediante el uso del analizador sintáctico ANTLR.
Validación	**Justificación**
OK	Como resultado de la iteración 2, se ha creado la gramática que permite reconocer Powerscript.

Objetivo	Descripción
3	Diseño y Desarrollo de un módulo para la generación de modelos basados en KDM a partir de la información extraída del analizador PowerBuilder.
Validación	**Justificación**
OK	El diseño del módulo para la generación de modelos se ha realizado con éxito

Una vez valorados cada uno de los objetivos parciales, y habiéndose superado todos ellos, puede decirse que **se ha cumplido el objetivo principal del PFC.**

6.2 Líneas futuras de trabajo

Una vez concluido el conjunto de objetivos que han marcado el desarrollo de este proyecto, se plantean las siguientes líneas de trabajo a futuro:

- Ampliación del sistema con el estudio de otros lenguajes de programación que puedan utilizarse como módulos en los proyectos PowerBuilder.
- Estudio de rendimiento y de su posible mejora mediante la utilización de *Streams* de Java 1.8 y procesamiento concurrente.
- Generación mediante el uso de MoDisco de un fichero UML que permita la utilización del modelo con otras herramientas.
- Generar el proyecto como paquete e incluirlo en un servidor de aplicaciones que pueda ser online.
- Crear una aplicación visual, que permita recuperar la información que el depositario final pueda necesitar de una manera accesible.

Capítulo 7

Listado de acrónimos

PUA	Proceso unificado ágil
MVC	Modelo Vista Controlador
SQL	Query Language
PFC	Proyecto fin de carrera
KDM	Knowledge-Discovery Metamodel
OMG	Organization modeling group
XMI	XML Metadata Interchange
UML	Unified Modeling Langauage
PUR	Proceso Unificado Rational
TDD	Test Driven Development
IDE	Integrated Development Enviroment
CASE	Computer Aided Software Engineering

Bibliografía

[Agi] Agile Modeling Best Practices. http://www.agilemodeling.com/essays/bestPractices.htm.

[Amb02] Scott Ambler. *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*. 2002.

[ANT] Repositorio de gramaticas del proyecto ANTLRv4. https://github.com/antlr/grammars-v4.

[Bec03] Kent Beck. *Test-Driven Development By Example*, volume 2. 2003.

[CC90] EJ Chikofsky y JH Cross. Reverse engineering and design recovery: A taxonomy. *Software, IEEE*, 1990. url: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=43044.

[CP07] Gerardo Canfora y Massimiliano Di Penta. New Frontiers of Reverse Engineering. 2007.

[Ecl] Eclipse Foundation. http://www.eclipse.org/.

[Gam] Johnson Vlissides Gamma, Helm. *Patrones de diseño*.

[Gro] OMG Group. *Architecture-Driven Modernization: Knowledge Discovery Metamodel*.

[Inc08] Sybase Inc. *PowerBuilder: Users reference guide*. 2008.

[KDM] Object Management Group - UML. http://uml.org/.

[OMG] The Object Management Group® (OMG®) is an international, open membership, not-for-profit technology standards consortium, founded in 1989. http://www.omg.org/technology/kdm/.

[Par] Terence Parr. *The Definitive ANTLR4 Reference Guide*.

[Sav] Soluciones software de recursos humanos y nómina. http://www.savia.net/.

[UML] Specification of way the world models not only application structure, behavior, and architecture, but also business process and data structure. http://www.uml.org/.

[Wul12] Hasselbring Wulf, Frey. A Three-Phase approach to efficiently transform c into KDM. 2012.

Este documento fue editado y tipografiado con LaTeX
empleando la clase **arco-pfc** que se puede encontrar en:
https://bitbucket.org/arco_group/arco-pfc