Universidad
Carlos III de Madrid

TRABAJO DE FIN DE GRADO

DEPARTAMENTO DE INFORMÁTICA

INGENIERÍA DIRIGIDA A MODELOS. SISTEMAS DE TRANSFORMACIÓN MODELO A TEXTO. COMPLEMENTO DE REFACTORIZACIÓN DE CÓDIGO C++ A C++11 PARA ELCIPSE.

Autor: Alejandro Tovar Moreno

Tutor: Luis Miguel Sánchez García

Colmenarejo, a 25 de Junio de 2013

Agradecimientos:

En primer lugar quiero agradecerle a mis padres, a mi hermana y a mi hermano todo el esfuerzo que han hecho para que yo pudiese llegar aquí y sobre todo estos últimos años para que no me faltase de nada y pudiese terminar esta carrera. Quiero agradecerles todo el apoyo que me han brindado cuando lo he necesitado y la fuerza que me han dado para seguir adelante cuando pensaba que me estaba estancando.

Quiero agradecer a amigos el haber estado ahí en todo momento, y confiar en mí y en que podría llegar hasta aquí y terminar esta carrera. El apoyo recibido y los buenos momentos que me han hecho pasar cuando necesitaba relajarme y despejarme mientras desarrollaba este proyecto.

También quiero recordar y agradecer a mi abuela el apoyo que me brindó y la capacidad de sacarme una sonrisa cuando las cosas flojeaban. Siempre confió en que terminaría esta carrera y ha sido la mejor motivación para conseguirlo.

Por otro lado, quiero agradecer a mi tutor Luis Miguel Sánchez García el esfuerzo, el tiempo y la paciencia invertida no solo en este proyecto de fin de grado sino a lo largo de la carrera en la que he tenido el gusto de tenerlo como profesor en varias ocasiones y al que le debo muchos de los conocimientos adquiridos durante estos cuatro años.

Quiero agradecer a mis compañeros de trabajo, la motivación y confianza que siempre me han transmitido para incitarme a terminar este trabajo y poder así terminar mis estudios. A veces resulta complicado cuando estás trabajando, pero ellos me han ayudado a ver lo importante que es terminarlo todo y afrontar el futuro.

Por último quiero agradecer a la Universidad y profesores en general los servicios y la ayuda prestada a lo largo de estos cuatro años y al SOPP de la universidad por las facilidades prestadas y el esfuerzo invertido durante mi periodo de prácticas en empresa, gracias a las cuales puedo contar a día de hoy con un trabajo estable en el sector informático y estar orgulloso de ello.

Resumen

En este Trabajo de Fin de Grado presentaremos los sistemas dirigidos por modelos (MDE) y las arquitecturas dirigidas por modelos (MDA). Trataremos de aplicar un sistema de transformación de modelo a texto dentro de una arquitectura MDA como una nueva alternativa a la generación de analizadores y conversores entre lenguajes de programación. El objetivo será conseguir un refactorizador de código que permita modelar una clase C++ y aplicar transformaciones para obtener código C++11.

Para ello, tras analizar y utilizar diferentes mecanismos y sistemas que podrían realizar estas transformaciones, nos decidiremos por utilizar un sistema basado en plantillas de transformación y el lenguaje de programación JAVA.

La situación ideal, consistiría en conseguir una arquitectura que nos permita realizar las tres actividades principales en estos sistemas de transformación modelo a texto de forma independiente, para que de esta manera, cualquier cambio que pretenda realizarse afecte a la menor parte de la arquitectura posible y sea fácil de modificar.

En nuestro caso en particular, presentaremos una arquitectura basada en esta separación de funcionalidades para conseguir de esta manera realizar algunas transformaciones entre dos lenguajes de programación C++03 y C++11.

Por un lado, procederemos a la lectura y modelado de ficheros C++ recurriendo a complementos de eclipse que nos permitan realizar esta función.

Una vez, consigamos extraer el modelo y seleccionemos las transformaciones que pretendemos realizar desde el lenguaje inicial C++03 a nuestro lenguaje final C++11, utilizaremos el complemento de eclipse XTEND2 que nos permitirá realizar conversiones utilizando un lenguaje de programación muy sencillo y similar a JAVA.

Para terminar, cuando todo el contenido del fichero C++ se halla analizado y se hayan realizado las transformaciones de lenguaje oportunas, la parte de la aplicación encargada de la generación de ficheros, se encargará de crear un nuevo fichero C++ con el contenido transformado a C++11.

En resumen, lo que se propone con este Trabajo de Fin de Grado es presentar una arquitectura que nos permita realizar transformaciones entre los distintos lenguajes de programación que conviven hoy en día; y como muestra de un caso aplicado basado en una arquitectura MDA, un sistema de refactorización y rejuvenecimiento de código C++03 a C++11.

ÍNDICE DE CONTENIDOS

ÍNDICE DE ILUSTRACIONES

1. INTRODUCCIÓN:

Este primer punto del documento nos muestra una perspectiva generalizada del trabajo de fin de grado a realizar. Explicaremos cuales son las motivaciones y los intereses que nos mueven al análisis de los sistemas de transformación “model to text” dentro de las arquitecturas MDA y los objetivos que pretendemos lograr con la realización de este proyecto.

Este apartado del documento, contendrá también un listado de acrónimos que se utilizarán en el documento y un análisis de la estructura del documento; cuyos objetivos son facilitar tanto la escritura como la lectura del documento.

1.1 Motivación

La principal motivación por la que se decide realizar esta aplicación es utilizar nuevas técnicas de modelado, refactorización y rejuvenecimiento de código que faciliten a los desarrolladores la adaptación al constante cambio al que se enfrente el sector informático.

Nuestro proyecto en concreto, pretende aplicar nuevas técnicas de refactorización y rejuvenecimiento de código que permitan adaptar código C++03 a C++11. Se pretende ofrecer una arquitectura basada en modelos que permita aplicar el mismo mecanismo de refactorización para cualquier otro proceso de refactorización entre lenguajes de programación.

El problema es que adaptarse al cambio, no es algo factible, ya que no es sencillo estar al tanto de todas las nuevas tecnologías aplicables al sector informático, aprenderlas y aplicarlas; sin olvidar y actualizar todos los conocimientos que ya poseas.

Debido a esto, se ha tratado de buscar una manera que nos permita buscar una solución alternativa a la necesidad del conocimiento de uno o varios lenguajes de programación concretos en un caso determinado.

En la actualidad conviven numerosos lenguajes de programación en el sector informático, con un porcentaje de uso muy variado.

Posición Feb 2013	Posición Feb 2012	Tendencia	Lenguaje de programación	Porcentaje Feb 2013	Diferencia Feb 2012	Estado
1	1	=	Java	18.387%	+1.34%	A
2	2	=	C	17.080%	+0.56%	A
3	5	↑↑	Objective-C	9.803%	+2.74%	A
4	4	=	C++	8.758%	+0.91%	A
5	3	↓↓	C#	6.680%	-1.97%	A
6	6	=	PHP	5.074%	-0.57%	A
7	8	↑	Python	4.949%	+1.80%	A
8	7	↓	(Visual) Basic	4.648%	+0.33%	A
9	9	=	Perl	2.252%	-0.68%	A
10	12	↑↑	Ruby	1.752%	+0.19%	A
11	10	↓	JavaScript	1.423%	-1.04%	A
12	16	↑↑↑↑	Visual Basic .NET	1.007%	+0.21%	A
13	13	=	Lisp	0.943%	+0.04%	A
14	15	↑	Pascal	0.932%	+0.12%	A
15	11	↓↓↓↓	Delphi/Object Pascal	0.886%	-1.08%	A
16	14	↓↓	Transact-SQL	0.773%	-0.07%	A-
17	75	↑↑↑↑↑↑↑↑↑↑	Bash	0.741%	+0.61%	A-
18	26	↑↑↑↑↑↑↑↑	MATLAB	0.648%	+0.15%	B
19	24	↑↑↑↑↑	Assembly	0.640%	+0.12%	B
20	19	↓	Ada	0.631%	0.00%	B

Tabla 1. Comparación de uso entre lenguajes de programación año 2013.

Por lo general, un desarrollador conocerá algunos de los lenguajes de programación anteriormente expuestos y será capaz de programar con ellos con mayor o menor agilidad.

Sin embargo, no tantos serán capaces de conocer todos estos lenguajes de programación y no serán capaces de desenvolverse con facilidad en ciertas ocasiones.

Por esto, planteamos los sistemas de transformación “model to text” dentro de arquitecturas dirigidas por modelos como alternativa al conocimiento de todos y cada

uno de los lenguajes de programación. Si nos basamos en estos sistemas y somos capaces de obtener un modelo del contenido de los ficheros programados en un determinado lenguaje de programación, podríamos analizarlos y crear transformaciones de código a otros lenguajes de programación, facilitando así el uso del resto de lenguajes a los usuarios que los desconozcan. De esta manera, un desarrollador java podría crear un mecanismo de transformación a otros lenguajes como pudiesen ser C, C++ o JavaScript.

El gran inconveniente que presentan los sistema de transformación "model to text", es que pueden llegar a ser algo muy difícil de implementar, ya que si pretendemos transformar por completo un fichero de un lenguaje inicial a otro lenguaje, debemos tener en cuenta que no todos los lenguajes de programación son similares y pueden requerir transformaciones muy complejas e incluso que requieran modificar el código inicial del usuario, con lo que puede que el código inicial no sea idéntico al código transformado. Además, el tiempo a invertir en las plantillas de transformación puede incrementarse de forma exponencial al nivel de precisión que busquemos en las transformaciones.

Por esto, lo que pretendemos a continuación es analizar estos sistemas de refactorización y llevarlos a la práctica aplicando una serie de transformaciones sobre un fichero con código inicial C++03 para obtener como resultado código C++11.

1.2 Objetivos

El objetivo principal del proyecto es crear una arquitectura basada en modelos (MDA) utilizando sistemas de transformación "model to text" y aplicar técnicas de refactorización y rejuvenecimiento de código que permitan adaptar programas escritos en C++03 a C++11. La consecución de este objetivo requiere otros objetivos secundarios previos que nos permitan conocer las arquitecturas MDA y los conceptos con los que se van a trabajar. Podemos desglosar los objetivos secundarios del proyecto como:

- Análisis de la ingeniería dirigida por modelos (MDE) y de las arquitecturas dirigidas por modelos (MDA).

- Estudio de las diferentes herramientas de transformación de modelo a modelo (M2M) y de modelo a texto (M2T). Selección del sistema de transformación de modelo a texto que más beneficie al refactorizador de código.

1.3 Estructura del documento

Este punto del documento detallará y definirá brevemente el contenido de cada uno de los apartados principales de la memoria. En este documento podremos encontrar el siguiente contenido:

- **Capítulo 1: Introducción.** Este apartado del documento indica las motivaciones por las que se ha decidido realizar este proyecto de fin de grado. Se marcan los objetivos que se pretenden conseguir con este proyecto y se definen la estructura del documento y un listado de acrónimos y definiciones que se utilizarán en el documento.

- **Capítulo 2: Estado de la cuestión.** En este apartado del documento se exponen los conceptos teóricos del proyecto. En primer lugar se definen los conceptos básicos que se necesita manejar para entender y utilizar el proyecto. Posteriormente se realiza un análisis sobre los sistemas dirigidos por modelos y sobre las arquitecturas dirigidas por modelos. Se analizan las transformaciones de modelo a modelo y de modelo a texto que se utilizan en estas arquitecturas. Se analizarán diferentes herramientas utilizadas para estos fines.

- **Capítulo 3: Proceso de desarrollo.** Se expone los objetivos que se pretenden conseguir con la creación de un complemento de eclipse que refactorice código C++03 a C++11 utilizando una arquitectura dirigida por modelos. Se realiza una planificación del proceso de desarrollo y se decide el diseño de la arquitectura final a utilizar en el refactorizador. Para terminar se definen y explican las acciones de refactorización que ofrecerá la herramienta.

- **Capítulo 4: Análisis y diseño.** Definición del marco regulador con las normativas aplicables al proyecto y al desarrollador. Listado de los requisitos de usuario del refactorizador de C++03 a C++11 y diagrama de casos de uso de la aplicación.

- **Capítulo 5: Conclusiones.** Propuesta de futuras mejoras para el complemento de eclipse creado, desglose presupuestario resultante del proceso de desarrollo del proyecto y exposición de conclusiones personales.

- **Anexo I: Manual de usuario.** Se añade un manual de usuario con las diferentes acciones que permite realizar el sistema de refactorización de código. Se indica al usuario como realizar cada una de ellas y se muestra gráficamente el resultado esperado en el entorno de trabajo para cada una de las acciones ofrecidas.

- **Anexo II: Manual de modificación de plantillas.** Se ofrece un manual de usuario para modificar o añadir nuevas plantillas de transformación al sistema de refactorización.

- **Bibliografía.** Referencias bibliográficas consultadas durante el desarrollo del complemento de refactorización y como documentación para la memoria del proyecto.

1.4 Definiciones y acrónimos

A continuación, mostraremos un listado de acrónimos y definiciones que utilizaremos a lo largo del documento. La finalidad de esta tabla de acrónimos es poder referirnos a algunos elementos que utilizaremos con frecuencia en este documento de forma rápida y de tal manera que el usuario pueda saber en todo momento a que nos estamos refiriendo.

El listado de acrónimos para este documento es el siguiente:

Acrónimo	Definición
C	Lenguaje de programación fuertemente tipificado de medio nivel.
C++	Lenguaje de programación que hace referencia a una extensión de C.
JAVA	Lenguaje de programación de alto nivel.
IDE	Integrated Development Environment.
CDT	C/C++ Development Tooling. Es un complemento de eclipse.
EMF	Eclipse Modeling Framework. Es un complemento de eclipse.
JDT	Java Development Tooling. Es un complemento de eclipse.
UML	Unified Modeling Languaje.
MODISCO	Complemento de eclipse que modela java con ficheros basados en xml.
XML	Extensive Markup Languaje. Lenguaje de marcas desarrollado por W3C.
W3C	World Wide Web Consortium. Consorcio internacional de recomendaciones para la world wide web (www).
HTML	HyperText Markup Languaje. Lenguaje de marcado predominante en el desarrollo de páginas web.
CSS	Cascading Style Sheets. Lenguaje de hojas de estilo utilizado para la maquetación de documentos basados en lenguajes de marcas, principalmente HTML, pero puede usarse también para el diseño de documentos XML.

JS	JavaScript. Lenguaje de programación interpretado, orientado a objetos, que se ejecuta en el lado del cliente.
MDA	"Model Driven Architecture". Arquitecturas dirigidas a modelo.
MDE	"Model Driven Engineering". Ingeniería orientada a modelos.
M2M	"Model to Model". Referencia a sistemas de transformación modelo a modelo en un proceso MDE o una arquitectura MDA.
M2T	Referencia a sistema de transformación modelo a texto (Model to Text).
MOF	"MetaObject Facility" Sistema de transformación de modelo a texto.
XTEND2	Sistema de transformación de modelo a texto basado en plantillas, con un lenguaje de programación similar a JAVA.
ATL	"ATLAS Transformation Language". Lenguaje de reglas aplicado a modelos para realizar transformaciones de modelo a modelo M2M.

2. EL ESTADO DE LA CUESTIÓN

2.1. Conceptos básicos.

En este apartado de documento procederemos a la descripción de diversos conceptos básico que deben tenerse en cuenta para la compresión y utilización del sistema refactorizador que se proporciona: entorno de programación, lenguajes de programación, complementos de eclipse (eclipse "plug-in"), etc.

2.1.1. IDE ECLIPSE:

Eclipse es un entorno de desarrollo integrado (IDE) de código abierto y multiplataforma para desarrollo de aplicaciones de cliente enriquecido.

El proyecto Eclipse fue creado originalmente por IBM en noviembre de 2001 con el apoyo de un consorcio de proveedores de software y en enero de 2004 se crea "The Eclipse Foundation" como una corporación independiente sin ánimo de lucro. En la actualidad esta corporación la forman individuos y organizaciones de la industria software y se financia con las cuotas anuales de sus miembros. [25]

Eclipse ha evolucionado mucho con el paso de los años y se ajusta mucho a las necesidades de los usuarios, ya que al tratase de un proyecto basado en software libre son los propios usuarios los que añaden nuevas funcionales creando nuevos complementos para la plataforma. Precisamente, este es uno de los puntos fuertes de la plataforma; ya que permite la creación de nuevos complementos que doten a la herramienta de nuevas funcionalidades, evolucionando así eclipse y ofreciendo constantemente nuevas prestaciones a los usuarios de este entorno de desarrollo.

Las principales diferencias entre Eclipse y otros IDEs de desarrollo, es que se trata de una plataforma bastante estable y que permite desarrollar de forma independiente para distintos lenguajes de desarrollo. Actualmente posee complementos de desarrollo para JAVA, C, C++, desarrollo web (PHP, HTML, CSS), Javascript y Python, entre otros muchos complementos.

Estas diferencias le permiten distanciarse del resto de IDEs como podemos ver en la siguiente imagen:

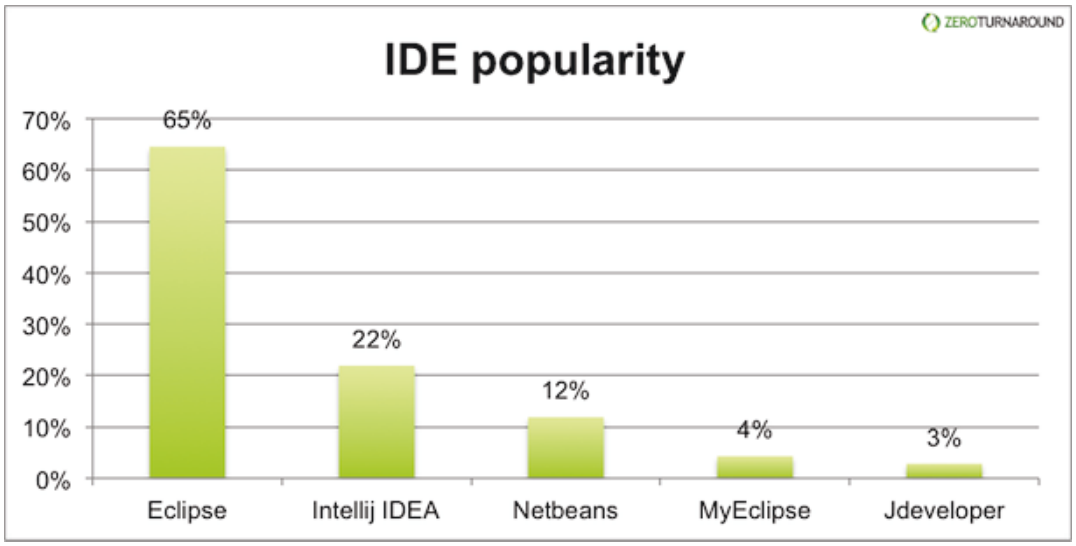


Tabla 2. Comparación de popularidad de IDEs de desarrollo año 2013.

2.1.2. LENGUAJE C/C++/C++11:

Lenguaje C:

Entre los años 1970 y 1972 Brian Kernighan y Dennos Ritchie crean el lenguaje de programación C en los Laboratorios Bell, como evolución del lenguaje B creado por Ken Thomson. Frente a su predecesor, C se presenta como un lenguaje altamente tipificado que permite desarrollar tanto en alto como bajo nivel sin tener que cambiar de lenguaje de desarrollo. [28]

Las principales características que presenta C son: [26]

- El núcleo está escrito en un lenguaje simple e incluye además manejo de archivos y funciones matemáticas mediante el uso de bibliotecas.

- Se trata de un lenguaje flexible que nos permite programar con diferentes estilos. Nos permitirá programar con un estilo estructurado o de forma no estructurada si el desarrollador lo prefiere.

- Tiene un sistema de tipos que no permite operaciones sin sentido.

- Nos permite la utilización de punteros como mecanismo de acceso a memoria de bajo nivel.

- Permite realizar interrupciones al procesador con uniones.

- Utiliza un conjunto reducido de palabras clave.

- Por defecto, el paso de parámetros a las funciones se realiza por valor y no por referencia.

- Permite encapsulado y polimorfismo con punteros a funciones y variables estáticas.

- Permite el uso de tipo de datos agregados que permiten la manipulación en conjunto de una serie de datos que guardan alguna relación. Un ejemplo de este tipo de dato sería una estructura profesor que nos permite manipular de forma conjunta sus datos relacionados “nombre de profesor”, “especialidad”, “aulas”, “asignaturas” y “alumnos”.

La descripción de C y todos sus contenidos se publican en 1978 de la mano de Brian Kernighan y Dennos Ritchie, en el libro *“The C programming Language”.*

En los años 80 se popularizó el lenguaje C desplazando al BASIC. Al mismo tiempo, Bjarne Stroustrup comenzó con el desarrollo del lenguaje C++ que complementaba al lenguaje C con clases, expresiones ADA y tipos genéricos; y añadía además un motor de objetos que ofrecía la posibilidad de combinar la programación de C con la programación orientada a objetos.

En 1983, el Instituto Nacional de Estándares (ANSI) organiza un comité para establecer una especificación estándar del lenguaje C, terminando de completarse esta estandarización en 1989.

El lenguaje de programación C durante todo este tiempo ha sido utilizado sobre todo en la programación de sistemas UNIX. También se ha utilizado este lenguaje en otros campos informáticos como el desarrollo de sistemas operativos, como LINUX/GNU y Windows; desarrollo de aplicaciones de escritorio, como GIMP; sistemas empotrados en ascensores, sistemas de monitorización… y como base de kits de desarrollo de micro controladores.

Se trata de un lenguaje muy interesante y conocido en el sector de la ingeniería y que sigue enseñando como lenguaje de programación base en muchos cursos y universidades.

Lenguaje C++:

El lenguaje C++ comenzó a desarrollarse en 1980 por Bjarne Stroustrup. La intención de Stroustrup en los comienzos del desarrollo de C++ era extender el exitoso lenguaje C de Kernighan y Ritchie con nuevas clases y con mecanismos que permitían la manipulación de objetos. En sus comienzos era conocido como “C with Classes”. El

nombre C++ hace referencia al carácter incremento (++), indicando su funcionalidad inicial, extender el lenguaje C.

En 1989 se formó un comité ANSI con el fin de estandarizarlo viendo la gran difusión y el gran éxito que había alcanzado entre los programadores del momento.

Se trata de un lenguaje versátil, potente y general, que gracias al éxito obtenido entre los desarrolladores ocupa los primeros puestos entre las herramientas elegidas para el desarrollo.

C++ mantiene las ventajas que ofrecía C respecto a la flexibilidad, eficiencia y riqueza de expresiones a las que añade nuevas clases, tipos genéricos y un motor que permite manipular la orientación a objetos.

Desde el punto de vista de los lenguajes de programación, se considera un lenguaje híbrido, ya que se trata a la vez de un lenguaje procedural (orientado a algoritmos) y un lenguaje orientado a objetos. Respecto al lenguaje procedural, es semejante y compatible con C; y como lenguaje orientado a objetos se basa en una filosofía diferente que requiere un cambio de mentalidad por parte del programador. [22]

Lenguaje C++11:

C++11 no es más que la cuarta revisión del lenguaje C++. Esta revisión ha sido muy lenta y durante muchos años ha sido conocido como C++0x, cuyo fin era sustituir la "x" por el año de la cuarta revisión del lenguaje (se esperaba para 2008/2009).

Finalmente, la revisión fue aprobada el 12 de Agosto del 2011 y de ahí que se conozca como C++11.

Con respecto a las antiguas revisiones del lenguaje, C++11 se presenta como un lenguaje más genérico y uniforme, que ofrece mejoras en los mecanismos de abstracción.

C++11 incluye varias adiciones al núcleo del lenguaje y extiende la biblioteca estándar de C++, incorporando gran parte de las bibliotecas de C++ Technical Report 1.

Respecto a la adiciones al núcleo del lenguaje, las áreas del núcleo que se mejoran significativamente es la inclusión de soporte multi-hilo, soporte de programación genérica, inicialización uniforme y mejoras de rendimiento.

2.1.3. LENGUAJE JAVA:

Java tiene su origen en enero de 1991 en Aspen (Colorado), dónde el grupo formado por Bill Joy, Andy Bechtolsheim, Wayne Rosing, Mike Sheridan, James Gosling, y Patrick Naughton, se reúnen para discutir sobre el futuro rumbo de la computación, sobre lo que les gusta y no les gusta de las tecnologías del momento y concluyen en la necesidad de desarrollar un entorno único que pudiera ser utilizado por todos los dispositivos de electrónica de consumo, ya que prevén este entorno como una de las principales tendencias de acercamiento del futuro. [19]

Con el objetivo marcado, comienzan a trabajar el 1 de Febrero de 1991. En un comienzo, C++ es el lenguaje referencia. Gosling y Joy, tratan de extenderlo sin éxito y deciden abandonar esta idea y crear un nuevo lenguaje desde cero al que se llama OAK (Roble).

Oak debe ser un lenguaje robusto a la vez que sencillo para evitar grandes problemas causados por los programadores y debe ser un lenguaje interpretado, ya que debe ser independiente de la plataforma debido a la necesidad de compatibilidad con el gran número de modelos del mercado.

Como resultado, un lenguaje similar a C++, no ligado a un tipo concreto de CPU, con el cambio de nombre a JAVA (**J**ames Gosling, **A**rthur **V**an Hoff, **A**ndy Bechtolsheim) ya que existía un lenguaje de programación llamado Oak.

En Agosto del 91, ya corrían los primeros programas en lenguaje Oak y en 1994, Joy, comienza el proyecto “Live Oak”, que consiste en el desarrollo de un pequeño sistema operativo usando este lenguaje de programación y analizar las posibilidades del negocio de Internet. En este mismo Año, Van Hoff, implementa un compilador de Oak en lenguaje Oak (Sustituyendo el compilador de Gosling, escrito en C); Naughton y Payne, escriben “WebRunner”, un navegador web escrito en Java, que termina por llamarse “HotJava”.

Finalmente, en Mayo de 1995, en la conferencia “SunWorld’95”, Sun Microsystems y Netscape, presenta oficialmente la primera versión de Java que funcionaba en Solaris y que se incorporaría en el navegador más utilizado del momento en Internet, “Netscape Navigator”.

Las siguientes versiones de Java, ofrecerán en ese mismo año soporte para Windows NT y para Windows 95.

En 1996, Sun Microsystems, crea versión 1.0 de la JDK.

El lenguaje Java es un lenguaje compilado e interpretado a la vez, y el entorno donde se ejecutan las aplicaciones desarrolladas en Java se conoce como *“Java Virtual Machine (JVM)”* (Máquina Virtual de Java). La JVM ejecuta las siguientes funciones:

- Reservar espacio de memoria para los objetos creados.
- Liberar memoria no usada, utilizando el recolector de basura (*Garbage Collector*).
- Asignar variables a registros y pilas.
- Llamar al sistema huésped para ciertas funciones.
- Vigilar el cumplimiento de las normas de seguridad de las aplicaciones Java. El tema de seguridad en la JVM es uno de los más importantes, y se respalda con las propias especificaciones del lenguaje que aportan en aspectos de seguridad:
    - Referencias de Arrays verificadas en ejecución.
    - No manipulación directa de punteros a memoria.
    - Gestión de la memoria por la JVM de manera que no queden huecos.
    - No se permiten ciertas conversiones entre tipos de datos.

Una vez compilados los ficheros ".java", a un conjunto de instrucciones denominadas *bytecodes*, independientes del tipo de máquina y almacenadas en un fichero ".class", el intérprete ejecuta estas instrucciones en el ordenador que corresponda.

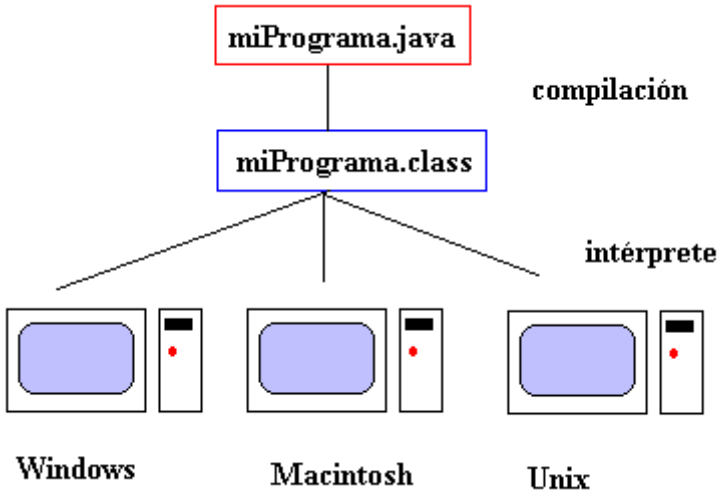


Ilustración 1. Proceso de compilación e interpretación de ficheros java.

2.1.4. COMPLEMENTOS DE ECLIPSE:

Tal y como se ha comentado anteriormente, eclipse es una plataforma de código abierto a la que los usuarios pueden contribuir añadiendo nuevas funcionalidades basadas en la utilización de elementos básicos de la plataforma. [25]

En la actualidad existen numerosos complementos o "plug-in" para eclipse que permiten muchos aspectos relacionados con el desarrollo de aplicaciones informáticas. Este amplio abanico abarca:

- **Complementos de lenguajes de programación:**

En la actualidad existen complementos para la plataforma eclipse que nos permitirán desarrollar de forma sencilla en lenguajes de programación como Java, C/C++, JavaScript, desarrollo web (HTML, CSS, PHP), PYTHON, ADT (Android Developpers Toolkit)…

Estos "plug-in" nos ofrecerán la posibilidad de desarrollar en cualquiera de estos lenguajes manteniendo una estructura, ofreciendo asistente de código, coloreado de palabras reservadas de cada lenguaje de programación, validaciones y otros elementos que facilitarán el proceso de desarrollo.

- **Versionado de aplicaciones:**

También podemos utilizar complementos de eclipse que nos permitan mantener un control mediante el uso de versionado de todo el proceso de desarrollo.

En este campo destacan principalmente los “plug-in” de eclipse CVS, SVN y GIT. Estos “plug-in”, nos permitirán mantener una organización mediante la utilización de repositorios de los proyectos que afectan a nuestro proceso de desarrollo. De esta manera serán accesibles para todos los participantes en el proceso y podremos crear varios procesos de desarrollo paralelos que podremos volver a juntar (“mergear”) en cualquier momento.

Estas herramientas facilitan tanto la organización de proceso de desarrollo como la posibilidad de recuperación de archivos mediante un histórico de contribuciones de código al repositorio de la aplicación, pudiéndose recuperar código que si por el contrario hubiésemos eliminado en local, acabaríamos perdiendo.

- **Modelado de aplicaciones:**

En este aspecto existen también numerosos “plug-in” para eclipse. Más adelante trataremos algunos de ellos que nos resultarán de interés en la realización desde este PFG. Actualmente existen “plug-in” que permiten modelar los distintos lenguajes de programación. Algunos de los más importantes son los ofrecidos por la JDT (nos permite acceder al modelo de Java) y la CDT (nos permite acceder al modelo de C/C++), EMF (Complemento de eclipse que facilita la ceración de modelos para el desarrollo de aplicaciones basada en modelos), UML (permiten la generación de diagramas UML e incluso en algunos casos la creación de código a partir de diagramas y de diagramas a partir de código), MODISCO (permite la creación de modelos basados en XML a partir de código JAVA).

Posteriormente analizaremos algunos de los complementos de eclipse orientado al modelado de lenguaje de programación y más en concreto, utilizados para el modelado de ficheros C y C++.

Además, de la cantidad de complementos para la plataforma eclipse a los que podemos acceder desde repositorios o en descarga desde internet. Eclipse nos ofrece la posibilidad de creación de nuestros propios "plug-in" de eclipse utilizando sus elementos básicos y dotándolos de nuevas funcionalidades para nuestro producto final. De esta manera, es posible crear acciones sobre botones, mostrar "wizards" de creación de artefactos, utilizar ventanas de comparación de ficheros y contribuir o utilizar en general cualquiera de los elementos que ofrece la plataforma, modificando su diseño y sus funcionalidades.

Este aspecto también lo trataremos más adelante, ya que dotaremos al entorno eclipse de nuevas funcionalidades que nos permitan ejecutar acciones que realicen transformación de código entre lenguajes de programación.

2.2. Ingeniería dirigida por modelos (MDE)

En este apartado del documento pretendemos explicar en qué consiste la ingeniería dirigida a modelos y que ventajas presenta el uso de modelos en un proceso de desarrollo.

En primer lugar, definimos un modelo de la siguiente manera:

> "Un modelo es una representación abstracta y parcial de un sistema o proceso"

Partiendo de esta definición, podemos concluir que un modelo es una representación parcial de la realidad, aplicable a cualquier elemento de nuestro entorno y que cambian en función de su objetivo. Un modelo es una simplificación de la realidad que nos facilita su comprensión.

De forma más específica, en el entorno de las tecnologías de la información, la utilización de modelo nos permite almacenar información ordenada y estructurada, facilitando la compresión del proceso que representa, permiten establecer un lenguaje único y mejorar el mantenimiento de los desarrollos en los que se utilizan.

Los expertos, consideran que la utilización de arquitecturas centradas en el uso de modelos es el mejor camino para satisfacer los requisitos, enfrentarse al aumento de la complejidad y a las limitaciones encontradas en los sistemas.

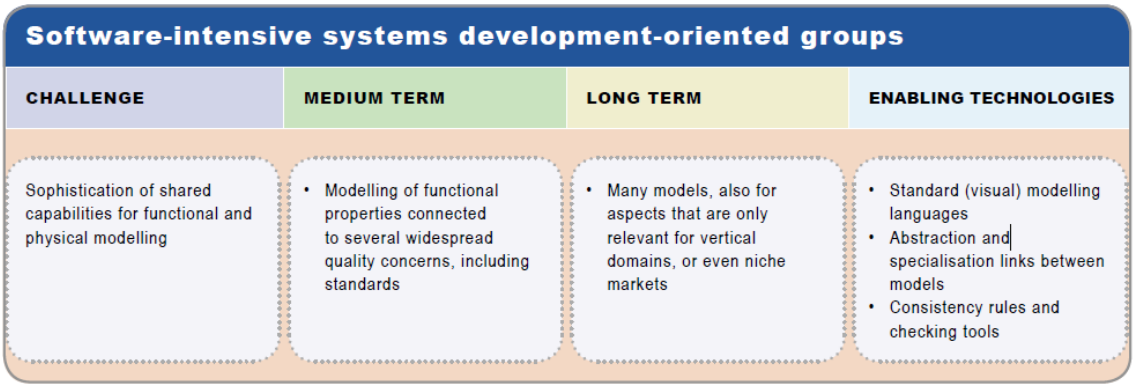


Ilustración 2. Sistemas de software orientados al desarrollo en grupo.

Cualquier proceso MDE, debe tener un objetivo, ya sea captura de requisitos, diseño/desarrollo de una aplicación, análisis y testeo de sistemas/aplicaciones…

Este objetivo será el que limite las condiciones de modelado, ya que definirá el lenguaje de modelado a utilizar que limitará a su vez las herramientas disponibles para modelar el proceso en base a dicho objetivo.

Otro factor a tener en cuenta en un proceso MDE es la representación de la misma realidad desde diferentes puntos de vista. El objetivo y la realidad que representa el modelo es el mismo, pero la percepción es diferente para cada miembro del proceso de desarrollo. Es por esto que pueden aparecer diferentes puntos de vista:

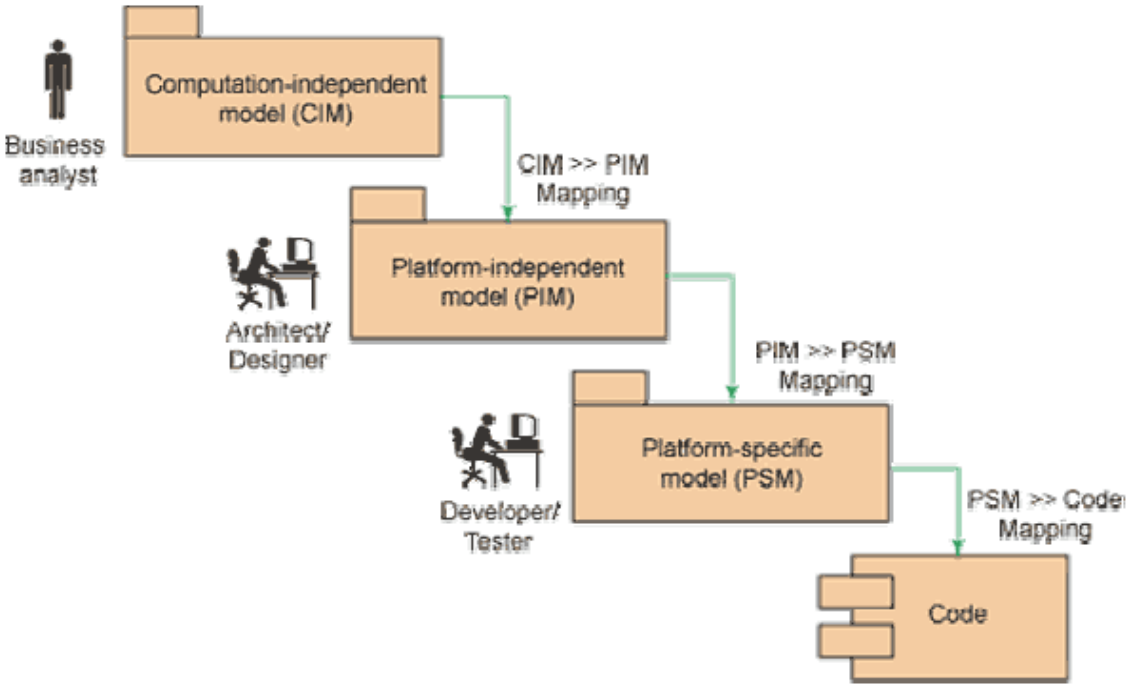


Ilustración 3. Misma realidad con diferentes puntos de vista durante el proceso de desarrollo.

Cada etapa del proceso puede mostrar un punto de vista diferente respecto a la misma realidad y por tanto el lenguaje de modelado puede ser diferente, pero debe existir un mapeo (relación) de conceptos en cada transición.

Para un analista de negocio el modelo y el desarrollo son totalmente independientes (CIM) puesto que sabe lo que quiere pero no repara en cómo debe conseguirse; para un arquitecto o diseñador la plataforma de desarrollo es independiente del modelo (PIM); pero un desarrollador necesita una plataforma específica para el modelo (PSM) pues ya sabe lo que se pretende conseguir, y necesita una plataforma concreta para trabajar con el modelo que se presenta en el proceso de desarrollo.

Cada una de estas etapas o vistas del proceso suele utilizar sus propias herramientas y su propia metodología. Existen herramientas que definen procesos MDE completos en base a diferentes objetivos (Simulink, Rational Software Architect), pero puesto que no siempre se ajustan a nuestras necesidades, podemos crear nuestra propia herramienta utilizando meta-modelos, definiendo los conceptos, las propiedades y las relaciones de nuestro modelo.

Cada modelo es conforme a su meta-modelo, que a su vez es conformo con su meta-meta-modelo de la siguiente manera:

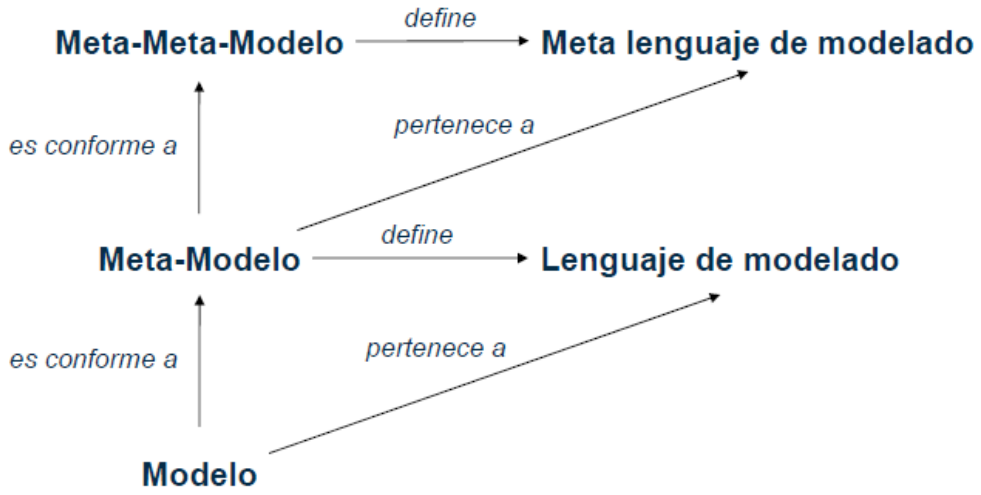


Ilustración 4. El mundo de los meta-modelos.

El proceso para la creación de meta-modelos requiere en primer lugar la definición de los conceptos que formarán parte de nuestro modelo. Es posible que algunos conceptos estén relacionados entre sí, por lo que estableceremos vínculos de herencia entre ellos; pero también es posible que algunos elementos contengan a otros, por lo que estableceremos vínculos de agregación entre ellos, o pueden estar relacionados entre sí de igual a igual.

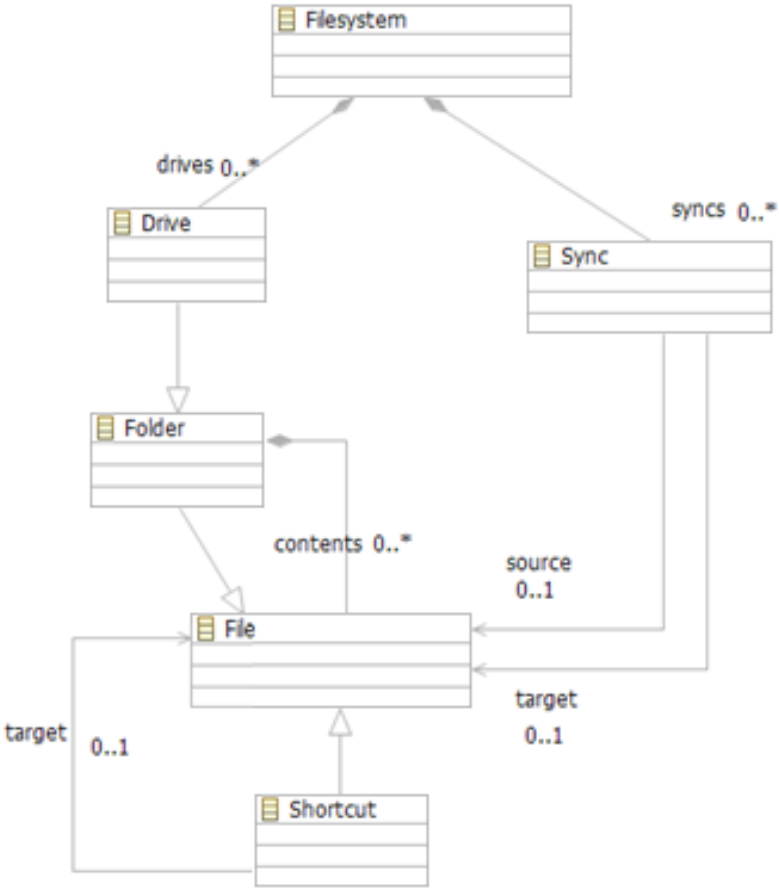


Ilustración 5. Representación gráfica de un meta-modelo.

La utilización de modelos, es beneficiosa para las empresas y para mejorar los procesos de desarrollo de aplicaciones. Un sistema basado en modelos es más sencillo de comprender y de utilizar y nos permite explotar de forma relativamente sencilla la automatización de herramientas, las herramientas de generación de código, configuraciones; si lo comparamos con otros mecanismos que tengan como objetivos estos mismos fines.

En la actualidad, una de las maneras más utilizada para la explotación de modelos y para sacar partido a los procesos de desarrollo basados en modelo son las transformaciones.

Estas arquitecturas basadas en modelos nos permiten realizar dos tipos de transformaciones: transformaciones modelo a modelo (M2M) y transformaciones de modelo a texto (M2T).

2.3. Transformaciones Modelo a Modelo (M2M)

Las transformaciones "model to model" (M2M), son uno de los principales mecanismo de explotación de las arquitecturas MDE/MDA.

Una transformación M2M consiste en la obtención a partir de uno o varios modelos, de un modelo más refinado.

En una transformación M2M se transforma un Modelo ***Ma*** instancia de un meta-modelo ***MMa*** en un modelo ***Mb*** instancia de un meta-modelo ***MMb***. Los meta-modelos ***MMa*** y ***MMb*** pueden ser meta-modelos iguales o diferentes.

Esta trasformación para obtener un modelo más refinado de una representación de la realidad requiere mapeos de conceptos, reglas de inferencia y restricciones entre los modelos de entrada y el modelo de salida.

Ilustración 6. Obtención de modelo más refinado a partir de un modelo de entrada.

En los procesos MDE, las transformaciones de modelo a modelo tienen tres objetivos:

- **Automatización de transiciones entre fases:** Mapeo de conceptos entre los modelos de dos vistas (puntos de vista) distintos. Este mapeo se realiza de una vista a su consecutiva.

- **Vinculación de modelos con herramientas externas:** ajuste de modelos para obtener uno más refinado compatible con herramientas externas que pretendemos utilizar.

- **Mezcla de varios modelos en uno:** pretendemos crear un modelo más completo a partir de uno o varios modelos base. Esto resultará beneficioso ya que tanto las consultas como las modificaciones se centralizarán en un solo modelo en lugar de tener varios.

Los motores de transformación se basan en reglas, por lo que si los modelos de entrada son incorrectos o representan distintas realidades para diferentes personas, todas las transformaciones fallarán.

Para intentar que esto no ocurra, deberíamos crear reglas que permitan distinguir los modelos válidos de los que no lo son, unificar los criterios para que el modelo represente la misma realidad desde todos los puntos de vista, definir una semántica y evitar errores en las etapas posteriores al proceso MDE.

Para establecer estas reglas se utiliza el estándar Object Constraint Language (OCL). OCL es un lenguaje muy completo pero también es muy complejo y muy difícil de manejar por completo, por lo que debemos tener en cuenta que aplicar reglas sobre nuestros modelos no será algo sencillo y trivial.

Algunos de los lenguajes más utilizados en las transformaciones de modelo a modelo son ATL, QVT y XTEND.

2.3.1. ATL:

Una transformación ATL se compone de un conjunto de "helpers" y de reglas de transformación.

Cada una de las reglas de transformación define como una parte del modelo de destino se debe generar en base a una parte del modelo de origen. En lenguaje ATL hay dos tipos de reglas "*lazy* y *matches* rules".

Las reglas de tipo *lazy* son ejecutadas de forma automática por el motor ATL. Cuando el motor de ATL encuentra un elemento de entrada del modelo base adecuado a una regla, la ejecuta.

Las reglas de tipo *matches* no se ejecutan de forma automática sino que deben ser llamadas desde otras reglas, otorgando un mayor control sobre la ejecución de la transformación.

Una regla se compone principalmente de un patrón de entrada y un patrón de salida:

- **Patrón de entrada:** filtro y definición del conjunto de elementos del modelo de entrada que serán transformados por la regla que se define.

- **Patrón de salida:** definición de la creación de los elementos de salida a partir de los elementos de entrada.

Respecto a los *helpers* que ofrece el lenguaje ATL, pueden considerarse funciones auxiliares que nos permiten factorizar código ATL que se utiliza en diferentes partes de la transformación.

```
module UML2ER;
create OUT : ER from IN : UML;

helper context UML!Class def: allClasses() : Sequence(UML!Class) =
    self.superClasses->iterate(e; acc : Sequence(UML!Class) = Sequence {} |
    acc->union(Set{e})->union(e.allClasses()) );
rule Class {
 from
  s: UML!Class
 to
  t: ER!EntityType (
   name <- s.name,
   features <- Sequence {attributes, weakReferences, strongReferences}
  ),
  attributes : distinct ER!Attribute foreach(a in
        s.allClasses().including(s).flatten()
        ->collect(e | e.ownedProperty).flatten()
        ->select (e | not e.primitiveType.oclIsUndefined())) (
    name <- a.name,
    type <- a.primitiveType
  ),
  weakReferences : distinct ER!WeakReference foreach(a in
        s.allClasses().including(s).flatten()
        ->collect(e | e.ownedProperty).flatten()
        ->select(e | not e.complexType.oclIsUndefined() and not e.isContainment)) (
    name <- a.name,
    type <- a.complexType
  ),
  strongReferences : distinct ER!StrongReference foreach(a in
        s.allClasses().including(s).flatten()
        ->collect(e | e.ownedProperty).flatten()
        ->select(e | not e.complexType.oclIsUndefined() and e.isContainment)) (
    name <- a.name,
    type <- a.complexType
  )
}
```

Ilustración 7. Ejemplo de transformación ATL de UML a ER.

El uso de transformaciones ATL, no es algo tan sencillo como parece y debemos tener en cuenta varios factores a la hora de implementarlas, ya que por ejemplo es conveniente realizar varias reglas con un trabajo de transformación limitado en lugar de crear una sola regla con mucho contenido que se encargue de todo el proceso de transformación.

Si algún día queremos reutilizar esta regla o refactorizar su contenido, tendríamos que volver a hacerlo por completo ya que esa regla es nuestro motor de transformación. Si por el contrario dividimos la transformación en varias reglas más pequeñas, podremos aprovecharlas en otros procesos de transformación y podremos refactorizar de forma más sencilla el código si hubiese que realizar cambios en la transformación del modelo.

Otro factor que deberíamos tener en cuenta, es el número de llamada que realizamos, ya que la utilización de llamadas innecesarias reduciría el rendimiento de la transformación.

2.3.2. QVT Transform

Las siglas QVT significan “Query Views Transformations”. Se trata de una lengua estándar OMG que se utiliza para expresiones de consulta, vistas y transformaciones en modelos MOF.

Las consultas de modelo y las vistas de modelo pueden ser consideradas un tipo especial de transformaciones de modelo, siempre que utilicemos la definición adecuada de transformación de modelo:

“Una trasformación de modelo es un programa que se ejecuta en modelos.”

El estándar QVT define tres lenguajes de transformación de modelos que se ajustan a meta-modelos MOF-2. Una transformación QVT en cualquiera de sus tres lenguajes puede considerarse un modelo, de acuerdo a los meta-modelos especificados en el estándar.

QVT está diseñado para construir modelos de destino a partir de estructuras complejas, pero pueden aparecer dificultades en la descripción cuando no existe correspondencia directa entre los elementos de origen y el modelo de destino.

Se trata de un lenguaje imperativo que especifica los pasos y el orden de ejecución para obtener el resultado.

Se define el proceso para convertir *n* modelos de origen en *n* modelos de destino, siendo el caso más común la transformación de un modelo ***Ma***, instancia de un meta-modelo ***MMa***; que se transforma en un modelo ***Mb*** instancia de un meta-modelo ***MMb***.

Como hemos dicho antes la arquitectura QVT está formada por tres lenguajes (QVT-Operational, QVT-Relations y QVT-Core) y un mecanismo de conexión con programas externos durante el proceso de transformación conocido como “Black Box”.

- **QVT-Relations:** Se trata de un lenguaje declarativo diseñado para permitir transformaciones de modelos unidireccionales y bidireccionales. Cada transformación hace referencia a una relación de coherencia entre un conjunto de modelos. Utiliza patrones de objetos para su adaptación y para crear instancias. Ofrece manejo automático de enlaces de trazabilidad. Las transformaciones son potencialmente multidimensionales. Soporta escenarios de ejecución:

  - **Chek-Only:** comprueba si el conjunto de modelos es coherente con la transformación.

  - **Transformaciones unidireccionales y multidireccionales.**

- **Actualización incremental de modelos existentes.**

- **QVT-Core:** Se trata de un lenguaje declarativo basado en las relaciones definidas en los elementos de los modelos del meta-modelo. Su objetivo es traducir las QVT-Relations. Utiliza patrones de objetos más simples y el manejo de los enlaces de trazabilidad es manual. Es un lenguaje más detallado que el de relaciones. En cuanto a los escenarios de ejecución que soporta, son los mismos que el lenguaje de relaciones; pero ofrece dos opciones de uso:

  - **Transformaciones simples de lenguaje.**

  - **Punto de referencia para la definición de la semántica del lenguaje de relaciones.**

- **QVT-Operational:** Se trata de un lenguaje imperativo diseñado para escribir transformaciones unidireccionales.

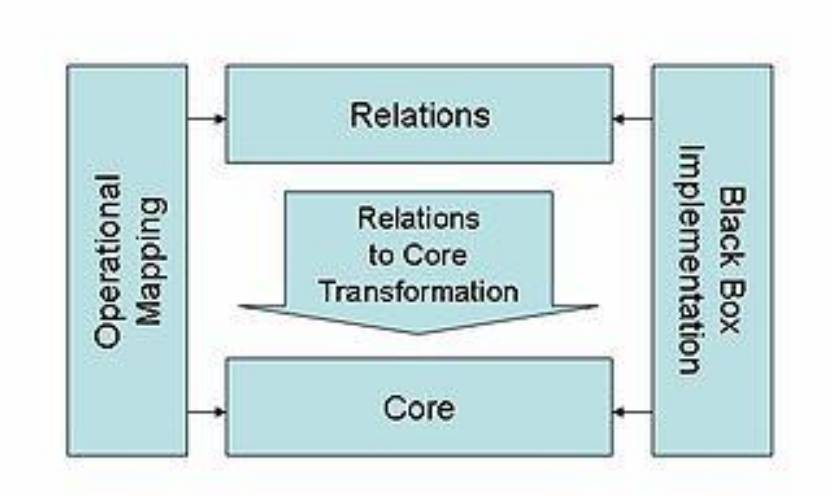


Ilustración 8. Arquitectura QVT.

2.4. Transformaciones Modelo a Texto (M2T)

En el mundo de la ingeniería dirigida por modelos (MDE) las transformaciones cumplen un papel fundamental, pudiendo establecer un vínculo entre los diferentes modelos que se generan durante todo el proceso de desarrollo.

Hemos de tener en cuenta, que por lo general un proceso MDE tiene como fin la obtención de código a partir de un modelo. El proceso puede ser más o menos largo pudiendo así implicar más o menos transformaciones entre modelos (M2M) pero la idea fundamental es obtener un modelo definitivo al final del proceso de desarrollo que posteriormente será transformado a código mediante una transformación de modelo a texto.

El funcionamiento de una transformación de modelo a texto (M2T) consiste en un proceso por el que uno o varios modelos de entrada se transforman para generar ficheros de texto plano. Para llevar a cabo estas transformaciones se aplican reglas de transformación sobre cada uno de los elementos del meta-modelo que nos interesan convertir a texto plano.

Durante la transformación del modelo a texto, se lleva a cabo un proceso de inferencia por el cual se deduce la salida que se espera para cada uno de los elementos del meta-modelo que se analizan.

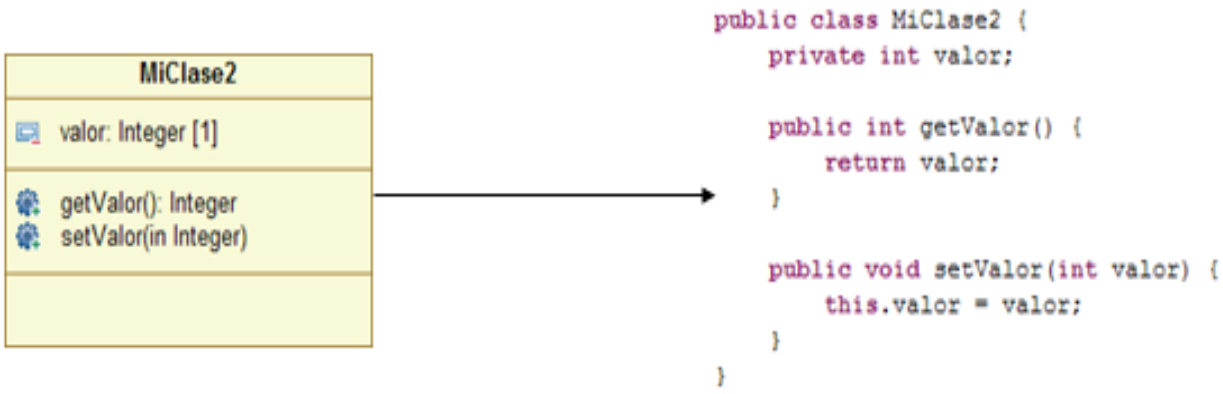


Ilustración 9. Trasformaciones de modelo a texto.

Estos procesos de transformación también son conocidos como transformaciones M2T, “model to text” o procesos M2C (“model to code” -> modelo a código).

Como hemos comentado antes, para este proceso solo necesitamos uno o varios modelos de entrada y definir las reglas de transformación que nos interesen. Estas reglas pueden corresponderse con reglas de transformación para obtención de código fuente en un lenguaje de programación conocido pero también debemos tener en cuenta que puede crearse cualquier tipo de regla con el fin de obtener el texto en plano que nos interese.

Este enfoque es interesante, porque de esta manera se globaliza el enfoque de las transformaciones M2T, ya que nos permite enfocar estas transformaciones muchos otros campos o fases del proceso de desarrollo.

Si por ejemplo, queremos enfocar nuestros proceso de transformación al lenguaje de programación JAVA, podríamos utilizar uno o varios modelo de entrada que se corresponderían con diferentes objetos que representarían una parte de la realidad; podríamos aprovechar una arquitectura MDE/MDA enfocada a diferentes fases del ciclo de vida de un proceso de desarrollo. Bastaría con aplicar diferentes reglas de transformación sobre los mismos modelos de entrada, generando así salidas en texto distintas.

Así pues, por ejemplo, podríamos aplicar las reglas necesarias para transformar esos modelos de entrada en sus clases ".java" correspondientes; pero podríamos aplicar otras reglas por las que para cada atributo y método del modelo de entrada se genere de forma automática clases ".java" con test de pruebas unitarias ***JUnit***; o cualquier tipo de regla que genere código fuente que pueda resultarnos de utilidad durante el ciclo de vida de nuestra aplicación.

Pero además, estos sistemas M2T, pueden enfocarse a temas totalmente contrarios a la generación de código fuente. Si aplicásemos las reglas correspondientes, podríamos por ejemplo crear ficheros ".txt" sobre cada elemento del modelo para generar documentación o información adicional al código que se está generando.

En general, podrían aplicarse estas transformaciones M2T a cualquier cosa que se nos ocurra y que por supuesto nos resulte interesante para nuestro desarrollo, como la generación de ficheros **XML,** modelos **XMI**… ya que, lo que en cierto modo consiguen estas reglas de transformación, es automatizar la generación de código aplicando reglas de transformación sobre los modelos de entrada.

Este es el enfoque que queremos darle en nuestro PFG, dónde pretendemos aplicar reglas de transformación sobre un modelo obtenido de una clase C++03 para obtener código C++11.

Al igual que en el caso de las transformaciones de modelo a modelo (M2M), los motores de transformación se basan en reglas, por lo que si los modelos de entrada son incorrectos o representan distintas realidades para diferentes personas, todas las transformaciones fallarán.

Para intentar que esto no ocurra, deberíamos crear reglas que permitan distinguir los modelos válidos de los que no lo son, unificar los criterios para que el modelo represente la misma realidad desde todos los puntos de vista, definir una semántica y evitar errores en las etapas posteriores al proceso MDE.

Para establecer estas reglas se utiliza el estándar Object Constraint Language (OCL). OCL es un lenguaje muy completo pero también es muy complejo y muy difícil de manejar por completo, por lo que debemos tener en cuenta que aplicar reglas sobre nuestros modelos no será algo sencillo y trivial.

Para realizar estas transformaciones, al igual que en las transformaciones M2M, pueden utilizarse diversas herramientas y lenguajes de transformación. A continuación, analizaremos algunas de ellas.

2.4.1. ACCELEO

Acceleo es un generador de código basado en las especificaciones de las transformaciones de modelo de texto de las OMG.

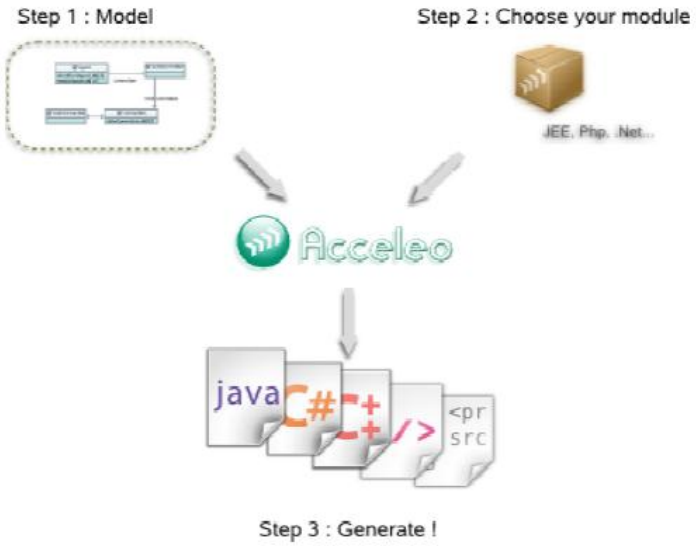


Ilustración 10. Funcionamiento ACCELEO

Acceleo contiene todas las herramientas que se esperan de un IDE de generación de código de calidad: sintaxis simple, generación de código eficiente, herramientas avanzadas, características similares a la JDT…

Su enfoque basado en prototipos y modelos facilita a los desarrolladores la creación de generadores de texto basados en el código fuente de los prototipos existentes. El resto de herramientas que ofrece Acceleo como la refactorización de código, una sintaxis estructurada y diferenciada por colores y la posibilidad de utilizar autocompletado de código al incorporarlo en el entorno de desarrollo eclipse, facilitan también la generación de plantillas generadoras de código.

Acceleo es un producto basado en eclipse e incluido entre las actualizaciones de eclipse desde su versión HELIOS (Eclipse 3.6.x), creado por OBEO.

Acceleo.org se creó en 2005, y con el paso del tiempo fue enfocándose poco a poco hacia temas relacionados con transformaciones de modelo a texto, debido al convencimiento de que este era el camino que deberían de seguir de cara a un futuro dónde los sistema MDE y las arquitecturas MDA resultarían muy interesantes. Consiguieron una sintaxis fácil y simple y un sistema de generación de código eficiente y potente.

Se evolucionó la primera versión de ACCELEO a una versión más refinada, con una sintaxis diferente, pero con un enfoque y una arquitectura prácticamente iguales en ambas versiones.

Posteriormente se incorporó como un proyecto de eclipse en su IDE de desarrollo. Tras la incorporación se realiza un cambio de sintaxis en sus plantillas, manteniéndose la interoperabilidad entre la sintaxis antigua y la nueva.

El equipo de Acceleo.org mantendrá la sintaxis antigua un par de años más fuera del entorno de desarrollo de eclipse, y posteriormente aportará una herramienta de migración de plantillas de la sintaxis antigua a la nueva.

Pretenden y consiguen que sea completamente compatible la versión de ACCELEO con la versión HELIOS de eclipse (v3.6.x) y poco a poco versionarán los complementos de ACCELEO para eclipse, mejorando algunos aspectos y dotando a las herramientas de nuevas funcionalidades y mejorando las ya incorporadas en el paquete ACCELEO.

En cuanto al funcionamiento de la herramienta es bastante sencillo. Se trata de una herramienta compatible con muchos de los modelos que actualmente podemos utilizar para representar realidades, entre los que se encuentran modelos UML, UML2, EMF, ECORE, XML, XMI, modelos generados por MoDisco (JAVAXMI)…

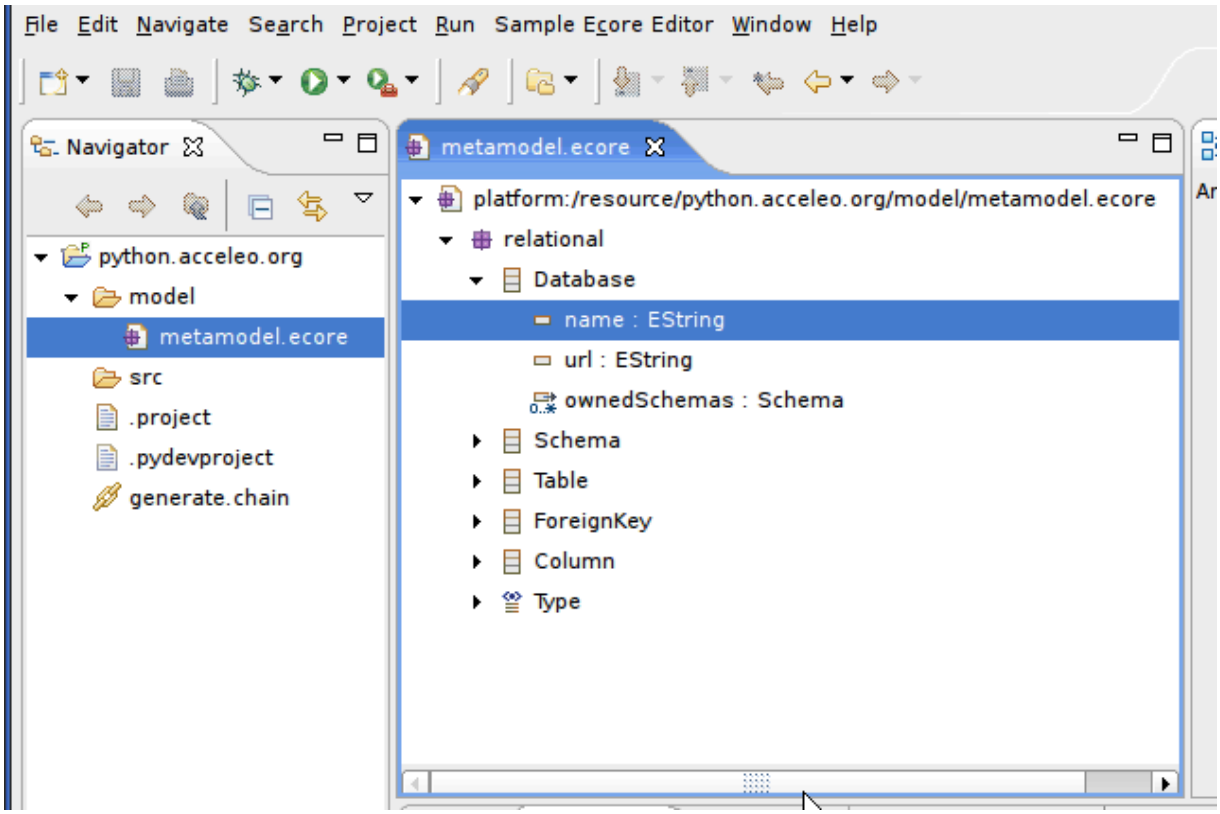


Ilustración 11. Meta-modelo de entrada ".ecore".

Cada uno de estos modelos presenta una estructura de contenido que referencia atributos o funciones de la realidad que representan. Todos estos atributos son lo que nosotros podremos iterar a través de nuestras plantillas de transformación para generar el texto en plano que nos interese en cada caso.

Estas iteraciones y sus transformaciones correspondientes se harán a través de plantillas ACCELEO. Para crear una plantilla, bastaría con crear un fichero ".mt" y rellenarla utilizando la sintaxis de ACCELEO con las transformaciones sobre el modelo que le interesen al usuario.

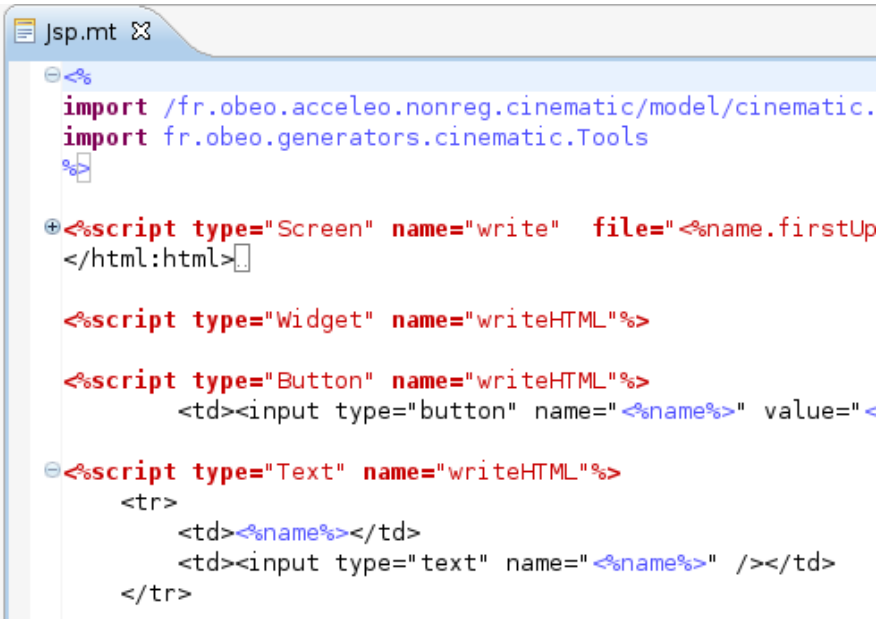


Ilustración 12. Plantilla transformación ACCELEO v2.x

```
[comment encoding = UTF-8 /]
[module generate('http://www.eclipse.org/emf/2002/Ecore')/]

[template public generate(e : EClass)]
[comment @main /]
[file (e.name + '.java', false, 'UTF-8')]
public class [e.name.toUpperFirst()/] {
    [e.generateAttributes()/]
    [e.generateMethods()/]
}
[/file]
[/template]

[template public generateAttributes (eClass : EClass) ]
[for (attribute : EAttribute | eClass.eAllAttributes)]
/**
 * The documentation of [attribute.name/]
 */
private [attribute.eType.instanceClassName/] [attribute.name/];

[/for]
[/template]

[template public generateMethods (eClass : EClass) ]
[for (operation : EOperation | eClass.eAllOperations)]
/**
 * The documentation of [operation.name/]
 */
public void [operation.name/]() {

}

[/for]
[/template]
```

Ilustración 13. Plantilla de transformación ACCELEO v3.x

Una vez analizado el modelo de entrada y definidas las reglas en la plantilla de transformación de ACCELEO, la herramienta analiza de forma automática el modelo y

le aplica las transformaciones correspondientes a cada elemento de entrada del modelo en base a las reglas de transformación que se han definido en la plantilla, obteniendo el texto en plano como salida.

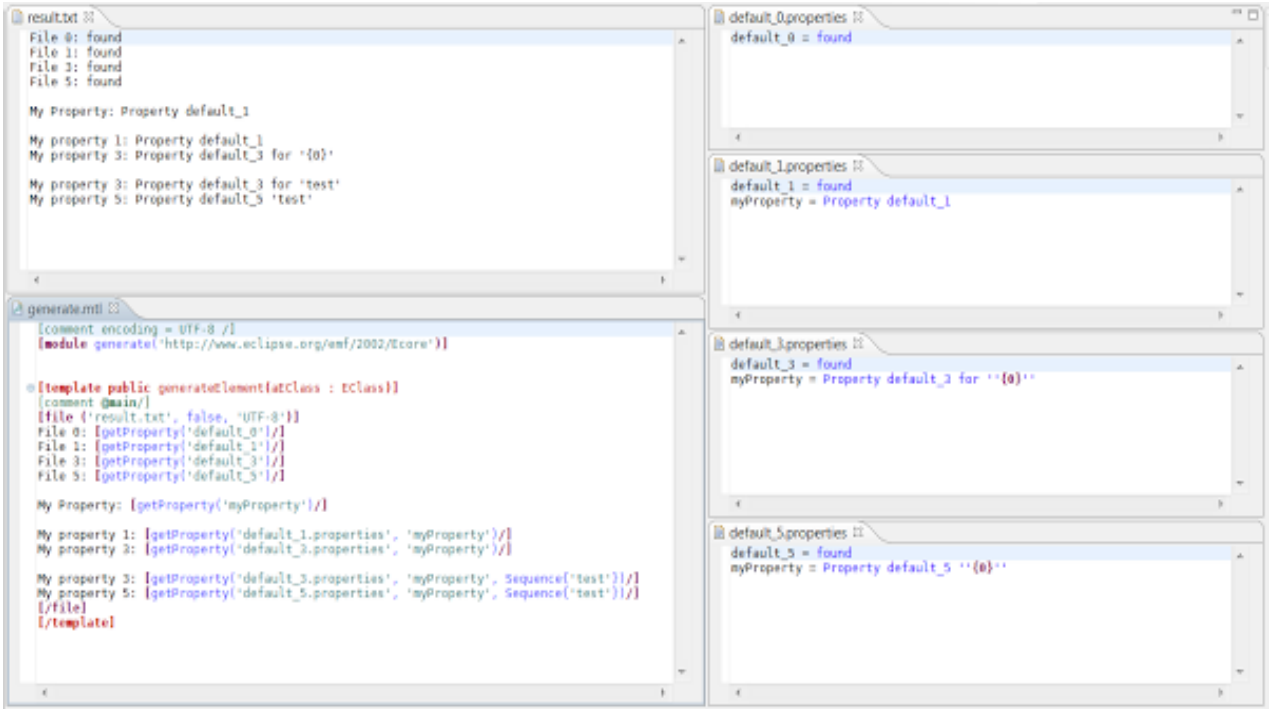


Ilustración 14. Proceso de transformación con ACCELEO.

ACCELEO es considerado por los desarrolladores como uno de los generadores de código para arquitecturas MDA más potentes de la actualidad. Para nuestro PFG en concreto, podría resultarnos de mucha utilidad, ya que lo que pretendemos es generar código a partir de un modelo existente, pero a continuación conoceremos otras herramientas que también podrían resultarnos de utilidad e incluso ser más potentes y eficientes para aplicar transformaciones de C++ a C++11.

2.4.2. XPAND/XTEND

XPAND es un framework de generación integrado en eclipse. La integración y utilización de XPAND va de la mano de otros dos elementos necesarios para el análisis de modelo y la realización de transformaciones sobre este, estos dos complementos son XTEXT y XTEND.

XTEXT permite definir una gramática para el análisis de los modelos y XTEND nos permitirá analizar el modelo y realizar transformaciones M2M o M2T.

Como el resto de herramientas que estamos exponiendo, es útil en varios puntos de un proceso MDE como las transformaciones entre modelos provenientes de diferentes puntos de vista de una misma realidad, para la obtención de modelos más completos; y para el proceso final de obtención de código a partir de un modelo.

Cada uno de estos dos lenguajes (XPAND/XTEND) están construidos en base a expresiones y tipos de sistema, globalizando y simplificando su utilización, ya que la manera de interactuar con los modelos, los meta-modelos y los meta-meta-modelos será exactamente la misma y no necesitaremos aprender el lenguaje de nuevo.

El framework ofrece un potente lenguaje de expresiones estáticas y proporciona una capa de abstracción uniforme para interactuar con diferentes meta-modelos con los que es compatible. Entre los modelos compatibles con estos lenguajes y herramientas se encuentra EMF Ecore, Eclipse UML, JavaBeans, XML, XMI…

La capa de abstracción se conoce como sistema de tipos. Esta capa es la que se encargará de dar acceso a los tipos de datos que tiene registrados y se corresponden con los tipos de datos del modelo o modelos concretos que se pretende analizar.

Es importante diferenciar el sistema de tipos del propio lenguaje de expresiones ya que se trata de dos elementos totalmente independientes. Mientras el sistema de tipos actúa como una especie de capa de reflexión que se puede ampliar con las implementaciones del meta-modelo; el lenguaje de expresiones solo define una sintaxis concreta para las expresiones que el sistema de tipos permite.

Cada objeto, en este caso cada elemento del modelo, tiene un tipo. Este tipo contiene propiedades y operaciones (funciones) y a su vez podría heredar de otros elementos.

Cada uno de estos tipos que están registrados en el sistema de tipos, podría ser iterado utilizando lenguaje XPAND.

El lenguaje XPAND se utiliza en plantillas para controlar y gestionar la generación de una salida a partir de un modelo de entrada. Una plantilla, es un fichero “.xpt”.

Esta sintaxis, permite realizar importaciones, bucles, análisis de objetos… pero siempre entre “Guillemets” («[expresión]»).

Partiendo de un modelo base reconocido por la herramienta, iteraremos utilizando estas expresiones entrecomilladas para gestionar la salida en texto que nos interese.

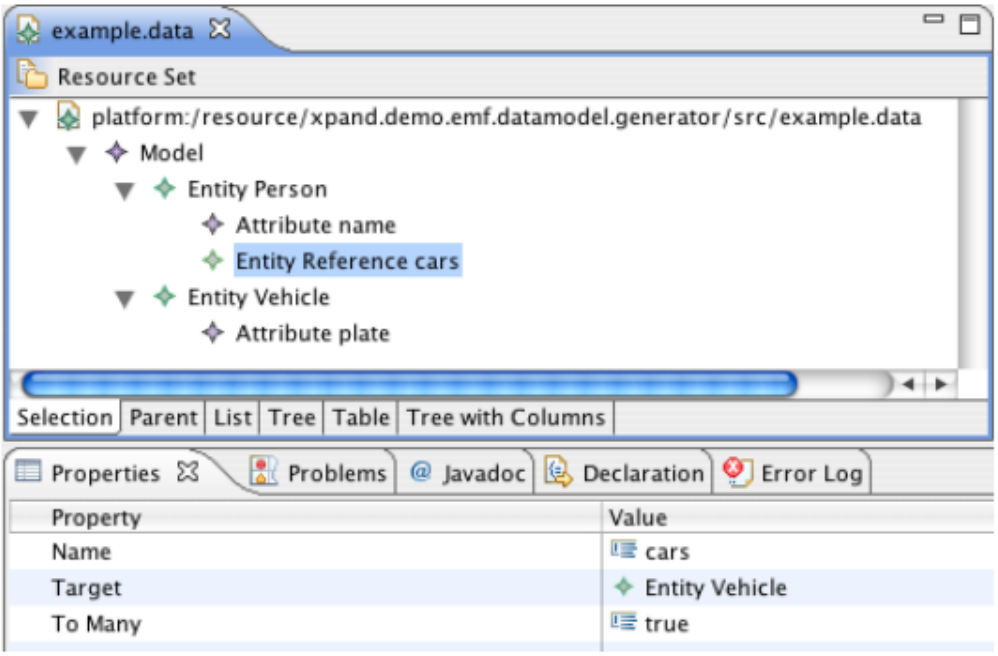


Ilustración 15. Modelo EMF Ecore.

Nosotros seremos quienes decidamos que elementos del modelo son susceptibles a ser transformados a código, o si por ejemplo, quisiésemos generar documentación, no deberíamos realizar transformaciones sino lecturas de atributos de estos elementos; pero aun siendo así, todas las iteraciones que se realizarán sobre el modelo.

Main.xpt

```
«IMPORT mydsl»

«DEFINE main FOR Model»
   «EXPAND javaBean FOREACH types.typeSelect(Entity)»
«ENDDEFINE»

«DEFINE javaBean FOR Entity»
«FILE name+".java"»

public class «name» {
    «FOREACH features AS f»
    private «f.type.name» «f.name»;

    public «f.type.name» get«f.name.toFirstUpper()»() {
        return «f.name»;
    }

    public void set«f.name.toFirstUpper()»(«f.type.name» «f.name») {
        this.«f.name» = «f.name»;
    }
    «ENDFOREACH»
}
«ENDFILE»
«ENDDEFINE»
```

Ilustración 16. Plantilla de transformación XPAND.

Uno de los principales problemas que pueden aparecer al utilizar plantillas XPAND es que el sistema de tipos no tenga registrados todos los elementos que queramos transformar, y necesitemos realizar una transformación más compleja.

Si llegamos a este punto, recurriremos a XTEND. Al igual que otros sublenguajes, XTEND nos ofrece un amplio abanico de expresiones en su sintaxis.

Este lenguaje nos permite definir nuestras propias librerías de operaciones independientes y extensiones del meta-modelo no invasivas. Estas bibliotecas podrán ser cargadas y llamadas en cualquiera de los demás lenguajes de expresiones del framework.

Un fichero XTEND tendrá extensión ".ext" y debe ubicarse en la ruta de la clase java que está en ejecución.

Desde su versión 4.1, XTEND soporta un apoyo adicional para realización de transformaciones sobre el modelo. Este concepto se conoce como creación de extensiones. Este sistema se utiliza cuando un mismo modelo se referencia varias veces. Si creamos extensiones podremos definir una transformación para todos los casos evitando la referencia múltiple.

El principal inconveniente que podemos encontrarnos es que nos embuclemos o tengamos referencias cíclicas. Para utilizar correctamente este elemento, debemos tener una estructura correctamente organizada y definida.

Helpers.ext

```
1 import metamodel;
2
3 getterName(metamodel::Attribute attribute):
4     "get" + attribute.name.toFirstUpper() + "()";
```

Simple.xpt

```
1  «IMPORT metamodel»
2
3  «EXTENSION template::Helpers»
4
5  «DEFINE javaClass FOR Entity»
6  «FILE name+".java"»
7  public class «name» {
8      «FOREACH attributes AS attr»
9          private «attr.type» «attr.name»;
10         public «attr.type» «attr.getterName()» {
11             return this.«attr.name»;
12         }
13     «ENDFOREACH»
14     «FOREACH references AS ref»
15         public «ref.type.name» «ref.name»;
16     «ENDFOREACH»
17 }
18 «ENDFILE»
19 «ENDDEFINE»
```

Ilustración 17. Fichero XPAND y librería XTEND.

Al igual que en el caso de ACCELEO, se trata de un motor de transformaciones M2T muy potente, pero algo obsoleto.

Además, el tiempo de ejecución en transformaciones grandes, que requieran numerosas interacciones entre el modelo y las reglas de transformación, era bastante elevado, por lo que la eficiencia en las transformaciones se ve bastante afectada.

Posteriormente a XPAND/XTEND, aparecerá XTEND2 de la mano de XTEXT y XBASE. Es una evolución de XPAND/XTEND basada en java, mucho más potente y eficiente que analizaremos más adelante.

2.4.3. XTEND2

El anterior sistema de transformaciones modelo a texto, M2T Xpand, es uno de los lenguajes de transformación a texto más amplios y más utilizados por los usuarios para este fin.

Sus ventajas respecto al resto de sistemas es que permite navegar de forma sencilla y rápida a través de los modelos. Esta facilidad de navegación del modelo en parte viene dada por la posibilidad de crear funciones de extensión a través de Xtend.

Otra de las ventajas por las que es conocido Xpand es por ofrecer una arquitectura muy abierta.

Pero, al analizar los inconvenientes que este sistema de transformaciones modelo a texto, y viendo el éxisto que estaba teniendo entre las herramientas de transformación utilizadas para este fin, fue cuando su creador se vió en la obligación de mejorar la arquitectura y evolucionarla en algo mucho más potente y sencillo de utilizar. De esta manera apareció XTEND2.

Los principales inconvenientes que ofrecía Xpand, y que se vieron obligados a mejorar en la nueva arquitectura XTEND2, fueron:

- **Rendimiento:** Xpand y Xtend, son lenguajes interpretados. Su intérprete está escrito sin tener en cuenta factores de rendimiento en su compilación, por lo que como ya contamos anteriormente, los tiempos de compilación/transformación son muy elevados en medida que el modelo a transformar se complica. Xpand, es muy lento y ofrece pocas prestaciones frente a grandes arquitecturas.

- **Herramientas:** La herramienta como tal, están hecha con JFace, con código muy viejo y poco probado. Por lo que es muy susceptible a errores cuando lo utilizamos.

- **Ayudas conceptualmente erróneas:** las herramientas de ayuda que ofrece la arquitectura no son correctas y en ocasiones interrumpen al usuario cuando está utilizando el sistema de transformaciones.

Debido a todo esto, y a lo complicado que había sido crear un IDE para Xpand/Xtend y el resultado obtenido, su creador decidió buscar una manera de evolucionarlo y mejorarlo, para poder así ofrecer mejores prestaciones al usuario como asistente de código, coloreado de sintaxis, vistas de esquemas y árboles de navegación de contenido…

Buscando todas estas mejoras, Sven Efftinge, su creador, presentó el primer prototipo de Xtext en 2006, con la suerte de que algunos de sus amigos y algunos patrocinadores se involucraron junto a él, en el proyecto más grande del momento para Eclipse.

Posteriormente, Xtext, estaría lista para ser la base de una nueva versión reescrita de Xpand, que hoy día se conoce como XTEND2. Esta nueva versión adquiere este nombre y en lugar de Xpand, porque la sintaxis de las plantillas de este último, podrá utilizarse como expresiones.

Tras haber diseñado Xpand/Xtend, tendrían que pensar que características debían mantener y eran productivas y cuáles eran las que deberían mejorarse.

Las principales diferencias que aporta XTEND2 con respecto a su anterior versión Xpand/Xtend son:

- **Basado en XBASE:** XTEND2, como es lógico, surge del uso de las expresiones definidas en XBASE. Gracias a esto, XTEND2 está estrechamente integrado con Java, teniendo cierre de llaves, tipos de datos y elementos de java bastante interesantes como el "switch".

- **Compila a Java:** los ficheros o plantillas de XTEND2, se compilan automáticamente a Java, generando ficheros ".java" con el parseo a este lenguaje del contenido y las funcionalidades que ofrecen las plantillas. Esto ofrece además otras ventajas, como la posibilidad de depurar el código en caso de problemas (al ser java podemos usar el depurador de este). Como principal inconveniente, el código java autogenerado es bastante "feo" y poco legible. A pesar de esto, el lenguaje compilado es bastante eficiente y rápido de ejecutar, incluso mejor en muchos casos del que podría crear un usuario. Además el lenguaje utilizado en las plantillas es muy similar a Java, lo que facilita su aprendizaje y su utilización si conoces este lenguaje o algún lenguaje orientado a objetos similar.

- **Contiene las buenas características que ofrecían XPAND/XTEND, pero mucho mejor todavía:** esta nueva reescritura de su predecesor, mantiene las características que lo llevaron al éxito como el envío polimórfico, métodos de extensión y la sintaxis de las antiguas plantillas; pero aún más potentes y ofreciendo muchas más posibilidades al usuario. Además de poder crear y aplicar funciones de extensión generadas sobre las plantillas u otros ficheros Xtend, es posible utilizar cualquier función estática de java, inyección de objetos y utilización de métodos inyectados, y tipos de datos nativos de java.

- **Herramienta sofisticada pero simple para el usuario:** pueden observarse, en general, muchas otras mejoras con respecto a la versión anterior, pero lo mejor del lenguaje no es que sea demasiado extenso, si no que ofrece los elementos necesarios muy bien estructurados y organizados. Elimina todo lo que considera innecesario en la sintaxis: no hay punto y coma, no son necesarios los paréntesis, no es necesaria la conversión entre tipos de datos en muchos casos. Estos elementos mejoran la visibilidad general de los ficheros.

- **Ofrece la posibilidad de mejorar las API de Java existentes:** permite mejorar las API de Java existentes mediante la utilización de métodos de extensión y expresiones lambda. Permite usar los operadores donde quieras y eliminar patrones redundantes utilizando anotaciones.

- **IDE Totalmente integrado con Eclipse:** todo el lenguaje y el entorno han sido diseñados de manera independiente, pero está integrado al cien por cien con el IDE de desarrollo Eclipse.

- **Fácil aprendizaje:** XTEND2 se basa en los conceptos del lenguaje Java añadiendo conceptos de lenguajes modernos por encima. En diferencia con otro lenguajes JVM, no agrega un nuevo sistema de tipos, ya que esto dificultaría el aprendizaje a los usuarios. De esta manera, podríamos decir que si conoces Java, sabrías crear tus propias plantillas XTEND2.

- **La ventaja de seguir siendo código java:** la compilación de XTEND2 a ficheros java de forma automática, que además son relativamente comprensibles nos permitirá utilizar este código java en otras plataformas basadas en él, como GWT.

Es por todas estas ventajas, por lo que XTEND2 se convierte en un mecanismo de transformación muy interesante para arquitecturas MDA y sistemas MDE que requieran transformaciones de modelo a texto en alguna de sus fases.

Se trata de un lenguaje muy potente que al ser compilado a java, reduce el tiempo de ejecución al mínimo, ya que estaríamos ejecutando java sobre java, en lugar de interpretar cada una de esas plantillas y compararla con el modelo.

Gracias a esto podemos obtener muy buenas prestaciones y como en casos anteriores aplicarlo en varias fases de nuestro proceso de desarrollo y con diferentes objetivos.

Podría interesarnos tener una plantilla que genere una clase java a partir de un modelo; pero también podríamos utilizar el mismo modelo de entrada sobre diferentes plantillas y generar para el mismo modelo un fichero JavaScript, HTML, XML, Documentación… Y sería tan sencillo como tener una plantilla común para todos ellos que se encargue de analizar cada elemento del modelo susceptible a ser transformado y llame a otras plantillas que añadan el texto en claro a cada uno de los ficheros de salida según el tipo de documento que pretendemos generar.

Desde mi punto de vista, es muy interesante, ya que la jerarquía que ofrece es similar a la que ofrece java. Gracias a esto se permite la abstracción o herencia, por lo que podremos generar plantillas extendidas a partir de una plantilla base. Además, al tratarse de lenguaje java y ser totalmente compartido, tendríamos la posibilidad de crear plantillas de utilidad, pero también podríamos aplicar clases de java de utilidad que formen parte de nuestra arquitectura. Es destacable también, la gran similitud con Java y la posibilidad que ofrece de trabajar con tipos nativos de java, o en su defecto declarar variables “automáticas” que luego el propio sistema se encarga de transformar al tipo de dato correspondiente.

En general, se trata de una herramienta muy potente e interesante, que de cara al futuro, es posible que se convierta en el principal sistema de transformación de modelo a texto en arquitecturas dirigidas por modelos y sobre todo en aplicaciones basadas en Java, ya que se trata de un lenguaje bastante potente y XTEND2 ofrece la posibilidad de realizar conversiones de modelo a texto, ejecutando Java generado desde el propio sistema Java, obteniendo una eficiencia y un tiempo de ejecución despreciable en comparación con los otros sistemas de transformación expuestos.

En cuanto a la imagen que tendría una plantilla creada en XTEND2, sería algo parecido a esto:

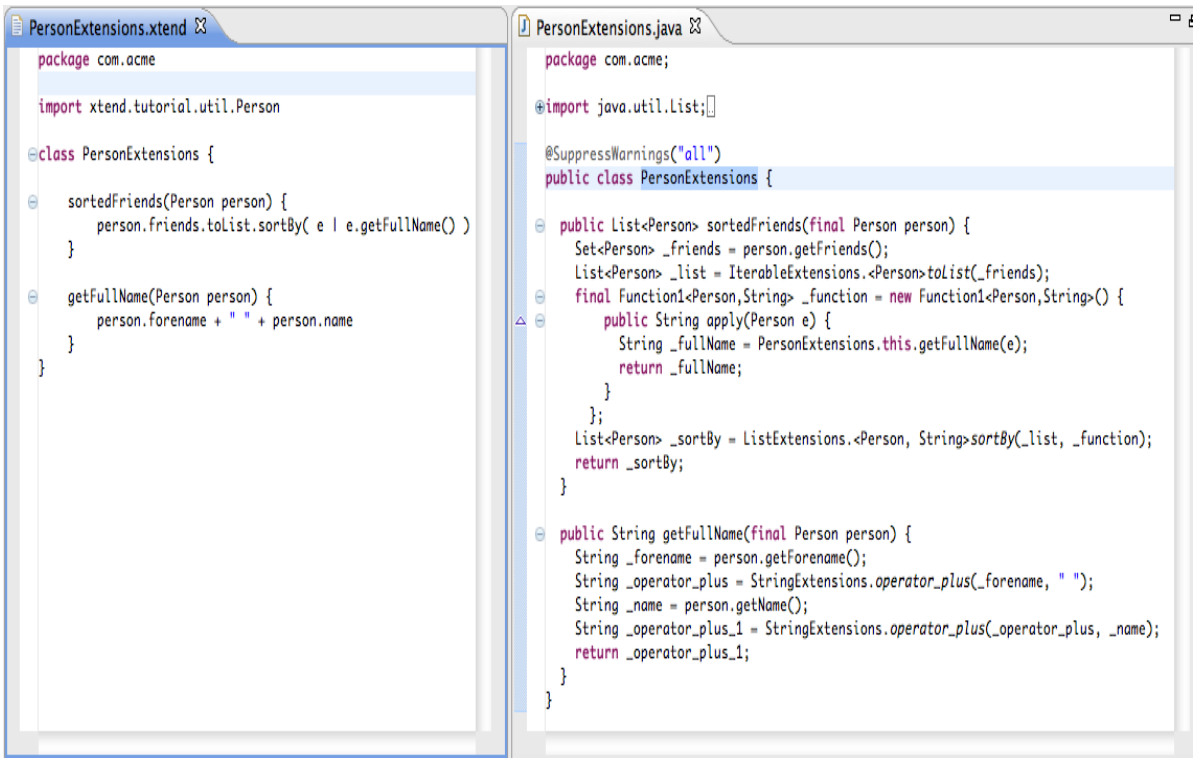


Ilustración 18. Plantilla Xtend y java generado.

En la parte izquierda podemos observar, una plantilla XTEND2. Como se puede ver, es similar a un fichero ***java***. Cuando se crea un fichero ".xtend" se crear un una definición de clase que puede completarse con los métodos o la declaración de variables que desee el usuario. En este caso, la clase PersonExtensions tendría dos métodos. Uno para obtener una lista ordenada de los amigos de una persona (Objeto persona java) y otro para obtener el nombre completo de una persona.

La parte derecha de la imagen se corresponde con el fichero ".java" autogenerado a partir del fichero ".xtend" por el compilador de XBASE. Este fichero, al tratarse de un fichero derivado autogenerado no debería ser manipulado por el usuario, y de ser manipulado debería sacarse el fichero una vez que haya sido modificado del paquete de destino "xtend-gen" en el que se crean los fichero generados, ya que si la plantilla se volviese a modificar y se salvase su contenido, la recompilación automática implicaría una reconstrucción de fichero ***java*** y por tanto la perdida de los cambios introducidos por el usuario manualmente.

3. PROCESO DE DESARROLLO

3.1. Objetivo

Una vez explicados y analizados los procesos MDE y las arquitecturas MDA, nos marcamos como objetivo la aplicación práctica de un sistema dirigido por modelos en un proceso de desarrollo.

La idea principal, consistiría en trabajar sobre ficheros C++ y conseguir realizar algunas transformaciones para obtener ficheros C++11.

Para llevar a cabo esta operación, podríamos utilizar diversos sistemas. Quizás una de las alternativas más rápidas consistiría en crear una aplicación basada en java que se encargase de leer el fichero en plano C++. Una vez leído el fichero C++ podríamos crear una arquitectura de ficheros ***java*** que se encargase de leer el fichero línea por línea y aplicar expresiones regulares para los casos que se deseen transformar y reemplazase el texto C++ por el nuevo texto C++11.

Esta opción, aunque aparentemente pueda resultar sencilla y de la sensación de que pudiese llegar a funcionar, debemos decir que es muy engorrosa, requiere demasiado esfuerzo y presenta muchos inconvenientes. Entre los principales inconvenientes que se nos pueden presentar tenemos los siguientes:

- **Dependemos totalmente del contenido del fichero**, pudiendo estar bien programado o mal programado en el lenguaje de origen.

- **Depende mucho de la programación del fichero**, la forma de declarar las variables, los nombre utilizados… por lo que requeriría contemplar varias opciones a la hora de leer y decidir cómo transformaremos el fichero.

- **El mantenimiento de la aplicación, sería nuestro principal inconveniente**. Mantener una aplicación mal estructurada puede llegar a convertirse en un infierno. En el caso ideal de conseguir que la aplicación funcione para realizar las transformaciones entre lenguajes de programación, si necesitásemos o decidiésemos realizar nuevas transformaciones, tendríamos que seguir añadiendo “parches” a nuestro código. Además, es posible que fuese necesario tocar el código en muchos puntos de nuestra aplicación, por lo que podemos concluir, que sería una aplicación muy difícil de mantener y de evolucionar.

- **Ausencia de reutilización de la arquitectura**. Una vez consigamos crear una transformador entre lenguajes de programación, si no mantiene

una arquitectura muy bien diferencia, que en este caso está claro que no la tiene, la reutilización de la arquitectura es nula. Cuando creamos un sistema de transformación entre lenguajes de programación, deberíamos tener en cuenta qué expectativas de futuro tenemos. El hecho de tener un proceso de desarrollo bien organizado, puede facilitar la reutilización de gran parte de la arquitectura si en un futuro nos planteásemos transformar ese lenguaje en otro nuevo, si deseamos generar documentación, crear ficheros de casos de prueba, entre otras muchas cosas.

Para solucionar estos problemas, lo que se plantea es orientar este transformador de código C++ a C++11 a un proceso dirigido por modelos (MDE). Si planteásemos una arquitectura dirigida por modelos (MDA) para este caso en concreto tendríamos que separar nuestro proceso en tres partes totalmente independientes pero interrelacionadas entre ellas.

Esta separación nos permitiría tener diferenciados los tres elementos claves del proceso MDE: el modelo, el sistema de transformación y la salida.

En nuestro caso aplicado, la separación quedaría de la siguiente manera:

- **Modelo.** En nuestro caso, el modelo a transformar no es un modelo creado por una o varias personas en concreto para representar una realidad; y tampoco será un modelo que requiera refinarse a lo largo del proceso de desarrollo. Se trata de un modelo que representa el contenido de un fichero C++ creado por el usuario, por lo que necesitamos obtener ese modelo de tal manera que podamos ser capaces de analizarlo para poder aplicarle las transformaciones que sean necesarias.

  El modelo es cerrado, y depende totalmente del usuario, ya que la realidad que representará ese modelo se corresponderá con el contenido del fichero C++ creado por este. Teniendo en cuenta esto, el modelo que obtengamos de este fichero es un modelo final, no susceptible al cambio y por tanto no será necesario aplicar transformaciones de modelo a modelo.

  Si por el contrario, nos interesase dejar la veda abierta a que en un futuro el contenido de ese fichero, y por lo tanto esa realidad y ese modelo, pudiese transformarse a otros lenguajes de programación podríamos intentar refinar ese modelo del fichero para abstraerlo hasta el punto de que ese modelo pudiese analizarse independientemente del lenguaje de programación en que estuviese programado y poder

generar así ficheros en otros lenguajes de programación utilizando ese modelo.

- **Sistema de transformación M2T.** Una vez conseguido el objetivo inicial de extraer el modelo del fichero C++, debemos crear un mecanismo que nos permita analizar ese modelo y realizar las transformaciones que nos interesen desde el lenguaje inicial C++ a C++11. Este sistema de transformación M2T, al igual que el extractor del modelo a partir del fichero, es independiente en nuestro proceso de desarrollo, pero necesita del modelo para poder funcionar. El hecho de que se trate de un sistema independiente, nos permite eliminar dos de los problemas que planteábamos anteriormente:

  - *Facilita el mantenimiento de la aplicación:* Si, una vez terminado el transformador "C++2C++11", se necesitase realizar modificaciones sobre los elementos a transformar, ya sea para añadir o para eliminar transformaciones, bastaría con modificar la plantilla de transformaciones únicamente y la aplicación seguiría funcionando con total normalidad. Al extraer e independizar la plantilla de transformaciones del resto de la aplicación, nos permite mayor libertad a la hora de modificar nuestro transformador.

    Esto además, es un valor añadido a la idea de utilizar un proceso MDE para desarrollar esta aplicación, ya que los cambios que requiere modificar los elementos de transformación, serían mínimos y no sería necesario tocar el resto de la arquitectura, ya que no implicaría ni al modelo ni al sistema de generación de ficheros de salida.

  - *Posibilita la reutilización de código:* al separar el modelo de la plantilla de transformaciones, también facilitamos la reutilización de código. Si el día de mañana deseásemos generar otros ficheros partiendo del mismo modelo, solo tendríamos que sustituir una plantilla de transformación por otra.

    Es cierto, que esto requiere la creación de dos plantillas, pero también tenemos que tener en cuenta, que el resto de la arquitectura se mantiene estable y que al crear varias plantillas independientes para cada transformación, también estamos

facilitando que si en un futuro alguna de las plantillas requiere cambios no afectaría para nada al resto.

Desde mi punto de vista, estos factores son algo muy interesante y muy beneficioso que ofrecen los procesos MDE como el que estamos proponiendo. Cuanto más estructurada esté el proceso de desarrollo y más diferenciados los diferentes puntos de la arquitectura menos cambios requerirá la evolución de la herramienta y por tanto más fácil será mantenerla y utilizarla y reutilizarla en el futuro.

- **Generador de ficheros de salida.** Este sería el tercer y último punto de nuestro proceso. Una vez extraído el modelo del fichero de origen y una vez realizado el análisis y transformación del modelo de entrada en base a la plantilla de transformación seleccionada, el resultado debemos mostrarlo en un fichero de texto plano.

  Esta es la función de nuestro de generador de salida, la creación del fichero ".cpp" de salida, y la adición a éste, del código C++11 obtenido tras la transformación. Es conveniente abstraer este sistema de los otros dos anteriores, ya que también es susceptible a sufrir modificaciones.

  En un principio solo estamos interesados en obtener un fichero ".cpp" con el código C++11 fruto de la transformación, pero puede que en un futuro queramos crear además ficheros de documentación o puede que queramos aprovechar la arquitectura para crear transformaciones del fichero de origen a otros lenguajes de programación. Si este caso se diera, podría resultar interesante tener un ente encargado de realizar la gestión de generación de ficheros en cada uno de los casos.

  Además, para la creación de ficheros actualmente en java, al igual que en los dos puntos anteriores, existen varias maneras de hacerlo. Puede que en un principio, nos interese utilizar una forma en concreto pero también puede que en un futuro pueda interesarnos modificar nuestro código para utilizar otro mecanismo de generación de ficheros por temas de mejora de rendimiento, facilitar el mantenimiento, mejorar la visión de la creación de ficheros a los usuarios… Si llegásemos a encontrarnos en este punto, sería interesante poder modificar nuestra arquitectura de forma rápida y limpia sin afectar al resto del proceso de desarrollo.

Basándonos en esta arquitectura del sistema transformador "C++2C++11", la representación gráfica de lo que queremos construir quedaría de la siguiente manera:

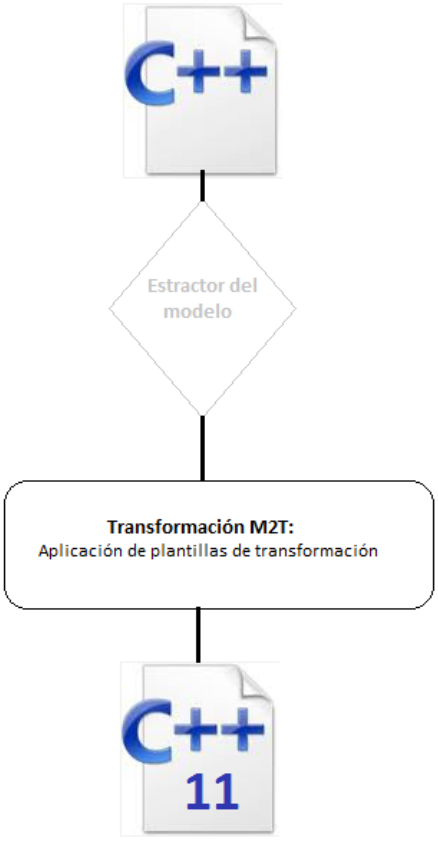


Ilustración 19. Funcionamiento del refactorizador de código C++ a C++11.

3.2. Planificación del trabajo

Este punto del documento detallará la el tiempo dedicado a cada una de las fases del proyecto que se ha desarrollado.

Mostrará las tareas que el equipo de desarrollo tiene que realizar y el tiempo estimado para completar cada una.

TAREA	SUBTAREAS	DESCRIPCIÓN
Investigación a fondo de entorno de trabajo eclipse		Investigación más a fondo sobre el IDE de programación de eclipse, sus principales virtudes, sus funcionalidades y sus posibilidades de mejora.
Investigación sobre creación de eclipse plug-in.		Investigación sobre la creación de complementos para eclipse y la posibilidad

		de contribuir a las funcionalidades que ofrece el IDE de desarrollo sobre el que se trabajará durante todo el proceso de desarrollo.
Investigación sobre contribución de acciones utilizando eclipse.		Investigación sobre la reutilización de las librerías de eclipse y la posibilidad de crear nuevas acciones que nos permitan realizar pruebas con nuestro refactorizador de código.
Investigación más a fondo sobre C++03		Proceso de investigación sobre C++03, sus librerías, su historia, sus principales virtudes y sus principales inconvenientes.
Investigación sobre nuevas actualizaciones de C++11 con respecto a C++		Proceso de investigación para conocer las nuevas librerías ofrecidas por C++11 y que elementos del anterior C++03 reemplazan.
Investigación sobre sistemas MDE y arquitecturas MDA		Se habré un proceso de investigación sobre la ingeniería dirigida por modelos y sobre las arquitecturas dirigidas por modelos. Debemos saber que son, en qué consisten y qué ventajas ofrecen con respecto al desarrollo no modelado.
Investigación sobre diferentes aplicaciones de Sistemas MDE y arquitecturas MDA		Investigación sobre casos concretos en los que se apliquen procesos MDE y si utilizan o no transformaciones entre modelos (M2M) y transformaciones de modelo a texto (M2T)
Investigación sobre herramientas de transformación M2M	• ATL • QVT Transform	Investigación sobre diferentes herramientas de transformación entre modelos. Estudiar la necesidad de utilizar

		transformaciones entre modelos en nuestro proyecto.
Investigación sobre herramientas de transformación M2T	• ACCELEO • XPAND/XTEND • XTEND2	Investigación sobre las diferentes herramientas para realizar transformaciones de modelo a texto. Sabemos que será necesario implementarlas en nuestro sistema. Elegir la mejor opción e indicar por qué lo es.
Información sobre los diferentes tipos de modelo.		Investigación sobre diferentes modelos y cuáles de ellos tiene librerías que pueden integrarse con nuestro entorno de desarrollo Eclipse.
Investigación de herramientas de extracción de modelos aplicables a eclipse.		Proceso de investigación sobre herramientas que permitan obtener modelos a partir de fichero de diferentes lenguajes de programación. Seleccionar las herramientas aplicables a nuestro sistema de refactorización de C++ a C++11. Buscamos extractores de modelos C++.
Investigación sobre arboles AST extraíbles desde CDT.		Investigación sobre la posibilidad de extraer un árbol AST de un fichero C++ utilizando el complemento de eclipse CDT. Comprobar si puede servirnos como modelo de entrada.
Implementación de Acciones de refactorización del menú "popUp".	• Compilar • Compilar en.. • Comparar • Añadir naturaleza	Implementación de las acciones contribuyendo al menú secundario de los ficheros del árbol de navegación de eclipse que nos interesen, los ficheros

		C++ y los proyectos C++.
Implementación de acciones de refactorización de la barra de menú de eclipse.	• Compilar • Compilar en… • Comparar	Implementación de las acciones contribuyendo al menú "toolbar" de eclipse, para permitir realizar las funciones de refactorización sobre los ficheros que están abiertos en el editor y son de tipo C++.
Implementación del sistema de transformación de modelo a texto.		Proceso de implementación del sistema de transformación de modelo a texto que permita realizar refactorizaciones de C++ a C++11 utilizando las acciones implementadas sobre eclipse.
Redacción de la memoria		Redacción de la memoria explicando el proyecto realizado, el proceso de investigación seguido y las conclusiones obtenidas, y explicación del sistema MDE implementado para refactorizar código C++ a C++11.

El diagrama de Gantt que representa el desarrollo de la aplicación en base a las tareas anteriormente especificadas es el siguiente

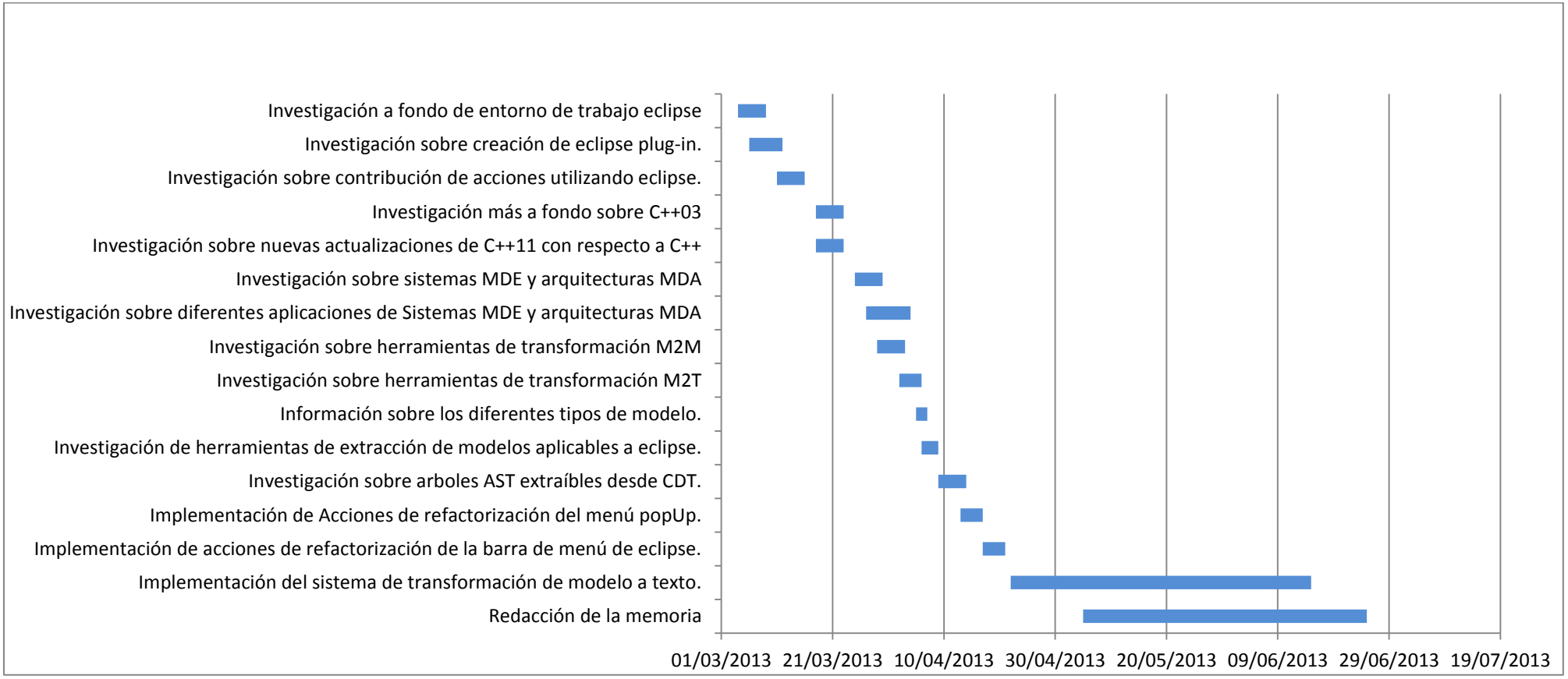


Ilustración 20. Diagrama de Gantt asociado al desarrollo del proyecto.

3.3. Arquitectura MDA seleccionada

En este punto del documento, basándonos en la arquitectura MDA anteriormente expuesta, contaremos que elementos hemos utilizado para la extracción del modelo del fichero C++, cual es la herramienta seleccionada para crear y aplicar las reglas de transformación y en que nos basaremos para la generación de los ficheros de salida con el contenido C++11 transformado.

3.3.1. Analizadores de modelo de ficheros C++

La primera parte de nuestra arquitectura, consistiría en crear una aplicación basada en java que nos permitiese obtener el modelo de un fichero C++ para poder analizarlo y transformarlo.

Como ya se ha comentado anteriormente, existen diferentes mecanismos para analizar modelos C++, y deberíamos seleccionar el que más nos convenga para utilizar en el transformador de código. Deberíamos utilizar, aquel que nos permita manejar todos los elementos que nos interesa transformar, pero que también nos permita un margen de maniobrabilidad de cara al futuro.

Si seleccionamos un mecanismo que nos limite algunas funciones, si el día de mañana deseamos evolucionar la aplicación y analizar otros elementos de un fichero C++ quizás tengamos problemas por este mecanismo y el trabajo de modificación implicaría grandes cambios. Por esto, debemos seleccionar una herramienta manejable, potente y que además sea relativamente sencilla de manejar.

Actualmente existen herramientas que permiten realizar un análisis del código fuente de un fichero con código fuente y crear un árbol sintáctico del contenido. Este árbol sintáctico, será el modelo de entrada que se pretende transformar. Entre las herramientas disponibles para analizar ficheros C/C++, se encuentran las siguientes:

- **CParser.** CParser es una biblioteca escrita en lenguaje C++ que permite realizar diferentes operaciones con código fuente escrito en C y C++. CParser permite realizar las siguientes acciones.
  - Construir un analizador de código.
  - Añadir criterios para buscar determinada semántica en los ficheros.
  - Reiniciar el analizador.
  - Buscar byte a byte en el código fuente e ir comparando con el criterio a buscar.

Para realizar el análisis del código es necesario proporcionar uno o más criterios de búsqueda.

Para crear los criterios, bastará con crear un elemento dentro del código que esté formado por un número identificativo. El analizador buscará dentro del código los criterios marcados y se podrá realizar una llamada a una función definida por el usuario para llevar a cabo las operaciones deseadas.

- **PYCParser.** PYCParser es un analizador escrito en Python del lenguaje C. Se ha diseñado en módulos pequeños, permitiendo integrarlo de forma sencilla dentro de otros programas que necesiten analizar código C en algún momento de su ejecución.

  Se genera un árbol sintáctico abstracto (AST) completo a partir del código C, que puede utilizarse como "front-end" para un compilador C, o como analizador de código estático de ficheros C++.

  PYCParser utiliza PLY para construir el léxico y el analizador de lenguaje C, pero no es compatible con C89.

  Se distribuye, como ya hemos dicho como pequeños modulitos, ya que según su creador, ofrece diferentes funcionalidades que pueden interesar a los desarrolladores y al estar escrito en lenguaje python, debería resultar relativamente sencillo para éstos, analizar el código y añadirle las funcionalidades que requieran partiendo de una base.

- **AST visitor.** Los árboles de sintaxis abstracta son representaciones en forma de árbol de la sintaxis estructural del código fuente de un fichero escrito en un lenguaje de programación. Cada uno de los nodos del árbol hace referencia a una sentencia del código fuente.

  La sintaxis abstracta indica que no se trata de una reprsentación detallada del código real que representa. Con esta sintaxis podremos tener una visión general del flujo de ejecución de los programas escritos en diferentes lenguajes de programación, de los elementos que contiene el código fuente y las relaciones entre ellos.

  Una utilidad común de los árboles AST es obtener poder realizar un análisis del código fuente de un fichero para buscar patrones de búsqueda concreto que interesen al desarrollador.

En nuestro sistema refactorizador trataremos de localizar estos patrones de búsqueda para analizarlos y realizar las transformaciones a C++11 que nos interesen en cada caso.

Las Imágenes 21 y 22 muestra árboles AST referentes a diferentes elementos contenidos en un código fuente determinado:

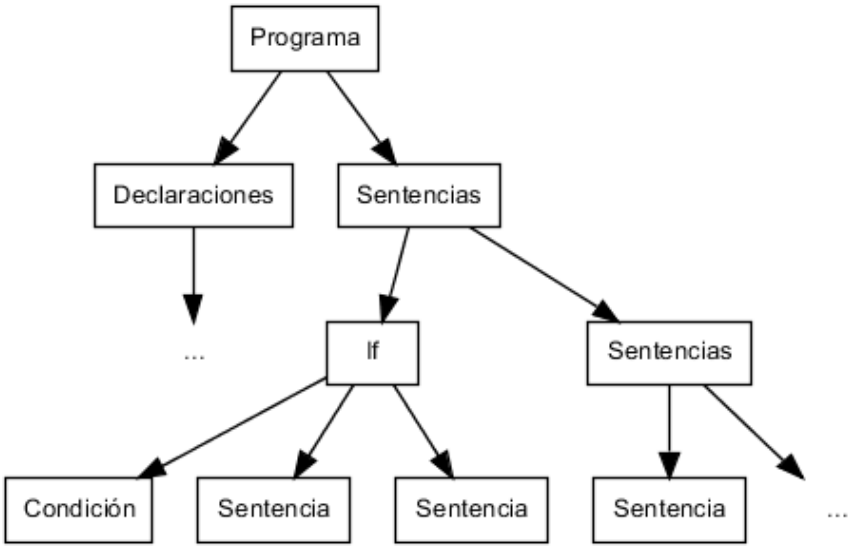


Ilustración 21. Ejemplo de árbol AST Asociado a un proceso.

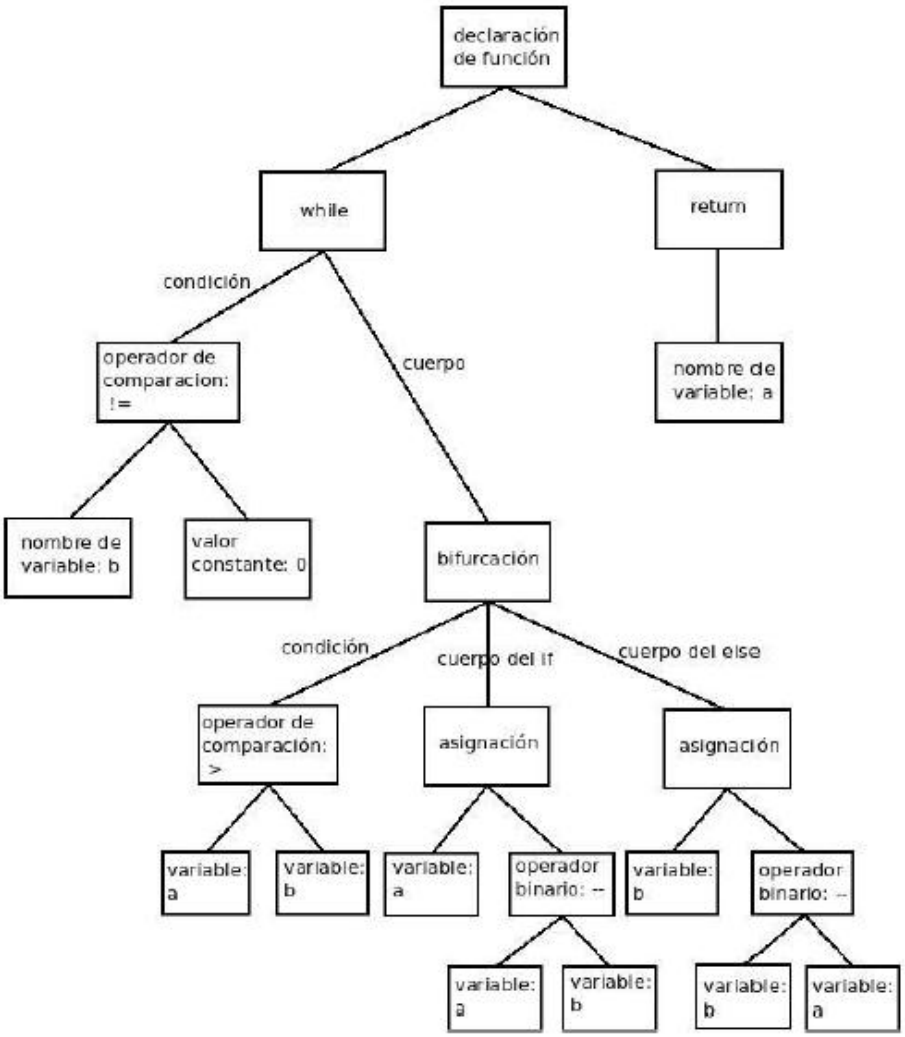


Ilustración 22. Ejemplo 2 de árbol AST asociado al proceso de ejecución de un programa.

- **CDT Plugin:** El plugin CDT nos ofrece un entorno completo y funcional de desarrollo integrado en Eclipse para trabajar con los lenguajes de programación C/C++.

  Algunas de las posibilidades que nos ofrecen son la creación y gestión de proyecto y ficheros C/C++.

  Además, ofrece herramientas de compilación, ejecución y construcción de código fuente, árboles de navegación sobre el contenido de ficheros, coloreo de sintaxis, validaciones de código, etc.

  Uno de los elementos que más nos interesa y que nos ofrece el complemento de eclipse CDT es la posibilidad de extraer el árbol AST de un fichero C++ perteneciente a un proyecto determinado.

Gracias a esto, decidimos utilizar este complemento de eclipse para obtener el modelo de los ficheros que queremos refactorizar.

Una vez extraído el árbol AST de un fichero C++, la plantilla de transformaciones podrá analizar los elementos del árbol y aplicar las refactorizaciones que tenga definidas, para generar posteriormente el fichero C++ con el contenido C++11 resultante del proceso.

3.3.2. Sistema de transformación M2T seleccionado

En puntos anteriores, propusimos varias alternativas a aplicar en un proceso MDE o en una arquitectura MDA para realizar unas transformación de modelo a texto.

Cualquiera de estas alternativas sería viable y una buena opción a aplicar cuando pretendamos obtener texto en plano a partir de un modelo, pero debemos decantarnos por una.

Puesto que nuestro entorno de desarrollo es eclipse, totalmente compatible con la JDT, y el lenguaje de programación seleccionado para desarrollar la aplicación es JAVA, deberíamos decantarnos por la opción que más nos favorezca como desarrolladores y que más beneficie la simplicidad del desarrollo de la aplicación ofreciendo el mejor rendimiento, o al menos uno bastante bueno.

Es interesante que la aplicación ofrezca un buen rendimiento, ya que sí el tiempo de compilación y transformación es demasiado elevado, el sistema pierde todo el sentido, porque el usuario tardaría demasiado tiempo en realizar sus transformaciones de lenguaje.

Si queremos un buen rendimiento, la opción de XPAND/XTEND queda totalmente descartada, ya que al tratarse de un lenguaje interpretado es bastante lento en su analizador y transformador de código y esto supondría tiempos de compilación y transformación demasiado elevados para lo que estamos buscando.

Cualquiera de las dos opciones restantes, ACCELEO y XTEND2 serían opciones acertadas con respecto al rendimiento, ya que suponen un tiempo de ejecución relativamente corto en ambos casos, pero ACCELEO supone mayores dificultades con las transformaciones de modelos. Como hemos contando anteriormente, ACCELEO provee un listado de posibles modelos con los que es compatible y con los que nos permite trabajar de forma rápida en sus plantillas de transformación. Es un sistema muy interesante para transformar a texto modelos UML, EMF, XMI y JAVAXMI (este último ofrecido por moDisco – generador de modelos a partir de código java).

Cualquiera de estos modelos, entre los muchos que permite analizar y transformar serían relativamente rápidos y sencillos de llevar a texto en plano utilizando ACCELEO, pero en nuestro caso, al utilizar el Árbol sintáctico obtenido de una clase C++, proporcionado por el "plug-in" CDT de eclipse, no es tan sencillo y directo de transformar como en los modelos anteriormente citados.

Debido a esto, y a la versatilidad que ofrece XTEND2, ya que permite generar texto plano a partir de modelos o a partir de cualquier elemento que se le quiera pasar y analizar; y puesto que nuestro sistema está implementado en JAVA, siendo totalmente compatibles; decidimos utilizarlo.

XTEND2, es la mejor de las opciones que podemos elegir. Al estar basado en JAVA y utilizar este lenguaje en nuestro sistema, nuestro árbol sintáctico de los ficheros a analizar estará compuesto de objetos JAVA. Debemos recordar, que XTEND2 nos ofrece la posibilidad de utilizar sus tipos de datos propios, dejando la libertad al desarrollador de poder o no utilizar tipos de datos nativos de JAVA y objetos creados por él.

Además, lo más importante de todo, es que nuestras plantillas ".xtend", al basarse XTEND2 en XBASE, se compilan de forma automática, generando ficheros java con el contenido de las plantillas.

Esto favorece en un 100% el rendimiento de la aplicación, ya que no estamos utilizando un sistema externo para interpretar un modelo, analizarlo y aplicarle las transformaciones que se requieran, sino, que lo que realmente haremos, será pasar nuestro árbol sintáctico como parámetro al método generador de texto definido en nuestra plantilla.

Este método generador, será un método java autogenerado por XTEND2, lo que supondrá que estaremos ejecutando código JAVA sobre código JAVA, que a su vez supondrá un tiempo de ejecución mínimo.

Cuantos menos cambios de lenguajes se utilicen en la aplicación, menos tiempo de transformaciones de lenguaje requerirá el sistema, y por tanto ofrecerá un mayor rendimiento.

De esta manera, podemos concluir, que para nuestro sistema, la herramienta de transformación de modelo a texto que más nos conviene es XTEND2, por facilidad de implementación, por versatilidad y compatibilidad con los modelos, y sobre todo por rendimiento que ofrece con frente a los otros sistemas.

3.3.3. Generador de fichero de salida

El último paso en una transformación modelo a texto es la obtención de texto en plano a partir del modelo y decidir que deseamos hacer con ello.

En el sistema que estamos trabajando, está claro que se debe hacer con el texto de salida del proceso de transformación. Debemos llevar todo ese contenido correspondiente a código fuente en C++11 equivalente al contenido inicial C++ a un fichero de texto ".cpp".

Para crear este fichero de texto, podremos recurrir a dos utilidades de eclipse:

- La primera opción que se nos presenta es utilizar la librería "java.util.*" de eclipse que nos permita en primer lugar crear un fichero con extensión ".cpp" en la ruta de destino que decidamos.

  Una vez creado el fichero de salida, bastaría con escribir sobre el fichero el contenido transformado y cerrar los descriptores de escritura del fichero una vez se haya escrito todo el contenido.

  Esta metodología es la que suele utilizarse cuando programando JAVA, se desea en un momento dado, crear un fichero y escribir sobre él, o cuando se desea tener acceso al contenido de un fichero.

  La mecánica es fácil de utilizar y fácil de programar. En nuestro caso, sería interesante y sencillo de incorporar a la aplicación, pero al utilizarlo, se nos plantea un problema. La creación del fichero se hacer de forma independiente al entorno, por lo que eclipse no recibe notificaciones de que deber refrescarse para mostrar los nuevos ficheros creados.

  Esto plantea el problema de que cada vez que transformemos código, el usuario de forma manual deberá refrescar el entorno de trabajo para mostrar las nuevas rutas, o que de forma programática extendamos el sistema de refresco de eclipse y lo lancemos una vez se hayan creado los ficheros con el código C++11. Esta última opción, requiere más trabajo y desde un punto de vista de calidad y rendimiento, es poco "viable", ya que no sería más que un "parche" que pondríamos para solucionar este problema de refrescos que podría solucionarse utilizando otras librerías.

- La segunda opción aparece tras programar y probar la primera opción y ver que no es viable debido a los problemas de refresco anteriormente

comentados. Investigando un poco sobre los mecanismos de creación de ficheros, se decide utilizar la clase IFile de "org.eclipse.core.resources". Si utilizamos la gestión de recursos que nos ofrece eclipse para la creación de ficheros, obtendremos bastantes ventajas.

Quizá, se trate de un mecanismo, menos conocido o algo más complejo de implementar que la librería de "java.util.*", pero también es más eficiente y permite sincronización y otras funcionalidades. La creación o acceso a recursos utilizando "org.eclipse.core.resources" es dependiente del dominio de edición, por lo que, por ejemplo, si tenemos un editor abierto, podríamos acceder al fichero que está abierto en ese editor y obtendríamos un IFile, que no es más que un recurso de eclipse de tipo fichero.

Entre las ventajas que nos ofrece con respecto al sistema expuesto anteriormente, creo que las más importantes son dos, la impresión visual que se le da a usuario y la eficiencia y eficacia del sistema.

En primer lugar, evitamos el problema de refresco que se planteaba antes. Al trabajar con recursos de eclipse y trabajar sobre el dominio de edición, la creación de un fichero conlleva el posterior refresco del árbol de navegación por lo que siempre aparecen los ficheros y no es necesario refrescar manualmente o implementar "parches" que solucionen los problemas de refresco que se presentaban anteriormente.

Además, el utilizar recursos propios de eclipse, permite utilizar el sistema de monitorización de progreso que ofrece el entorno. Esto, es algo visual para el usuario, pero creo que es interesante con respecto al diseño de la aplicación y al nivel de usabilidad de esta, ya que cuando eclipse está creando los ficheros de salida, nos muestra un dialogo de proceso por el que podemos saber en qué estado se encuentra el proceso de transformación en todo momento y el usuario podrá hacerse una idea del tiempo que queda para que termine de realizar las transformaciones de ficheros.

Para transformaciones individuales, esto suele ser imperceptible ya que es un proceso muy rápido, pero si por ejemplo queremos transformar un gran número de ficheros, podremos saber el estado y el punto en el que se encuentra el proceso.

Dicho esto, es obvio, que la mejor opción es la segunda que se propone porque pese a ser algo más complejo de programar y utilizar, por motivos de desconocimiento, resulta ser más rápido, más eficiente y permite la monitorización del progreso de proceso de transformación.

Aun así, en nuestro proyecto utilizaremos ambos sistemas. El primero para la creación de ficheros en directorios externos al espacio de trabajo de eclipse y el segundo para la creación automática de ficheros dentro del espacio de trabajo que ofrece el entorno.

3.4. Funcionamiento de la herramienta.

Este punto del documento, consiste en una explicación a alto nivel del funcionamiento de la herramienta y de las funcionalidades que nos ofrece el transformador de C++ a C++11.

Como se ha comentado en los puntos anteriores, el sistema está divido en tres partes diferencias.

La primera parte consiste en obtener un modelo estático de un fichero C++ con el que podamos trabajar durante el resto del proceso de refactorización. Para conseguir esto, se utiliza el árbol abstracto que nos ofrece el "plug-in" de eclipse CDT.

Si somos capaces de obtener el árbol abstracto de un fichero C++, obtendremos las relaciones padre/hijo de cada uno de los elementos del fichero. De esta manera, una vez obtenido el árbol AST, se recorren cada uno de los nodos que lo forman y sus hijos analizando los elementos para comprar si requieren o no requieren transformaciones.

Una vez obtenido el árbol AST con todo el contenido del fichero, entramos en la segunda parte del sistema, el analizador y transformador de modelo a texto. Existían dos posibilidades para analizar cada uno de los elementos del árbol abstracto de contenido. La primera opción era analizarlo en la parte java y en caso de necesitar transformación, solicitarle a la plantilla de transformaciones que hiciese el cambio correspondiente. La segunda opción que se plantea, es pasarle a la plantilla de transformación el árbol completo de nodos y que fuese está quien se encargase de analizar los nodos y aplicarle las transformaciones que fuesen necesarias.

Puesto que lo que se busca con los procesos MDE, es diferenciar del resto cada una de las partes del proceso, se decide utilizar la segunda opción. Esta opción nos permite dotar a la plantilla de transformación de todo el proceso de análisis y transformación de código. Al asignarle toda la funcionalidad a la plantilla liberamos el proceso de recuperación del modelo del fichero y solo le pasamos a la plantilla los elementos que

nos interesan que son las declaraciones de variables, las estructuras y los métodos de las clases C++.

Puesto que es la plantilla quien recorre todos los nodos hijo y va analizándolos y viendo si necesitan transformación, si en un futuro quisiésemos analizar nuevos elementos del árbol o quisiésemos eliminar o modificar las transformaciones existentes, bastaría con modificar únicamente el fichero plantilla, reduciendo al mínimo el número de cambios. Si por el contrario, optásemos por la primera opción, tendríamos el problema de tener que estar cambiando la parte JAVA que se encarga de analizar los elementos y solicitar las transformaciones; y además, tendríamos que cambiar la plantilla para añadirle las nuevas transformaciones que queramos hacerle. Requiere más cambios que la segunda opción, y limita la independencia entre el modelo y el transformador.

La plantilla de transformación recibirá todos los elementos que podemos encontrar a primer nivel y que nos interesan las declaraciones de variables, las estructuras y las funciones de una clase C++. Una vez recibe el nodo a analizar, comprueba de qué tipo de nodo se trata y comprueba si se le puede aplicar algún tipo de transformación o debemos dejar el contenido como está. Si es posible aplicarle alguna transformación, se transforma. Si no es posible aplicar transformaciones, se mantiene como está. Posteriormente se comprueba si tiene hijos. Si tiene hijos, se llama de nuevo al transformador con cada uno de sus hijos de manera que se vuelve a comprobar de forma recursiva si cada uno de los nodos hijos del modelo es susceptible a ser transformador o no.

Cuando la plantilla termina de analizar todos los nodos del árbol y ha terminado de aplicar todas las transformaciones que pueda realizar, devuelve el contenido en texto plano correspondiente al código C++11 equivalente al código C++ de entrada.

Este contenido devuelto como una cadena de texto, entraría en la tercera parte del sistema, el manejador de ficheros.

Este manejador de ficheros será el encargado de recibir el contenido transformado del fichero de salida y la ruta de destino dónde generar el fichero. Dependiendo de la opción de refactorización que elijamos la ruta de destino se verá modificada.

Si se selecciona la opción “Compilar”, se realizará el proceso de transformación y el manejador de ficheros se encargará de generar una carpeta sobre el proyecto C++ con el nombre “cpp11-gen” y dentro de ella almacenará todos los ficheros transformados con la nomenclatura **<nombre_fichero_c++>.cpp.**

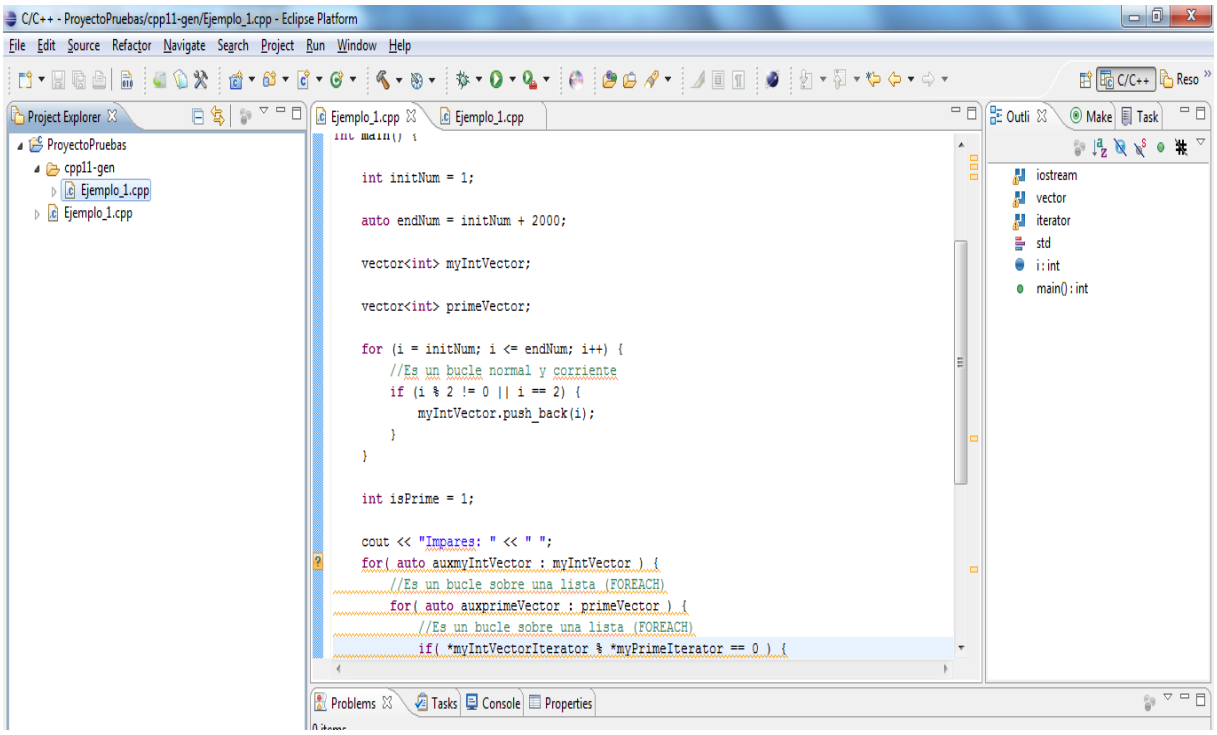


Ilustración 23. Ejemplo del resultado de realizar una compilación de un fichero C++. Se genera una carpeta "cpp11-gen" dónde se almacenan los ficheros C++11 fruto del proceso de refactorización.

Si por el contrario, se selecciona la opción "Compilar en…", el sistema analizará y transformará todos los ficheros C++ seleccionados, y generará el listado de ficheros transformados en la ruta que el usuario introduzca en la ventana emergente que el sistema le mostrará. Esta ruta puede formar parte del mismo entorno de trabajo en el que estamos trabajando, o podría corresponderse con cualquier otra ruta del equipo en el que se trabaja, generándose el código en la carpeta y ruta introducidas por el usuario.

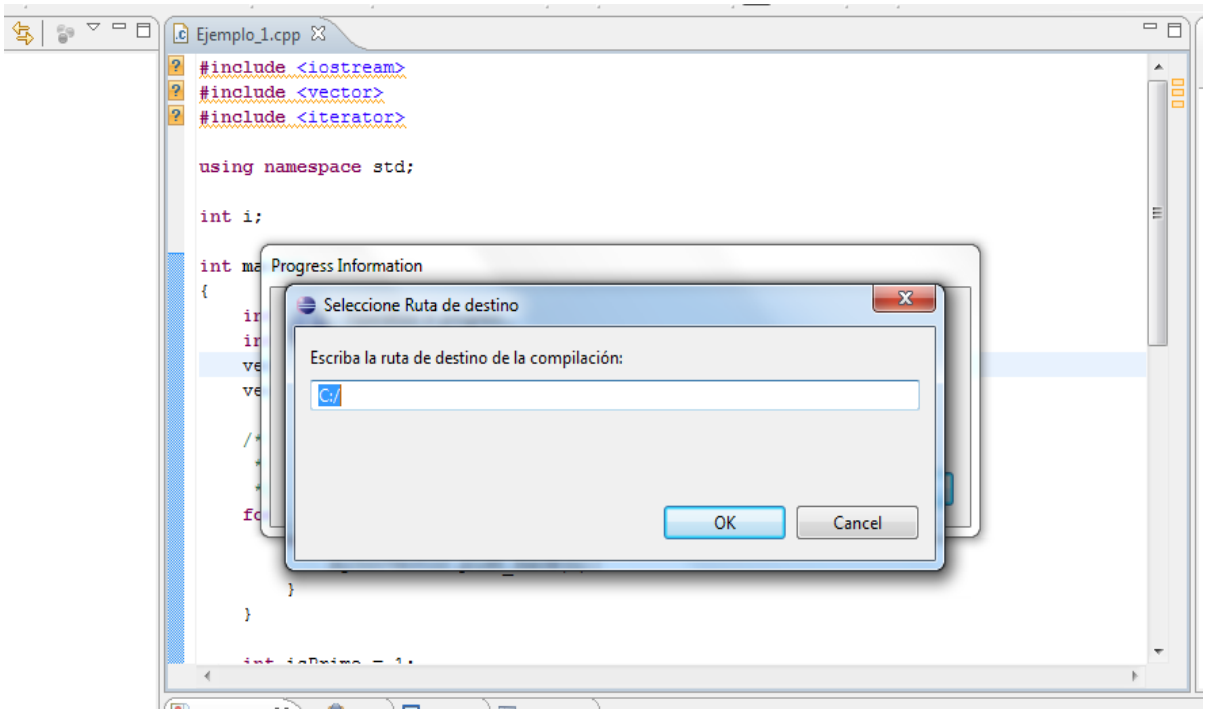


Ilustración 24. Se solicita al usuario la ruta de destino de los ficheros C++11 resultantes de la compilación.

La tercera opción que se plantea al manejador de ficheros, es que el proyecto C++ esté marcado como compilador automático. Si el proyecto contiene la naturaleza "Cpp11Nature", cada vez que se realicen cambios sobre un fichero y se salve el editor, se lanzará una delta de cambio que avisará a nuestro sistema de que deber recompilar el fichero. De forma automática, se lanzará por detrás el proceso de transformación. Se generará un modelo AST del fichero abierto en el editor, se pasará el modelo al transformador para que realice los cambios que sean necesarios y por último el manejador de ficheros añadirá el fichero autogenerado o modificará el contenido del fichero en caso de que ya exista y lo dejará en la carpeta **cpp11-gen** del proyecto C++ al que pertenece el fichero de origen modificado.

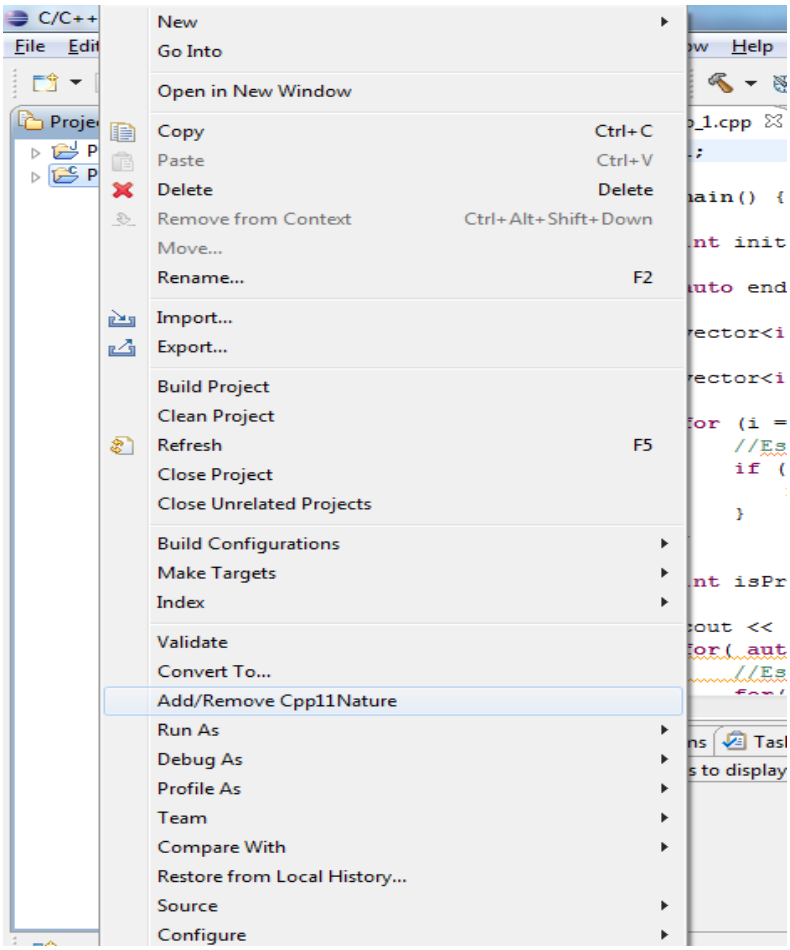


Ilustración 25. Proceso a seguir para marcar un proyecto C++11 con la naturaleza que permita realizar el proceso de refactorización de forma automática.

Por último, tenemos la opción de que el usuario analice los cambios que podría realizar y decidir si llevarlos a cambio o por el contrario no hacerlo. Para esto, si se selecciona la operación “Comparar”, se lanzarán en primera instancia las dos primeras fases del sistema. Se obtendrá el modelo del que está abierto en el editor en ese momento y se realizarán las transformaciones necesarias. Una vez analizado el contenido del fichero, este contenido se maqueta y se muestra en una ventana comparadora frente al código fuente de origen.

En esta ventana comparadora, el usuario puede decidir si llevarse los cambios C++11 al fichero C++ o no. Una vez terminado, el proceso de comparación se genera el fichero con los cambios aplicados por el usuario en la carpeta **cpp11-gen** en el mismo proyecto al que pertenece el fichero C++ que se está analizando. El fichero C++ de origen, mantendrá su código original, mientras que el fichero autogenerado, contendrá el texto con los cambios aplicados por el usuario sobre el texto de origen, con los cambios del código transformado a C++11.

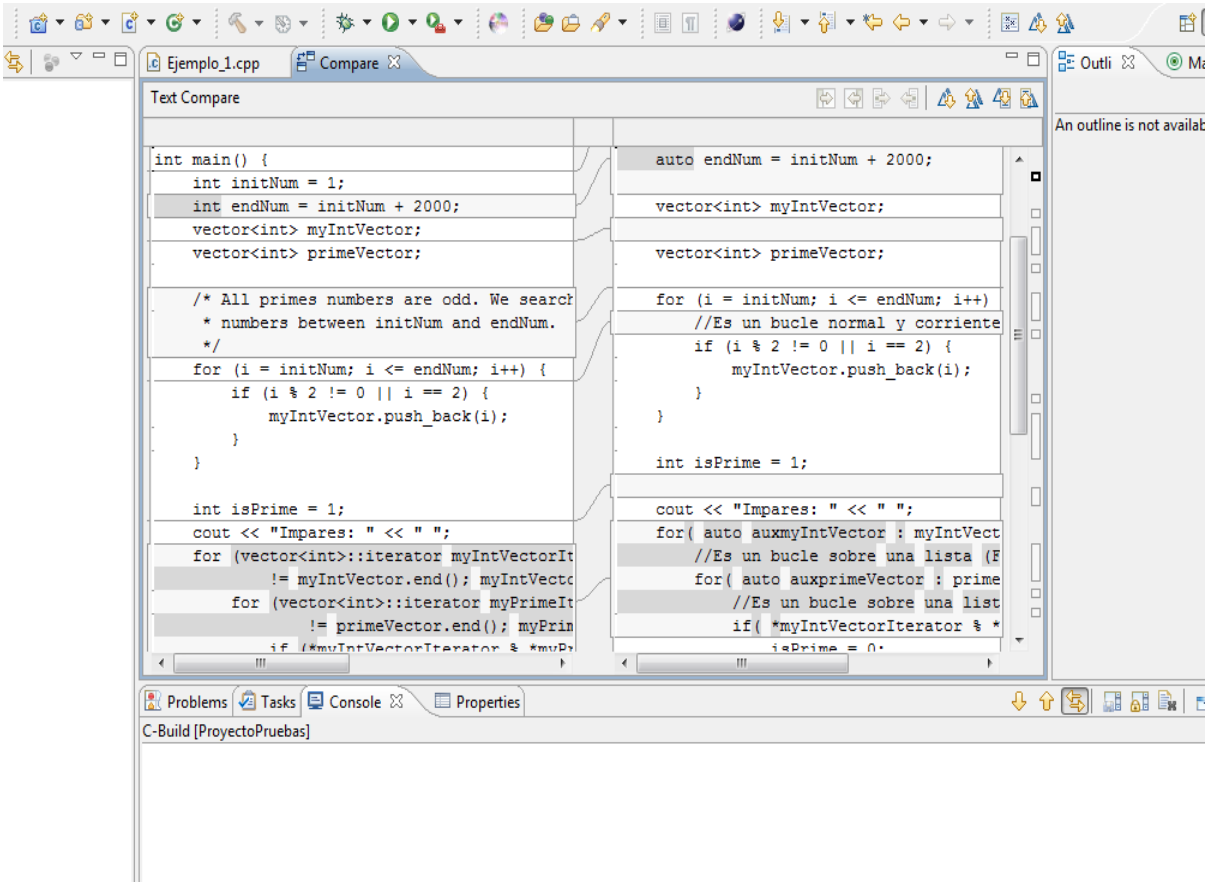


Ilustración 26. Editor de comparación entre el fichero de entrada C++03 y el fichero resultante C++11.

3.4.1. Acciones del sistema de refactorización de código.

Como se ha comentado anteriormente, el sistema de refactorización de código ofrece varias operaciones para obtener código C++11 a partir del código inicial C++03.

Estas acciones estarán accesibles desde varios puntos. Una posibilidad es utilizar la barra de navegación de la parte superior de eclipse dónde se mostrarán las diferentes posibilidades de iteración del sistema, o podremos hacer clic con el botón secundario sobre un proyecto del espacio de trabajo de eclipse y elegir las distintas operaciones desde las submenús "Refactorizar" o "Añadir naturaleza **Cpp11Nature**" al proyecto para que se generen automáticamente los ficheros C++11 al salvar un fichero de tipo C++.

Compilar

Al seleccionar esta operación se generará en la ruta predefinida, **<nombre_proyecto>/cpp11-gen/<nombre_fichero>.cpp**, un fichero con el contenido C++11 correspondiente a la transformación del código inicial C++

Esta operación no permite al usuario la posibilidad de elegir una ruta de destino, sino que se entiende que el usuario quiere seguir trabajando sobre el mismo proyecto en el que se está creando código C++.

Si no es así, y lo que se quiere es exportar ese contenido a una ruta del sistema fuera del entorno de trabajo deberíamos seleccionar la siguiente operación.

Esta acción de refactorización será accesible desde dos puntos:

- Botón Secundario -> Refactorizar -> Compilar
- Barra de navegación superior -> Compilar

Compilar en…

Al seleccionar esta operación, se lanzará una venta emergente para que el usuario seleccione la ruta del sistema dónde deben generarse los ficheros compilados. Esta ruta debe ser absoluta, del tipo **"C:/<folder_1>/<folder_2>/<nombre_fichero>.cpp"**. El nombre de los ficheros se generará automáticamente en base al nombre de fichero de origen, siendo del tipo **<nombre_del_fichero>.cpp.** Si la ruta no cumple con los, requisitos establecidos no podrá continuarse con la operación.

Una vez se ha introducido la ruta de destino de los ficheros compilados y se pulsa el botón aceptar, se lanza el proceso de transformación de los ficheros seleccionados y se generan en la ruta de destino introducida por el usuario los ficheros con el contenido inicial transformado.

Esta acción de refactorización será accesible desde dos puntos:

- Botón Secundario -> Refactorizar -> Compilar en…
- Barra de navegación superior -> Compilar en…

Comparar

Lo que se pretende con esta acción, es que cuando el usuario la seleccione, se le pueda mostrar la información del contenido inicial que puede transformarse y de cómo quedaría el fichero una vez transformado.

Para ello, cuando se lanza la acción, el proceso de transformación varía un poquito. En primer lugar se crean dos contenedores virtuales para código fuente. El primer contenedor guarda la información relativa al contenido del fichero. En cuanto al segundo contenedor, guarda la información correspondiente a la transformación del modelo del fichero C++ que se está analizando.

Es decir, el proceso que se lleva a cabo es prácticamente el mismo que en los casos anteriores, pero sin llevarnos el contenido directamente a un fichero de texto plano. En este caso, se guarda el contenido transformado y se muestra en un diálogo de comparación de eclipse frente al contenido del fichero original.

Una vez el contenido inicial y el final se encuentran enfrentados, podremos llevarnos los cambios de un lado al otro del fichero.

Al pulsar el botón **Aceptar**, los cambios que se hayan realizado en el cuadro comparador se llevarán a un fichero ".cpp" en la ruta de ficheros autogenerados sobre el propio proyecto que se está analizando **<nombre_proyecto>/Cpp-gen/<nombre_fichero>.cpp**

Esta acción de refactorización será accesible desde dos puntos:

- Botón Secundario -> Refactorizar -> Compilar
- Barra de navegación superior -> Compilar

Compilador automático

Esta última opción consiste en quitarle al usuario el trabajo que ocasiona tener que estar compilando de forma manual todos los ficheros C++ que vaya construyendo.

Para quitarle al usuario este trabajo, suponiendo el caso en el que éste decida querer transformar todos los ficheros C++ de un proyecto C; se ofrece la posibilidad de crear una especie de compilador automático.

Esta especia de compilación automática, consiste en añadir al proyecto o proyectos C++ con los que se está trabajando una nueva naturaleza, la naturaleza **"Cpp11Nature"**. El hecho de añadir al "classpath" de un proyecto una nueva naturaleza, nos permite crear un compilador propio para todos los ficheros que formen parte de este proyecto.

De esta manera, cada que un fichero C++ se ve modificado, se lanza una delta de cambio que nos alerta de que un fichero de ese proyecto está siendo modificado. Estas deltas de cambio pueden analizarse e interpretarse, de forma que, si lo que recibimos es una delta de cambio porque se ha salvado el contenido de un fichero, y si ese fichero es del tipo que queremos compilar, es decir un fichero C++, el programa nos avisará de que debemos compilar ese fichero porque ha sufrido cambios.

Al recibir el aviso, se lanza el sistema de refactorización, obteniéndose el árbol AST del fichero que está abierto en el editor en ese momento y que acaba de salvarse. Se manda el árbol a la plantilla de transformación y se realizan las transformaciones

correspondientes. Finalmente, se creará de forma automática un fichero ".cpp" en la ruta **<nombre_proyecto>/Cpp-gen/<nombre_fichero>.cpp**

Este proceso se repetirá automáticamente cada vez que se salve el contenido de un fichero C++ que pertenezca a un proyecto al que se le haya añadido la naturaleza de nuestro "compilador automático". Esto, en un primer análisis, puede parecer pesado en cuanto a lo que respecta eficiencia y velocidad del sistema de refactorización, pero tenemos que decir que el sistema no se ve afectado. Siempre estará analizándose un fichero al mismo tiempo, por lo que el tiempo de transformación con la arquitectura seleccionada es despreciable.

Esta acción de refactorización será accesible desde:

- Botón Secundario -> Refactorizar -> Añadir naturaleza **Cpp11Nature**.

4. ANALISIS Y DISEÑO

4.1. Marco Regulador

Este apartado del documento será el que presente las normativas legales y las normativas técnicas que afecten al trabajo del desarrollador y al proceso de desarrollo.

Debería considerarse necesario contemplar la ***Ley Orgánica 15/1999 de 13 de diciembre de Protección de Datos de Carácter Personal****, (LOPD), que tiene por objeto garantizar y proteger, en lo que concierne al tratamiento de los datos personales, las libertades públicas y los derechos fundamentales de las personas físicas, y especialmente de su honor, intimidad y privacidad personal y familiar.*

Puesto que el proyecto que se desarrolla, consiste en la investigación sobre los procesos MDE y la arquitectura MDA; y puesto que la aplicación práctica consiste en un complemento para eclipse de refactorización de C++ a C++11, siendo eclipse un IDE de software libre, no es necesario tener en cuenta esta ley.

Además, se entiende que el uso de la aplicación, salvo publicación posterior será de uso personal y por tanto no accesible a cualquier persona.

Para terminar, con el fin de estandarizar el trabajo de implementación se seguirá el estándar internacional C++ ISO/IEC 14882:2011

4.2. Requisitos de Usuario

Este apartado del documento mostrará un listado de los requisitos de usuario que tendrá nuestro caso práctico presentado como arquitectura MDA.

Los requisitos de usuario, referenciarán las distintas funcionalidades que se desean del refactorizador de código C++ a C++11.

Para representar estos requisitos de usuario utilizaremos el siguiente formato tabla:

IDENTIFICADOR: <id_requisito>	
Prioridad:	Alta / Media / Baja
Necesidad:	Esencial / Deseable / Opcional
Descripción:	<descripción_requisito>

La definición de los diferentes puntos referentes a la tabla anterior es la siguiente:

- **Identificador:**
  Identificará y diferenciará los requisitos de usuario. Para ello recurrirá la nomenclatura **RUX-WW**, donde se tomarán los valores:

    - **X:** indicará el tipo de requisito, siendo posibles los siguientes valores:
        - **F:** identificará un requisito de tipo funcional.
        - **N:** identificará un requisito de tipo no funcional.

    - **WW:** indicará el número del requisito del tipo anteriormente indicado en el que nos encontramos, de forma consecutiva.

- **Necesidad:**
  Este campo nos indicará si un atributo es no imprescindible para el correcto funcionamiento de la aplicación. Un atributo imprescindible se marcará como esencial, mientras que los atributos prescindibles se marcarán como opcionales.

- **Descripción:**
  Definición escueta pero clara de la funcionalidad que se desea obtener de la aplicación.

En base a esta definición de tabla de requisitos, el refactorizador de C++ a C++11 cumplirá con los siguientes requisitos:

IDENTIFICADOR: RUN-01	
Prioridad:	Alta
Necesidad:	Esencial
Descripción:	Compatibilidad con la versión Helios de Eclipse IDE (v3.6.x).

IDENTIFICADOR: RUF-01	
Prioridad:	Alta
Necesidad:	Esencial
Descripción:	Buscar la manera de extraer en un modelo el contenido de una clase C++ e implementarlo.

IDENTIFICADOR: RUF-02	
Prioridad:	Alta
Necesidad:	Esencial
Descripción:	Crear una plantilla de transformación utilizando XTEND2.

IDENTIFICADOR: RUF-03	
Prioridad:	Alta
Necesidad:	Esencial
Descripción:	Crear un manejador de ficheros que nos permita generar los ficheros y añadirle el código transformado.

IDENTIFICADOR: RUF-04	
Prioridad:	Alta
Necesidad:	Esencial
Descripción:	Organizar una arquitectura MDA abstrayendo los tres elementos del proceso: extracción de modelo, plantilla de refactorización y generador de ficheros.

IDENTIFICADOR: RUF-05	
Prioridad:	Alta
Necesidad:	Esencial
Descripción:	Organizar un sistema de comunicación entre los tres elementos de la arquitectura MDA generada.

IDENTIFICADOR: RUF-06	
Prioridad:	Alta
Necesidad:	Esencial
Descripción:	Añadir una opción al “toolbar” de eclipse que ejecute la acción de refactorización de código con la etiqueta “Compilar”.

IDENTIFICADOR: RUF-07	
Prioridad:	Alta
Necesidad:	Esencial
Descripción:	Añadir una acción al “toolbar” de eclipse que permita ejecutar la acción de refactorización de código con la etiqueta “Compilar en…”

IDENTIFICADOR: RUF-08	
Prioridad:	Media
Necesidad:	Deseable
Descripción:	Añadir una acción al “toolbar” de eclipse que permita ejecutar la acción de comparar.

IDENTIFICADOR: RUF-09	
Prioridad:	Media
Necesidad:	Esencial / Deseable / Opcional
Descripción:	Añadir una acción al menú de acciones del botón secundario sobre un proyecto C++ para añadirle una naturaleza que permita lanzar el compilador automático.

IDENTIFICADOR: RUF-10	
Prioridad:	Alta
Necesidad:	Esencial
Descripción:	Añadir una acción al menú de acciones del botón secundario sobre un fichero C++ para ejecutar la acción de refactorización con la etiqueta “Compilar”.

IDENTIFICADOR: RUF-11	
Prioridad:	Alta
Necesidad:	Esencial
Descripción:	Añadir una acción al menú de acciones del botón secundario sobre un fichero C++ para ejecutar la acción de refactorización con la etiqueta “Compilar en…”.

IDENTIFICADOR: RUF-12	
Prioridad:	Media
Necesidad:	Deseable
Descripción:	Añadir una acción al menú de acciones del botón secundario sobre un fichero C++ para ejecutar la acción de refactorización con la etiqueta “Comparar”.

IDENTIFICADOR: RUF-13	
Prioridad:	Alta
Necesidad:	Esencial
Descripción:	Implementar la acción de refactorización que se ejecute al pulsar el botón del “toolbar” o en el menú secundario “Compilar”.

IDENTIFICADOR: RUF-14	
Prioridad:	Alta
Necesidad:	Esencial
Descripción:	Implementar la acción de refactorización que se ejecute al pulsar el botón del “toolbar” o en el menú secundario “Compilar en…”.

IDENTIFICADOR: RUF-15	
Prioridad:	Media
Necesidad:	Deseable
Descripción:	Implementar la acción de refactorización que se ejecute al pulsar el botón del “toolbar” o en el menú secundario “Comparar”.

IDENTIFICADOR: RUF-16	
Prioridad:	Media
Necesidad:	Deseable
Descripción:	Implementar el compilador automático que refactorice un fichero C++ cuando se salve el editor del fichero. Debe realizar el proceso de refactorización al salvar.

IDENTIFICADOR: RUF-17	
Prioridad:	Media
Necesidad:	Deseable
Descripción:	Mostar un diálogo al pulsar la acción de “Compilar en…” desde cualquiera de sus puntos de acceso que muestre un diálogo emergente que pida al usuario la ruta de destino del fichero autogenerado.

IDENTIFICADOR: RUF-18	
Prioridad:	Baja
Necesidad:	Opcional
Descripción:	Implementar un sistema de auto formateo de código que de formato al código generado C++11 que se incluirá en el fichero de salida.

IDENTIFICADOR: RUF-19	
Prioridad:	Media
Necesidad:	Deseable
Descripción:	Mostrar un diálogo de comparación de código de eclipse al seleccionar la opción “Comparar”.

IDENTIFICADOR: RUF-20	
Prioridad:	Media
Necesidad:	Deseable
Descripción:	Dotar de funcionalidades de comparación al diálogo de comparación que aparece al pulsar “Comparar”.

IDENTIFICADOR: RUF-21	
Prioridad:	Alta
Necesidad:	Esencial
Descripción:	Aplicar transformaciones para declaración de variable auto cuando sea necesario.

IDENTIFICADOR: RUF-22	
Prioridad:	Alta
Necesidad:	Esencial
Descripción:	Aplicar transformaciones para definición de variables null por std::nullptr.

IDENTIFICADOR: RUF-23	
Prioridad:	Alta
Necesidad:	Esencial
Descripción:	Aplicar transformaciones para generación de bucles forEach cuando se utilicen iteradores sobre listas en un bucle For.

IDENTIFICADOR: RUF-24	
Prioridad:	Alta
Necesidad:	Esencial
Descripción:	Aplicación de transformaciones de sustitución de la declaración typdef por using.

4.3. Diagrama de casos de uso

En este punto del documento se mostrará un diagrama de casos de uso detallando las diferentes funciones e iteraciones que puede tener el usuario con la aplicación.

El diagrama de casos de uso asociado al refactorizador de C++ a C++11 es el siguiente:

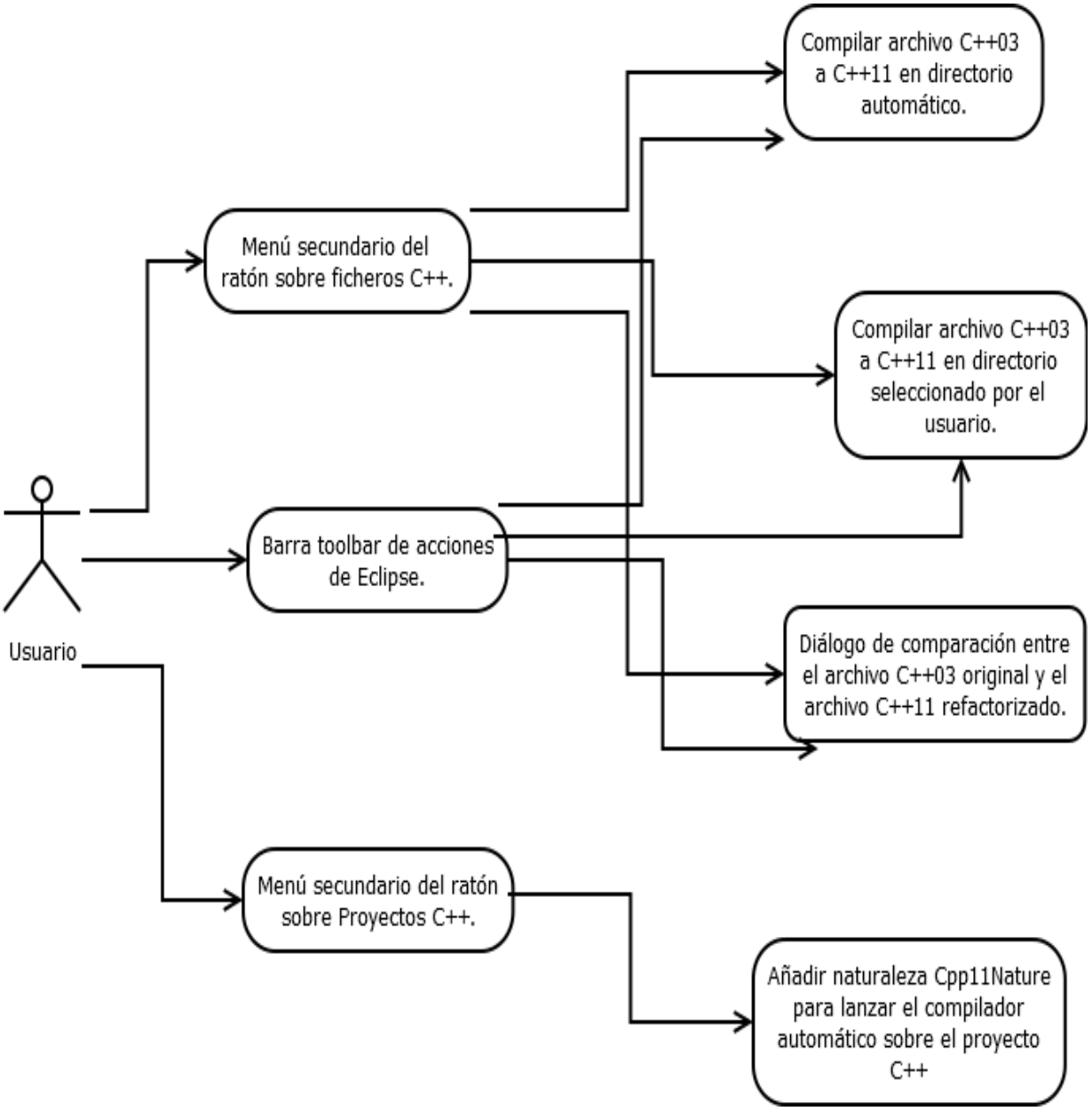


Ilustración 27. Diagrama de casos de uso del refactorizador.

En el caso aplicado del refactorizador C++ a C++11 solo tendremos un actor que se corresponderá con el usuario del "plug-in" de eclipse que nos permite transformar código.

El complemento de eclipse permite realizar cuatro funciones finales de refactorización.

Las funciones de refactorización de ficheros que pueden tener lugar son Compilar el fichero en el directorio automático (**cpp11-gen**), compilar el fichero en la ruta indicada por el usuario, mostrar un diálogo de comparación entre el fichero de origen y el código C++11 generado; y, por último, añadir una naturaleza (**Cpp11Nature**) a un proyecto C++ para que se lance el refactorizador de forma automática al salva un fichero C++ que pertenezca a ese proyecto. Los ficheros generados al añadir una naturaleza se generarán de forma automática en la carpeta **cpp11-gen**.

La manera que el usuario tiene de acceder a estas funcionalidades es a través de las acciones habilitadas para ello mediante la contribución a los diferentes menús de acciones que ofrece eclipse.

En este caso, tendremos dos maneras de hacerlo. Una a través de la barra de acciones rápidas ("toolbar") de eclipse, y otra a través de los diferentes menús pop-up que muestra eclipse al clicar con el secundario del ratón en los diferentes tipos de archivos (ficheros cpp, ficheros java, proyectos cpp, proyectos web…).

La tabla de casos de uso resultante es la siguiente:

Caso de Uso	Acción	Descripción
Menú secundario del ratón sobre ficheros C++.	Compilar archivo C++ a C++11 en directorio automático.	Permite al usuario refactorizar un fichero C++03 a C++11 y se generará de forma automática en una carpeta sobre el propio proyecto con el nombre **cpp11-gen**.
	Compilar archivo C++ a C++11 en directorio seleccionado por el usuario	Permite al usuario refactorizar un fichero C++03 a C++11, en la ruta que el usuario indica en el diálogo que se muestra al ejecutar la acción.
	Diálogo de comparación entre el archivo original y el archivo refactorizado.	Muestra un diálogo de comparación de ficheros de eclipse con el fichero de entrada y con el fichero C++11 resultante de la

		refactorización.
Barra de acciones rápidas (“toolbar”) de Eclipse.	Compilar archivo C++ a C++11 en directorio automático.	Permite al usuario refactorizar un fichero C++03 a C++11 y se generará de forma automática en una carpeta sobre el propio proyecto con el nombre **cpp11-gen**.
	Compilar archivo C++ a C++11 en directorio seleccionado por el usuario	Permite al usuario refactorizar un fichero C++03 a C++11, en la ruta que el usuario indica en el diálogo que se muestra al ejecutar la acción.
	Diálogo de comparación entre el archivo original y el archivo refactorizado.	Muestra un diálogo de comparación de ficheros de eclipse con el fichero de entrada y con el fichero C++11 resultante de la refactorización.
Menú secundario del ratón sobre proyectos C++.	Añadir naturaleza Cpp11Nature para lanzar el compilador automático sobre el proyecto C++.	Permite al usuario añadir una naturaleza a los proyectos C++. Gracias a esta naturaleza podemos crear un “Builder” que detecte los cambios sobre los ficheros de un proyecto C++ y genere de forma automática los ficheros C++11 en la carpeta **cpp11-gen** del proyecto C++.

5. CONCLUSIONES

5.1. Futuras mejoras

El proceso de implementación del sistema de refactorización no es un proceso sencillo. Tenemos que tener en cuenta que un lenguaje de programación es demasiado extenso y que el código que podemos encontrarnos depende mucho del programador que lo desarrolle.

Al tratarse de lenguajes de programación extensos, la realización de un sistema de compilación de lenguaje C++ a lenguaje C++11 requiere mucho tiempo para analizar cada uno de los elementos, cada uno de los posibles casos de uso de esos elementos y la implementación de un elemento compatible en C++11.

Debido al alcance del trabajo fin de grado no ha sido posible plantear objetivos basados en la mejora del rendimiento de la aplicación, pero podría ser una futura mejora directa de este trabajo.

Otra posible futura mejora directa de esta aplicación es permitir a los usuarios del sistema la posibilidad de cambiar las plantillas en cualquier momento desde el propio entorno de trabajo. La implementación de esta funcionalidad es complicada, ya que no solo requiere conocer el contenido de la plantilla que se está utilizando en el sistema refactorizador, sino que además, requiere la creación de algún compilador automático y un manejador de ficheros que almacene la nueva plantilla y el nuevo fichero java autogenerado tras salvar el contenido, en el fichero ".jar" que contiene las plantillas de refactorización que se están aplicando. Es una nueva funcionalidad interesante que podría dotar de mayor versatilidad al sistema, y podría permitir al usuario una mayor libertad en sus procesos de transformación.

Durante el desarrollo de la aplicación el principal problema con el que nos encontramos es la compatibilidad entre los nuevos elementos de C++11 y los anteriores disponibles en C++03. Lo ideal, para sacar el máximo rendimiento sería transformar los elementos C++03 por los elementos compatibles en C++11 que ofrezcan un mayor rendimiento a la aplicación, pero aquí es dónde surge el problema. Para poder, sustituir por completo unos elementos por otros necesitaríamos generar modelos intermedios que relaciones los métodos de un elemento con los de otro, ya que debemos sustituir no solo la declaración de variables, sino los métodos que ejecutan. Para esto es posible aplicar modelos intermedios utilizando el sistema que se propone y que sean estos quienes contengan las referencias entre los métodos de los dos elementos comparados.

De esta manera podríamos sustituir los elementos que tenga la aplicación del usuario por elementos compatibles al cien por cien, sin alterar el funcionamiento de la

aplicación y dotándola de una mayor rapidez y eficiencia. En el caso ideal tendríamos una refactorización completa de un código de un lenguaje a otro.

Como se comenta, conseguir esto es algo prácticamente inviable salvo en el caso de dedicarse única y exclusivamente a esto, pudiendo convertirse en algo inviable en cualquier momento, pero sería interesante seguir trabajando sobre esto.

Como posibles mejoras futuras, indicaría lo anterior expuesto. Tratar de dotar poco a poco a la aplicación de nuevas transformaciones. Lo ideal sería observar los nuevos elementos de C++11, analizarlos, compararlos con los existentes en C++03 e intentar incorporarlos en una aplicación.

Poco a poco, la aplicación será más consistente y podría realizar mejores transformaciones, pero aun así, debemos tratarlo como un tema muy delicado, ya que la posibilidad de que el sistema falle aumenta considerablemente en medida que el código de entrada se vuelve más complicado o se incrementa considerablemente.

Es posible conseguir un refactorizador de C++03 a C++11 bastante completo dedicándole muchas horas de trabajo, pero posiblemente para pequeñas aplicaciones y no tanto para sistemas completos.

Por lo demás, es una experiencia interesante que te hace pensar en que la posibilidad de generar nuestros propios compiladores entre lenguajes de programación no está tan lejos ni es necesariamente tan complicado como aparenta en primera instancia. Aunque sí es cierto, que no es nada fácil conseguir un compilador compatible y funcional completo sin mucha dedicación, mucho esfuerzo y muchas pruebas y errores.

5.2. Presupuesto.

Este punto del documento muestra el cálculo del presupuesto del desarrollo de este proyecto de fin de grado. Los datos que tendremos en cuenta para calcular el presupuesto referente a recursos humanos, se tomarán de dos partes. La primera parte es referente a las horas dedicadas al proceso de investigación y conocimiento de las arquitecturas dirigidas por modelos y al estudio de la ingeniería dirigida por modelos. La segunda parte del presupuesto referente a los recursos humanos se obtendrá a partir de los datos introducidos en la parte de planificación del trabajo del apartado 3.2, dónde se detalla cada una de las tareas de la fase de desarrollo del caso práctico aplicando MDE (Refactorizador C++ a C++11).

El desarrollo del proyecto conlleva un total de ciento ocho días con una media de trabajo de cuatro horas diarias, implican un total de cuatrocientas treinta y dos horas de trabajo.

Respecto al hardware necesario para el desarrollo del proyecto, se ha utilizado un ordenador portátil **HP EliteBook 8470p**. Con este ordenador se ha llevado a cabo tanto la parte de desarrollo de la aplicación de prueba como la parte de la documentación del sistema.

En cuanto a las características del ordenador, son las siguientes:

- Procesador Intel Core i5 3320M.
- 4GB de Memoria Ram.
- 500 GB de Disco Duro.
- Grafica Intel HD 4000.

En cuanto al software utilizado, se ha tratado de recurrir a software libre en medida de lo posible. La parte referente al entorno de implementación Eclipse IDE y a todos los complementos necesarios para poder desarrollar este proyecto son considerados software libre y accesible para cualquier usuario, por lo que podemos concluir que su coste de uso es 0.

El sistema operativo utilizado para realizar el proceso de **I+D sobre MDE/MDA**, para realizar la implementación del caso práctico (Refactorizador C++ a C++11) y para la redacción de la documentación referente al proyecto ha sido Windows 7 Enterprise.

Se ha utilizado para los temas relacionados con ofimática Microsoft Office 2010 Enterpise. En lo referente al diseño de diagramas de casos de uso y los esquemas que aparecen en la documentación, se ha utilizado la herramienta DIA con licencia gratuita.

Por último, el grupo de desarrollo encargado de implementar este proyecto estaría formado por un analista desarrollador. Esta persona se encargaría de realizar el proceso de **I+D sobre MDE/MDA** y posteriormente se encargaría de implementar un

sistema de prueba utilizando los conocimientos adquiridos sobre esta metodología de trabajo. El caso aplicado resultante, un **refactorizador de C++ a C++**11, consistiría en crear un complemento para eclipse que lo dote de nuevas funcionalidades y que permitan realizar pequeñas refactorizaciones de código de C++ a C++11.

Desglose Presupuestario del proyecto

1. **AUTOR:** Alejandro Tovar Moreno

2. **DEPARTAMENTO:** Informática

3. **DESCRIPCIÓN DEL PROYECTO:** Ingeniería dirigida a modelos. Sistemas de transformación de modelo a texto. Complemento de refactorización para eclipse de código C++ a C++11 para Eclipse.

4. **PRESUPUESTO TOTAL EN EUROS:** Nueve mil cuarenta y siente con siete céntimos.

Presupuesto Referente a RRHH

Desglose presupuestario referente a RRHH				
Función desarrollada	**Categoría**	**Dedicación (Horas)**	**Coste/Hora**	**Coste Total**
I+D sobre MDE/MDA	Programador Junior	184	20,45€/hora	3.762,80
Desarrollo refactorizador C++ a C++11	Programador Junior	248	20,45€/hora	5.071,60
Coste Total de RRHH:				**8.834,40€**

Presupuesto referente al uso de Equipos y Software

Desglose presupuestario referente a Equipos y Software						
Descripción	**Coste (Euros)**	**Tiempo de vida (meses)**	**Dedicación al proyecto (%)**	**Tiempo de uso (meses)**	**Coste de uso mensual (Euro)**	**Coste imputable (Euro)**
Hp EliteBook 8470P	1097,00	42	100	4	26,11	104,44
Licencia Microsoft Office 2010 Entrerprise	199,00	36	100	4	5,52	22,08
Licencia Widows 7 Enterprise	199,00	36	100	4	5,52	22,08
Eclipse IDE	0,00	50	100	4	0,00	0,00
Eclipse Plug-in	0,00	50	100	4	0,00	0,00
DIA	0,00	50	100	4	0,00	0,00
Coste Total de Equipos y software:						**148,60€**

Presupuesto referente a otros costes aplicables al proyecto

Desglose presupuestario con otros costes aplicables				
Descripción	**Coste mensual (Euros)**	**Uso (en % por mes)**	**Tiempo de uso (meses)**	**Coste imputable (Euros)**
Conexión a Internet	24,99	35%	4	34,98
Electricidad	35,00	20%	4	28
Coste total:				**62,98€**

Resumen de costes

Desglose total del proyecto	
Descripción de costes totales	**Costes totales del proyecto (Euros)**
Costes de RRHH	8.834,49
Costes de Equipo y Software	148,60
Otros costes	62,98
TOTAL:	**9.046,07€**

5.3. Conclusiones personales

En primer lugar, destacar que los objetivos principales de esta aplicación se han cumplido.

En un primero momento, se trataba de investigar acerca de las arquitecturas dirigidas por modelos (MDA) y la ingenierías dirigida por modelos (MDE). Es complicado encontrar información concreta sobre este tema, ya que es un sistema en expansión actualmente que, desde mi punto de vista, será todo un éxito en un futuro si las empresas se conciencian con este sistema de trabajo.

Se trata de un sistema de trabajo muy abierto, basado en modelos, pero no en un tipo de modelo concreto, sino considerando como modelos cualquier elemento o elementos que nos permitan representar de alguna manera una realidad. De esta manera, llevándolo al punto informático, podríamos trabajar con modelos de tipo UML, modelos ECORE, árboles sintácticos AST, modelos XML/XMI, o en general cualquier elementos que nos permita una fácil manipulación para añadir o eliminar propiedades y que nos permita iterar para obtener resultados en un futuro.

Desde mi punto de vista, organizar un proceso de desarrollo, abstrayendo el trabajo hasta el punto de representar realidades sin tener en cuenta el lenguaje de desarrollo, el entorno de programación o el nivel de los desarrolladores y personas que intervienen en el proceso de desarrollo, es algo muy interesante. Al elevar el proceso a una capa de abstracción tan elevada, sin entrar en las posibilidades a bajo nivel, cualquier persona que intervenga en el proceso puede opinar acerca de un elemento y proponer modificaciones aunque no conozca el sistema de desarrollo final.

Gracias a los sistemas de transformación de modelo a modelo (M2M) que se han contado en este documento, se puede ir refinando el modelo en cada fase del proceso de desarrollo si se desea, hasta obtener un modelo definitivo. Si lo miramos desde el punto de vista de un desarrollador Java, cada uno de estos modelos sería una representación de un objeto java pero no tendríamos que implementarlos ni mantenerlos, ya que actualmente existen complementos que te permiten trabajar y modificar el modelo como tal, refinándolo hasta el punto que se desee. Posteriormente, utilizando los sistemas de transformación de modelo a texto (M2T) que se señalan en este documento, podríamos generar cualquier tipo de fichero que nos interese en base a los modelos. Bastaría con iterar todos los elementos del modelo y generar el tipo de fichero que nos interese. Un punto muy interesante que pienso que ofrecen los sistemas de transformación de modelos a texto utilizando modelos es que permiten una salida muy abierta, pudiéndose generar a partir de un modelo ficheros de salida de cualquier tipo: HTML, XHTML, XML, Java, C++, C, documentación...

Si por ejemplo trabajamos con modelos que referencian componentes como cajas de texto, botones, tablas, listas… y deseamos que se generen ficheros que se ejecuten en cliente y servidor, podría interesarnos a partir de ese modelos generar un fichero HTML o XHTML que pinte una página web en base a esos componentes y a sus atributos, pero también podría interesarnos generar ficheros JavaScript para cada una de los eventos de estos componentes y podría interesarnos crear un documento JavaScript que genere y pinte ese HTML. Pero si por ejemplo, el modelo de componentes también permitiese salvar los estilos de los componentes, podríamos crear una plantilla de transformación de modelo a texto que nos genere un fichero CSS que le asocie el estilo correspondiente a cada componente. Este proceso podría alargarse tanto como nos interese y tanto como refinemos nuestro modelo de componentes. Cuanto más refinado esté y cuanto más completo sea, más provecho podríamos sacarle a los sistemas de transformación de modelos y menos carga de trabajo tendría los desarrolladores.

Sin duda, teniendo en cuenta estas explicaciones, creo que nos enfrentamos a un gran sistema de rejuvenecimiento de código y a una gran posibilidad de avance tecnológico. Con esto, quiero decir, que gracias a estos sistemas, nos alejamos poco a poco de las formas de trabajo y los sistemas de trabajo orientados a objetos “manuales”, al “infierno” de crear objetos y mantenerlos. Cada vez que deseamos añadir nuevos atributos o funcionalidades a un objeto es posible que un desarrollador tenga que tocar varios ficheros de código, probar que la nueva funcionalidad es correcta y comprobar que el resto del sistema funciona correctamente. Si utilizamos un sistema basado en modelos, por el contrario, cada una de las nuevas funcionalidades requeriría tan solo una pequeña modificación del modelo y en algunas ocasiones, si por ejemplo trabajamos con java, existen herramientas como EMF que generan los ficheros java correspondiente a esos “objetos” que representan los modelos.

Tan solo al modificar el modelo, y manteniendo la arquitectura anteriormente establecida, añadiendo la nueva modificación a las plantillas de salida y las plantillas de transformación de modelos (si el proceso requiere de estas transformaciones) tendríamos totalmente funcional la aplicación y no requeriría tantos cambios ni tan manuales como el caso anterior.

Además, como es obvio, permite la posibilidad de automatizar parte del proceso de desarrollo, evitando tener que generar de forma manual o programática algunos ficheros que gracias a las plantillas de transformación podríamos obtener de forma automática, más limpia y más rápida.

Además, la utilización de plantillas de transformación también las considero un gran avance. Si pretendemos generar cualquier fichero y queremos modificar la externalización del contenido del fichero, si utilizamos arquitecturas bien diferenciadas separando el modelo, la plantilla y el sistema generador, los cambios requeridos son

mínimos. Bastaría con modificar la plantilla de transformación para obtener el resultado deseado.

El principal inconveniente que le encuentro, es que al no ser un sistema aún demasiado extendido, puede resultar complicado empezar a utilizarlo si pretendemos crear una arquitectura MDA muy grande y muy sólida. Además, la incorporación de los complementos de eclipse que hemos estudiado y la utilización de sus librerías no es sencilla tampoco.

Otro inconveniente importante es el desconocimiento total de los lenguajes de programación de plantillas, pero es necesario indicar que con un poco de uso se aprenden a utilizar rápido, son bastante intuitivas ya que los modelos suelen ser simples basados en atributos y funciones, aunque sean densos. Además, algunos de los sistemas propuestos como XTEND2 son similares a Java, en incluso compatibles con él, y es posible programarlos de forma muy parecida.

Además, creo que es interesante invertir tiempo en aprender a programar plantillas, ya que si conseguimos dominarlas y decidimos utilizar este tipo de arquitecturas ahorraremos tiempo de trabajo y obtendremos una mejor organización del proceso de desarrollo y por tanto mejores resultados.

Por lo demás, creo que es un buen sistema de trabajo, para controlar y facilitar el desarrollo de proyectos grandes en grandes empresas. Creo que es digno de estudiar y analizar, ya que el proceso de desarrollo puede resultar más sencillo para el equipo de trabajo y el tiempo de desarrollo puede reducirse gracias a la automatización de parte del proceso. Sin duda, el sistema sería mucho más fácil de mantener gracias a su organización y además, en muchos casos podría resultar incluso más fácil de probar ya que pueden realizarse pruebas automáticas sobre modelos establecidos frente a los modelos transformados o frente a los ficheros de salida.

Con respecto al segundo objetivo de este trabajo de fin de grado, la creación de un sistema refactorizador de código C++ a C++11, basado en una arquitectura MDA y plantillas de transformación de modelo a texto también se han obtenido los resultados esperados.

No ha sido sencillo crear una arquitectura MDA muy diferenciada debido a las limitaciones de los ficheros de entrada. Actualmente, para lenguajes de programación como JAVA existen numerosas herramientas que nos permiten crear un modelo que sea fácil de iterar y al que sea fácil a aplicar transformaciones de modelo a texto. Debido al desconocimiento sobre el tema, mucho tiempo se ha invertido en buscar alguna manera de obtener un modelo al que aplicar estas transformaciones.

Finalmente, se ha decidido utilizar el árbol AST ofrecido por la propia CDT de eclipse. Este árbol nos ofrece un modelo y las relaciones entre los elementos de un fichero C++

de forma estática. Este modelo es el que se ha analizado y recorrido para poder realizar modificaciones a C++11.

Como se contó en la sección de posibles mejoras futuras, otro problema fundamental en la creación de refactorizadores de código es el tiempo. La realización de un sistema refactorizador completo, efectivo y sin errores requiere mucho tiempo y muchas pruebas. Estudiando las nuevas incorporaciones de C++11 y teniendo en cuenta las posibilidades de refactorización de elementos C++03 con respecto a estas incorporaciones se han realizado transformaciones tales como la declaración de variables automáticas, la declaración de punteros a NULL, la utilización de bucles forEach dónde antes se utilizaban iteradores y la utilización de la etiqueta **using** donde antes utilizábamos al etiqueta **typedef**.

En cuanto al resto de elementos que ofrece C++11, son realmente interesantes, lo ideal sería seguir avanzando este transformador y ofrecer la posibilidad de refactorizar elementos **char*** con elementos **std::String**, modificar los vectores por "Arrays" y buscar mejoras de rendimiento que se puedan obtener con C++11, pero esto supone inconvenientes. Los principales problemas que aparecen son la modificación del código original del usuario, que igual no desea que modifiquemos su sistema de programar o sus elementos pero además el principal problema es que necesitaríamos crear dos modelos que muestren las compatibilidades entre los elementos que se comparan, para evitar problemas. Es decir, un **char***podríamos querer cambiarlo por un **std::String** pero para poder hacer esto tendríamos que tener en cuenta que las funciones que están disponibles para un elemento no tienen por qué estarlo para el otro elemento. Sería necesario tener modelos intermedios y un motor de transformaciones que indique que función del elemento de destino es compatible con la de origen y en caso de que no existe implementarle al modelo de destino una función compatible con la del elemento de origen.

Como explicaba antes, la creación de un refactorizador bastante completo, requiere mucho tiempo de análisis y requiere tener en cuenta muchos factores y muchos elementos.

Sin embargo, estoy muy contento con el resultado obtenido, ya que se ha podido crear una arquitectura MDA con los tres elementos (modelo, plantilla y salida) diferenciados y se ha creado un sistema de transformación basado en XTEND2 que permite realizar transformaciones básicas entre ficheros C++03 y C++11. Sería interesante intentar avanzarlo un poco más en el futuro y ver si realmente es viable obtener buenos resultados y un sistema de refactorización completo e interesante para los usuarios.

Los objetivos propuestos para esta segunda parte del trabajo fin de carrera, la creación de complementos de eclipse, la creación de acciones basadas en las librerías de eclipse, la utilización de una arquitectura MDA, la creación de un sistema de

transformación de modelo a texto de C++ a C++11 basado en plantillas, el aprendizaje del lenguaje de plantillas XTEND2 y la utilización de éstas, la creación de recursos utilizando el sistema eclipse y los elementos de monitorización de progreso que ofrece se han cumplido de forma satisfactoria, a pesar del desconocimiento inicial de las posibilidades que ofrece el entorno eclipse, de los sistemas MDE/MDA y del funcionamiento a elevado nivel de los sistemas de transformación de modelo a texto.

Estoy muy satisfecho con el esfuerzo realizado y los resultados obtenidos. Además, considero este proyecto como un buen sistema para conocer el funcionamiento de la ingeniería dirigida por modelos y la creación de arquitecturas MDA, que han resultado ser conceptos muy interesantes y con gran proyección de cara al futuro.

Anexo I. Manual de usuario.

Este anexo del documento tiene como finalidad crear un manual de usuario con las diferentes acciones de refactorización que se podrán realizar sobre el entorno de trabajo eclipse tras la instalación de los complementos necesarios y de los complementos de refactorización que se han creado.

Como hemos explicado antes, se ha tratado de contribuir a los menús de operaciones de eclipse para poder dotar así al entorno de trabajo de eclipse de nuevas funcionalidades que permitiesen realizar un proceso de refactorización de ficheros C++ a C++11.

De esta manera, una vez instalado el complemento de refactorización sobre eclipse, el entorno de trabajo que nos encontraremos será como este:

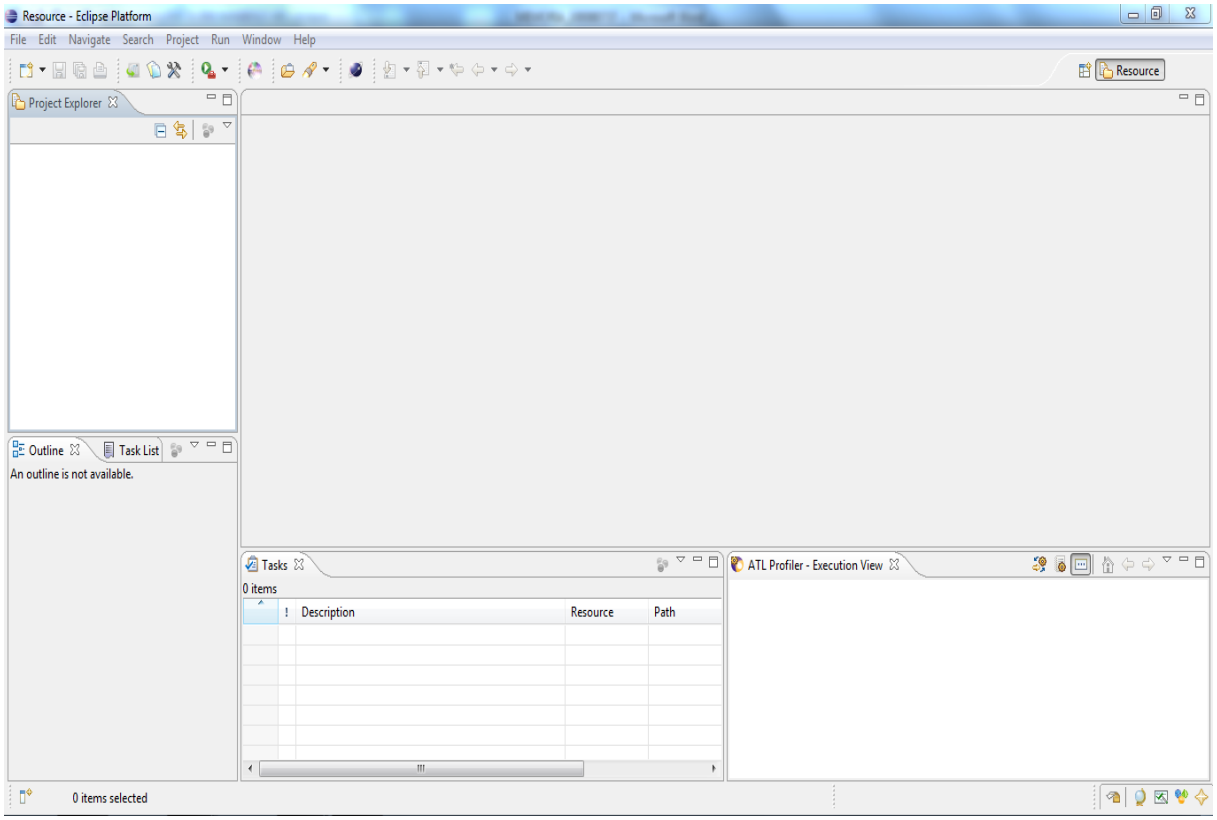


Ilustración 28. Entorno de trabajo con el complemento de refactorización de C++03 a C++11.

Como se puede observar en la imagen anterior, el entorno de trabajo no difiere en mucho del entorno de trabajo clásico que ofrece la plataforma Eclipse, salvo porque ofrece como novedad un nuevo conjunto de acciones en el menú de acciones rápidas ("toolbar") de eclipse.

Este nuevo conjunto de acciones es el siguiente:

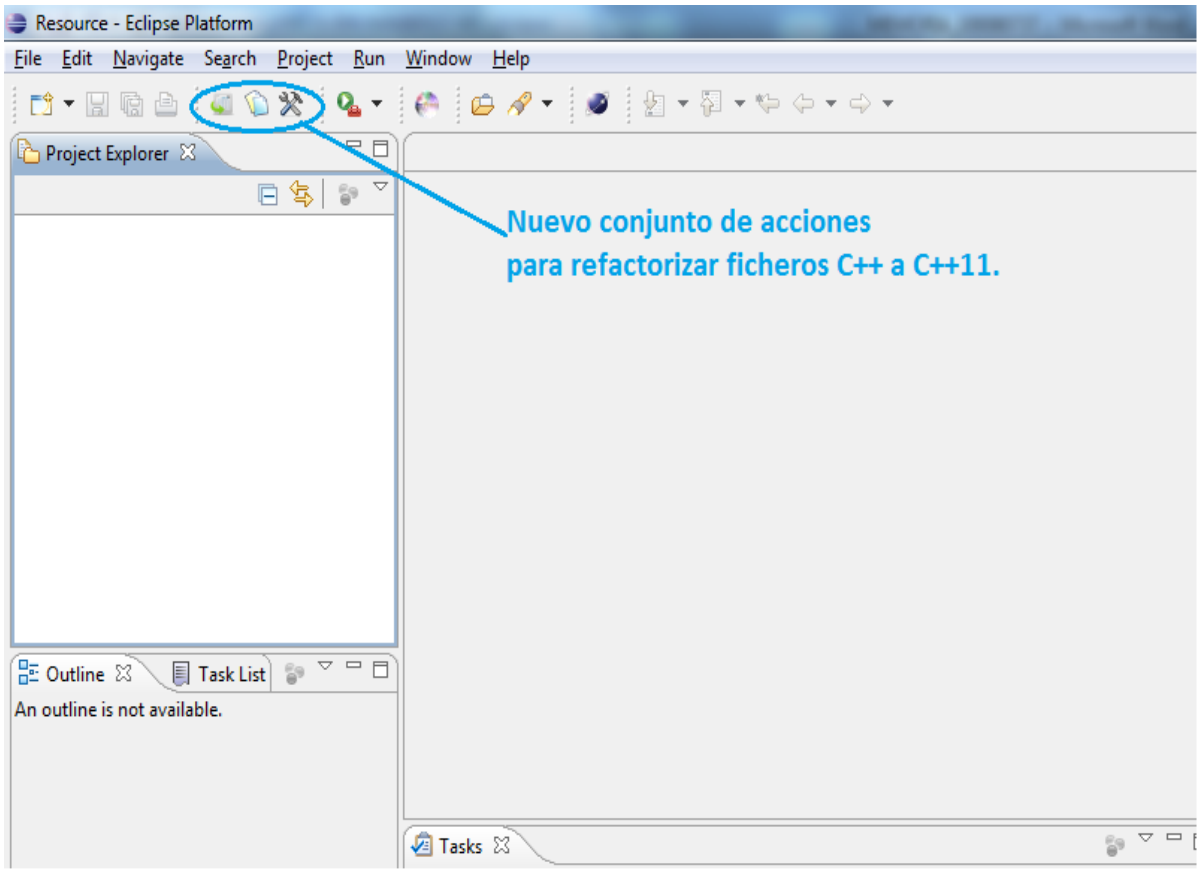


Ilustración 29. Conjunto de acciones añadido a la barra de acciones rápidas ("toolbar") de Eclipse.

Este nuevo conjunto de acciones que se muestra sobre la barra de acciones de eclipse es el que nos va a permitir de forma rápida operar con los ficheros C++ cuando queramos refactorizarlos.

Podremos realizar tres operaciones distintas desde este conjunto de acciones que presentamos. De izquierda a derecha, podremos realizar las acciones:

- **Compilar en…**
- **Comparar**
- **Compilar**

A continuación contaremos de forma detallada en que consiste cada una de estas operaciones y cuál sería el proceso a seguir para realizar refactorizaciones de código C++ a C++11.

En primer lugar, para poder realizar operaciones de refactorización sobre ficheros C++, sería necesario preparar el espacio de trabajo de forma adecuada. Necesitaremos crear un proyecto C++ y añadirle al menos un fichero C++ para poder interactuar con él.

Para ello seguimos el siguiente proceso:

- Seguimos la secuencia: File -> New -> Other (Ctrl + N si utilizamos atajos de teclado).
  Una vez abierto el cuadro de dialogo de nuevos elementos seleccionamos **C/C++ -> C++ Project.**

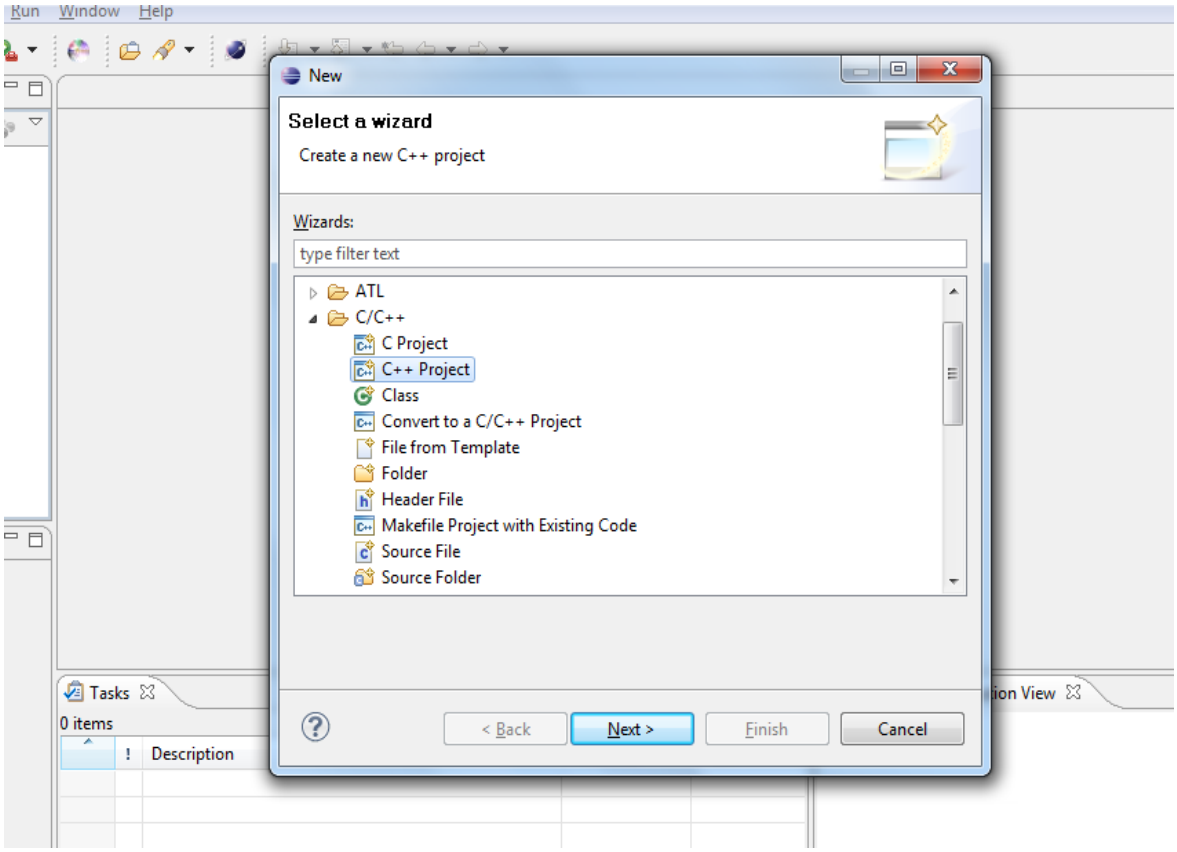


Ilustración 30. Creación de un proyecto C++.

Configuramos el proyecto dándole el nombre que nos interese, por ejemplo **ProyectoPruebas**, y pulsamos el botón **Finish.** Una vez hecho esto, se añadirá un nuevo proyecto C++ a nuestro espacio de trabajo de eclipse, quedando de la siguiente manera:

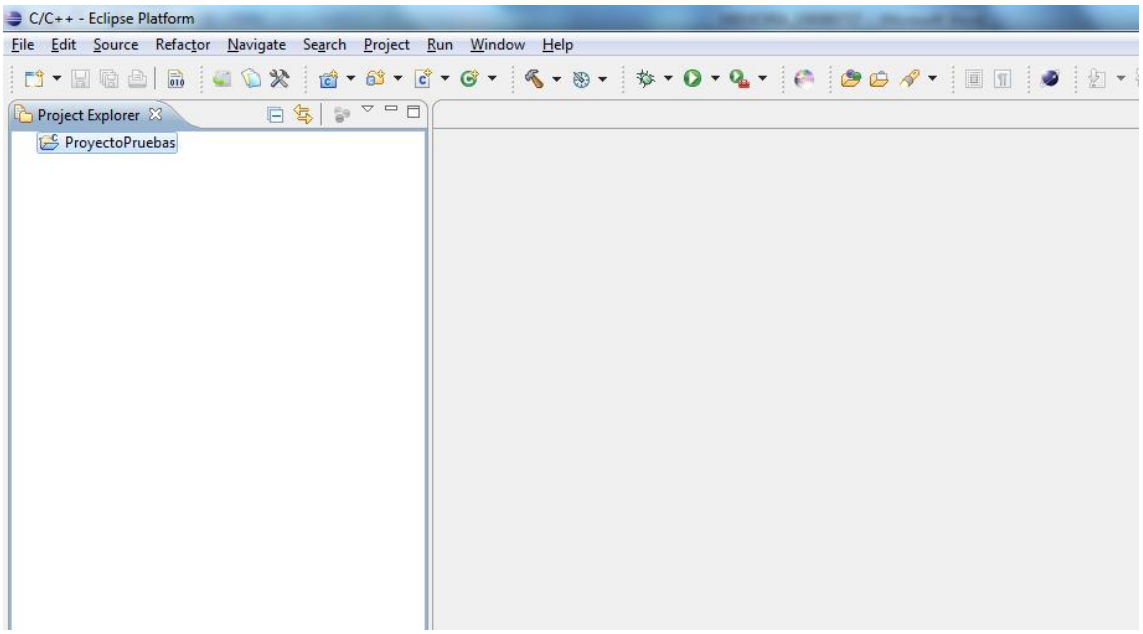


Ilustración 31. Entorno de trabajo tras añadir un proyecto C++.

- Añadimos un nuevo fichero C++ al proyecto creado anteriormente. Para ello seguimos la secuencia: File -> New -> Other -> C/C++ -> Source File.

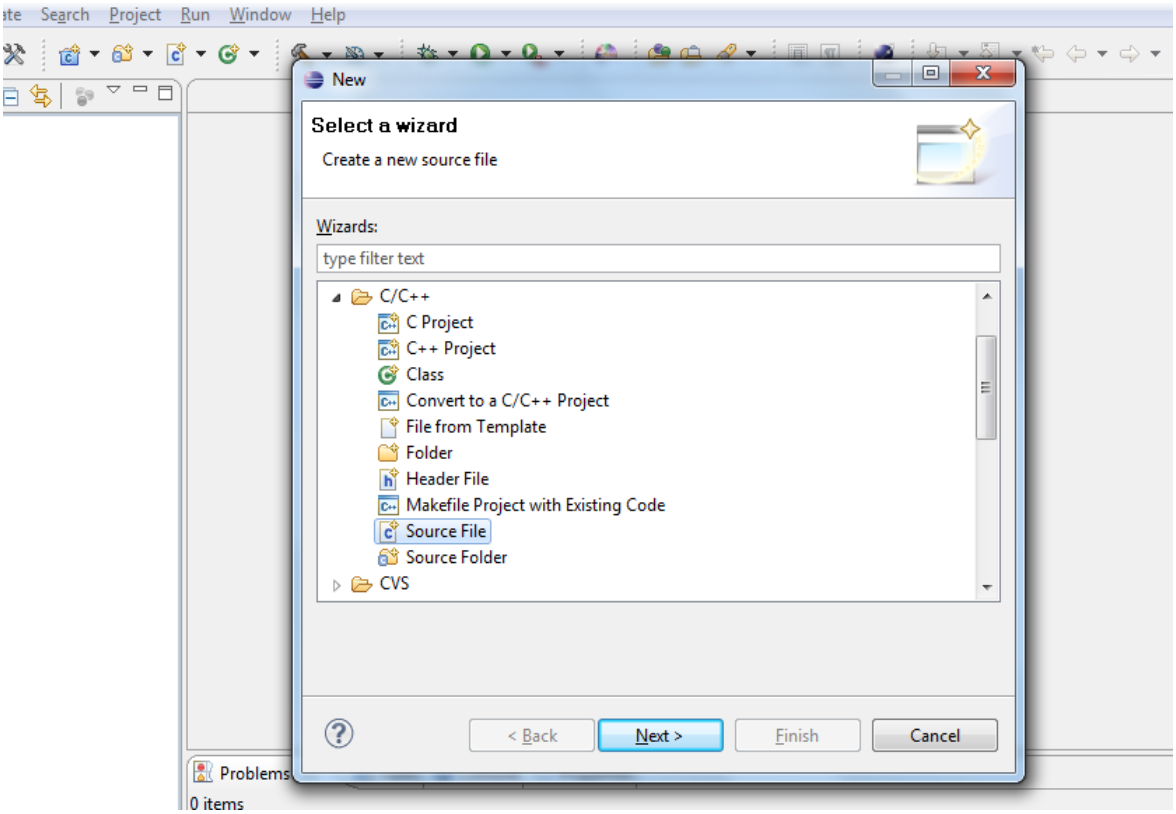


Ilustración 32. Añadimos un fichero C++ al proyecto anterior.

Configuramos el nombre del fichero, por ejemplo, “Ejemplo_1.cpp” e indicamos que se trata de un archivo C++ y pulsamos el botón **Finish.** Se añadirá el fichero al espacio de trabajo sobre el proyecto anteriormente creado, quedando el espacio de trabajo de la siguiente manera:

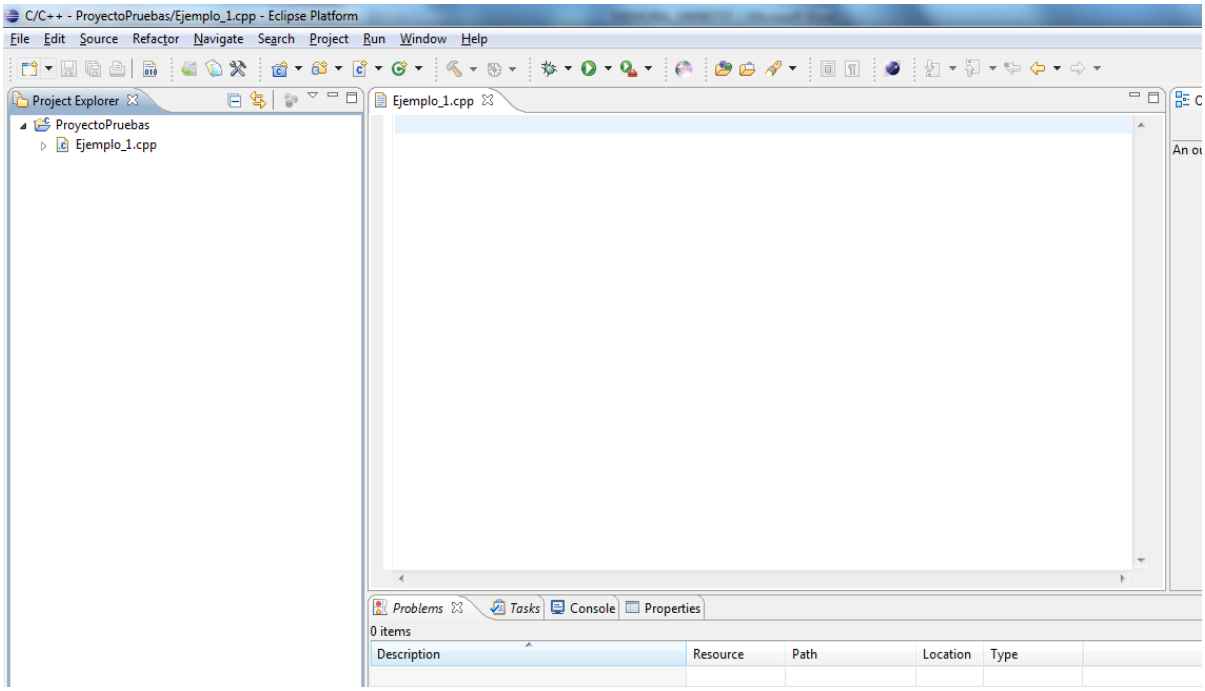


Ilustración 33. Entorno de trabajo tras añadir un fichero C++.

- Añadimos contenido al fichero C++ para poder realizar acciones de refactorización sobre el fichero y poder así aplicar transformaciones a C++11.

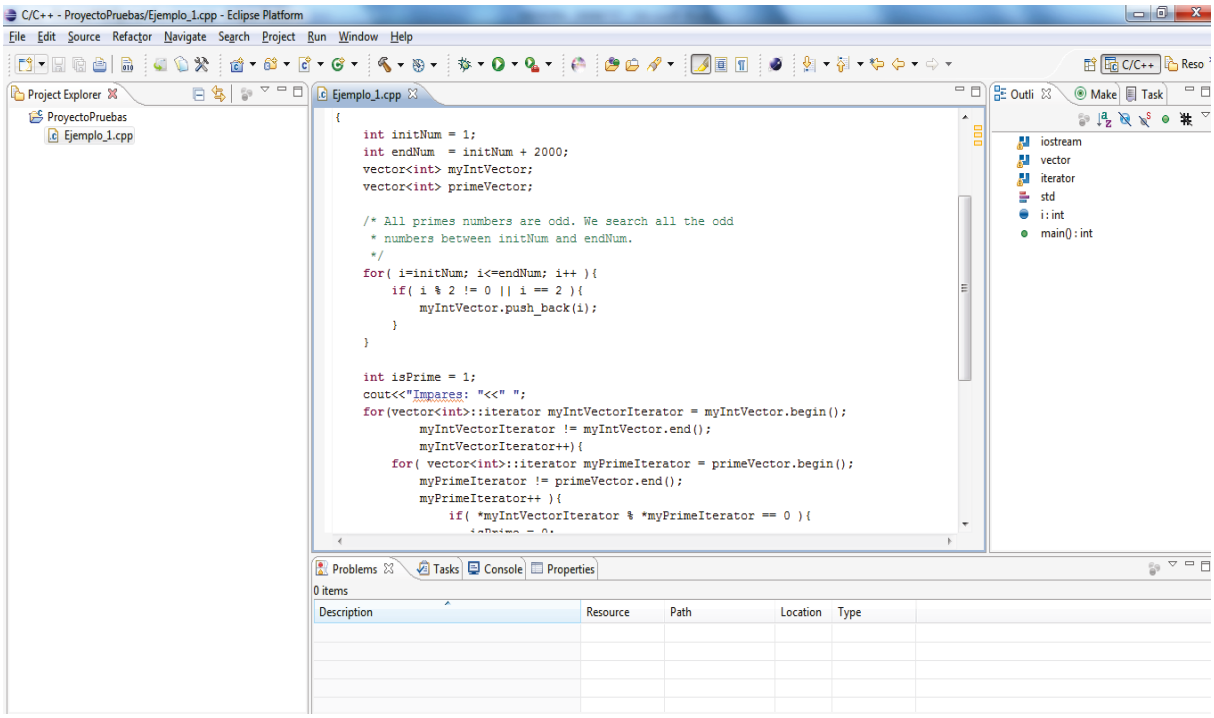


Ilustración 34. Añadimos contenido al fichero C++.

- Una vez configurado el entorno de trabajo y añadido contenido C++ al fichero creado, podremos empezar a realizar operaciones sobre el fichero. En primer lugar, como hemos contado antes, explicaremos las operaciones que pueden realizarse sobre un fichero abierto en el editor de trabajo utilizando el conjunto de acciones rápidas añadido al "toolbar" de eclipse. Las acciones que pueden realizarse son las siguientes:

Compilar

La acción de **Compilar** se ejecutará cuando pulsemos sobre el botón cuyo icono es  . La acción de compilar implica lanzar el motor de transformaciones creado.

El motor de transformaciones se encargará de obtener el modelo del árbol abstracto AST del contenido del fichero C++. Este árbol AST será analizado por la plantilla de trasformaciones y aplicará los cambios que puedan realizarse sobre el fichero de origen obteniéndose como resultado una cadena de texto con el contenido C++11 que tendrá en nuevo fichero de texto.

Esta acción es una acción automática en la que el usuario no puede elegir la ruta de destino del nuevo fichero "**.cpp**" con contenido C++11, por lo que se generará de forma automática un fichero con el mismo nombre del fichero de origen en la ruta **<nombre_proyecto>/cpp11-gen/<nombre_fichero>.cpp**.

En nuestro caso, la ruta de destino del nuevo fichero C++ con el contenido refactorizado a C++11 será **ProyectoEjemplo/cpp11-gen/Ejemplo_1.cpp.**

El resultado obtenido, tras pulsar el botón de **Compilar** será el siguiente:

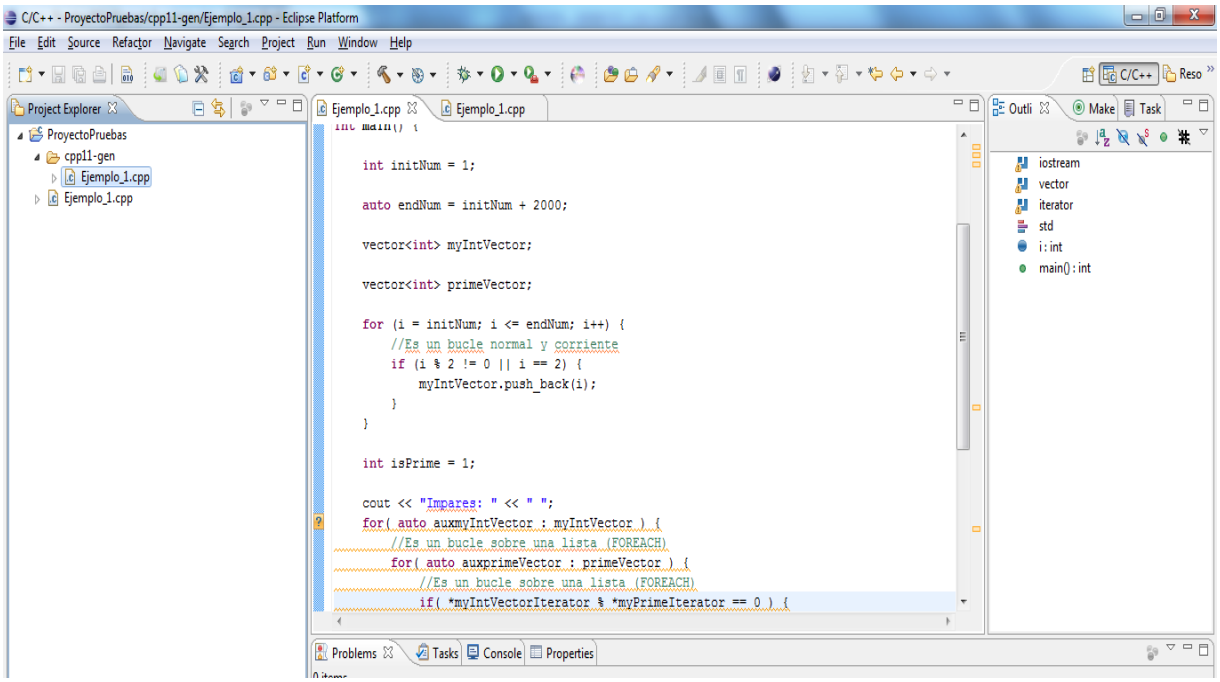


Ilustración 35. Resultado obtenido tras realizar la acción de "Compilación".

Nuestro nuevo fichero estará accesible en la ruta anteriormente indicada desde cualquier parte de nuestro espacio de trabajo.

Si nuestro espacio de trabajo tuviese otro tipo de ficheros que no pudiesen compilarse, por ejemplo un fichero de tipo "**.java**", al pulsar sobre la acción de **Compilar**, se nos mostrará un diálogo informativo que indicará que el fichero que tratamos de refactorizar no es de tipo C++ y por tanto no es posible realizar la acción de refactorización.

El resultado obtenido en esos casos, sería el siguiente:

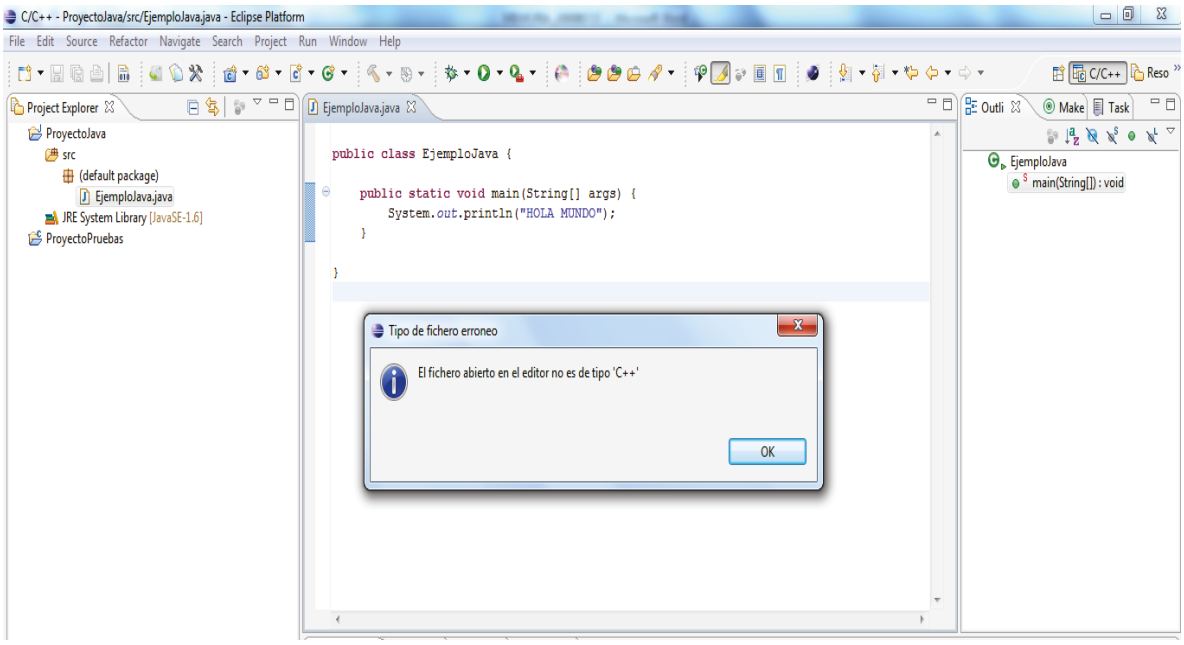


Ilustración 36. Mensaje de error al intentar realizar la acción de "Compilación" sobre un fichero ".java".

Compilar en…

La acción de **Compilar en…** se ejecutará cuando pulsemos sobre el botón cuyo icono es  . Esta acción implica lanzar el motor de transformaciones que se ha creado en este proyecto y almacenar el contenido resultante en un directorio fuera del espacio de trabajo de Eclipse, seleccionado por el usuario.

Para ello, lo primero que nos aparecerá al pulsar sobre el botón correspondiente a esta acción es un cuadro de diálogo, pidiendo al usuario que inserte la ruta de destino de creación del fichero C++11 resultante del proceso de transformación. El cuadro que se muestra es el siguiente:

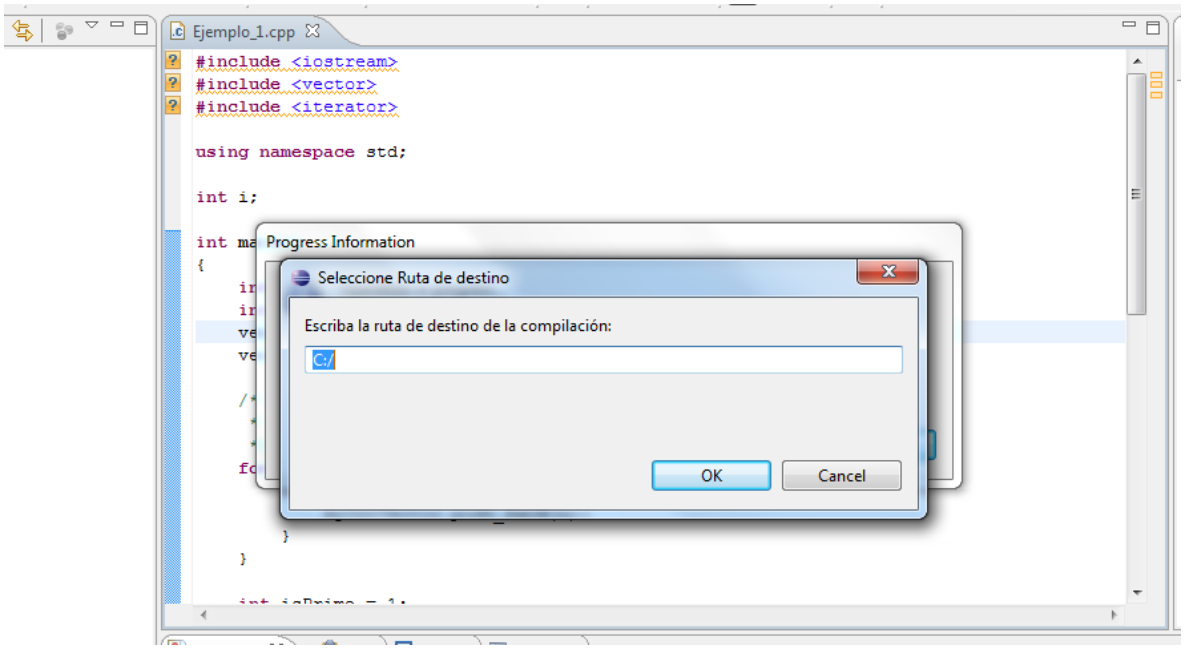


Ilustración 37. Diálogo de solicitud al usuario de la ruta de destino de los ficheros C++11.

La ruta de compilación de destino que introduzca el usuario debe ser completa, incluyendo el nombre del fichero con su extensión, pudiéndose generar y almacenar el contenido en un fichero "**.cpp**", en un fichero "**.txt**", o con la extensión que lo quiera el usuario, del tipo:

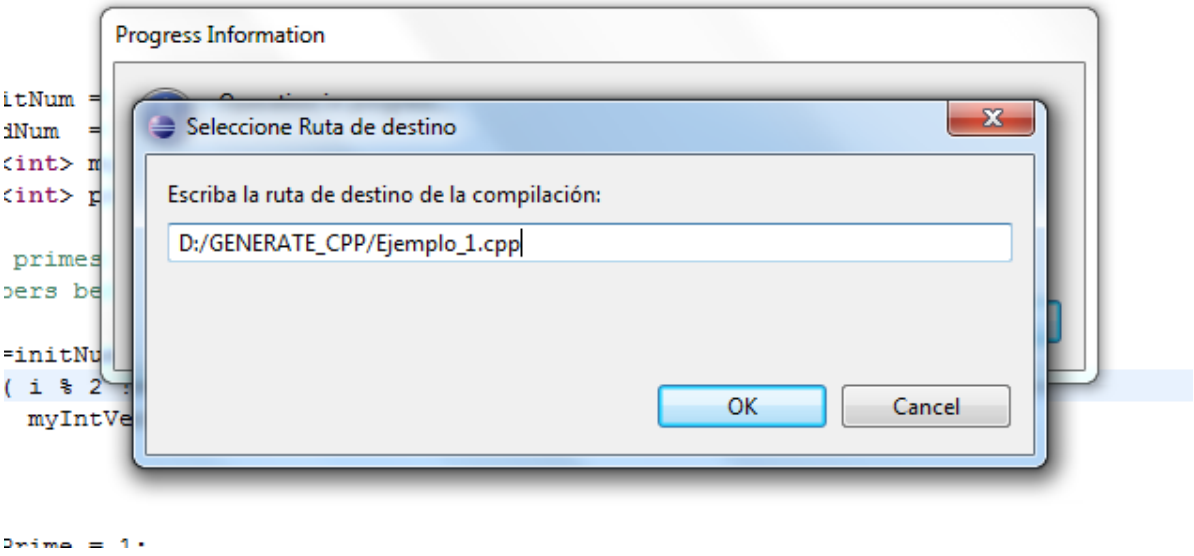


Ilustración 38. Ejemplo de ruta de destino completa y válida.

Una vez introducida la ruta de destino, si la ruta no es vacía, comenzará el proceso de refactorización.

El motor de transformaciones se encargará de obtener el modelo del árbol abstracto AST del contenido del fichero C++. Este árbol AST será analizado por la plantilla de trasformaciones y aplicará los cambios que puedan realizarse sobre el fichero de origen obteniéndose como resultado una cadena de texto con el contenido C++11 que tendrá en nuevo fichero de texto.

El resultado obtenido del proceso de transformación se almacenará en un fichero que se creará en la ruta de destino que introdujo el usuario al comenzar el proceso.

El resultado obtenido, podrá comprobarse en el fichero creado en la ruta de destino, de tal forma:

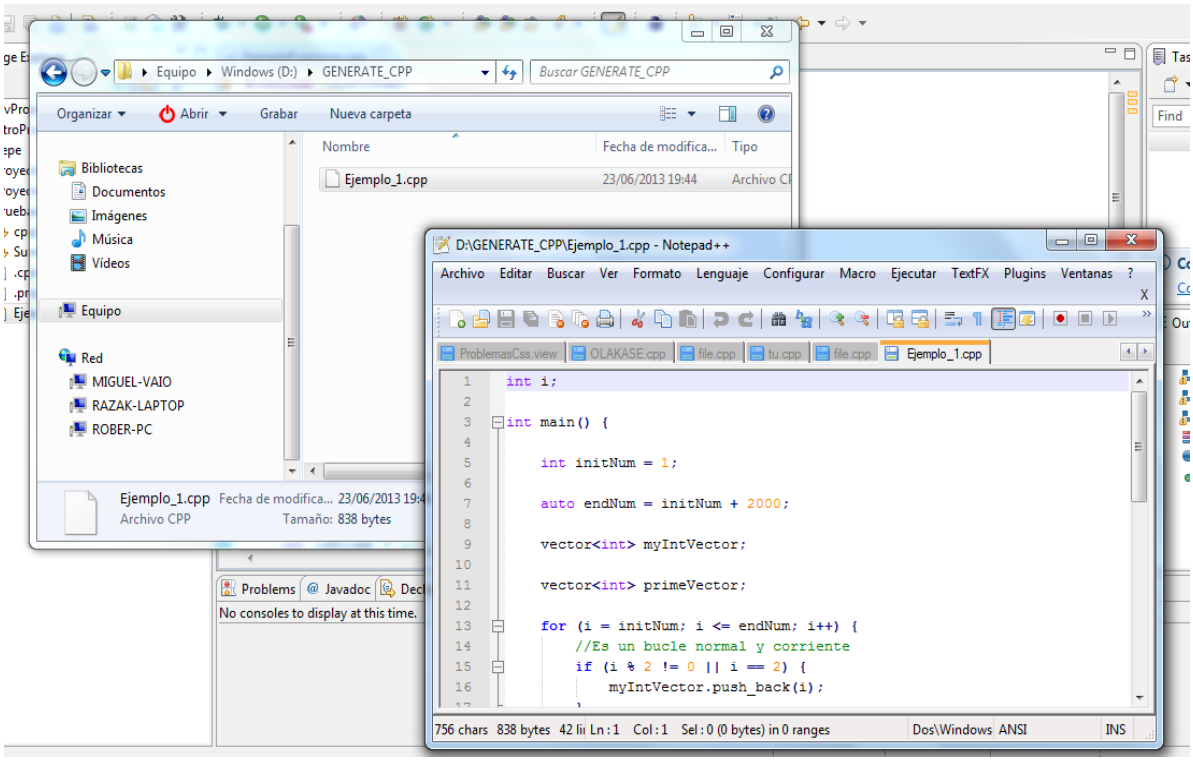


Ilustración 39. Resultado obtenido tras realizar la acción de "Compilar en..." con una ruta de destino elegida por el usuario.

Al igual que en el caso anterior, si tratamos de ejecutar la acción de **Compilar en…** con un fichero abierto en el editor que no sea del tipo valido, se mostrará una ventana informativa indicando que el tipo de fichero del editor no es un tipo válido para ejecutar el sistema de transformaciones.

El resultado obtenido sería el siguiente:

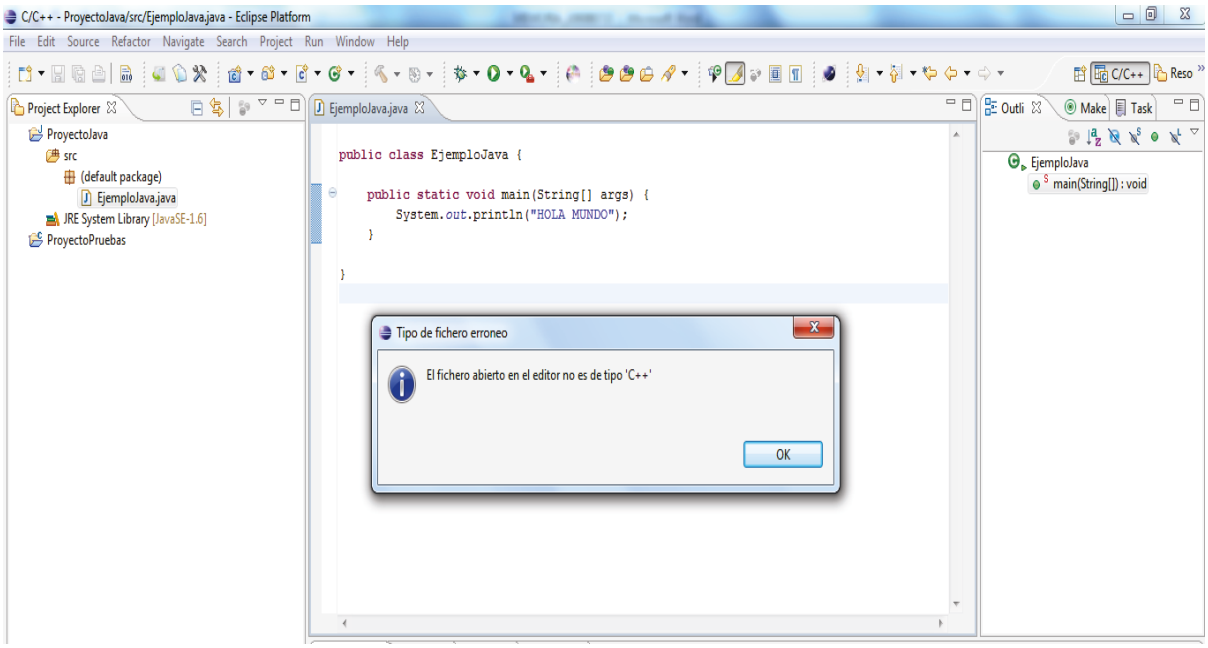


Ilustración 40. Mensaje de error al intentar realizar la acción de "Compilar en..." sobre un fichero ".java".

Comparar

La acción de **Comparar** se ejecutará cuando pulsemos sobre el botón cuyo icono es  .

Esta acción, nos permite abrir un editor de comparación de eclipse. Este diálogo de comparación de eclipse nos mostrará en el lado izquierdo del editor el contenido original del fichero C++ y en la parte derecha del editor el contenido C++11 procedente del proceso de refactorización.

Para rellenar este cuadro de diálogo, lo primero que se hace al pulsar el botón es obtener el fichero C++ abierto en el editor y lanzar el motor de transformaciones con este fichero.

El motor de transformaciones se encargará de obtener el modelo del árbol abstracto AST del contenido del fichero C++. Este árbol AST será analizado por la plantilla de trasformaciones y aplicará los cambios que puedan realizarse sobre el fichero de origen obteniéndose como resultado una cadena de texto con el contenido C++11 que tendrá en nuevo fichero de texto.

Con la cadena de texto del contenido del fichero de origen y con la cadena resultante del proceso de refactorización a C++11, se crea un diálogo de comparación.

El usuario podrá gracias a esta acción, comparar el contenido entre el fichero de origen y el futuro contenido que se obtendría al aplicarle las transformaciones a C++11 correspondientes.

Además podría seleccionar el código que le interese y copiarlo sobre el fichero de origen, llevándose así los cambios que le interesen.

El resultado obtenido, sería un editor de comparación del tipo:

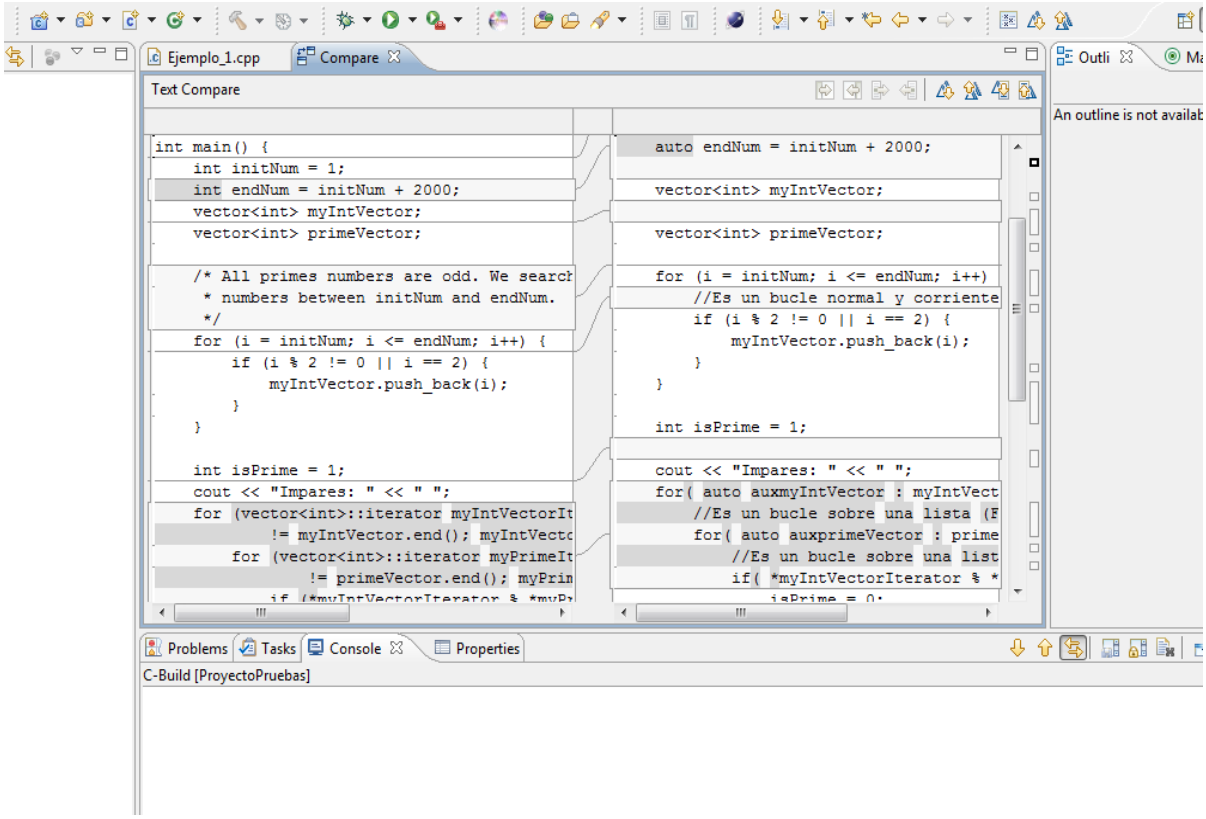


Ilustración 41. Editor de comparación del fichero original C++03 y el fichero refactorizado a C++11.

Al igual que en los dos casos anteriores, si el fichero abierto en el editor no es de tipo C++, se mostrará un diálogo informativo indicando que el fichero abierto en el editor no es del tipo válido y se cancelará la ejecución de la acción y la apertura del editor de comparación.

El resultado obtenido en este caso sería:

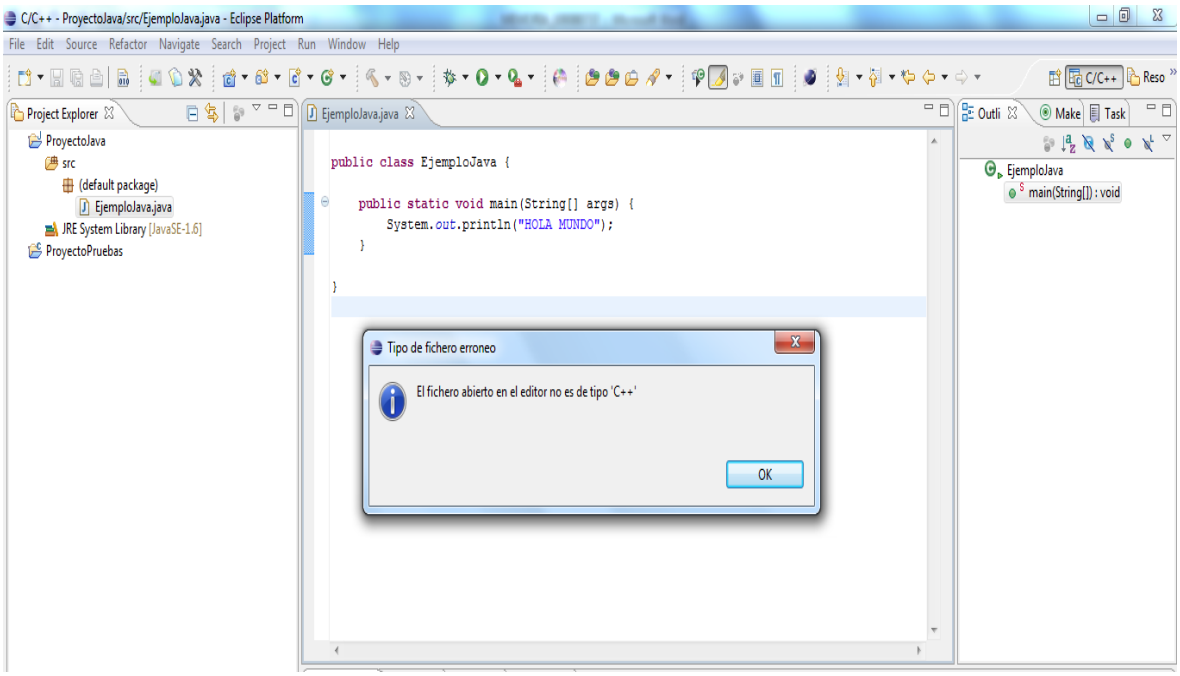


Ilustración 42. Mensaje de error al intentar realizar la acción de "Comparar" sobre un fichero ".java".

Como se ha contado al principio del manual, se han añadido nuevas funcionalidades al entorno de trabajo de eclipse. Para ofrecer diferentes funcionalidades al usuario, y que puedan acceder a ellas y ejecutarlas desde distintos puntos, mejorando así la usabilidad y la funcionalidad del complemento de eclipse creado en este proyecto, se ha contribuido al menú secundario de acciones sobre ficheros y proyectos de tipo C++.

De esta manera, no sería necesario tener abierto el fichero en el editor, para poder realizar un proceso de refactorización a C++11 sobre él.

Las funcionalidades que se ofrecen sobre los ficheros de tipo C++, al igual que en la barra de acciones rápidas de eclipse (el "toolbar" con acciones anteriormente expuesto), son tres: **Compilar, Compilar en..** y **Comparar.** El funcionamiento y el resultado de cada una de ellas es exactamente el mismo que el anteriormente contado, pero con la diferencia de que no es necesario tener el fichero abierto en el editor, sino que puede ejecutarse en segundo plano.

Para añadir estas funciones se ha contribuido al pop-up para archivos de tipo **IFile** de eclipse. Esto nos asegura que todos los elementos del entorno de trabajo que sean de tipo IFile tendrán este conjunto de operaciones al realizar clic con el botón secundario del ratón sobre ellos.

Si realizamos clic con el secundario del ratón sobre un fichero de tipo C++, obtendremos el siguiente resultado:

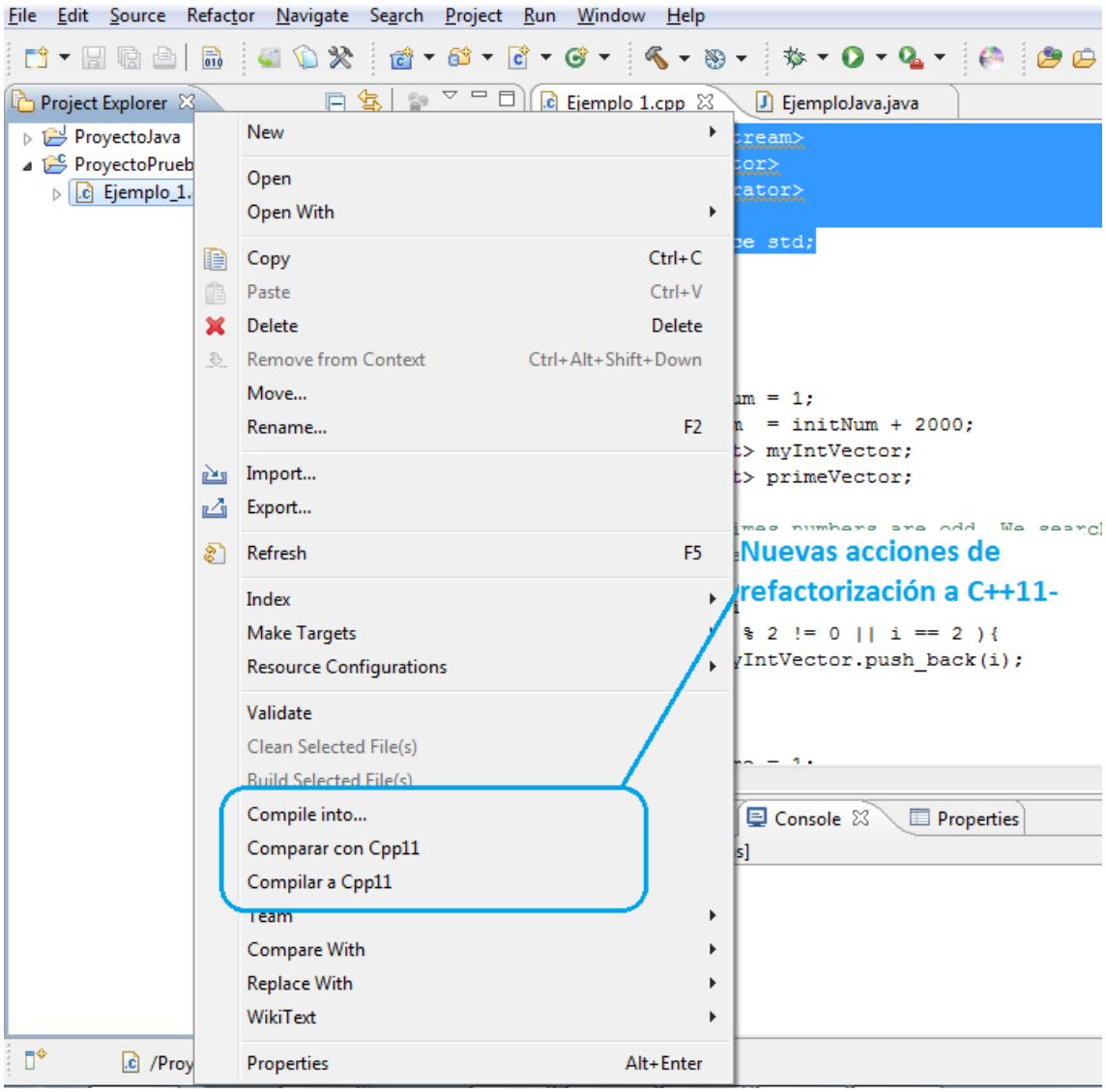


Ilustración 43. Conjunto de acciones añadidas en el menú secundario sobre los ficheros de tipo C++.

Como es obvio, al contribuir al menú pop-up de los archivos de tipo **IFile** de eclipse, estas acciones se mostrarán para todos los archivos de este tipo que se encuentre en nuestro espacio de trabajo. Para eclipse, un archivo de este tipo es cualquier tipo de fichero final con extensión, por lo que por ejemplo, se consideraría de tipo **IFile** los ficheros "**.java**".

Es por esto, que para que solo puedan ejecutarse estas operaciones sobre ficheros de tipo C++, se ha añadido a la funcionalidad un filtro que se encarga de desactivar las acciones cuando el archivo que se selecciona es de cualquier otro tipo que no sea C++.

En caso de que el fichero que se selecciona, no tenga un tipo válido, nos encontraremos ante algo como esto (resultado de clicar sobre un fichero **java**):

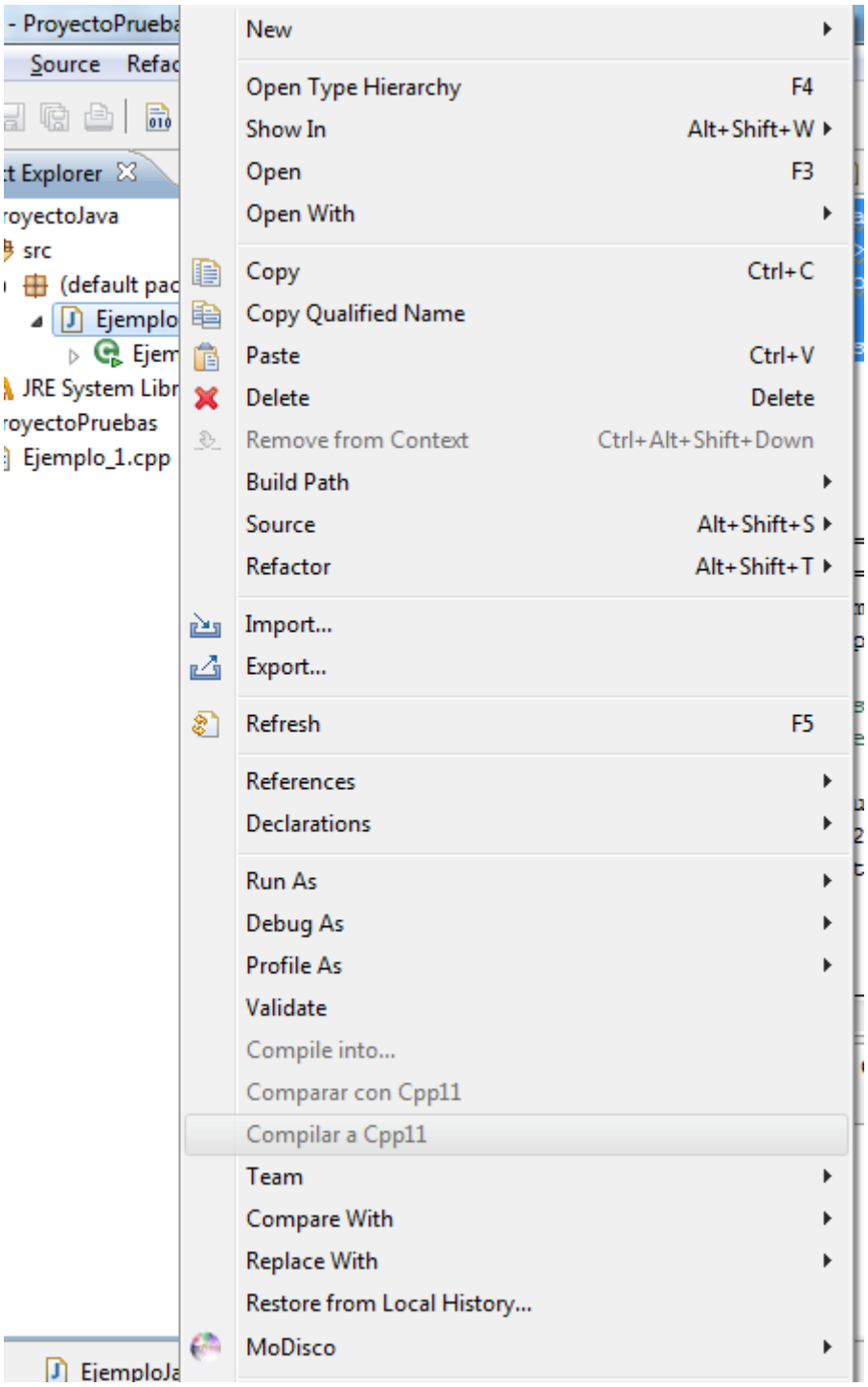
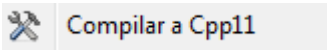


Ilustración 44. Conjunto de acciones deshabilitado si el fichero seleccionado no es tipo C++.

Las acciones que podemos realizar sobre un fichero de tipo C++ con el menú pop-up resultante de pulsar con el botón secundario sobre el fichero en el árbol de navegación de eclipse son:

Compilar

Esta acción se ejecuta cuando pulsamos sobre la etiqueta 

.

Al pulsar esta etiqueta, el proceso de refactorización a C++11 que se lleva a cabo es el mismo que el que hemos contado anteriormente para la barra de acciones rápidas de eclipse. Se generará un fichero con el contenido compilado en la carpeta **cpp11-gen** con el mismo nombre del fichero de origen y con el contenido refactorizado a C++11.

El resultado obtenido es del tipo:

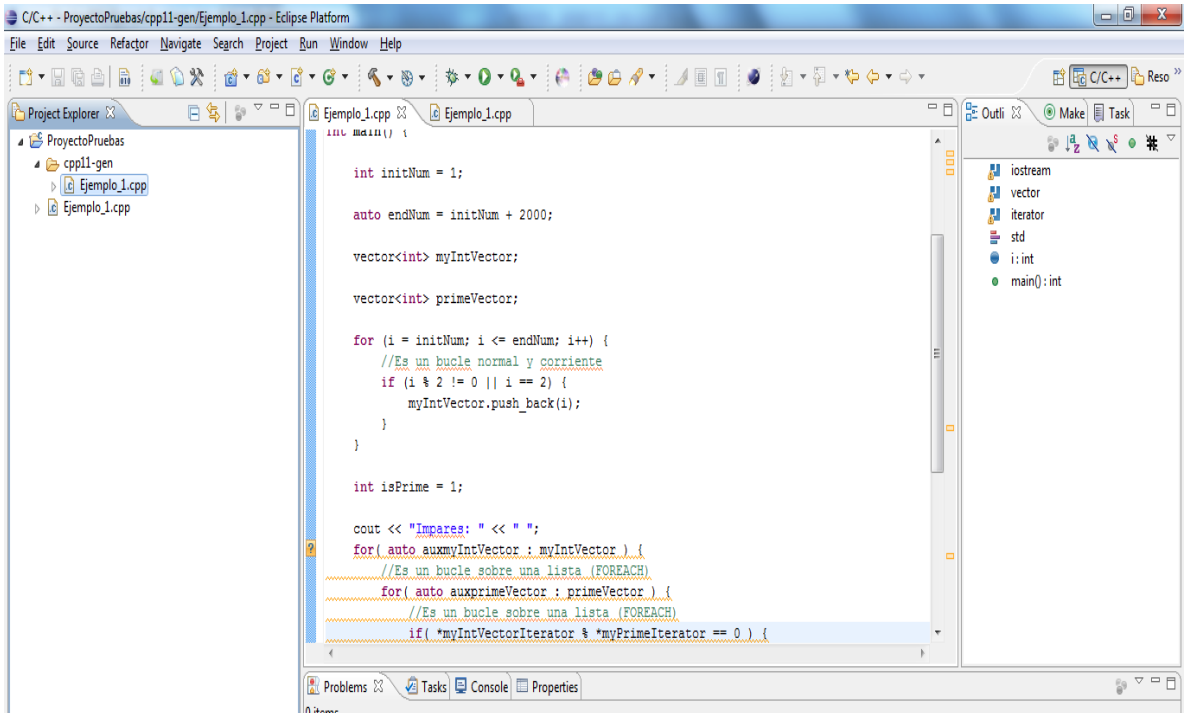
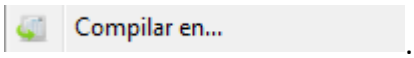


Ilustración 45. Resultado obtenido tras realizar la acción de "Compilación".

Compilar en…

Esta acción se ejecuta cuando pulsamos sobre la etiqueta 

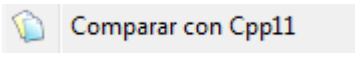
.

Al pulsar esta etiqueta, el proceso de refactorización a C++11 que se lleva a cabo es el mismo que el que hemos contado anteriormente para la barra de acciones rápidas de eclipse en la operación **Compilar en…**

En primer lugar se mostrará el diálogo de selección de ruta de destino. Una vez el usuario indique la ruta de destino para el fichero refactorizado a C++11, se inicia el proceso de transformación del fichero de origen seleccionado

Se generará un fichero con el contenido C++11 resultante de la transformación del fichero de origen en la ruta especificada.

Comparar

Esta acción se ejecuta cuando pulsamos sobre la etiqueta 

.

Al pulsar esta etiqueta, la acción ejecutada es la misma que la correspondiente **Comparar** de la barra de acciones rápida que hemos creado y hemos contando anteriormente.

Se realizará el proceso de refactorización a C++11 con el fichero de origen. Se generar dos cadenas de texto, la primera con el contenido del fichero de origen y la segunda con el contenido C++11 resultante del proceso de transformación. Ambas cadenas de texto se mostrarán en un editor de comparación de eclipse, y el usuario podrá decidir si quiere llevarse el contenido al fichero de origen o no.

El resultado de la operación será el siguiente:

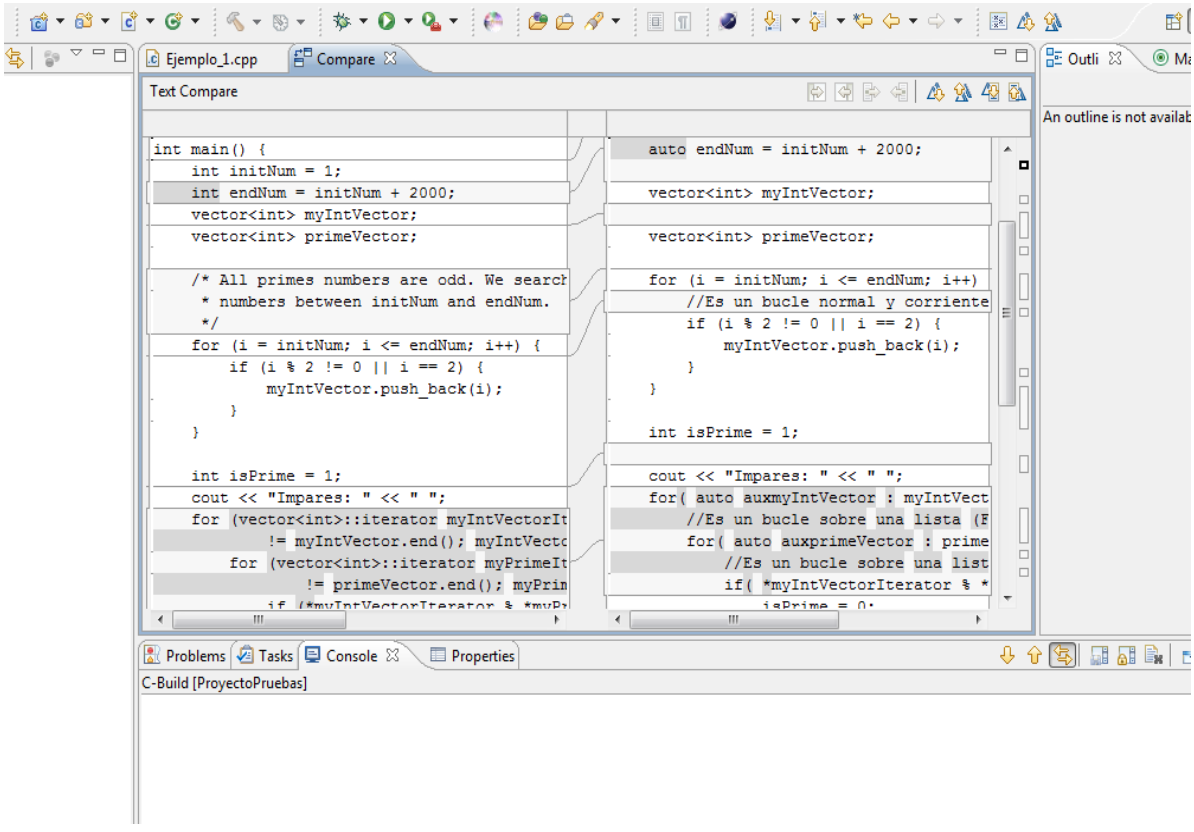


Ilustración 46. Editor de comparación del fichero original C++03 y el fichero refactorizado a C++11.

Por último, se han realizado contribuciones también al menú pop-up que aparece al realizar clic con el botón secundario del ratón sobre proyecto de tipo C++. Pretendemos añadir una nueva funcionalidad más a los proyectos C++ para poder añadirle una naturaleza propia para poder crear así nuestro propio compilador automático.

Añadir naturaleza

Añadir una naturaleza, no es otra cosa que añadir una marca a nuestros proyectos C++, gracias a la cual, cada vez que se produzca un cambio en este proyecto, nosotros podremos analizarlo e interpretarlo. Si se realiza un cambio en el que un fichero del proyecto se salva, podremos crear un "builder" propio que ejecute de forma automática el proceso de refactorización para el fichero que se ha modificado. De esta manera, automatizaríamos por completo el proceso de refactorización para todos los ficheros C++ que están en ese proyecto.

La nueva acción añadida es la siguiente:

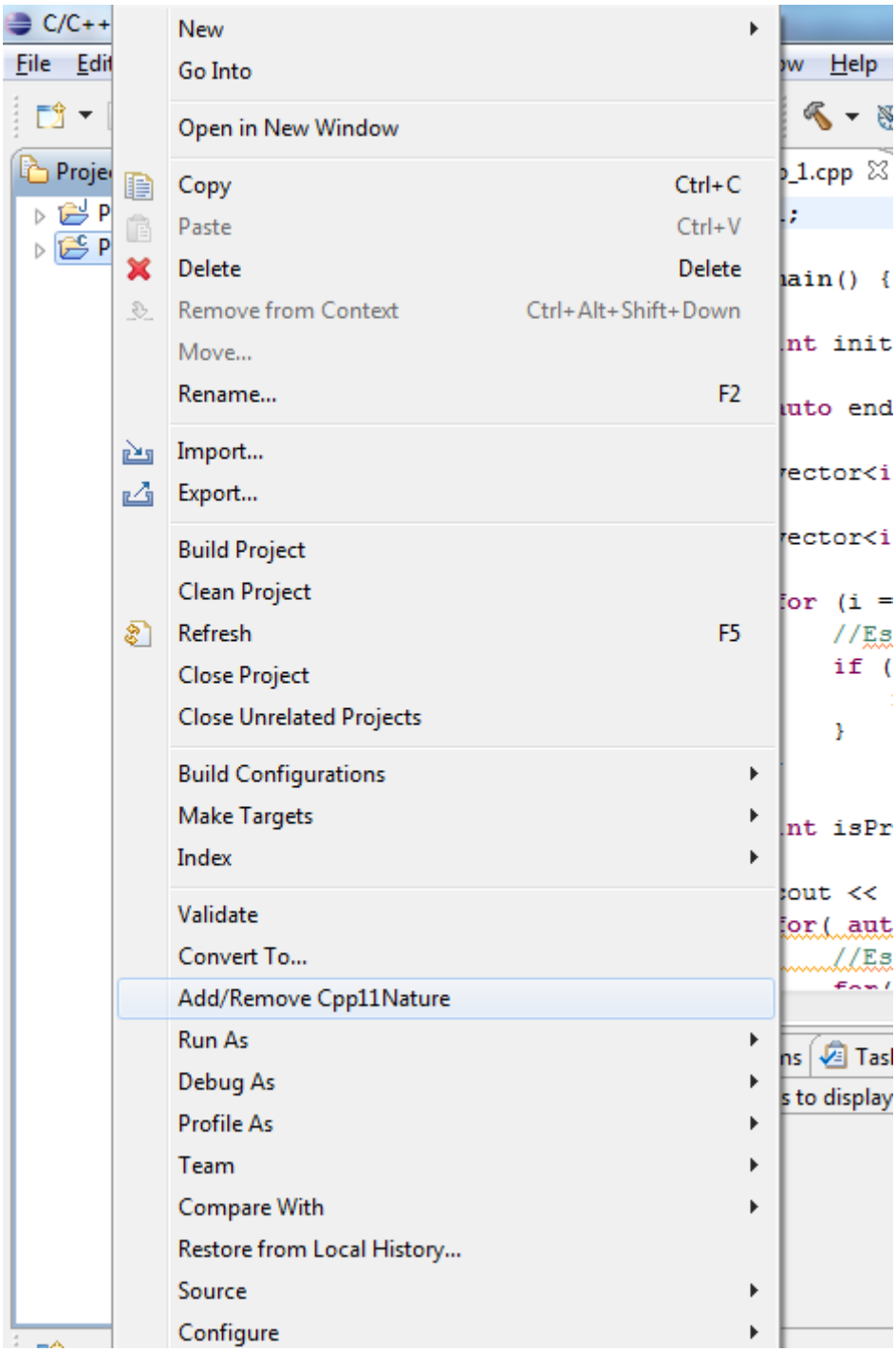


Ilustración 47. Acción de añadir naturaleza que se muestra sobre los proyecto de tipo C++.

Si abrimos el fichero ".project" de un proyecto C++, nos encontramos con formato XML en el que encontraremos las naturalezas de los proyectos C++. Cada una de esas naturalezas puede o no estar asociadas a un constructor (Builder) que realice operaciones de forma automática. El fichero contendrá algo como esto:

```
<natures>
    <nature>org.eclipse.cdt.core.cnature</nature>
    <nature>org.eclipse.cdt.core.ccnature</nature>
    <nature>org.eclipse.cdt.managedbuilder.core.managedBuildNature</nature>
    <nature>org.eclipse.cdt.managedbuilder.core.ScannerConfigNature</nature>
</natures>
```

Ilustración 48. Naturalezas por defecto en el fichero ".project" de un proyecto C++.

Como podemos observar, el fichero por defecto no contiene nuestra naturaleza propia, por lo que si modificamos ficheros de este proyecto, no se ejecutará el constructor que refactorice los ficheros C++ que contenga este proyecto.

Al pulsar sobre la acción que indicamos arriba, se realiza una operación para añadir la naturaleza o eliminarla en caso de que la tuviese ya, resultando algo como esto en el fichero ".project" del proyecto C++:

```
<natures>
    <nature>org.eclipse.cdt.core.cnature</nature>
    <nature>org.eclipse.cdt.core.ccnature</nature>
    <nature>org.eclipse.cdt.managedbuilder.core.managedBuildNature</nature>
    <nature>org.eclipse.cdt.managedbuilder.core.ScannerConfigNature</nature>
    <nature>com.uc3m.atovar.menu.Cpp11Nature</nature>
</natures>
```

Ilustración 49. Naturalezas del fichero ".project" tras añadirle la nueva "Cpp11Nature".

Ahora el proyecto sí que tiene la naturaleza Cpp11Nature, nuestra propia y personalizada. Al añadir esta naturaleza, se le está asignado un constructor automático por detrás que se ejecuta cada vez que se modifica un fichero C++ contenido en ese proyecto.

Gracias a ese "builder" automático, cada vez que salvamos un fichero C++, se está lanzando el motor de transformaciones de C++ a C++11 y por tanto se están generando continuamente los ficheros transformados en la carpeta del proyecto **cpp11-gen** con los nombres de los ficheros C++.

Si tenemos un proyecto con esta naturaleza y un editor abierto y sucio (tiene el * en la pestaña del editor) con un fichero C++ como el siguiente:

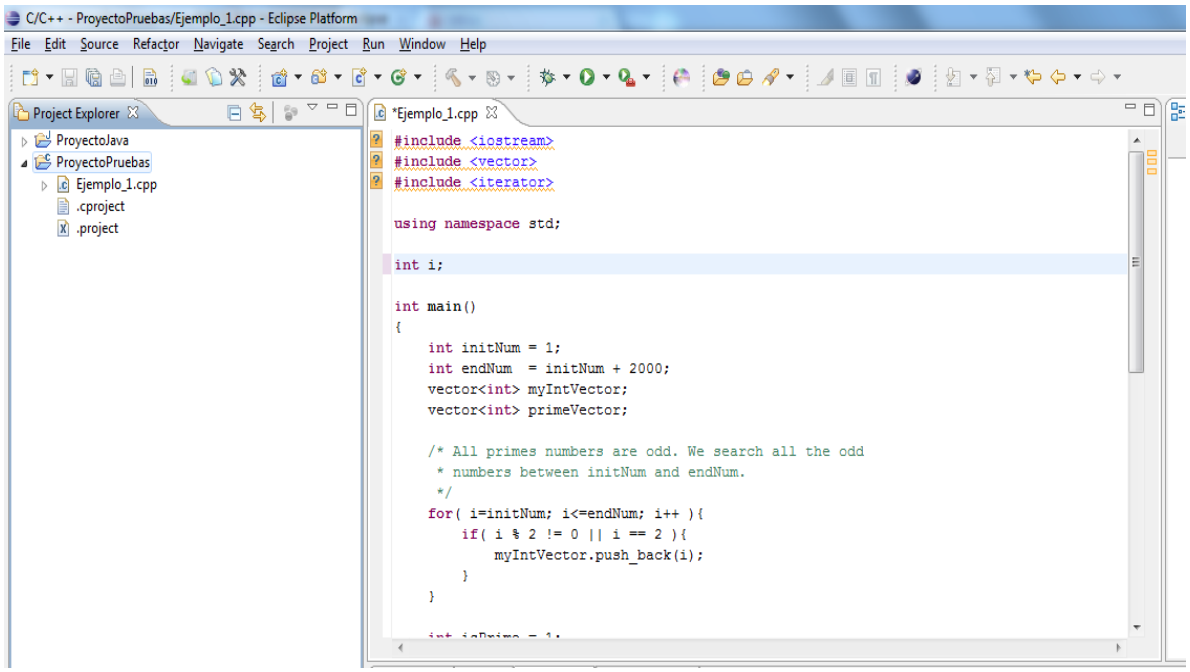


Ilustración 50. Muestra de un editor con contenido C++ sin salvar el fichero.

Y salvamos el editor, el resultado obtenido será el siguiente:

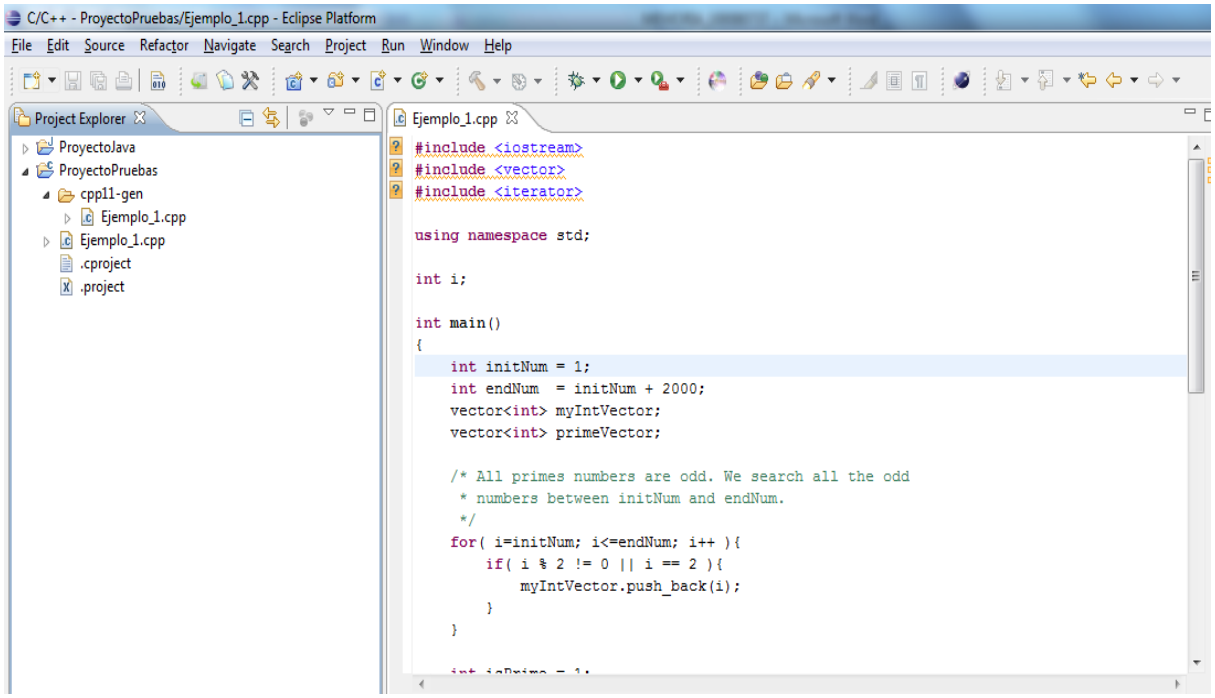


Ilustración 51. Estado final del proyecto tras salvar el fichero.

Gracias a esta funcionalidad, podemos darle la libertad al usuario de olvidarse de que tiene que estar transformando manualmente todos los ficheros que vaya creando, ya que gracias al compilador automático asociado a la naturaleza añadida al proyecto, se reconstruirán los ficheros C++ del proyecto de forma automática cada vez que se realizan cambios.

Anexo II. Manual de modificación de plantillas.

Uno de los principales beneficios que hemos contado que ofrecen las arquitecturas MDA y que nosotros hemos tratado de aprovechar en nuestro proyecto es la libertad de realizar cambios en la plantilla de transformaciones sin tener que realizar cambios en la arquitectura del proyecto.

Se ha tratado de extraer al máximo la plantilla de transformación del resto del proyecto, por ello, se han realizado dos plug-in o complementos para eclipse dependientes entre sí.

El primero de los complementos es el que contiene todas las acciones de la barra de acciones rápida ("toolbar") de eclipse y las de los menús pop-up de los ficheros y proyectos C++.

Además de las acciones, este plug-in contiene todo lo relacionado con la arquitectura MDA del complemento de transformación, y por tanto contiene dos de los tres principales puntos de la arquitectura, el extractor del modelo y el generador de ficheros de salida.

El extractor de modelos es el encargado de obtener el árbol abstracto AST del fichero C++ que se pretende refactorizar. Este árbol AST será el que nuestra plantilla de transformación se encargue de analizar y sobre el que aplicará los cambios que se considere necesario en base a las reglas de transformación que contiene.

El segundo elemento contenido en este complemento es el generador de ficheros. Este generador de ficheros será quien genere el fichero resultado con el contenido C++11 una vez la plantilla de transformación haya realizado los cambios oportunos sobre el contenido de origen.

Para ofrecer la libertad al usuario de modificar la plantilla de transformación o incluso cambiarlo por las suyas propias o añadir nuevas plantillas de transformación al proyecto, se ha extraído la plantilla de transformaciones del resto de la arquitectura y se ha implementado en un plug-in independiente con el nombre **com.uc3m.atovar.xtext.cpp**.

Este nuevo plug-in no es más que un proyecto de tipo XTEND2 con una plantilla de transformaciones con el nombre **Uc3mAtovarCppGeneratorTemplate.xtend.** Este fichero es la plantilla de transformaciones del motor de transformación que ofrecen en conjunto estos dos "plug-in".

El contenido de esta plantilla es similar a este:

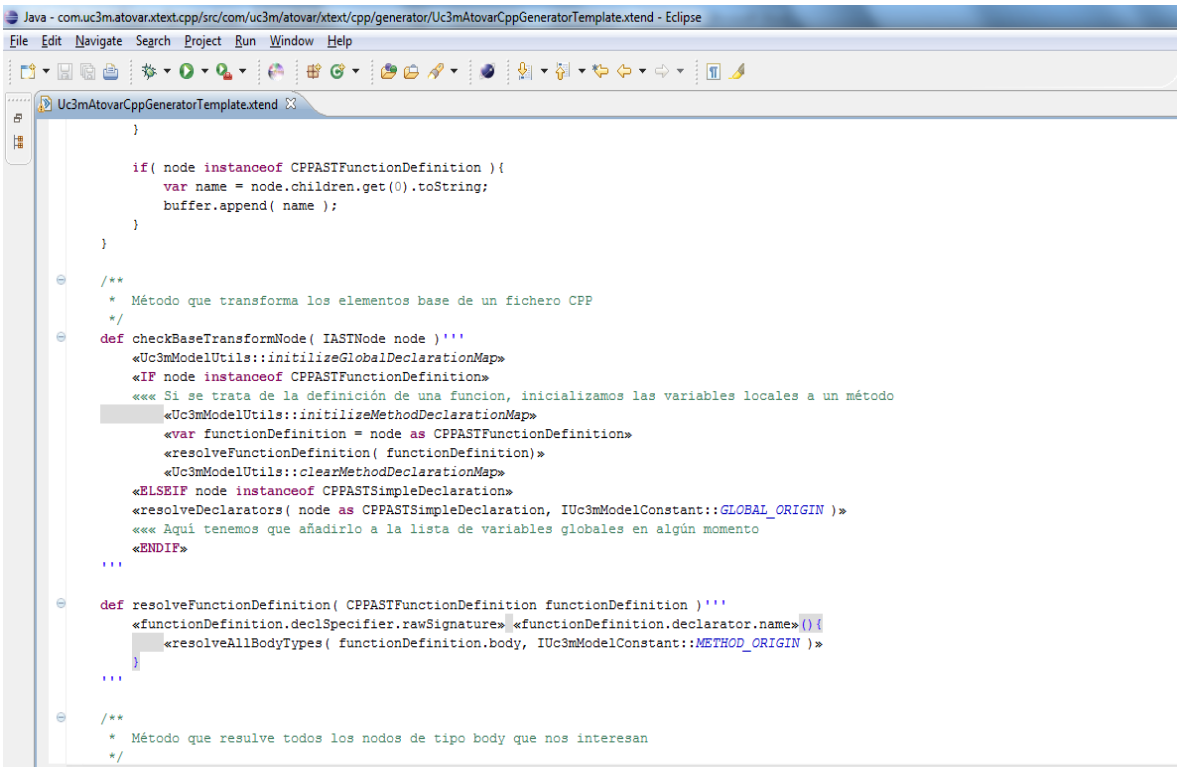


Ilustración 52. Ejemplo de plantilla de transformación XTEND.

Este código es el que se encarga de analizar cada uno de los nodos del árbol AST y aplicarle las transformaciones que no interesen, por lo que si se desea añadir o modificar transformaciones al sistema de refactorización que actualmente existe bastaría con modificar la plantilla y añadir nuevas transiciones que analicen los nodos AST sobre los que deseemos añadir nuevas transformaciones.

Actualmente, no existe ningún sistema que permita modificar las plantillas en caliente y añadirlas al sistema transformador. Una posible mejora futura, sería crear un proyecto con la plantilla de transformación que se está aplicando y crearle una naturaleza y un builder al igual que tenemos para transformar de forma automática C++03 a C++11 de tal manera que al modificar la plantilla se salve automáticamente en el fichero ".jar" del plug-in que contiene las plantillas (**com.uc3m.atovar.xtext.cpp**).

Puesto que este sistema actualmente no está disponible, si lo que queremos es modificar las plantillas de transformaciones deberíamos en primer lugar coger la plantilla que se está aplicando sobre el entorno de trabajo Eclipse. Para ello accedemos a la carpeta **plugins** del directorio dónde está el ejecutable de Eclipse. En nuestro caso en la ruta **D:\PFG\eclipse\plugins**:

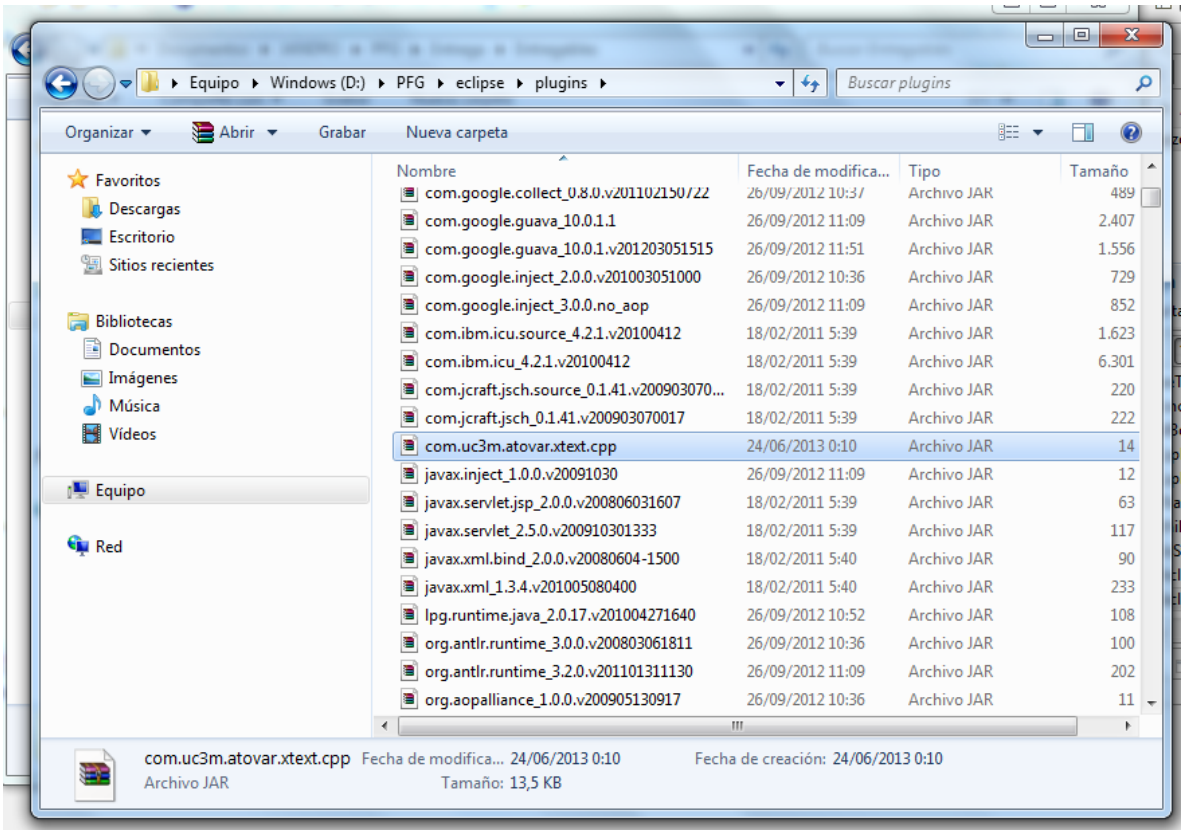


Ilustración 53. Muestra los complementos instalados en el IDE Eclipse. Se trata de un listado de ficheros ".jar". El que contiene la plantilla es "com.uc3m.atovar.xtext.cpp".

Si abrimos este fichero ".jar" nos encontraremos el siguiente contenido:

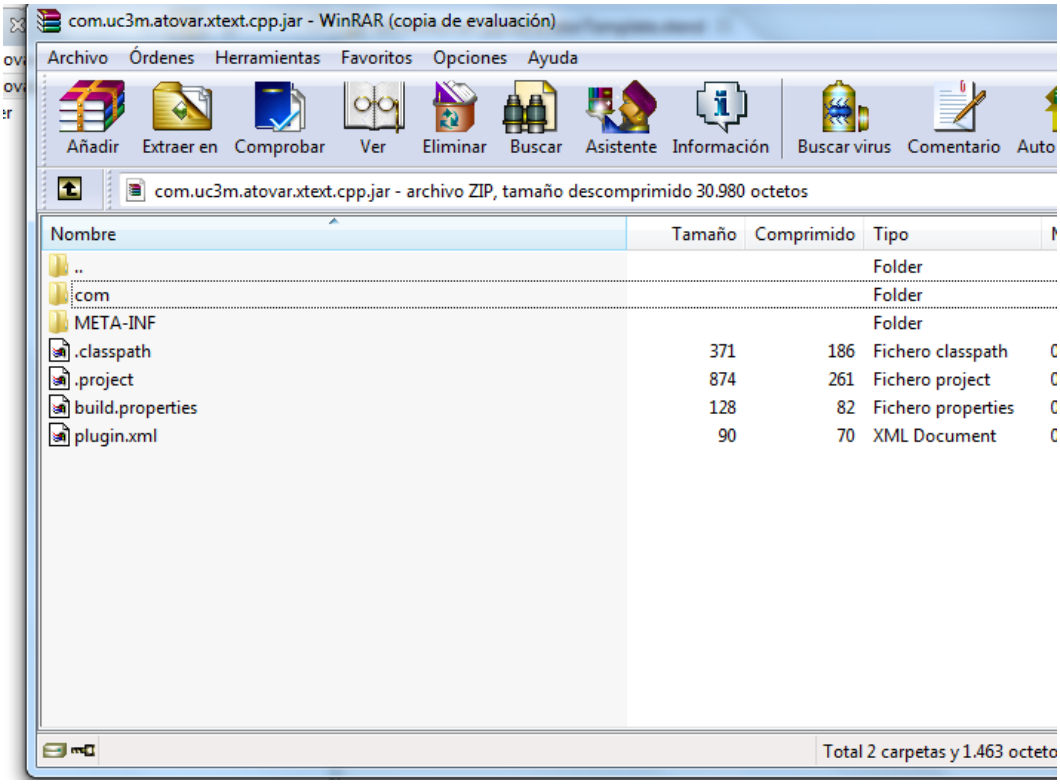


Ilustración 54. Contenido del archivo ".jar" de las plantillas.

Si recorremos la ruta **com.uc3m.atovar.xtext.cpp.generator** nos encontraremos nuestra plantilla de transformaciones **Uc3mAtovarCppGeneratorTemplate.xtend.**

Si copiamos esta plantilla en un proyecto XTEND en nuestro Eclipse, podremos editarlo y modificarlo a nuestro gusto.

Cada vez que se modifica la plantilla se genera de forma automática un fichero ".java" en la ruta **xtend-gen** con el mismo nombre de paquete y el mismo nombre que la plantilla de transformaciones. Bastaría con sustituir en el fichero ".jar" los ficheros **Uc3mAtovarCppGeneratorTemplate.xtend/Uc3mAtovarCppGeneratorTemplate.class** por los nuevos ficheros con los mismos nombres generados en el entorno eclipse al modificar la plantilla.

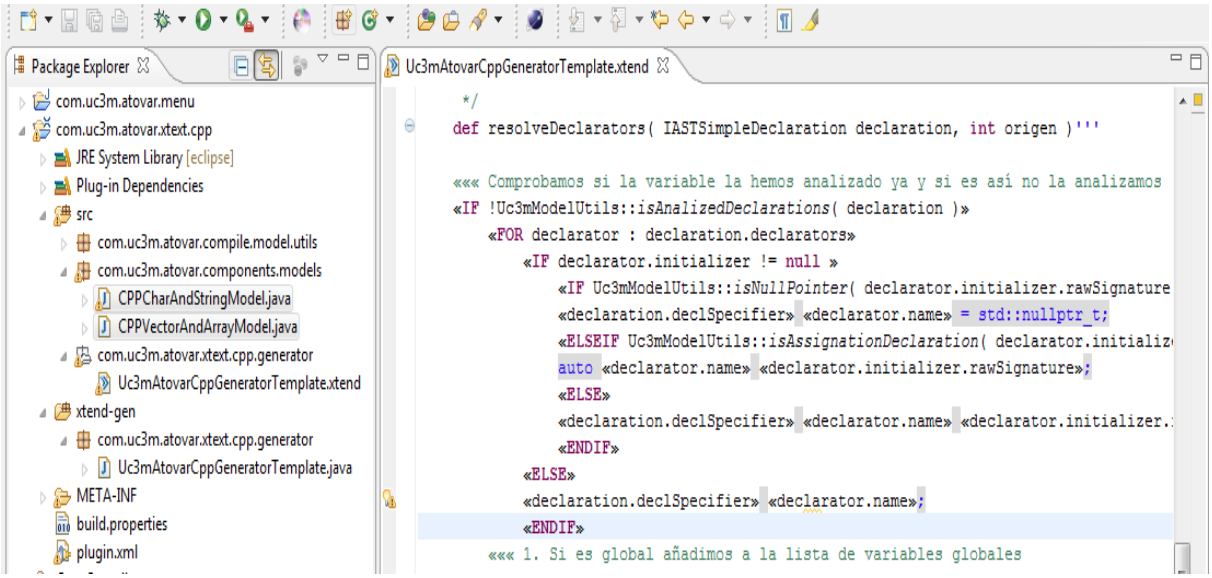


Ilustración 55. Ejemplo de modificación de plantillas de transformación.

Si reiniciamos el entorno de trabajo Eclipse podremos realizar transformaciones de C++03 a C++11 con las nuevas transformaciones añadidas.

Si no se desea trabajar con la misma plantilla de transformaciones y se prefiere añadir una plantilla totalmente nueva, bastaría con renombrar la nueva plantilla creada con los nombres anteriormente expuestos y sustituir los ficheros **.class** y **.xtend** originales del **jar** por los nuevos ficheros.

REFERENCIAS BIBLIOGRÁFICAS

[1] Model-Driven Architecture: Vision, Standars and Emergin Technologies. John D. Poole-. Abril 2001.

[2] Catálogo de refactorizaciones para transformaciones de Modelo a Modelo. Salvador Martínez, Manuel Wimmer, Frédéric Joualult, Jordi Cabot.

[3] Ingeniería de modelos con MDA. Jesús Rodríguez Vicente. Junio 2004.

[4] Traceability in Model to Text Transformation. Jon Oldevik, Tor Neple.

[5] Overview of The New C++ (C++0x). Scott Meyers. Abril, 2010.

[6] Acceleo Code Generation. Stéphane Bégaudeau.

[7] EMF, XPAND/XTEND. Diego Sevilla Ruiz. DITEC facultad de informática. Noviembre, 2010.

[8] Un método de desarrollo dirigido por modelos de arquitectura para aplicaciones web. Santiago Meliá.

[9] Aplicación de ingeniería dirigida por modelos (MDA), para la construcción de una herramienta de modelado de dominio específico (DSM) y la creación de módulos en sistemas de gestión de aprendizaje (LMS) independientes de la plataforma. Carlos Enrique Montenegro Marín, Paulo Alonso Gaona García, Juan Manuel Cueva Lovelle, Oscar Sanjuan Martínez.

[10] Model transformation with Operational QVT. Radomil Dvorak.

[11] An introduction to the MOF 2.0 QVT standard with focus on the operational Mappings. Ivan Kurtev.

[11] Herramientas Eclipse para desarrollo de software dirigido por modelos. Cristina Vicente Chicote, Diego Alonso Cáceres.

[12] Article: Model-Driven Engineering. Douglas C.Schmidt. Vanderbilt University.

[13] Desarrollo dirigido por modelos para la creación de laboratorios Virtuales. Yois Smith Pascuas Rengifo, José Joaquín Bocanegra García, Edson Johann Ortiz Lozada, José Nelson Pérez Castillo.

[14] Conceptos fundamentales de Ingeniería dirigida por Modelos y Modelos de dominio específico. John Ledgard Trujillo Trejo, Armando David Espinoza Robles.

[15] Como aumentar la productividad a través de las tecnologías de modelado. Adrián Noguero.

[16] Xtend2 – The successor to XPAND: http://blog.efftinge.de/2010/12/xtend-2-successor-to-xpand.html. Sven Efftinge.

[17] Xtend2- Sven Efftinge's Blog: http://blog.efftinge.de/2013_03_01_archive.html. Sven Efftinge.

[18] Xbase – A new programming language?. http://blog.efftinge.de/2010/09/xbase-new-programming-language.html. Sven Efftinge.

[19] Historia de Java: http://mundogeek.net/archivos/2004/10/04/una-no-tan-breve-historia-de-java/

[20] Características de C++11: http://msdn.microsoft.com/es-es/library/hh567368.aspx

[21] Historia de C++11: http://vitaminacpp.wordpress.com/2012/01/06/que-es-cpp11-historia-de-cpp/

[22] Historia de C++: http://vitaminacpp.wordpress.com/2012/01/06/que-es-cpp11-historia-de-cpp/

[23] ACCELEO: http://www.acceleo.org/pages/accueil/fr

[24] Introducción a EMF: http://blog.farmerdev.com/?p=95

[25] Eclipse IDE: http://eclipse.org/

[26] Características del lenguaje de programación C: https://es.wikipedia.org/wiki/C_(lenguaje_de_programaci%C3%B3n)

[27] Evolution of C: http://www.faqs.org/docs/artu/c_evolution.html

[28] Historia de C: http://sopa.dis.ulpgc.es/so/cpp/intro_c/