PLANTILLA PARA MEMORIA DE PROYECTO FIN DE CARRERA

UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

INGENIERÍA

EN INFORMÁTICA

PROYECTO FIN DE CARRERA

Plantilla para Memoria de Proyecto Fin de Carrera

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

Plantilla para Memoria de Proyecto Fin de Carrera

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

El presente documento es un ejemplo de memoria del Proyecto Fin de Carrera o Trabajo de Fin de Máster según el formato y criterios de la Escuela Superior de Informática de Ciudad Real. La intención es que este texto sirva además como una serie de consejos sobre tipografía, LaTeX, redacción y estructura de la memoria que podrían resultar de ayuda. Por este motivo, se aconseja al lector consultar también el código fuente de este documento.

Este documento utiliza la clase LaTeX *arco-pfc*, disponible como paquete Debian/Ubuntu, consulta:

https://bitbucket.org/arco_group/arco-pfc.

Si encuentra cualquier error o tiene alguna sugerencia, por favor, utilice el *issue tracker* del proyecto *arco-pfc* en:

https://bitbucket.org/arco_group/arco-pfc/issues

El resumen debería estar formado por dos o tres párrafos resaltando lo más destacable del documento. No es una introducción al problema, es decir, debería incluir los logros más importantes del proyecto. Suele ser más sencillo escribirlo cuando la memoria está prácticamente terminada. Debería caber en esta página (es decir, esta cara).

Índice general

Índice de cuadros

Índice de figuras

Índice de listados

Listado de acrónimos

RPC	Remote Procedure Call

Agradecimientos

Escribe aquí algunos chascarrillos simpáticos. Haz buen uso de todos tus recursos literarios porque probablemente será la única página que lean tus amigos y familiares. Debería caber en esta página (esta cara de la hoja).

Juan[1]

[1] Sí, los agradecimientos se firman

A alguien muy querido y/o respetado

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.

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 desaparción del desarrollador.

Cuando se afronta una situación como la descrita anteriormente, es necesario realizar un estudio y valoración pormenorizado de la estratégia 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.

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 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:

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

- Alta acomplación entre las tres capas definidas por el patrón MVC (Modelo Vista Controlador).
- 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 preven 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 ??: ??

Explica herramientas y aspectos básicos de edición con LaTeX.

Capítulo 2: Objetivos

Finalidad y justificación (con todo detalle) del presente documento.

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:

Diseñar y desarrollar un sistema de soporte para la migración de sistemas heredados en explotación desarrollados en PowerBuilder y escritos en PowerScript hacia platformas actuales que están actualmente en explotación, a una tecnología de código abierto actual.

De esta forma se posibilitará el aprovechamiento del conocimiento acumulado en las reglas de negocio que 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 (Knowledge-Discovery Metamodel)[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 projecto, 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 Ingenieria 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 códido 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 apliación de 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 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 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®.

Objectos PowerBuilder

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

Objeto	Uso	Prefijo	Extensión
Aplicación	Punto de entrada a la aplicación		.pbt
Window	Intefaz 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
Projecto/Librerias	Objectos de apoyo a la compilación que indican los objetos que serán compilados conjuntamente		.pbg

Cuadro 3.1: Semánticas de RPC en presencia de distintos fallos (PUDER [PRP05])

Objetos de tipo Aplicación Estos objetos representan el punto de entrada a una aplicación. Se trata de un objeto que lista el conjunto de librerias 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.

Objectos 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. Normamente 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 puden 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 imágen, 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 usuo en cualquier script de la aplicación.

Objectos 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 apliación que lo necesiten. Los objetos definidos por el usuario pueden agruparse en objetos de usuario estándarizados 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 PBL 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 librerias a generar y el nombre del producto final. Por otro lado los ficheros PBL definen las librerias resultantes de la compilación de objetos relacionados, que pueden encontrarse en el mismo directorio o en directorios separados. [Inc08]

3.5 KDM

En Junio de 2003, OMG[OMG] creó un equipo de trabajo para modelar artefactos software en el contexto de sistemas obsoletos. Incialmente 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] En Noviembre de 2003 la *ADMTF*[3] incorporó la solicitud de propuesta para la especificación *Knowledge Disco-*

[1] traducción de «Legacy Transformation Task Force»
[2] traducción de «Architecture-Driven Modernization Task Force»
[3] Siglas en inglés del nombre del equipo de trabajo

very Metamodel(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 interactuen 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 ssepara 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 traves 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,

[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 intefaces. 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 automaticamente 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 traves 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.

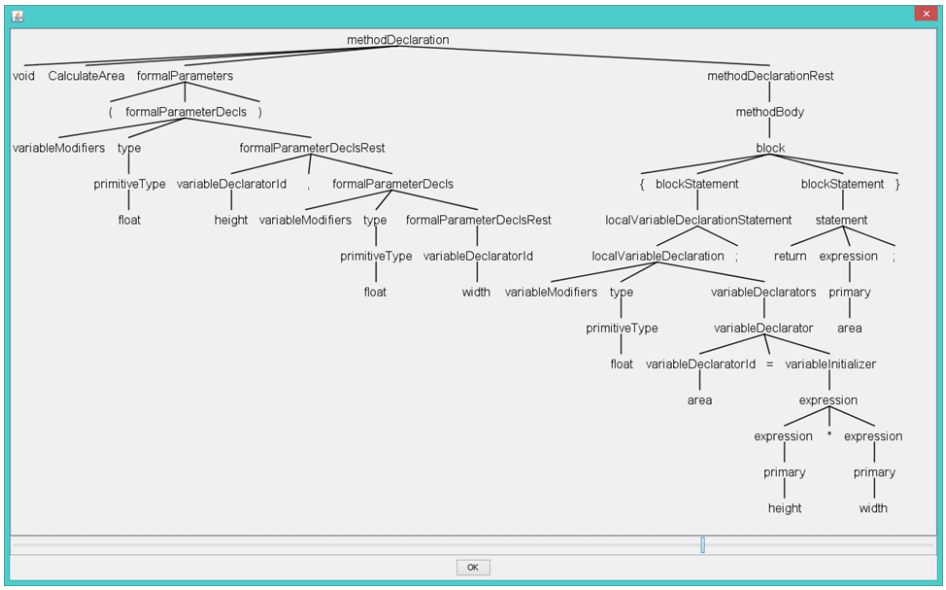

Figura 3.1: Vista de un árbol generado mediante ANTLR4

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 gramaticas 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 ambiguedad 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.

```
expr : expr '*' expr
 | expr '+' expr
 | INT
 ;
```

Listado 3.1: ALL(*)

- El mayor cambio de la nueva versión v4 es que se desenfatiza 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 Java 1.8 y Lambdas

3.8 MoDisco

MoDisco es un projecto «open source» que forma parte de manera oficial de la *Eclipse Foundation*(EF)[Ecl] y está integrado en el projecto base de modelado de dicha fundación, promocionando las técnicas de *Ingeniería dirigida a modelos*(MDE) [5] dentro de la comuni-

[5] Traducción de *Model Driven Engineering*

dad de la comunidad de Eclipse. Además está reconocida por la OMG como proveedor de referencia para la implementación de diversos estándares como:

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

MoDisco provee de una serie de componentes que permiten eleborar soluciones de ingeniería inversa para la transformación, con idependencia del lenguaje en el que esté desarrollado, de sistemas obsoletos utilizando metamodelos. 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 funcion 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 traves de él, únicamente con su registro en el *Discoverer Manager*.

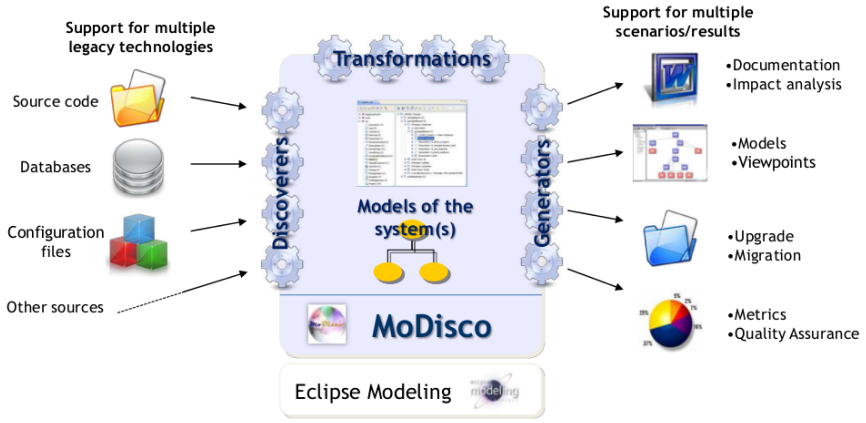

Figura 3.2: Estructura MoDisco

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).

3.9 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:

- **Tranformación 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.

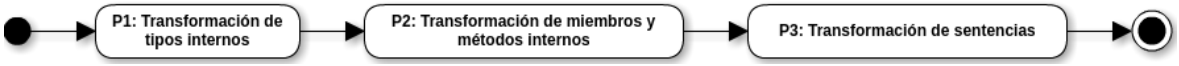

Figura 3.3: Esquema de los pasos llevados a cabo en la transformación

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 de Rational [Amb02] que desarrolló Scott Ambler, conocida como Proceso Unificado Ágil[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, *"(TDD)"* [2] [Bec03].

4.1 Proceso Unificado Ágil

Por tanto en el desarrollo del software se mantendrán los principios base del Proceso Unificado de Software:

- 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
[2] Test Driven Development

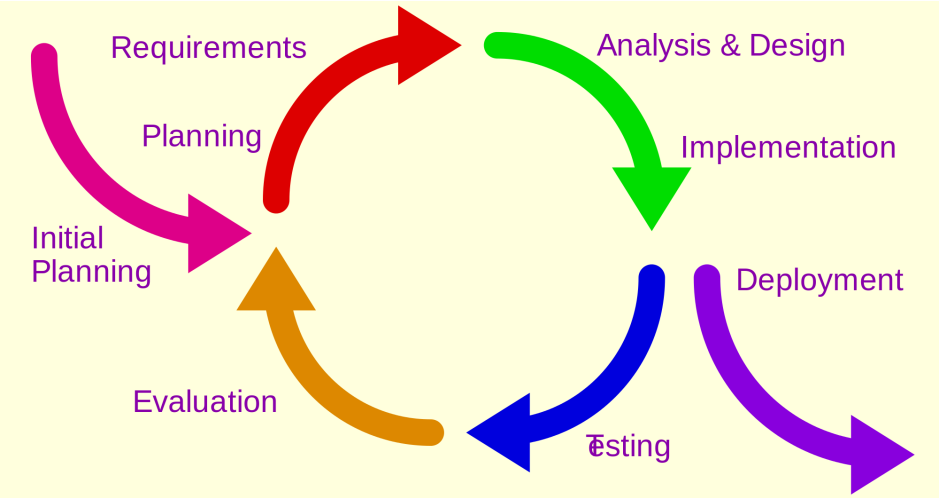

Figura 4.1: Ciclo de vida PUA

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"[3] 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 iteracciones 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

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

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

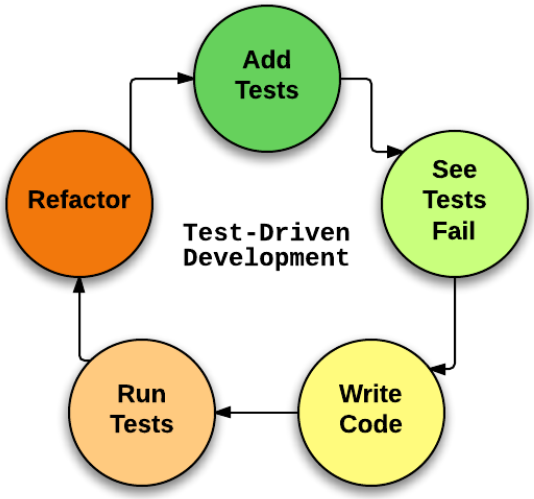

Figura 4.2: Flujo de las iteracciones del desarrollo basadas en TDD

Los procesos 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"* [4].
- **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.

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

4.3 Evolución del Proyecto

Como consecuencia de la metodología seleccionada para la construcción del proyecto se mostrará a continuación el listado de iteracciones 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 librerias 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, ordenandolos 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 establecios han sido superados por el resultado del proceso.

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 generar un sistema de comprobación que el proyecto que se analizará está correctamente establecido. 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 § 3.9), 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: Tranformación de tipos internos

En la siguiente iteración, se

4.3.6 Iteración 5: Transformación de miembros internos y métodos

4.3.7 Iteración 6: Transformación de sentencias

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

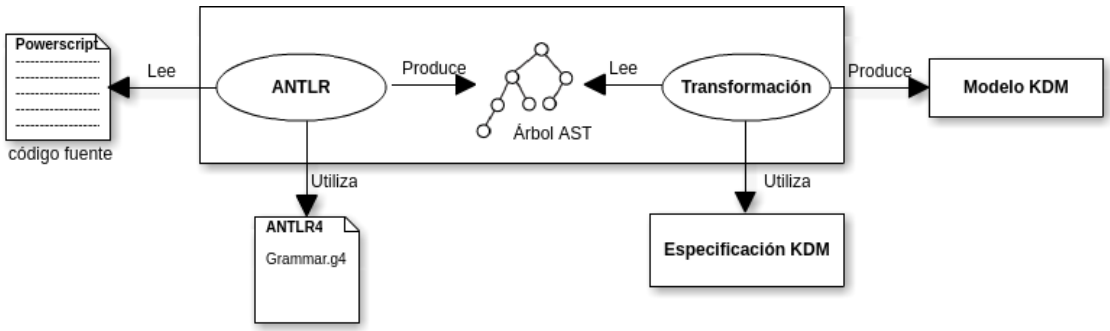

Figura 5.1: Base arquitectural del sistema

5.2 Iteracción 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*.

5.2.1 Planificación y requerimientos

Para abordar esta parte de la iteración, inicialmente se realiza una busqueda 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 y tras descartar las que se puediera encontrar una propia del lenguaje de programación a estudio en el repositorio github del proyecto[] se pasa a realizar un análisis pormenorizado de una colección de ficheros de código extraidos del sofware heredado. 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 iteracción permitan alcanzar el objetivo propuesto.

Estructuras comunes

Se analizarán ahora todas aquellas estructuras que se puedan relacionar con las mas comunes de otros lenguajes de programación conocidos.

Palabras reservadas del lenguaje, tipos de datos y comentarios Tanto las palabras reservadas como los tipos de datos, pueden encontrarse facilmente 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 iteracción anterior.

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

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.

```
type variables
boolean ib_Painting
integer width = 251, height = 72
fontcharset fontcharset = ansi
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\constants

Bucles Dentro de Powerscript encontramos los siguientes bucles de flujo:

- IF-THEN-ELSEIF-THEN
- FOR
- DO-WHILE
- TRY-CATCH
- CHOOSE

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 de una base de datos un proceso muy tedioso.

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

Funciones

Expresiones

Estructuras propias

Estructura «forward»

Estructura «type»

Estructura «variables»

Estructura «framework»

Estructuras «event» y «on»

Estructuras «prototipos de funciones»

Comandos propios

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:

Como muestra de ello se adjunta el listado5.2 resultante del primer bloque de testing que incluye las «Estructuras comunes»5.2.2

5.3 Iteracción 2: Sistema de validación del software heredado

5.4 Iteracción 3: Generación del InventoryModel

5.5 Iteracción 4: Tranformación de tipos internos

5.6 Iteracción 5: Transformación de miembros internos y métodos

5.7 Iteracción 6: Transformación de sentencias

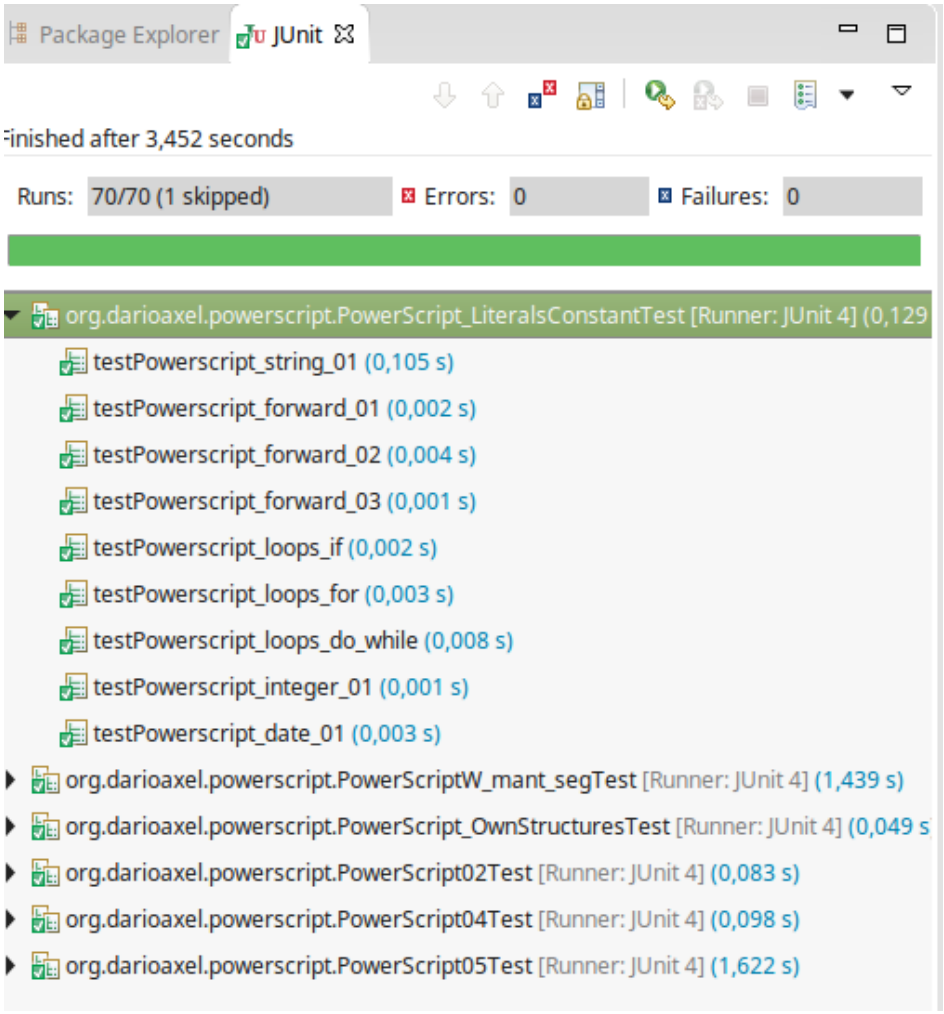

Figura 5.2: Resultados del testing de estructuras comunes

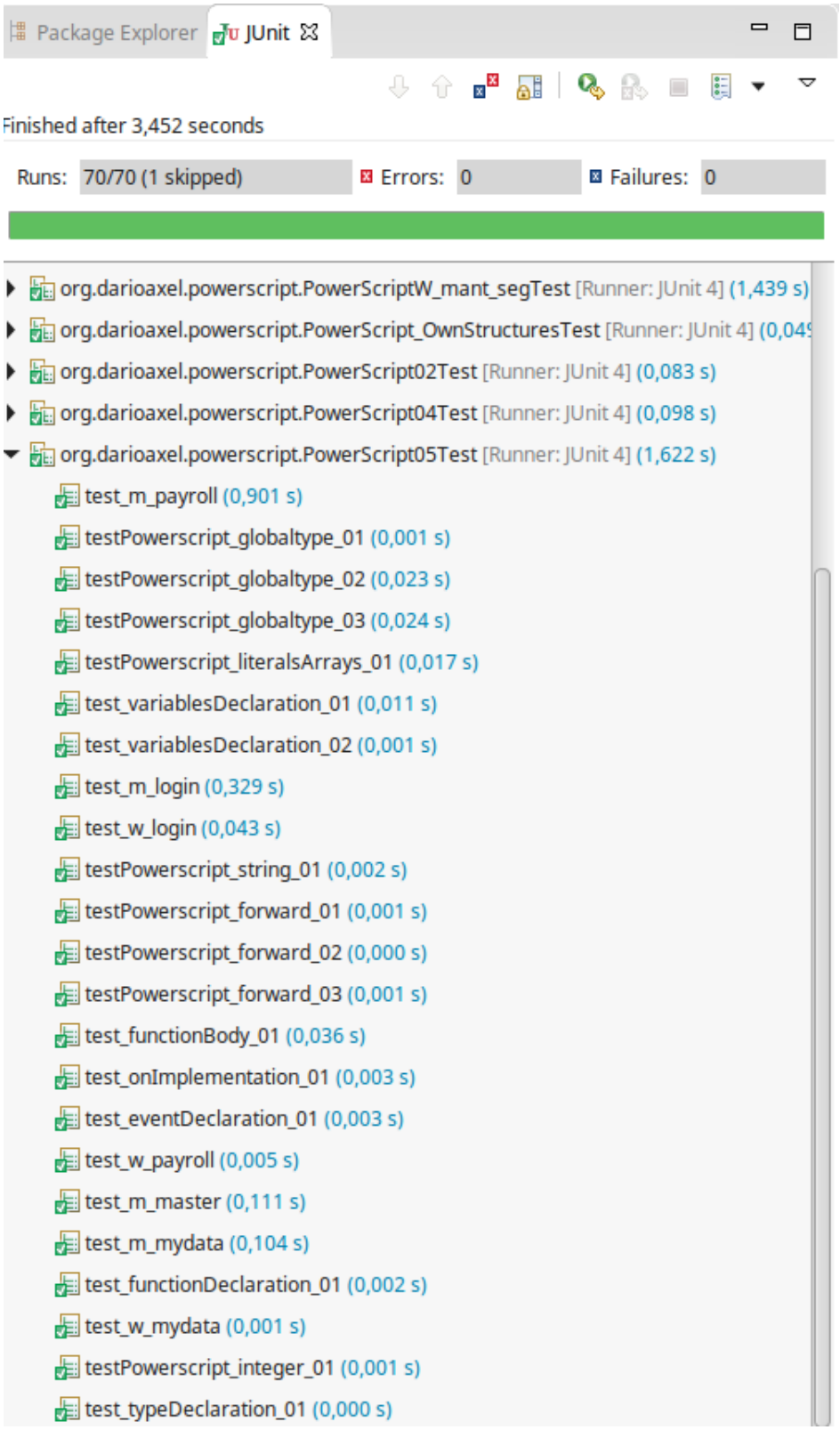

Figura 5.3: Resultados del testing de la gramática completa

Capítulo 6

Conclusiones y propuestas

6.1 Conclusiones

ANEXOS

Anexo A

Gramática resultante de la iteración 1

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.

[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/.

[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*.

[PRP05] A. Puder, K. Römer, y F. Pilhofer. *Distributed Systems Architecture: A Middleware Approach*. Morgan Kaufman, 2005.

[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