Facultad 2

GENERADOR AUTOMÁTICO DE CÓDIGO FUENTE PARA SOPORTAR EL LENGUAJE DE MODELADO ApEM-L 2.0

Trabajo de Diploma para optar por el título de Ingeniero en Ciencias Informáticas

Autores: Dayana León González
Liusvel Socarrás Sánchez

Tutores: Dr.C. Febe Angel Ciudad Ricardo
Ing. Yosnel Herrera Martínez

La Habana, junio 2014

"Año 56 de la Revolución"

La ciencia es el alma de la prosperidad de las naciones

y la fuente de vida de todo progreso.

Louis Pasteur

De Dayana:

Quiero agradecerle primero que todo a mi familia por estar siempre a mi lado y apoyarme en todas mis decisiones. A mi tío Iván que más que un tío ha sido un padre, gracias por tus consejos, por tu preocupación y por tu ayuda, te estaré eternamente agradecida. A mi mamá por ser tan recta y a la vez tan cariñosa; gracias por la educación que me diste, por tu comprensión, por tus regaños gracias por estar ahí cuando más lo necesité. A mi hermana Daylén por ser siempre tan buena conmigo, porque sé que siempre puedo contar contigo para lo que necesite y aunque a veces nos peleemos nos queremos muchísimo. A mi hermanita Daliana que aunque te conozco hace poco has sabido ganarte mi corazón con tu forma de ser, te quiero mucho. A mi abuelita Orquídea por ser tan especial, por sacarme siempre de apuros y sobre todo por quererme tanto. A mi novio por estar a mi lado durante estos 5 años de carrera en las buenas y en las malas, gracias por apoyarme siempre, por ayudarme a salir de los problemas, por darme fuerzas cuando más lo necesite, por hacerme sonreír por esto y mucho más Te Amo. A mis suegros Mercy y Lázaro por ser tan cariñosos y considerarme una más de la familia.

A Antonio Rey por haberme apoyado en el momento más duro de mi carrera, gracias por confiar en mí y ayudarme cuando de veras lo necesité. A mi tutor Febe por en tan pocos meses haberme enseñado a ser mejor profesional y persona, gracias por darme siempre tantos ánimos y apoyo. A mis amigos que vienen conmigo desde el IPI y nunca dejaron de preocuparse por mí: el Nino, Handy, Hayron, Anniel y Chavelis, gracias por su ayuda y compañía. A Karina y Ardelio por ser incondicionales y ayudarme en todo, sepan que tienen aquí una amiga para toda la vida. A mis compañeros de aula por los buenos y malos momentos que compartimos durante estos años. A todos los que de una forma u otra contribuyeron a este logro y a los que no porque gracias a ellos tropecé, caí y luego me levanté y gracias a eso soy hoy más fuerte.

De Liusvel:

A mi madre, quien ha sabido guiarme y apoyarme en tantos años de estudios, por su lucha incansable y brindarme todo su amor; gracias por ser tan especial mami.

A Lázaro por el apoyo que siempre me ha dado y confiar en mí, además de ser un ejemplo a seguir. A mi hermana por ser tan atenta y brindarme su ayuda para el desarrollo de esta investigación, además de saber ocupar el lugar de mami con tanta madurez. A mi abuelo quien desde pequeño me inculco el amor por los estudios y siempre estuvo tan orgulloso de su nieto. A mi familia toda, por ser tan maravillosa y ayudarme siempre. A mi novia, por hacer más hermosos los días y brindarme tanto amor. A Yaima y Reny, por ser tan preocupados y atentos en esta recta final. A mis maravillosos suegros y a Orquídea por ser siempre tan atentos y ayudarme en todo lo que necesité. A mi tutor Febe, por su apoyo para el desarrollo de la presente investigación y brindarnos la oportunidad de desarrollar esta investigación. A todos los amigos que de una forma u otra han convertido mi tránsito por el estudio superior, en una experiencia inolvidable.

A todos: muchas gracias.

Declaramos que somos autores del presente trabajo de diploma y reconocemos a la Universidad de las Ciencias Informáticas los derechos patrimoniales sobre la misma, con carácter exclusivo. Para que así conste, firman la presente a los ___ días del mes de ________ del año _____.

______________________________	______________________________
Autora: Dayana León González	Autor: Liusvel Socarrás Sánchez
______________________________	______________________________
Tutor: Dr.C Febe Angel Ciudad Ricardo	Tutor: Ing. Yosnel Herrera Martínez

Tutor: Dr.C Febe Angel Ciudad Ricardo

Graduado como Ingeniero Informático en el año 2004 por la Universidad de Holguín “Oscar Lucero Moya” (UHOLM) y el Instituto Superior Politécnico “José Antonio Echeverría” (CUJAE). Titulado como Máster en Informática Aplicada en el año 2007 por la Universidad de las Ciencias Informáticas (UCI) y obtuvo el grado científico de Doctor en Ciencias de la Educación – Especialidad Tecnología Educativa en el año 2012 por la Universidad de La Habana (UH). Imparte su docencia de pregrado en las disciplinas de Ingeniería y Gestión de Software, Metodología de la Investigación Científica y Formación Pedagógica. Es miembro de los claustros de las maestrías de Informática Aplicada, Informática Avanzada, Gestión de Proyectos y Educación a Distancia de la UCI. Desarrolla sus investigaciones en las temáticas de Ingeniería y Gestión de Software, con énfasis en el área del Software Educativo; así como en la Tecnología e Informática Educativas. Ha publicado diversos artículos científicos y ha participado en diferentes eventos nacionales e internacionales en estas áreas del conocimiento. Ha sido arquitecto, analista y líder de proyectos de desarrollo de software, jefe de departamento docente y asesor técnico – docente. Actualmente se desempeña como Director del Centro de Innovación y Calidad de la Educación (CICE) de la UCI.

Email: fciudad@uci.cu

Tutor: Ing. Yosnel Herrera Martínez

Graduado como Ingeniero en Ciencias Informáticas en el año 2006 por la Universidad de Ciencias Informáticas (UCI). Obtuvo la categoría docente de Profesor Asistente en el año 2013. Imparte su docencia de pregrado en la disciplina de Ingeniería y Gestión de Software. Desarrolla sus investigaciones en las temáticas de Ingeniería y Gestión de Software, con énfasis en el área del modelado de Software Educativo. Ha publicado diversos artículos científicos y ha participado en diferentes eventos nacionales e internacionales en estas áreas del conocimiento. Ha sido analista, diseñador y probador de proyectos de desarrollo de software; así como jefe de asignatura. Actualmente se desempeña como profesor del Dpto. de Ingeniería y Gestión de Software de la Facultad 4 de la UCI.

Email: yherrera@uci.cu

Para el proceso de desarrollo de software se utilizan los lenguajes de modelado, los cuales facilitan la comunicación entre los integrantes del equipo de trabajo a través del uso de una semántica común. ApEM-L constituye un lenguaje de dominio específico enfocado al modelado de aplicaciones educativas y multimedia; esta actividad se realiza utilizando herramientas CASE, las cuales brindan un conjunto de funciones automatizadas, siendo la generación de código una de las más preciadas. La presente investigación tiene como objeto de estudio el proceso de generación de código con el uso de ApEM-L 2.0 como lenguaje de modelado. Se propone el desarrollo de un plugin para Eclipse que permita generar el código fuente en lenguaje PHP 5 para aplicaciones educativas, elevando los grados de consistencia y completitud del código generado. Ambas variables fueron seleccionadas a partir de la sistematización de los principales referentes teórico-metodológicos relativos al proceso de generación de código basado en modelos. Para valorar la contribución y calidad de la solución propuesta, se realizaron pruebas de caja blanca y caja negra, así como un experimento a través de un caso de estudio, comprobándose el incremento experimentado por las variables de la investigación al utilizar el plugin desarrollado.

PALABRAS CLAVE: ApEM-L, CASE, código, modelado

INTRODUCCIÓN 1

CAPÍTULO 1. EL MODELADO DE SOFTWARE Y SUS TECNOLOGÍAS ASOCIADAS 6

1.1. Conceptos asociados al desarrollo de software 6

1.1. Pilares del MDD: modelos, transformaciones y herramientas 8

1.2. Lenguajes de Modelado de Dominio Específico 15

1.3. La Generación de Código Fuente basada en Modelos 18

1.4. Metodología para el desarrollo del generador automático de código 22

1.5. Tecnologías a utilizar para el desarrollo del generador automático de código 25

Conclusiones Parciales 30

CAPÍTULO 2. DESCRIPCIÓN Y DISEÑO DEL PLUGIN PARA LA GENERACIÓN DE CÓDIGO PHP 5 31

2.1. Caracterización del proceso de generación de código 31

2.2. Propuesta de Solución 32

2.3. Modelo del Dominio 33

2.4. Diagrama de Flujo 36

2.5. Especificación de Requisitos 37

2.6. Patrón Arquitectónico: Modelo-Vista-Controlador 39

2.7. Patrones de Diseño 40

2.8. Diagrama de clases del diseño 42

2.9. Diagrama de Interacción 45

Conclusiones parciales 46

CAPÍTULO 3. IMPLEMENTACIÓN Y PRUEBAS DEL PLUGIN PARA LA GENERACIÓN DE CÓDIGO PHP 5 ..... **48**

3.1. Modelo de Implementación 48

3.2. Reglas para la generación automática de código 51

3.3. Plantillas para la generación de código 59

3.4. Pruebas de Software 60

Conclusiones parciales 66

CAPÍTULO 4. APLICACIÓN DEL PLUGIN A UN CASO DE ESTUDIO **72**

4.1. Caso de estudio: Proyecto “A Jugar” 72

Conclusiones parciales 77

CONCLUSIONES FINALES **78**

RECOMENDACIONES **79**

REFERENCIAS BIBLIOGRÁFICAS **80**

ANEXOS **84**

GLOSARIO DE TÉRMINOS **92**

Tabla 1: Ejemplos de herramientas de modelado y meta-modelado 11
Tabla 2: Comparación de las herramientas analizadas 14
Tabla 3: Diferencias entre metodologías tradicionales y ágiles [Tomado de (Figueroa et al., 2011)] 22
Tabla 4: Comparación de metodologías ágiles 23
Tabla 5: Comparación de herramientas de soporte para la generación de código [Tomado de (Riba, 2007)] 27
Tabla 6: Valores tomados por los indicadores de las variables definidas en la investigación 32
Tabla 7: Listado de los requisitos funcionales del plugin a desarrollar 38
Tabla 8: Descripción de las clases del diseño 44
Tabla 9: Descripción de las variables 61
Tabla 10: Matriz de Datos 61
Tabla 11: Descripción de las Variables 62
Tabla 12: Matriz de datos 62
Tabla 13: Dificultades encontradas con las pruebas de caja negra 63
Tabla 14: Comparación de los resultados de la aplicación de ambas herramientas 76
Tabla 15: Comparación de los valores tomados por las variables de la investigación al utilizar ambas herramientas 76
Tabla 16: Operacionalización de la variable independiente de la investigación 84
Tabla 17: Operacionalización de las variables dependientes de la investigación 85
Tabla 18: Listado de las No conformidades detectadas con las pruebas de Caja Negra 86

Figura 1: Desarrollo de Software Dirigido por Modelos [Tomado de (Pons et al., 2010)] 7
Figura 2: Herramientas en la arquitectura de 4 capas [Tomado de (Pons et al., 2010)] 11
Figura 3: Elementos en la generación mediante plantillas [Tomado de (Molina, 2003)] 21
Figura 4: Capas de la Metodología OpenUP [Tomado de (Balduino, 2007)] 25
Figura 5: Modelo del Dominio 34
Figura 6: Diagrama de flujo del proceso de generación de código 37
Figura 7: Diagrama de clases del diseño 43
Figura 8: Paquete de plantillas para cada tipo de clases 44
Figura 9: Diagrama de colaboración 46
Figura 10: Diagrama de componentes 49
Figura 11: Diagrama de clases modelado con ApEM-L 56
Figura 12: Aplicación de las reglas de transformación para la obtención del código fuente 57
Figura 13: Aplicación de las reglas de transformación para la obtención del código fuente 58
Figura 14: Plantilla para la generación de clases interfaces 59
Figura 15: Código fuente del método run 65
Figura 16: Gráfico de flujo y complejidad ciclomática correspondiente al método run 66
Figura 17: Rutas linealmente independientes correspondientes al método run 66
Figura 18: Resultado del Caso de Prueba 1 69
Figura 19: Resultado del Caso de Prueba 2 70
Figura 20: Diagrama de Clases del Caso de Uso “Presentar Software” [Tomado de (Ciudad, 2007)] 73
Figura 21: Código generado por Visual Paradigm para la clase CMEP_Video 74
Figura 22: Fragmento de código generado por el plugin desarrollado para la clase CMEP_Video 75
Figura 23: Comparación de las variables de la investigación 77
Figura 24: Primer fragmento de código generado por el plugin 87
Figura 25: Segundo fragmento de código generado por el plugin 88
Figura 26: Tercer fragmento de código generado con el plugin 89
Figura 27: Cuarto fragmento de código generado con el plugin 90

INTRODUCCIÓN

Durante los primeros años de la era de las computadoras, el desarrollo de software se realizaba sin ninguna planificación. Posteriormente con la introducción de nuevos conceptos de interacción hombre – máquina aparece el software como producto, por lo que comienza la búsqueda de procedimientos que optimizaran el proceso de construcción del software. (Guillén, 2001) Con el propósito de lograr un avance significativo en la industria del software e introducir mejoras en la productividad y calidad, actualmente existe la tendencia a seguir un enfoque dirigido por modelos donde estos últimos asumen un rol protagónico en el proceso de desarrollo del software.

Con la evolución de la Ingeniería de Software como disciplina, surgen las herramientas CASE[1] brindando un apoyo computarizado a los especialistas, durante todas o algunas de las etapas del ciclo de vida del software. Desde finales de la década del 80 del siglo XX, las herramientas CASE incorporaron la generación automática de código a partir de especificaciones de diseño, facilitando la transición entre la fase de diseño y la fase de implementación de los sistemas de software. (Bell, 1998) expone tres enfoques para la generación de código basada en modelos: el enfoque estructural, el enfoque de comportamiento y el enfoque de traducción. Según (Muñeton *et al.*, 2007) existen diversas herramientas que generan código a partir de representaciones de estructura estática y algunas complementan el código con el comportamiento de los objetos.

(Campoalegre, 2008) plantea que la generación automática de código trae consigo beneficios considerables para el desarrollo de software tales como: la aceleración del proceso de desarrollo al automatizar la tarea de crear diagramas y posteriormente codificarlos, el aseguramiento de una estructura estándar y consistente lo que ayuda en la fase de mantenimiento, y la disminución de la ocurrencia de errores, mejorando de esta manera la calidad del software desarrollado.

Todas las herramientas CASE prestan soporte a un lenguaje de modelado que permitirá la representación de conceptos utilizando una semántica común, facilitando la comunicación entre los beneficiarios y los integrantes de un proyecto. El Lenguaje Unificado de Modelado (UML – Unified Modeling Language) está diseñado para ser utilizado con múltiples propósitos y debido a sus grandes potencialidades es un

[1] Ingeniería de Software Asistida por Computadora (Computer Aided Software Engineering)

lenguaje estandarizado por el OMG[2] desde el año 1997. Sin embargo, aunque UML es un lenguaje universal que puede expresar un elevado número de propiedades fundamentales de modelado, a veces resulta necesario un lenguaje que permita el modelado de un dominio de aplicación concreto. Los lenguajes de modelado para estos fines son denominados de Dominio Específico.

Gracias a la facilidad de extensión que provee UML a través de los perfiles, surge en el año 2007 como propuesta en la Universidad de las Ciencias Informáticas (UCI) el Lenguaje de Modelado Orientado a Objetos para Aplicaciones Educativas y Multimedia (ApEM-L 1.0), creado por el Dr.C. Febe Angel Ciudad Ricardo, como respuesta a la «*necesidad de utilizar un lenguaje notacional que se ajustara a las características del software educativo cubano y que representara en modelos: la estructura lógica, el comportamiento y las funciones del futuro software a desarrollar*».(Ciudad y Herrera, 2008, p 52)

Una de las ventajas notables que suponía para los integrantes de un proyecto la aplicación de ApEM-L para el modelado de aplicaciones educativas era la posibilidad de utilizar las herramientas CASE existentes para UML. Sin embargo, en la versión 2.0 del lenguaje, debido a los cambios realizados en el metamodelo y la incorporación de nuevos elementos sintácticos y semánticos, esto no es posible. Por lo tanto, el proceso de desarrollo de aplicaciones educativas, en lo referente al modelado y la generación de código en particular, presenta hoy las siguientes insuficiencias:

- ✓ Las herramienta CASE no soportan el modelado y la generación de código para aplicaciones educativas que utilicen ApEM-L en su versión 2.0.
- ✓ Deficiente correspondencia entre los artefactos generados en la fase de diseño y la fase de implementación debido a que los estereotipos restrictivos definidos por el lenguaje en su versión 2.0 no resultan reflejados en el código.

Según (Rumbaugh *et al.*, 2007) *si no existe una herramienta genérica que comprenda los estereotipos del lenguaje de modelado, no puede verificarse la consistencia entre el modelo y el código*, viéndose afectada también la completitud del código generado.

[2] Object Management Group, es una asociación estándar de la industria del software libre y sin fines de lucro que se dedica a la producción y mantenimiento de especificaciones tecnológicas en el campo de las ciencias de la computación.

A partir de lo planteado anteriormente, se identificó el siguiente **problema científico**: ¿Cómo generar código fuente consistente y completo para las aplicaciones educativas desarrolladas en la UCI que utilizan el lenguaje de modelado ApEM-L 2.0?

Se plantea como **objeto de estudio** el proceso de generación de código fuente con el uso de ApEM-L 2.0 como lenguaje de modelado. Surgiendo como **campo de acción** las herramientas CASE para el modelado del dominio de las aplicaciones educativas.

El **objetivo de la investigación** es desarrollar un generador automático de código fuente para ApEM-L 2.0 que garantice consistencia y completitud en el código generado.

De esta manera se plantea como **hipótesis de trabajo**: el desarrollo de un generador automático de código fuente para aplicaciones educativas que usen ApEM-L 2.0 elevará los grados de consistencia y completitud del código generado.

Para el cumplimiento del objetivo de la investigación se plantean las siguientes **tareas investigativas**:

1. Establecimiento de los referentes teóricos-metodológicos para la construcción de herramientas de generación de código.
2. Caracterización del proceso de generación de código en los proyectos productivos que usen ApEM–L en la UCI.
3. Construcción de la herramienta de generación de código para ApEM–L 2.0 como lenguaje de modelado.
4. Validación de la herramienta, en lo referente a su aporte a la consistencia y completitud del código generado.

Para la ejecución de la investigación se trabajó con una **población** donde cada unidad muestral tuviera ambas de las siguientes características:

- Documentos científicos de la UCI que contuvieran los resultados de investigaciones desarrolladas utilizando el lenguaje ApEM-L en cualquiera de sus versiones 1.0 o 1.5.
- Documentos científicos o cualquier otra forma de socialización de resultados científicos en la UCI que contuvieran el código fuente como parte de los resultados de investigaciones desarrolladas utilizando el lenguaje ApEM-L en cualquiera de sus versiones 1.0 y 1.5.

El análisis arrojó un total de 10 unidades muestrales. Por considerarse que este tamaño de la población es manejable por los investigadores y que se necesitaba obtener la mayor cantidad de información relativa al objeto de estudio, se decidió utilizar la totalidad de la población como **muestra** de la investigación.

Se realizó un **muestreo** intencional seleccionando los códigos generados en 10 tesis que usaron ApEM-L como lenguaje de modelado.

Durante el desarrollo de la investigación se utilizaron los siguientes **métodos científicos**:

Métodos teóricos:

Hipotético–Deductivo: para la determinación de la hipótesis de esta investigación y su posterior comprobación.

Análisis y Síntesis: para el procesamiento de la información y el arribo a las conclusiones de la investigación, así como para precisar la tecnología para la generación de código.

Histórico-Lógico: para hacer un estudio de los antecedentes del proceso que se desea automatizar y el análisis de otras soluciones existentes a problemas similares.

Modelación: para la creación de modelos con vistas a investigar la realidad. Este método permitió a los autores determinar las características de la herramienta, mediante la modelación de los conceptos del entorno objeto de informatización a través de diagramas.

Métodos Empíricos:

Observación: para el análisis del comportamiento lógico de las variables de la investigación en el transcurso del tiempo, dentro del campo del objeto de estudio seleccionado, permitiendo comparar resultados en las distintas fases de la investigación.

Entrevistas: utilizado para obtener información verbal de los desarrolladores de los proyectos productivos seleccionados, que permita conocer el comportamiento lógico de las variables de la investigación.

Experimento: permitió realizar la verificación del estado de las variables utilizadas en la investigación.

El trabajo que se presenta a continuación está conformado por cuatro capítulos:

En el primer capítulo se abordan los conceptos fundamentales asociados al desarrollo de software dirigido por modelos, los lenguajes de dominio específico, la generación de código basada en modelos así como las tecnologías utilizadas para darle cumplimiento al objetivo propuesto. El segundo capítulo contiene la caracterización del proceso de generación de código en la actualidad y el diseño del plugin desarrollado para la generación automática de código. Posteriormente en el tercer capítulo se presenta el diagrama de componentes generado durante la fase de Implementación, el conjunto de reglas de transformación implementadas en las plantillas, para la traducción a código de cada uno de los componentes del diagrama de clases del diseño de ApEM-L 2.0 y las pruebas realizadas al plugin con el fin de detectar

errores que afecten la calidad de la solución. Por último en el cuarto capítulo se realiza la comprobación de la hipótesis a través de un caso de estudio donde se demuestra cómo la aplicación de la solución brindada eleva los grados de las variables tratadas durante la investigación.

CAPÍTULO 1. EL MODELADO DE SOFTWARE Y SUS TECNOLOGÍAS ASOCIADAS

En este capítulo se hace un análisis de todos los conceptos asociados al dominio del problema, pues esta es la base para el entendimiento del objeto de estudio que rige la presente investigación, el cual se centró en la generación de código fuente a través de herramientas CASE para el desarrollo de aplicaciones educativas. Por lo tanto, se hizo necesario profundizar acerca del paradigma de desarrollo dirigido por modelos y los pilares que sustentan el mismo, haciendo énfasis en el proceso de generación de código y los enfoques existentes para llevar a cabo este proceso.

1.1. Conceptos asociados al desarrollo de software

La Ingeniería del Software: una tecnología estratificada

Según (Pressman, 2002) la Ingeniería del Software es una tecnología estratificada compuesta por:

Proceso: Constituye el estrato base de la Ingeniería del Software y el encargado de mantener juntos los estratos de la tecnología. El proceso define un marco de trabajo de las tareas que se requieren para construir software de alta calidad.

Método: Indica cómo construir técnicamente el software. Los métodos se basan en un conjunto de principios que gobiernan cada área de la tecnología e incluye actividades de modelado y otras técnicas descriptivas.

Herramientas: Son las que proporcionan el soporte automatizado o semiautomatizado para el proceso y los métodos.

A partir de lo antes descrito se puede concluir que las herramientas utilizadas en el desarrollo de software constituyen un eslabón fundamental para la sostenibilidad del proceso y los métodos.

El desarrollo de software dirigido por modelos

(Pons *et al.*, 2010) plantean que el Desarrollo de Software Dirigido por Modelos (MDD[3]) se ha convertido en un nuevo paradigma; el uso del adjetivo "dirigido" (*driven)* enfatiza en el rol central y activo que tienen los modelos en este paradigma. MDD promete mejorar el proceso de construcción de software basándose en un proceso guiado por modelos y soportado por herramientas. Los modelos se van generando desde los más abstractos a los más concretos, a través de pasos de transformación y/o refinamientos hasta llegar al código. En la Figura 1 se muestra la parte del proceso de desarrollo de software en donde la intervención humana es reemplazada por herramientas automáticas. Los modelos pasan de ser entidades contemplativas para convertirse en entidades productivas a partir de las cuales se deriva la implementación en forma automática.

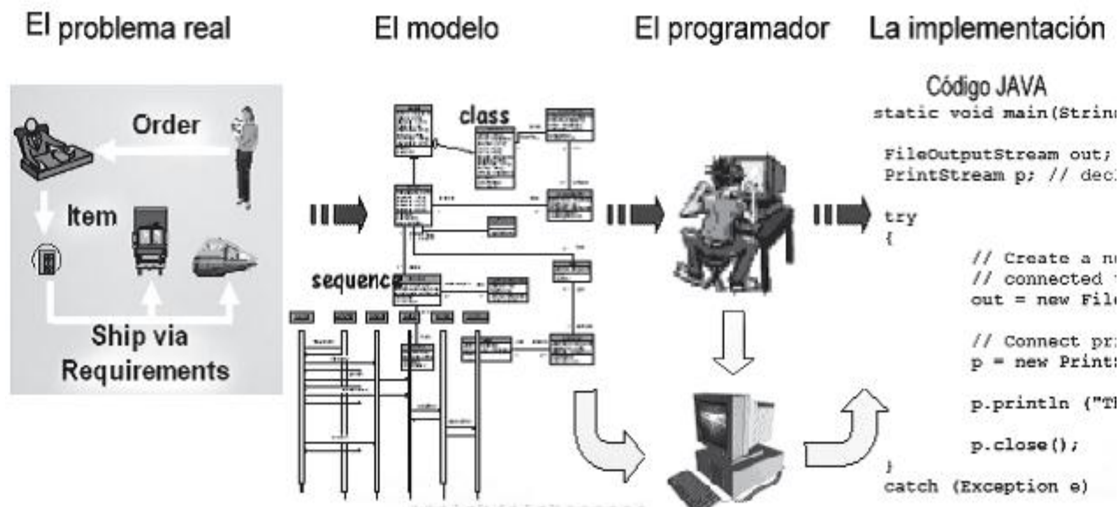

Figura 1: Desarrollo de Software Dirigido por Modelos [Tomado de (Pons et al., 2010)]

Según (Pons *et al.*, 2010) las propuestas concretas más conocidas y utilizadas en el ámbito de MDD son, por un lado la Arquitectura Dirigida por Modelos (MDA)[4] desarrollada por el OMG y por otro lado el modelado específico del domino (DSM)[5] acompañado por los lenguajes específicos del dominio.

---

[3] Acrónimo en inglés de Model Driven Software Development

[4] Acrónimo en inglés de Model Driven Architecture

[5] Acrónimo en inglés de Domain Specific Modeling

1.1. Pilares del MDD: modelos, transformaciones y herramientas

El proceso de desarrollo de software dirigido por modelos se apoya sobre los siguientes pilares:

- Modelos con diferentes niveles de abstracción, escritos en un determinado lenguaje.
- Definiciones de cómo un modelo se transforma en otro modelo más específico.
- Herramientas de software que den soporte a la creación de modelos y su posterior transformación.

¿Qué es un modelo?

Debido a la importancia que tienen los modelos como mecanismo para la representación simplificada de una determinada realidad y facilitador de la comprensión de los sistemas modelados, diversos autores han aportado sus teorías acerca de este tema.

En la investigación se adopta la definición brindada por (Sommerville, 2005, p 5) donde expresa que los modelos son «*la representación simplificada de un proceso del software, presentada desde una perspectiva específica*». Por su parte (Rumbaugh *et al.*, 1999, p 11), plantea que los modelos son los encargados de *«captar los aspectos importantes de lo que se está modelando, desde cierto punto de vista, y simplificar u omitir el resto».* Estos modelos adquieren diversas formas y aparecen en diversos niveles de abstracción para cumplir propósitos como el de guiar el proceso de pensamiento y la descripción completa o parcial de un sistema.

Sin embargo, a consideración de (Selic, 2003) para que un modelo sea útil debe tener las siguientes características: ante todo debe ser abstracto (es decir, ser versión reducida del sistema que representa), y también comprensible (expresado de tal forma que se pueda entender fácilmente), preciso (representa fielmente el sistema modelado) y predictivo (se puede utilizar, para obtener conclusiones correctas sobre el sistema).

Aunque frecuentemente se utilizan como sinónimos, los términos “modelo” y “diagrama” no significan lo mismo. Un diagrama es la representación gráfica de un modelo, o de una parte del modelo. Los modelos pueden representarse gráficamente mediante diagramas, en una estructura de árbol, o incluso en forma puramente textual. En otras palabras, el modelo es independiente de su representación gráfica o textual.(Génova, 2013)

Los modelos se expresan a través de un lenguaje de modelado y este último concepto ha sido

apropiadamente interpretado por (Herrera, 2014,p 16) quien lo define como un *«sistema de conceptos de modelado que a través de sus símbolos, reglas sintácticas y semánticas, posibilitan la representación visual o gráfica de la estructura y el comportamiento de los programas de computadora a través de modelos»*.

Transformación de modelos

Cualquier propuesta MDD implica trabajar con varios modelos relacionados entre sí, lo cual requiere mucho esfuerzo para abordar la complejidad que implican las diferentes tareas relacionadas con la gestión de modelos como es el refinamiento y la validación. Por lo tanto, uno de los principales retos para potencializar el papel de los modelos en la Ingeniería del Software es automatizar dichas tareas para reducir la complejidad asociada. Esta automatización se lleva a cabo a partir de las transformaciones de modelos.

En (Kleppe *et al.*, 2003) se definen los siguientes conceptos de interés para la presente investigación:

- Una transformación es la generación automática de un modelo destino desde un modelo fuente, de acuerdo a una definición de transformación.
- Una definición de transformación es un conjunto de reglas de transformación que juntas describen como un modelo en el lenguaje fuente puede ser transformado en un modelo en el lenguaje destino.
- Una regla de transformación es una descripción de como una o más construcciones en el lenguaje fuente pueden ser transformadas en una o más construcciones en el lenguaje destino.

Según (Pons *et al.*, 2010) existen dos tipos fundamentales de transformaciones: de Modelo-a-Modelo (M2M)[6] donde se convierte la información de un modelo origen a un modelo destino y de Modelo-a-Texto (M2T)[7] donde se convierte cada elemento del modelo origen en definiciones de texto; en cualquiera de los dos casos existen lenguajes que han sido creados específicamente para expresar dichas transformaciones, tal es el caso de ATL[8] y QVT[9], el cual constituye el estándar del OMG para la

[6] Acrónimo en inglés de Model to Model

[7] Acrónimo en inglés de Model to Text

[8] Acrónimo en inglés de Atlas Tranformation Language

transformación de M2M. Para el caso de las transformaciones de M2T, en el 2004 el OMG hizo una petición de propuestas siendo el lenguaje MOFScript uno de los presentados. El proceso de combinación de MOFScript con las restantes propuestas produjo como resultado al estándar llamado MOF2Text.

Herramientas de soporte

El desarrollo de software dirigido por modelos, no tendría sentido sin la disponibilidad de herramientas que brinden soporte a la creación de modelos y su posterior transformación. Actualmente existe una gran variedad de herramientas CASE las cuales han sido definidas por (Sommerville, 2005, p 79) como «el software utilizado para apoyar las actividades del proceso de software, tales como la ingeniería de requerimientos, diseño, implementación y pruebas»; brindando así un soporte automatizado al desarrollo de software para contribuir a mejorar la calidad y productividad en el desarrollo de sistemas de información.

No existe una única forma de clasificar las herramientas CASE. Sin embargo, las herramientas de modelado son de las más utilizadas en la actualidad para el proceso de desarrollo de software. Las mismas ofrecen una serie de funciones automatizadas donde una de las más preciadas es la generación de código, la cual se realiza a partir de reglas de transformación que están programadas dentro de la propia herramienta y que por lo general no son accesibles al modelador.
(Pons *et al.*, 2010) plantea que la mayoría de las CASE que se utilizan actualmente basadas en UML poseen una arquitectura de dos niveles expuesta en la Figura 2. En esta arquitectura, el metamodelo está programado y compilado dentro de la herramienta, determinando la clase de modelos que se pueden hacer y la manera en que se procesan, por lo que solo la empresa proveedora de la herramienta es la que puede modificar el lenguaje de modelado definido en el código. Sin embargo, la tecnología basada en metamodelos elimina esta limitación permitiendo lenguajes de modelado flexibles y reglas de transformación editables. Dicha transformación se lleva a cabo adoptando la arquitectura de capas definida por el OMG. A diferencia de las demás herramientas, una herramienta basada en metamodelos permite al usuario acceder y modificar las especificaciones del lenguaje.

---

[9] Acrónimo en inglés de Query View Transformation

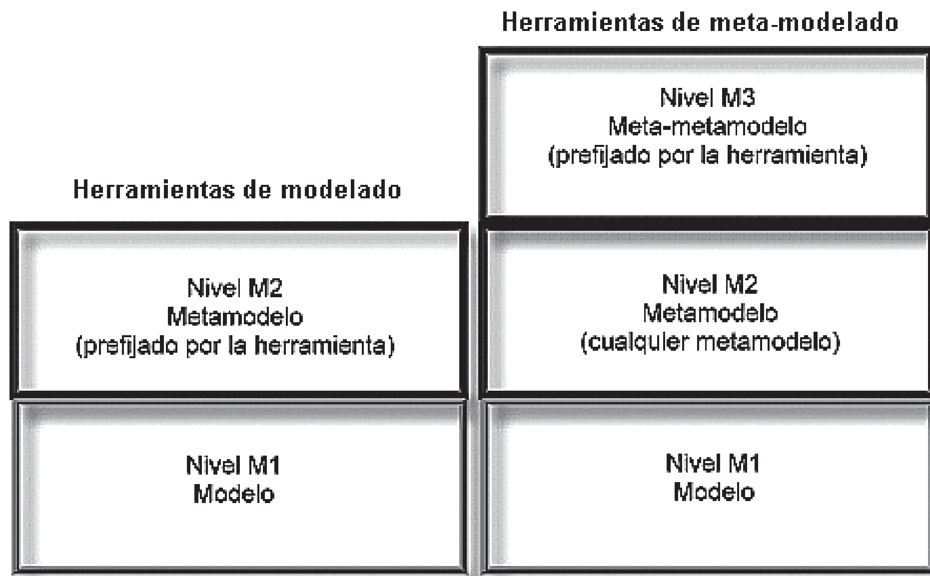

Figura 2: Herramientas en la arquitectura de 4 capas [Tomado de (Pons et al., 2010)]

Actualmente, existe una gran variedad de herramientas de modelado y meta-modelado de las cuales se exponen en la siguiente tabla las más reconocidas.

Tabla 1: Ejemplos de herramientas de modelado y meta-modelado

Tipo de Herramienta	Herramientas disponibles
Modelado	• Visual Paradigm • ArgoUML • Enterprise Architect • Rational Rose
Meta-Modelado	• Eclipse Modeling Framework • MetaEdit+ • DSL Tools

Autores como (Domingo et al., 2012) y (Pons *et al.*, 2010) caracterizan las herramientas de modelado y meta-modelado respectivamente de la siguiente manera:

ArgoUML: Es una aplicación de modelado de UML escrita en Java y publicada bajo la licencia BSD[10]. Permite crear diagramas como: diagrama de Casos de Uso, diagrama de Clases, diagrama de Secuencia, diagrama de Colaboración, diagrama de Estado, diagrama de Actividades y diagrama de Despliegue. Es una herramienta extensible y permite la generación automática de código a lenguajes como Java, C++, C#, PHP4, PHP5 y SQL. Se caracteriza por: Uso de arquitectura basada en componentes, control de cambios, modelado visual del software y verificación de la calidad del mismo.

Visual Paradigm for UML: Es una herramienta que soporta el ciclo de vida completo en el desarrollo de software: análisis y desarrollos orientados a objetos, construcción, prueba y despliegue. Permite el modelado de diagramas UML tales como diagrama de paquetes, de clase, de objetos, de componentes, de despliegue, de casos de uso, diagrama de actividades, entre otros. También permite generar código a lenguajes como Java, C++, PHP y la generación de documentación a partir de dichos diagramas, además permite realizar ingeniería inversa.

Enterprise Architect (UML): Enterprise Architect es una herramienta comprensible de diseño y análisis UML, cubriendo el desarrollo de software desde el paso de los requerimientos a través de las etapas del análisis, modelos de diseño, pruebas y mantenimiento. Enterprise Architect es una herramienta multi-usuario, basada en Windows, diseñada para ayudar a construir software robusto y fácil de mantener. Ofrece salida de documentación flexible y de alta calidad. Algunas de sus características son: diseño y construcción de UML, Casos de Uso, Modelos Lógico, Dinámico y Físico, Extensiones personalizadas para modelado de procesos y más, soporte para ActionScript 2.0, Java, C#, C++, VB.Net, Delphi, Visual Basic, Python y PHP. Facilidad de Importación/Exportación XMI[11].

Rational Rose: Es una herramienta para modelado, que soporta UML 2.0 y permite la creación de diferentes diagramas tales como: diagrama de Casos de Uso, de Interacción, de Actividad, de Clases, de Estado, de Componentes y de Despliegue, así como la generación automática de código a diferentes lenguajes tales como: Ada, ANSI C++, C++, CORBA, Java/J2EE, Visual C++ y Visual Basic, además de brindar la posibilidad de que varias personas trabajen a la vez, permitiendo que cada desarrollador opere en un espacio de trabajo privado que contiene el modelo completo y permite que tenga un control

[10] Acrónimo en inglés de Berkeley Software Distribution

[11] Acrónimo en inglés de XML Metadata Interchange

exclusivo sobre la propagación de los cambios en ese espacio de trabajo. Permite realizar ingeniería inversa y generar documentación.

Eclipse Modeling Framework (EMF): Es un framework de código abierto para modelado donde la generación de código es posible a partir de la especificación de un modelo; los modelos se especifican usando Ecore el cual constituye una versión simplificada del lenguaje de metamodelado MOF[12], esto permite un soporte para la interoperabilidad con otras herramientas. El generador de código de EMF produce archivos que pretenden que sean una combinación entre las partes generadas y las partes modificadas por el programador. Siempre se puede regenerar desde el modelo cuando se necesite, y las partes agregadas serán preservadas durante la regeneración; además existe una gran disponibilidad de asistentes y tutoriales sobre EMF. Como contraparte tiene como debilidad la existencia de una gran cantidad de plugins por lo que es difícil elegir cual es más conveniente para una tarea en particular y combinar las versiones adecuadas de cada uno de los plugins.

MetaEdit+: Es una herramienta comercial, basada en repositorios, que usa una arquitectura cliente/servidor. La misma facilita la propagación de cambios del metamodelo hacia sus instancias; posee un editor integrado para la definición de la sintaxis concreta del lenguaje y el generador de código está integrado en la herramienta. Sin embargo no es una herramienta de código abierto; los editores generados necesitan de MetaEdit+ para funcionar y se dificulta la portabilidad de proyectos debido a que usa un lenguaje de metamodelado propio que no cumple con el estándar.

DSL Tools: Las herramientas provistas en Visual Studio son denominadas colectivamente las “DSL Tools”. Esta propuesta soporta plantillas predefinidas para asistir a las distintas actividades durante el proceso de creación del lenguaje; cuenta con buena documentación para su utilización; posee un robusto soporte para especificar y evaluar restricciones sobre los modelos; cuenta con mapeo visual muy amigable para definir la conexión entre la sintaxis abstracta y la concreta. Sin embargo no es una herramienta de código abierto; los editores generados necesitan de Visual Studio para funcionar y se dificulta la portabilidad de proyectos debido a que usa un lenguaje de metamodelado propio que no cumple con el estándar.

[12] Acrónimo en inglés de Meta Object Facility

A partir de lo antes descrito se puede concluir que la mayoría de las herramientas de modelado prestan soporte a la generación automática de código. Tomando en consideración las herramientas más utilizadas en la Universidad se hizo necesario realizar una comparación en cuanto a los siguientes indicadores:

- **Plataforma**: Indica la posibilidad de ejecutar la herramienta en los distintos sistemas operativos.
- **Código Abierto**: Indica la facilidad de acceso al código fuente de la herramienta para realizar modificaciones del mismo.
- **Lenguaje de modelado**: Se refiere al lenguaje de modelado soportado por la herramienta.
- **Documentación**: Indica el grado de documentación disponible para facilitar la comprensión de la herramienta.
- **Posibilidad de extensión**: Indica si la herramienta permite extender o no sus funcionalidades.

Tabla 2: Comparación de las herramientas analizadas

Indicadores / Herramientas	Plataforma	Código Abierto	Lenguaje de Modelado	Documentación	Posibilidad de extensión
ArgoUML	Multiplataforma	sí	UML	Media	sí
Visual Paradigm for UML	Multiplataforma	no	UML	Alta	sí
Enterprise Architect	Windows, (compatible con Linux y Mac)	no	UML	Alta	no
Rational Rose	Multiplataforma	no	UML	Alta	no
Eclipse	Multiplataforma	sí	UML	Alta	sí

Luego del análisis realizado se concluye que:

- ✓ ArgoUML según su página oficial (www.argouml.tigris.org), brinda mecanismos de extensión sin embargo esta herramienta se encuentra descontinuada desde el año 2010 y no cuenta de una documentación detalla que permita la comprensión de la misma para el desarrollo de plugins.
- ✓ Tanto Rational Rose como Enterprise Architect no brindan facilidades de extensión.
- ✓ Visual Paradigm aunque posibilita el desarrollo de extensiones no permite la modificación del metamodelo del lenguaje por lo tanto no es posible su utilización en la investigación.
- ✓ Eclipse aunque comúnmente es utilizado como un Entorno Integrado de Desarrollo (IDE) brinda la posibilidad de incorporar nuevas funcionalidades a través del mecanismo de plugin para el modelado de aplicaciones y permite, a través de un conjunto de herramientas la definición de metamodelos y la creación de generadores de código.

La herramienta seleccionada para darle cumplimiento al objetivo de la investigación es Eclipse debido a que es multiplataforma, de código abierto, brinda facilidades de extensión para incorporar nuevas funcionalidades y además la documentación disponible es abundante y actualizada.

1.2. Lenguajes de Modelado de Dominio Específico

La iniciativa DSM plantea la creación de modelos utilizando un lenguaje focalizado y especializado para modelar un dominio de aplicación concreto. Estos lenguajes son denominados DSL[13] y como subconjunto de los mismos aparece el término: Lenguaje de Modelado de Dominio Específico *(DSML[14])*. Estos lenguajes surgen debido a que en muchas ocasiones la sintaxis o la semántica de lenguajes de propósito general, tales como UML, no permiten expresar los conceptos específicos del dominio, o es necesario restringir y especializar los constructores propios de UML, que suelen ser demasiado genéricos y numerosos.

Algunos de los conceptos más importantes que forman parte del modelado de dominio específico son: modelo, metamodelo y meta-metamodelo; siendo la “Arquitectura de 4 capas”, definida por el OMG, la usada tradicionalmente para establecer la relación entre estos conceptos.

---

[13] Acrónimo en inglés de Domain-Specific Languages

[14] Acrónimo en inglés de Domain- Specific Modeling Language

Los cuatro niveles definidos en esta arquitectura se denominan M3,M2, M1, M0 y son descritos por (Pons *et al.*, 2010) de la siguiente forma:

- **M0. Instancia:** caracteriza los objetos del mundo real que son manipulados por el software. El término “objeto” es utilizado en un amplio sentido para significar también “procesos”, “conceptos”, “estados”, etc.
- **M1. Nivel de modelo:** caracteriza a los modelos que representan los datos del nivel M0. Es decir, cada elemento de M0 es una instancia de un elemento de M1.
- **M2. Nivel del metamodelo:** caracteriza a metamodelos que describen los modelos del nivel M1. Es decir los elementos del nivel M1 son a su vez instancias del nivel M2.
- **M3. Nivel de meta-metamodelo:** caracteriza a los meta-metamodelos que describen los metamodelos del nivel M2. Es decir la relación entre un meta-metamodelo y un metamodelo es análoga a la relación entre un metamodelo y un modelo. Un meta-metamodelo es un modelo que define el lenguaje para representar un metamodelo.

Según (Herrera, 2014; Koch *et al.*) el metamodelo es el modelo que especifica la sintaxis abstracta del DSL y su representación se hace usualmente a través de diagramas de clases UML donde se definen las entidades (metaclases) que identifican a los conceptos de modelado, y de asociaciones (meta-asociaciones) que identifican las relaciones entre estos conceptos. Además los metamodelos juegan un papel fundamental en la construcción de herramientas CASE pues son usados para la actividad de diseño, la definición de transformaciones del modelo y la generación automática de código.

UML está definido utilizando un metamodelo, es decir, un modelo del propio lenguaje de modelado. Sin embargo, dado que en ocasiones no es posible expresar las características de un determinado dominio a través de UML, el propio lenguaje ofrece un mecanismo de extensión del metamodelo con el fin de adaptarlo a situaciones o necesidades específicas. Este mecanismo es conocido como **perfil UML.**

Sintácticamente, los perfiles UML son definidos como un paquete del UML estereotipado <<profile>>. Estos se componen estructuralmente de tres elementos de extensión estándares: estereotipos, restricciones y valores etiquetados.

(Rumbaugh *et al.*, 2007) define cada uno de los elementos de extensión de la siguiente forma: una restricción es una declaración de texto de una relación semántica expresada en algún tipo de lenguaje formal o en lenguaje natural. Un estereotipo es un nuevo tipo de elemento del modelo concebido por el modelador y basado en un tipo de elemento del modelo existente pero con restricciones adicionales, una

interpretación, un icono diferentes, y un tratamiento distinto por parte de los generadores de código y otras herramientas de bajo nivel. Por otra parte, un valor etiquetado es una pieza de información con nombre vinculada a cualquier elemento del modelo.

Autores como (Fuentes y Vallecillo, 2003), mencionan en su investigación las dos posibilidades establecidas por el OMG para definir lenguajes de dominio específico: o bien se define un nuevo lenguaje (que sería una alternativa al UML), o de lo contrario se extiende el UML a través de los perfiles.

A partir de lo planteado, (Herrera, 2014) considera que las soluciones científico-tecnológicas de lenguajes de modelado dentro de un dominio de aplicación específico se pueden clasificar en tres categorías en dependencia al grado de profundidad de las extensiones al UML:

1. Los perfiles UML creados para un dominio específico (**Perfil**$_{uml}$).
2. Los DSML como extensión del UML que surgen a partir de los perfiles UML (**DSML**$_{uml}$).
3. Los DSML que dejaron de ser extensiones del UML y por tanto son alternativas de este (**DSML**$_{ad\text{-}hoc}$).

Como ejemplos de **DSML**$_{uml}$ se encuentra el Lenguaje Orientado a Objetos para la Modelación de Aplicaciones Multimedia (OMMMA-L) definido por (Engels y Sauer, 2000) y al Lenguaje de Modelado Orientado a Objetos para Aplicaciones Educativas y Multimedia (ApEM-L). OMMMA-L tiene como propósito facilitar el modelado de un gran rango de aspectos de aplicaciones multimedia interactivas de una forma integrada y comprensiva. Por otra parte, ApEM-L surgió como propuesta del Dr.C. Febe Angel Ciudad Ricardo debido a la «*necesidad de utilizar un lenguaje notacional que se ajustara a las características del software educativo cubano y que representara en modelos: la estructura lógica, el comportamiento y las funciones del futuro software a desarrollar*» (Ciudad y Herrera, 2008, p 52). Dicho lenguaje toma como bases teóricas principales OMMMA–L (2001) y OCL[16]–2.0 (2003) y al ser una extensión de UML no modifica su semántica, sino que trabaja con estereotipos restrictivos.(Ciudad, 2007)

A partir de un estudio realizado por (Herrera, 2014) de la literatura referente a los Perfiles$_{uml}$ y DSML$_{uml}$, así como soluciones dentro del dominio de las aplicaciones educativas, se constató que el más completo de los elementos de extensión es el estereotipo y los más utilizados son éste y las restricciones; aunque los

---

[16] Acrónimo en inglés de Object Constraint Language

valores etiquetados se aplican con moderación. En el caso del estereotipo, la razón fundamental es su nivel de expresividad por acoger a los otros mecanismos de una manera sistémica.

(Berner *et al.*, 1999) proponen la siguiente clasificación de estereotipos de acuerdo al grado de expresividad:

- **Estereotipo decorativo**: Se trata solamente de reemplazar la forma decorativa en la cual un elemento del lenguaje es visualmente representado sin implicar cambios semánticos. En dependencia de la buena o mala selección del símbolo de remplazo puede que mejore o empeore la representatividad del modelo, teniendo en cuenta que la interpretación del símbolo depende de la cultura del usuario.
- **Estereotipo descriptivo**: Extiende la sintaxis del lenguaje para expresar información adicional sin implicar restricciones semánticas, lo cual significa que está a un nivel puramente sintáctico. Su expresividad se materializa en proponer nuevas clasificaciones a las metaclases o describir información adicional de éstas.
- **Estereotipo restrictivo**: Extiende la sintaxis del lenguaje, pero a su vez impone restricciones semánticas. Esto significa que puede definir cambios estructurales a las propiedades de la metaclase o simplemente imponer restricciones de asociación con otras metaclases, por citar algunos ejemplos.
- **Estereotipo re-definitorio**: Modifica el núcleo semántico del lenguaje. Se trata del estereotipo cuya expresión cambia completamente la semántica original de la metaclase extendida, definiendo un nuevo concepto de modelado y su significado. Los estereotipos que redefinen la semántica de las metaclases son los que al final convierten, por ejemplo, un $DSML_{uml}$ en un $DSML_{ad\text{-}hoc}$.

1.3. La Generación de Código Fuente basada en Modelos

La generación automática de código es definida por (Rincón *et al.*, 2011, p 406) como «*el proceso mediante el cual un programa produce, de manera automática, código en un lenguaje, a partir de un esquema expresado en otro lenguaje*».

Según (Rincón *et al.*, 2011) las herramientas de desarrollo de software actuales, muestran la tendencia a facilitar el desarrollo basado en modelos, permitiendo a los desarrolladores trabajar a un nivel de abstracción más alto. Las herramientas de generación de código fuente basadas en modelos son las encargadas de la producción de código fuente en lenguajes de alto nivel de manera automática, a partir de

modelos gráficos que describen la estructura, el comportamiento, o la arquitectura de los sistemas; facilitando así la transición entre la fase de diseño y la fase de implementación de los sistemas de software.

Según (Bell, 1998) existen tres enfoques para la generación de código basada en modelos:

- ✓ **Enfoque estructural**: este enfoque permite generar el código correspondiente a definiciones de clases y sus relaciones estáticas, evitándole a los desarrolladores la tediosa tarea de escribir el código para las estructuras que se derivan de un diagrama de clases. Debido a que en este enfoque el código para capturar el comportamiento del sistema debe ser escrito a mano, se ofrecen mecanismos para integrar código escrito manualmente junto con la estructura generada además de protegerlo para evitar reescribirlo en sucesivas generaciones.
- ✓ **Enfoque de comportamiento**: este enfoque permite generar código para especificaciones de comportamiento y de acción expresadas en un lenguaje de alto nivel. Un beneficio adicional obtenido es que se puede simular y verificar el comportamiento del sistema (a partir de los modelos) antes de que el código sea generado.
- ✓ **Enfoque de traducción**: este enfoque proporciona dos puntos de entrada al proceso de generación de código. El primero de ellos siguen siendo los modelos que representan la estructura y comportamiento del sistema. El segundo punto de entrada es un conjunto de plantillas de traducción independientes de la arquitectura de software de la aplicación, para dar a los usuarios mayor control sobre la traducción a código fuente para arquitecturas de software específicas.

Tomando en cuenta los diagramas resultantes modelados con ApEM-L así como la utilidad que proveen los mismos para generar las estructuras de clases, se realizó una evaluación de los enfoques anteriormente descritos tomando en consideración que cada uno provee un determinado grado de completitud en dependencia del tipo de diagrama utilizado. Por lo antes expuesto, se adoptó para la solución propuesta el enfoque estructural debido que permite generar el esqueleto del código referente a cada clase con sus atributos, relaciones y declaración de los métodos a partir del diagrama de clases. Esto posibilita una mayor aceptación del lenguaje de modelado ApEM-L por parte de los desarrolladores y proyectos que lo utilicen debido a la disminución de esfuerzos destinados al desarrollo, la reducción del tiempo y la facilidad de mantenimiento, traduciéndose así a un aumento de la productividad.

Las herramientas que dan soporte a la generación de código estructural lo hacen a través de un motor de traducción y plantillas preexistentes para especificar correspondencias con un código fuente en particular. Escritas en un lenguaje de script, las plantillas guían la traducción de los modelos en estructuras de código, como cabeceras de clases o declaraciones de funciones. Los lenguajes de scripts permiten a los diseñadores seguir estándares de codificación y crear plantillas nuevas para lenguajes no soportados. La generación de código estructural es incompleta, pero ahorra esfuerzo de codificación manual y proporciona un marco de trabajo inicial, consistente con los modelos.(Molina, 2003)

Por otra parte según la interacción con el código generado (Herrington, 2003) clasifica a los generadores de código en activos y pasivos:

Los **generadores activos** son aquellos que permiten generar varias veces sobre el mismo código generado a partir de cambios en la entrada. Estos generadores definen espacios de código seguros donde el programador puede hacer los cambios que desee sin que éstos se pierdan en las sucesivas generaciones de código.

Los **generadores pasivos** generan el código una vez y no vuelven a tener interacción con él. Tienen la desventaja de que si se corrige un error en los mecanismos de generación o se cambia el diseño y se vuelve a generar se pierde todo lo que se codificó manualmente.

Dada la necesidad de integrar el código escrito manualmente con las estructuras de clases generada se decidió brindar la característica de generación activa en la solución propuesta.

Generando código a través de plantillas

Las transformaciones de modelo a texto se basan en un conjunto de plantillas que siguiendo un determinado metamodelo definen la estructura del código o texto a generar. Por lo tanto, en las plantillas se distingue una parte estática y otra parte que tendrá que ser sustituida por la información extraída del modelo. Según (Molina, 2003) la generación mediante plantillas puede ser empleada cuando el código destino a producir tiene un patrón de repetición bien caracterizado, de modo que todos los ficheros generados son idénticos salvo los datos procedentes directamente del modelo a generar. En este caso puede definirse una plantilla genérica a partir de ejemplares del código objetivo. En esta plantilla, las dependencias del modelo son sustituidas por marcadores (o huecos) con nombre único.

Aparejado a la plantilla podemos definir un proceso de generación que dado un elemento de modelo, la plantilla sea instanciada a código mediante un proceso de sustitución de cadenas: los marcadores son sustituidos por los datos del modelo correspondiente.

Para la solución propuesta se utilizarán los elementos involucrados en esta aproximación, los cuales son descritos por (Molina, 2003) de la siguiente forma :(véase Figura 3)

1. *Documentos* en un lenguaje dado (objetivo de la generación).
2. Una *plantilla de documento* que da cuenta de la parte común y de los puntos donde aparecen las variabilidades.
3. Un *modelo o especificación*, que describe las variabilidades.
4. Un *metamodelo* que establezca las reglas de formación de los modelos.
5. Un *algoritmo de transformación* que traduce la plantilla en un documento a partir de información de un modelo.

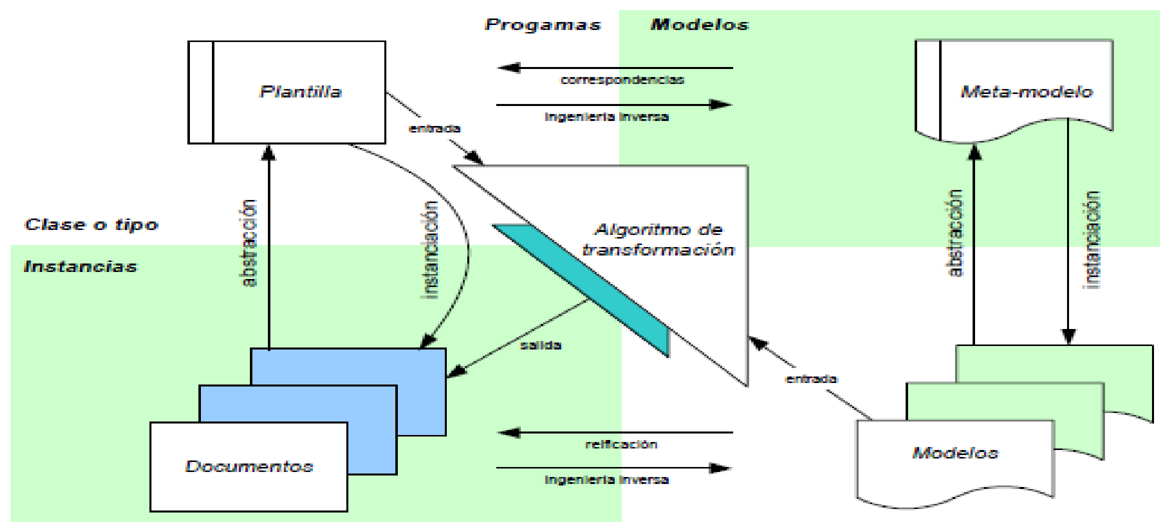

Figura 3: Elementos en la generación mediante plantillas [Tomado de (Molina, 2003)]

1.4. Metodología para el desarrollo del generador automático de código

Según (Figueroa *et al.*, 2011) la selección de la metodología de desarrollo a utilizar en un determinado proyecto es crucial para garantizar el éxito del producto. Comúnmente las metodologías son agrupadas en dos enfoques: el tradicional y el ágil.

A continuación la Tabla 3 recoge las principales diferencias de las metodologías ágiles con respecto a las tradicionales, estas diferencias afectan no sólo al proceso en sí, sino también al contexto del equipo y su organización.

Tabla 3: Diferencias entre metodologías tradicionales y ágiles [Tomado de (Figueroa et al., 2011)]

Metodologías Tradicionales	Metodologías Agiles
Basadas en normas provenientes de estándares seguidos por el entorno de desarrollo	Basadas en heurísticas provenientes de prácticas de producción de código
Cierta resistencia a los cambios	Especialmente preparados para cambios durante el proyecto
Impuestas externamente	Impuestas internamente (por el equipo)
Proceso mucho más controlado, con numerosas políticas/normas	Proceso menos controlado, con pocos principios.
El cliente interactúa con el equipo de desarrollo mediante reuniones	El cliente es parte del equipo de desarrollo
Más artefactos	Pocos artefactos
Más roles	Pocos roles
Grupos grandes y posiblemente distribuidos	Grupos pequeños (<10 integrantes) y trabajando en el mismo sitio
La arquitectura del software es esencial y se expresa mediante modelos	Menos énfasis en la arquitectura del software
Existe un contrato prefijado	No existe contrato tradicional o al menos es bastante flexible

La investigación responde a necesidades y condiciones reales del equipo de desarrollo, determinadas por el dominio de un lenguaje común, considerable experiencia en el desarrollo de soluciones de software y el amplio dominio de las tecnologías lo que contribuye a la confianza entre los integrantes, destreza para la toma de decisiones y organización propia. Además, el cliente está altamente comprometido y motivado

con el desarrollo del software y el equipo de trabajo está caracterizado por una elevada disposición y capacidad ante los nuevos cambios surgidos durante la investigación. Asimismo, el equipo de trabajo es pequeño y dispone de poco tiempo para el desarrollo. Lo antes expuesto determina la utilización de una **metodología ágil**.

A partir de la selección del enfoque ágil, se analizan varios exponentes de este tipo de metodología, tales como: XP (eXtreme Programming), Scrum y OpenUp, por su gran aceptación en el desarrollo de proyectos de software. A continuación se establece una comparación de dichas metodologías a partir de la información brindada en (Gimson, 2012):

Tabla 4: Comparación de metodologías ágiles

Indicadores	XP	Scrum	OpenUP
Nivel de documentación	Se sustituye la documentación escrita por la comunicación directa entre clientes y desarrolladores o entre los propios desarrolladores.	Poca	Suficiente y necesaria
Participación del cliente	Realimentación continua. El cliente tiene que estar presente y disponible todo el tiempo para el equipo.	Al final de cada iteración	Pertenece al equipo de trabajo
Realización de pruebas	Son ejecutadas constantemente ante cada modificación del sistema	Al final de cada iteración	Muchas veces en cada iteración
Nivel de flexibilidad	Bajo	Bajo	Alto
Enfoque	Centra su atención a la programación	Centra su atención a la gestión de proyectos	Ciclo Completo

Debido a las ventajas que ofrece OpenUP con respecto a las demás metodologías comparadas en cuanto a la documentación que genera, la flexibilidad que brinda al equipo de trabajo para la generación de artefactos, así como la posibilidad de organizar el trabajo y determinar la mejor forma de alcanzar los objetivos. También la realización de pruebas constantes en cada iteración garantiza la elaboración de una solución estable y disponible durante su desarrollo. Además el cliente puede ser consultado tantas veces

como sea necesario aunque no esté disponible a tiempo completo para el equipo de desarrollo y se le realizan continuas entregas del software que le aporten un valor.

OpenUP es una variante ágil del Proceso Unificado que aplica el desarrollo iterativo e incremental, dirigido por casos de uso y centrado en la arquitectura durante el ciclo de vida del software. Es una metodología ligera y extensible ya que los procesos se pueden agregar o adaptar según lo vayan requiriendo los sistemas. Dicha metodología se centra en balancear las prioridades para maximizar las necesidades de los stakeholders[17], además tiene como ventajas importantes la mitigación de riesgos y su utilización en proyectos con un reducido número de integrantes. Sin embargo si se maneja con cuidado y con profesionalismo se puede desarrollar un software de gran calidad, a pesar de que se le diseñe en poco tiempo y con poca documentación.

Esta metodología constituye un proceso iterativo cuyas iteraciones se distribuyen a través de cuatro fases: Inicio, Elaboración, Construcción y Transición las cuales son descritas a continuación:

- ✓ Inicio: Es la primera fase del ciclo de vida del proyecto, donde se produce el entendimiento del propósito y los objetivos, obteniendo suficiente información para confirmar lo que el proyecto debe hacer. El objetivo de esta fase es capturar las necesidades de los stakeholders durante el ciclo de vida del proyecto.
- ✓ Elaboración: Es la segunda fase del ciclo de vida del proyecto donde se tratan los riesgos significativos para la arquitectura. El propósito de esta fase es establecer la base para la elaboración de la arquitectura del sistema.
- ✓ Construcción: Esta fase está enfocada al diseño, implementación y prueba de las funcionalidades para desarrollar un sistema completo. El propósito de esta fase es completar el desarrollo del sistema basado en la arquitectura definida.
- ✓ Transición: Es la última fase, cuyo propósito es asegurar que el sistema es entregado a los usuarios, y evalúa la funcionalidad y el desempeño del último entregable de la fase de construcción.

OpenUP dirige la organización del trabajo en tres niveles: el personal, el equipo y el stakeholders como se muestra en la Figura 4:

[17] Stakeholder es un término inglés utilizado para referirse a “quienes pueden afectar o son afectados por las actividades de una empresa”

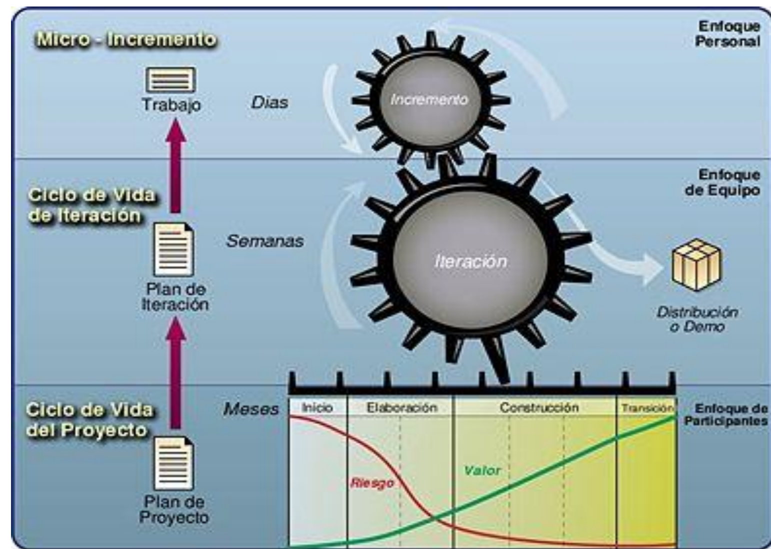

Figura 4: Capas de la Metodología OpenUP [Tomado de (Balduino, 2007)]

Desde la perspectiva personal de cada miembro del equipo el proyecto consta de micro-incrementos, pequeñas unidades de trabajo, que producen un continuo y mesurable ritmo de progreso del proyecto generalmente contado en horas o pocos días. Una colaboración intensiva entre los comprometidos en el micro-incremento le da el carácter incremental al desarrollo. Cada micro-incremento provee información de retroalimentación que permite tomar decisiones adaptativas al desarrollo en la iteración.

Según (Balduino, 2007) desde la perspectiva del equipo el proyecto consta, de iteraciones planeadas con encajonamiento de tiempo en intervalos que no pasan de unas pocas semanas y centradas en producir de una manera predecible un incremento del valor para los stakeholders. El equipo se auto-organiza centrado en cómo lograr los objetivos de la iteración y obtener el entregable.

1.5. Tecnologías a utilizar para el desarrollo del generador automático de código

A partir del objetivo propuesto en la investigación se determinó utilizar como plataforma Eclipse, facilitando la aceptación de la herramienta por parte de los desarrolladores al brindar un ambiente donde pueda modificarse, compilarse y probarse de manera inmediata el código generado. Eclipse debido a su diseño permite ser fácilmente extendido a través de plugins, los cuales son aplicaciones que se integran al IDE

para aportarle funcionalidades específicas; además de ser considerados la unidad mínima de funcionalidad que puede ser distribuida de manera separada.(Echemendia y Hervis, 2010)

Posteriormente se realizó un estudio de las tecnologías existentes para la implementación de generadores de código, acotando la búsqueda a aquellas que sean de código abierto, se puedan integrar con Eclipse, que ofrezcan mecanismos para dar soporte a la generación incremental, permitiendo así la personalización del código generado para adaptarlo a cualquier necesidad, estilo de programación o plataforma y por último que su funcionamiento se base en el uso de plantillas. A partir de lo planteado se tomó en consideración dos variantes: JET y Acceleo. Luego se realizó una comparación entre ambas herramientas, donde uno de los criterios más importantes para la selección fue que se pudiera integrar de manera inmediata con el editor visual desarrollado para ApEM-L sobre las bases del GMF[18].
(Riba, 2007) realiza una comparación entre ambas propuestas teniendo en cuenta los siguientes indicadores obteniendo como resultado los datos expuestos en la Tabla 5:

- **Integración con GMF:** Este aspecto evalúa la facilidad de integración con el editor gráfico.
- **Implementación de reglas de transformación:** Este aspecto evalúa el soporte que provee la herramienta con respecto a la implementación de las reglas que definen qué bloques de código se generará para cada elemento en el modelo.
- **Independencia de la plataforma del código generado:** Este aspecto evalúa la capacidad de la herramienta de generar cualquier tipo de archivo de texto que se necesite.
- **Fácil navegación por los elementos del modelo creado con el editor:** Este aspecto evalúa la facilidad que provee la herramienta para acceder a los elementos del modelo creado con el editor.

Facilidad de uso y aprendizaje: Este aspecto evalúa la facilidad de uso de la herramienta así como el nivel de programación que se requiere para hacer uso de ésta sin tener que llevar una curva de aprendizaje muy alta.

Concluyendo que Acceleo es de las dos herramientas la que más facilidades provee. Además, permite la generación de código a partir de un modelo creado en el editor gráfico generado por GMF siendo este uno

[18] Acrónimo en inglés de Graphical Modeling Framework

de los principales factores a tener en cuenta durante la investigación. Otra de las razones que justifican su selección, es el apoyo de la herramienta en un lenguaje estandarizado por la OMG, lo que le proporcionará una mayor aceptación de la comunidad científica. A continuación se describen las herramientas a utilizar para darle cumplimiento al objetivo propuesto.

Tabla 5: Comparación de herramientas de soporte para la generación de código [Tomado de (Riba, 2007)]

Indicadores	JET	Acceleo
Integración con GMF	Puede integrarse con GMF sin embargo la integración no es tan simple y se requiere de un alto nivel de programación para lograrlo.	La integración es inmediata y sencilla.
Implementación de las reglas de transformación	Debido a la riqueza del lenguaje utilizado por JET, cualquier regla de transformación puede ser implementada de manera rápida.	Acceleo tiene un lenguaje más restringido que JET, sin embargo, reglas de transformación bastante complejas pueden implementarse con este.
Independencia de plataforma del código generado	Se puede generar cualquier archivo de texto, en cualquier formato.	Se puede generar cualquier archivo de texto, en cualquier formato.
Fácil navegación por los elementos del modelo creado con el editor	La navegación por los elementos del modelo creado con el editor es una tarea de programación.	La navegación por lo elementos del modelo es completamente natural y simple, a través de editores especialmente diseñados para ello.
Facilidad de uso y aprendizaje	Se requiere leer mucha información y tener muy buenas bases de programación en Java para utilizar la herramienta.	Solo se requieren elementos básicos de programación para crear las plantillas e integrarlo con otras herramientas. Esto se debe a que sigue un enfoque de modelos como GMF.

Entorno integrado de desarrollo: Eclipse Galileo

Eclipse es una plataforma de software libre que alberga numerosos proyectos tales como *EMF*[19]*,* que es la base del DSM en Eclipse. En este IDE se puede encontrar todas las herramientas y funciones necesarias

---

[19] Acrónimo en inglés de Eclipse Modeling Framework

para trabajar, además tiene una atractiva interfaz que lo hace fácil y agradable de usar. Una de las ventajas que provee Eclipse es la posibilidad de ampliación y mejora mediante el uso de plugins que son usados para realizar actividades específicas tales como la transformación de modelo a texto, entre otras. La versión Galileo fue liberada en el 2009 y presenta entre sus principales tendencias la innovación de las tecnologías de modelado de Eclipse.(Moreno, 2013)

Acceleo

Según (Pérez, 2012) Acceleo es una herramienta para la creación de generadores de código, que implementa el lenguaje estándar para la transformación de modelo a texto desarrollado por el OMG y se denomina MOFM2T. Es suministrado como un plugins de Eclipse con licencia EPL[20] y desarrollado en el lenguaje Java bajo el marco de trabajo de EMF.

Este plugin proporciona herramientas para la generación de código a partir de modelos basados en EMF permitiendo la generación incremental. Esta forma de generación de código, brinda a los usuarios la capacidad de generar un fragmento de código, modificar el código generado y finalmente regenerar el código una vez más, sin perder las modificaciones anteriores.

Algunas de las características más importantes que posee Acceleo son las siguientes:

- Generación de código a través de cualquier tipo de metamodelo compatible con EMF como UML1, UML2, e incluso metamodelos adaptados a un dominio específico.
- Personalización de la generación con las plantillas definidas por el usuario.
- Generación de cualquier lenguaje textual (C, Java, Python).

Acceleo está basado en los principales estándares MDA, lo que garantiza su compatibilidad e interoperabilidad con otras aplicaciones, como GMF. Es compatible con XMI, lo que asegura compatibilidad con muchos de los modeladores de UML en el mercado. Además, permite extender la funcionalidad ofrecida mediante la importación de librerías de Java, que pueden utilizarse para agregar funcionalidad a las plantillas y la generación de código. En resumen, es una herramienta muy completa cuya principal fortaleza es basarse en un estándar internacional del OMG.

[20] Acrónimo en inglés de Eclipse Public Licence

Lenguaje de Programación: Java

Java es un lenguaje de Programación Orientado a Objeto (POO) y de propósito general. Fue desarrollado por Sun Microsystems[21] y constituye uno de los lenguajes más utilizados por los desarrolladores, debido al conjunto de características que este posee: (Zukowski, 2003)

- **Orientado a Objetos:** Se apoya en todos los conceptos asociados a la técnica orientado a objeto (OO) como por ejemplo el encapsulamiento, herencia, polimorfismo, etc.
- **Disponibilidad de un amplio conjunto de bibliotecas:** Pone a disposición del programador un amplio conjunto de clases con las cuales es posible realizar prácticamente cualquier tipo de aplicación.
- **Simplicidad:** Java posee una curva de aprendizaje muy rápida.
- **Interpretado y compilado a la vez:** Java es compilado, en la medida en que su código fuente se transforma en una especie de código máquina, los bytecodes[22], semejantes a las instrucciones de ensamblador. Por otra parte, es interpretado, ya que los bytecodes se pueden ejecutar directamente sobre cualquier máquina a la cual se hayan portado el intérprete y el sistema de ejecución en tiempo real.
- **Robusto:** Java fue diseñado para crear software altamente fiable. Para ello proporciona numerosas comprobaciones en compilación y en tiempo de ejecución.
- **Independiente de la arquitectura:** Java está diseñado para soportar aplicaciones que serán ejecutadas en los más variados entornos como Unix, Mac o Windows; el motivo de esto es que el que realmente ejecuta el código generado por el compilador no es el procesador del ordenador directamente, sino que este se ejecuta mediante una máquina virtual.

  Multitarea: Java soporta sincronización de múltiples hilos de ejecución a nivel de lenguaje, especialmente útiles en la creación de aplicaciones de red distribuidas.

---

[21] Empresa informática dedicada a vender estaciones de trabajo, servidores, componentes informáticos, sistemas operativos y otros servicios informáticos. En el año 2009, fue comprada por Oracle Corporation.

[22] El bytecode es un código intermedio más abstracto que el código máquina. Habitualmente es tratado como un archivo binario que contiene un programa ejecutable similar a un módulo objeto, que es un archivo binario producido por el compilador cuyo contenido es el código objeto o código máquina.

Herramienta de Modelado: Visual Paradigm for UML 8.0

Visual Paradigm es una herramienta CASE que soporta el ciclo de vida completo del desarrollo de software: análisis y diseño orientados a objetos, implementación y pruebas, facilitando la rápida construcción de aplicaciones de calidad. Debido al soporte que provee esta herramienta para la construcción de diagramas necesarios en el desarrollo de la investigación, la facilidad de generación automática de código en lenguajes como PHP, C++ y Java , así como el amplio conocimiento por parte de los desarrolladores al ser una de las herramientas más utilizadas en la universidad. Se decidió utilizar Visual Paradigm como herramienta de modelado y por consecuente UML.

Conclusiones Parciales

- Para el proceso de generación de código con el uso de ApEM-L 2.0, se analizaron un conjunto de herramientas de modelado. Estas herramientas presentan limitaciones para darle solución al problema de la investigación por brindarle soporte a UML como lenguaje de modelado y no permitir la definición de nuevos metamodelos.
- Las herramientas CASE para el modelado de aplicaciones brindan un conjunto de funciones automatizadas siendo la generación de código una de las más preciadas, ya que permite generar las estructuras de código a partir de especificaciones de diseño y para esto se utilizan las reglas de transformación.
- Existen tres enfoques para la generación de código basado en modelos. El enfoque estructural el que mejor se adapta a las necesidades de la investigación teniendo en cuenta los tipos de diagramas obtenidos de la actividad de modelado usando ApEM-L 2.0.
- Las características del generador automático de código a desarrollar durante la investigación determinó el uso de las siguientes herramientas y tecnologías: Eclipse como Entorno Integrado de Desarrollo, Acceleo como herramienta para la creación del generador de código fuente, Java como lenguaje de programación, UML como lenguaje de modelado y Visual Paradigm como herramienta CASE asociada al proceso de desarrollo del software.

CAPÍTULO 2. DESCRIPCIÓN Y DISEÑO DEL PLUGIN PARA LA GENERACIÓN DE CÓDIGO PHP 5

En un primer momento del capítulo se realizará una caracterización del proceso de generación de código en las tesis de grado de la UCI que han utilizado ApEM-L en su versión 1.0, para el modelado de las aplicaciones educativas. Posteriormente, se presenta una descripción de los conceptos más importantes para facilitar la comprensión del contexto del sistema. Se definen tanto los requisitos funcionales como los no funcionales, los patrones de diseño y la arquitectura a utilizar, así como los diagramas generados durante la fase de diseño.

2.1. Caracterización del proceso de generación de código

Luego de una exploración de la realidad mediante entrevistas realizadas a los profesores y especialistas informáticos que han usado ApEM-L en las versiones 1.0 o 1.5 **(Anexo 4)**, así como el análisis de las tesis de perfil educativo desarrolladas en la UCI desde el año 2008 hasta el 2011, se pudo determinar que el 70% utilizó ApEM-L en su versión 1.0 como lenguaje de modelado de dichas aplicaciones, el otro 17% y 13% utilizaron OMMMA-L y UML respectivamente.

El análisis de las diez tesis de grado que implementaron las aplicaciones educativas propuestas arrojó como resultados que:

- ✓ Un 50% utilizó Visual Paradigm y el otro 50% utilizó Rational Rose como herramienta CASE asociada al proceso de desarrollo del software. Sin embargo, ninguna de estas herramientas utilizan el metamodelo de ApEM-L 2.0; así como el grado de transformación a código fuente es bajo, debido a que no se codifican todos los componentes del diagrama de clases del diseño.
- ✓ Basado en los valores tomados por los indicadores definidos para medir las variables de la investigación que se muestran en la Tabla 6, se determinó que la consistencia y completitud del código generado es media y baja respectivamente, debido a que los estereotipos aplicados al diagrama no son comprendidos por la herramienta y por lo tanto estos no se ven reflejados en el código.

Tabla 6: Valores tomados por los indicadores de las variables definidas en la investigación

Variables	Indicadores	Valores
Consistencia	Cantidad de clases y relaciones del diagrama que han sido codificadas	Alta
	Representación de los estereotipos	Baja
Completitud	Calidad de la documentación del código	Media
	Transformación a código de los componentes del diagrama	Baja

2.2. Propuesta de Solución

Llegado a este punto de la investigación y teniendo en cuenta que ApEM-L es el lenguaje más utilizado para el modelado de aplicaciones educativas en la UCI, y sobre la base de lo que se plantea en los fundamentos referentes a las tecnologías seleccionadas en el capítulo anterior, se propone como solución: el desarrollo de un plugin para Eclipse que permita la generación automática de código fuente PHP 5 a partir del diagrama de clases del diseño modelado con ApEM-L 2.0. Para la implementación del plugin se utilizó Acceleo donde se definieron un conjunto de plantillas que contienen las reglas de transformación, encargadas de generar las estructuras de código correspondiente a cada uno de los componentes del diagrama. El plugin dará soporte a la generación incremental y será capaz de comprender los estereotipos restrictivos definidos por el lenguaje de modelado, garantizando así la consistencia y completitud del código. Además la solución propuesta detectará las violaciones realizadas al metamodelo de ApEM-L 2.0 una vez regenerado el código y ofrecerá una exhaustiva documentación del mismo con el objetivo de guiar a los programadores durante su trabajo.

2.3. Modelo del Dominio

Para construir un sistema correcto los desarrolladores requieren un firme conocimiento del contexto en el que se emplaza el sistema, existiendo dos formas de expresar el mismo: el modelado del negocio y el modelado del dominio donde la selección de una u otra variante estará en dependencia de las características del entorno organizacional en que se esté trabajando. En (Ciudad y Herrera, 2006) se plantea la utilización de un diagrama conceptual del modelo de dominio para la representación de entornos con flujos de información complejos y difusos, por lo tanto debido a que el entorno en presencia no se comporta como un negocio donde se tienen definidos los flujos de información y las responsabilidades, los autores decidieron realizar un modelo del dominio(ver Figura 5) para capturar los tipos más importantes de objetos en el contexto del sistema. Los objetos del dominio representan las "cosas" que existen o los eventos que suceden en el entorno en que se trabaja. Según (Jacobson *et al.*, 2000) este modelo se describe la mayoría de las veces mediante el diagrama de clases de UML y tiene como objetivo fundamental comprender y describir las clases más importantes del entorno objeto de informatización.

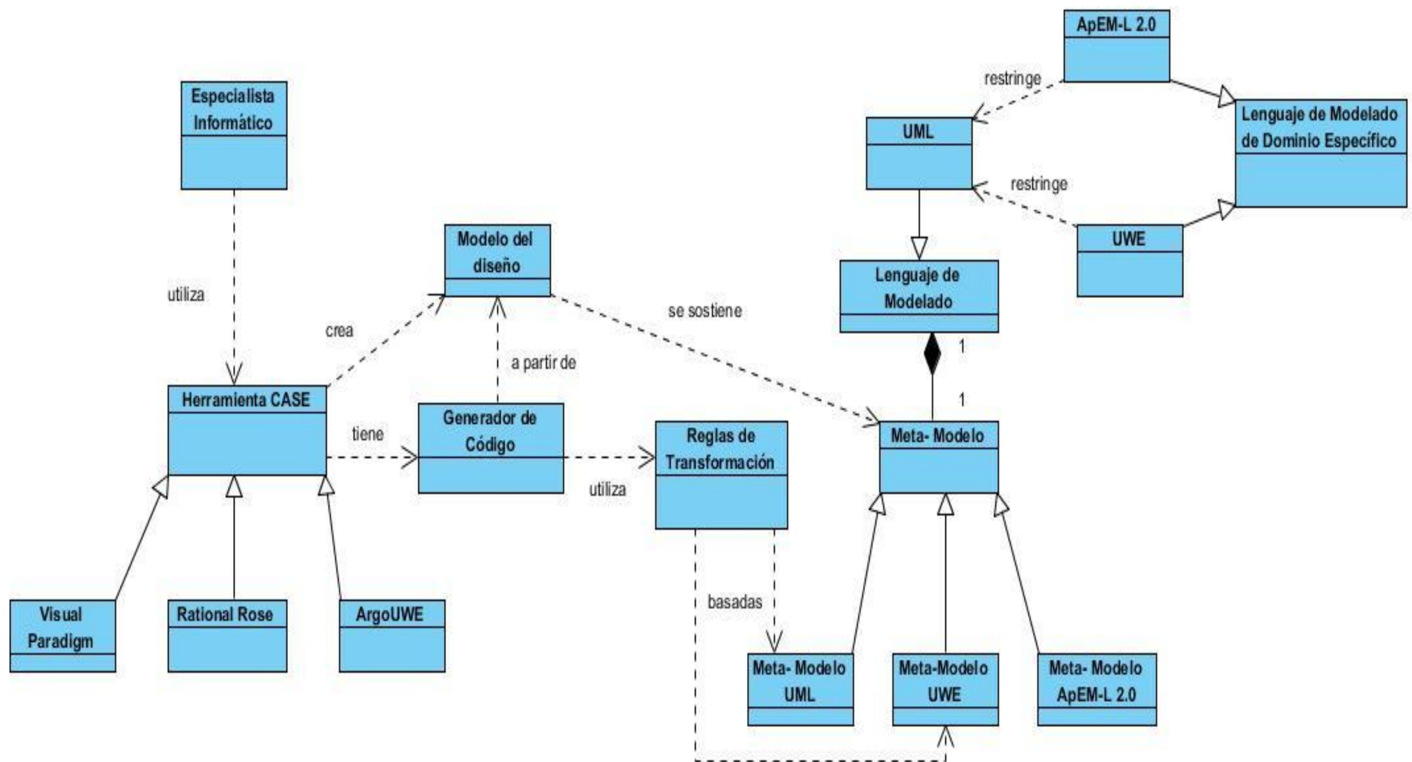

Figura 5: Modelo del Dominio

Definición de los conceptos del modelo de dominio:

- ✓ **Especialista Informático**: Es cualquier persona que desempeñe un determinado rol dentro del proceso de desarrollo de software ejemplo: el analista, el desarrollador, etc.
- ✓ **Herramienta CASE**: Es el software que permite al especialista informático modelar a través de diagramas el sistema a desarrollar.

- ✓ **Visual Paradigm**: Es una herramienta CASE que se utiliza para realizar la actividad de modelado. Esta herramienta utiliza UML como lenguaje de modelado, además de que permite dibujar todos los tipos de diagramas, generar código desde diagramas y realizar la documentación.
- ✓ **Rational Rose**: Es una herramienta CASE que se utiliza para realizar actividad de modelado. Esta utiliza el lenguaje de modelado UML, además de que permite dibujar todos los tipos de diagramas, generar código desde diagramas y realizar la documentación.
- ✓ **ArgoUML**: Herramienta para modelar sistemas, mediante el cual se realizan diseños en UML. Esta herramienta puede crear la mayoría de los diagramas estándares de UML.
- ✓ **Generador de Código**: Es la herramienta que permite realizar automáticamente el proceso de generación de código a partir de diagramas previamente elaborados.
- ✓ **Modelo del diseño**: Es el modelo que captura las decisiones de representación interna con un alto nivel de detalles.
- ✓ **Reglas de transformación**: Describen como un modelo en el lenguaje fuente puede ser transformado en definiciones de texto en el lenguaje destino.
- ✓ **Meta-Modelo:** Es el modelo que especifica la sintaxis abstracta del lenguaje de modelado por lo tanto cada lenguaje de modelado tiene un metamodelo asociado.
- ✓ **Meta-Modelo UML**: Es el modelo que especifica la sintaxis abstracta del lenguaje de modelado UML.
- ✓ **Meta-Modelo UWE**: Es el modelo que especifica la sintaxis abstracta del lenguaje de modelado UWE.
- ✓ **Meta-Modelo ApEM-L 2.0**: Es el modelo que especifica la sintaxis abstracta del lenguaje de modelado ApEM-L 2.0.
- ✓ **Lenguaje de modelado:** Es el lenguaje en el cual son expresados los modelos siendo UML el lenguaje de propósito general más utilizado en la actualidad para el modelado de software.
- ✓ **Lenguaje de Modelado de Dominio Específico:** Es el lenguaje en el cual son expresados los modelos de un dominio de aplicación concreto.

- ✓ **UML**: Es uno de los lenguajes de modelado más utilizados por la industria del software para realizar el modelado de los sistemas, pero a pesar de que presente un gran número de ventajas, presenta limitaciones para la creación de modelos detallados de un dominio de aplicación concreto.
- ✓ **UWE:** Es un lenguaje de modelado de dominio específico, para el desarrollo de aplicaciones web orientado a objetos y constituye una extensión de UML.
- ✓ **ApEM-L 2.0**: Es la versión actual del lenguaje de modelado de dominio específico, para el desarrollo de aplicaciones educativas y multimedia; constituye una extensión de UML.

2.4. Diagrama de Flujo

El diagrama de flujo según (FUNDIBEQ, 2011) es la representación gráfica de la secuencia de pasos que se realizan para obtener un cierto resultado. Este puede ser un producto, un servicio o bien una combinación de ambos. Los diagramas de flujos se caracterizan por su:

Capacidad de comunicación: Permite presentar los conocimientos individuales sobre un proceso facilitando así la comprensión global del mismo.

Claridad: Proporciona información sobre los procesos de forma clara, ordenada y concisa.

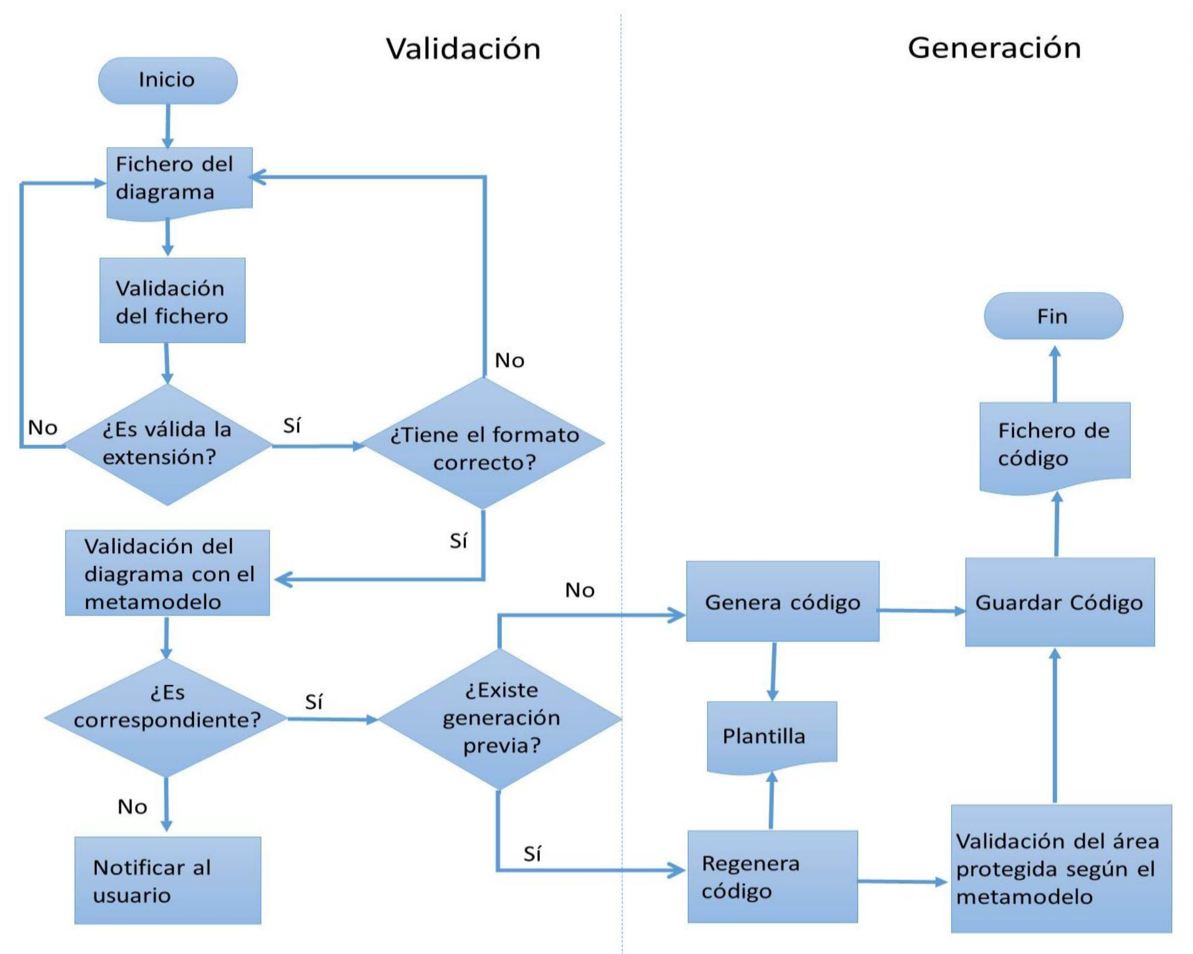

Figura 6: Diagrama de flujo del proceso de generación de código

2.5. Especificación de Requisitos

(Pressman, 2005) plantea que la especificación de requisitos constituye una de las tareas más difíciles e importante que debe enfrentar un ingeniero de software. La ingeniería de requisitos proporciona el mecanismo apropiado para entender lo que el cliente quiere, analizar las necesidades y especificar la solución sin ambigüedades; esta especificación será la base para las actividades subconsecuentes de la

Ingeniería del Software ya que en ella se describe la función y desempeño del sistema así como las restricciones que regirán su desarrollo.

Requisitos Funcionales

Los requisitos funcionales (RF) representan las capacidades o funciones que el sistema debe cumplir por lo tanto en el presente trabajo se identificaron los siguientes requisitos:

Tabla 7: Listado de los requisitos funcionales del plugin a desarrollar

Requisito Funcional	Descripción
RF1- Generar código PHP 5.	Se genera el código fuente en lenguaje PHP 5 correspondiente a las clases del diagrama de clases del diseño modelado con ApEM-L 2.0.
RF1.1- Validar el fichero de entrada.	Se valida la extensión y formato del fichero de entrada a partir del cual se genera el código.
RF1.2- Validar la correspondencia del diagrama con el metamodelo.	Se valida que el diagrama a partir del cual se genera el código no viole el metamodelo de ApEM-L 2.0.
RF1.3- Validar las áreas protegidas de código según el metamodelo.	En cada regeneración se valida que en las áreas protegidas de código el desarrollador no haya introducido violaciones del metamodelo de ApEM-L 2.0
RF1.4- Guardar el código generado.	Se guardan en una dirección física los ficheros de código generados.

Requisitos No Funcionales

Los requisitos no funcionales (RNF) son propiedades o cualidades que el producto debe tener y representan las características del producto. Estos requisitos pueden clasificarse en cuanto al software, el hardware, la usabilidad y seguridad entre otras clasificaciones; a continuación se presentan los requisitos no funcionales identificados en el presente trabajo:

Software:

RNF1- Requiere la instalación de la Máquina Virtual de Java.

RNF2- Para su funcionamiento debe ser instalado algunas de las versiones del IDE Eclipse de la 3.4 – 3.6. y Acceleo en su versión 3.0.

Hardware:

RNF3- Mínimo 1 GB de RAM (Random Access Memory, por sus siglas en inglés), pero se recomienda 2,0 GB.

Interfaz:

RNF4- El plugin debe brindar una interfaz lo más descriptiva y amigable posible, permitiendo la fácil interacción con la misma, así como que las operaciones a realizar por los usuarios estén bien descritas, de manera que no existan ambigüedades.

Usabilidad:

RNF5- El código generado debe tener una adecuada documentación de su estructura, una indentación consistente y utilizar estilos de codificación que permitan la legibilidad del código.

RNF6- La herramienta debe proteger el código implementado por el desarrollador para posteriores regeneraciones.

Portabilidad:

RNF7- El plugin podrá ser instalado en Eclipse y utilizado en diferentes sistemas operativos por ser Eclipse una herramienta multiplataforma.

2.6. Patrón Arquitectónico: Modelo-Vista-Controlador

Para la solución del problema se hizo necesario dividir las responsabilidades asociadas a los datos de la aplicación, la interfaz de usuario y el control del flujo de datos, siendo este uno de los enfoques mayormente utilizados en las aplicaciones de generación de código. A partir de lo planteado anteriormente, el patrón arquitectónico con mayor probabilidad de ofrecer resultados satisfactorios es modelo-vista-controlador debido a que el mismo permite definir tres componentes fundamentales descritos en (Ciudad, 2006):

- ✓ **Modelo:** es la representación específica de la información con la cual el sistema opera. El modelo encapsula los datos y las funcionalidades, además de ser independiente de cualquier representación de salida y/o comportamiento de entrada.
- ✓ **Vista:** presenta el modelo en un formato adecuado para interactuar con el usuario, es responsable de la lógica de presentación y captura de datos del sistema.
- ✓ **Controlador:** recibe las entradas como eventos, estos eventos son traducidos a solicitudes de servicio para el modelo y/o la vista.

Cada uno de los componentes antes mencionados se encuentran representados en el plugin desarrollado con Acceleo donde las plantillas constituyen los modelos, estas son las encargadas de acceder a los datos contenidos en el diagrama para generar las estructuras de código. Por otra parte la vista se encarga de mostrar las opciones que le brinda el plugin a los usuarios y de la captura de datos para guardar el código, por último las controladoras capturan los eventos producidos por el usuario para posteriormente ejecutar las plantillas y generar así el código deseado.

2.7. Patrones de Diseño

En (Pressman, 2005) se plantea que el diseño basado en patrones es una técnica que reutiliza elementos de diseño que han probado ser exitosos en el pasado. Los patrones de diseño son aplicados a elementos específicos del diseño como un agregado de componentes para resolver algún problema de diseño, relaciones entre los componentes o los mecanismos para efectuar la comunicación de componente a componente.

Patrones de asignación de responsabilidad (GRASP)

Bajo acoplamiento

Según (Larman, 1999) el acoplamiento mide qué tan fuerte se encuentra una clase conectada con otras. Una clase con bajo acoplamiento no depende de muchas otras clases, por lo que en caso de producirse una modificación en alguna de estas, se tendrá la mínima repercusión posible en el resto de clases,

potenciando la reutilización. Este patrón se tuvo en cuenta para el diseño del diagrama de clases de la solución con el fin de lograr una dependencia mínima entre las clases posibilitando que se adapten mejor a los cambios.

Alta cohesión

Según (Larman, 1999) la cohesión es una medida de cuán relacionadas y enfocadas están las responsabilidades de una clase. Una alta cohesión caracteriza a las clases con responsabilidades estrechamente relacionadas que no realicen un trabajo enorme. Este patrón se utilizó en la solución para la creación de las plantillas, donde cada una se enfoca en un componente específico del diagrama pero se encuentran estrechamente relacionadas para generar las estructuras de código correspondientes.

Experto

Según (Larman, 1999) el uso de este patrón permite asignar las responsabilidades a la clase que cuenta con la información necesaria para cumplir dicha responsabilidad. Dicho patrón se evidencia en las siguientes clases: *BuscadorDeFicheros* la cual tiene la responsabilidad de buscar recursivamente en una dirección especificada los archivos con extensión .php; *ServiciosJava* que brinda un conjunto de funciones que son complejas de realizar utilizando plantillas.

Creador

Según (Larman, 1999) el uso de este patrón permite identificar quién debe ser el responsable de la creación de nuevos objetos o clases. Dicho patrón se evidencia en la clase *GenerateAll* la cual es responsable de crear instancias de las clases *Principal y ApemlParser,* a su vez esta última clase también crea una instancia de la clase *BuscadorDeFicheros.*

Controlador

Según (Larman, 1999) este patrón se encarga de asignar la responsabilidad de controlar el flujo de eventos del sistema, a clases específicas. En el plugin desarrollado este patrón es aplicado a las clases *GenerateAll* y *Principal*, donde se delega la responsabilidad del manejo de mensajes favoreciendo así la reutilización de la lógica para manejar los procesos a fines en aplicaciones futuras.

2.8. Diagrama de clases del diseño

El diagrama de clases es uno de los más importantes de la fase de diseño, siendo esta la etapa en la que se fomentará la calidad en la Ingeniería del Software. El diseño es la única forma en que, de manera exacta un requisito del cliente puede convertirse en un sistema o producto de software terminado. Los elementos del modelo del diseño utilizan muchos de los diagramas de UML aplicados en el modelo de análisis. La diferencia es que estos diagramas están refinados y elaborados como parte del diseño; se proporciona un mayor detalle para la implementación específica y se resaltan la estructura y el estilo arquitectónico.(Pressman, 2005)

A continuación se muestra en la Figura 7 el diagrama de clases del diseño para la actividad más significativa del plugin: **Generar código**

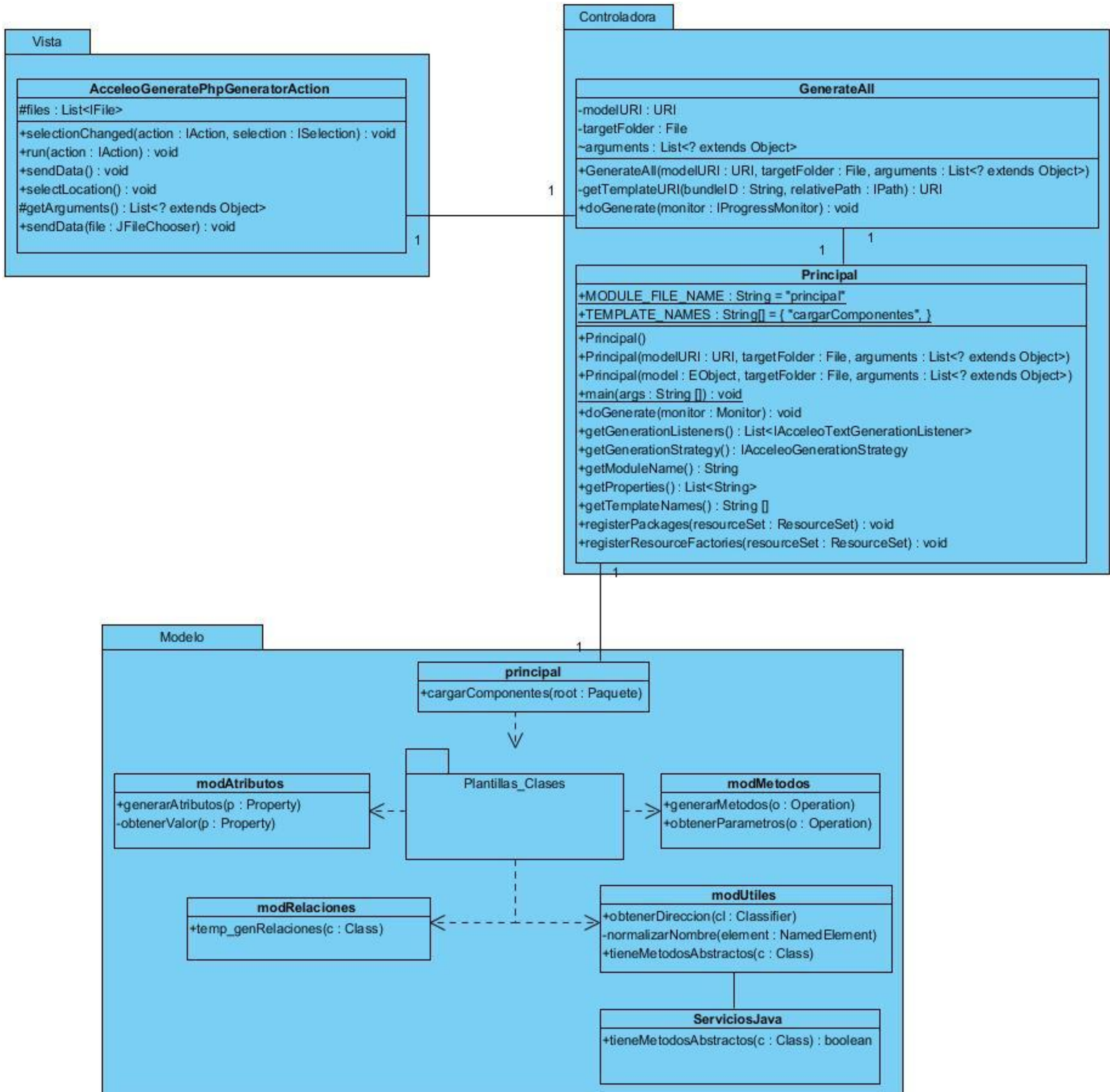

Figura 7: Diagrama de clases del diseño

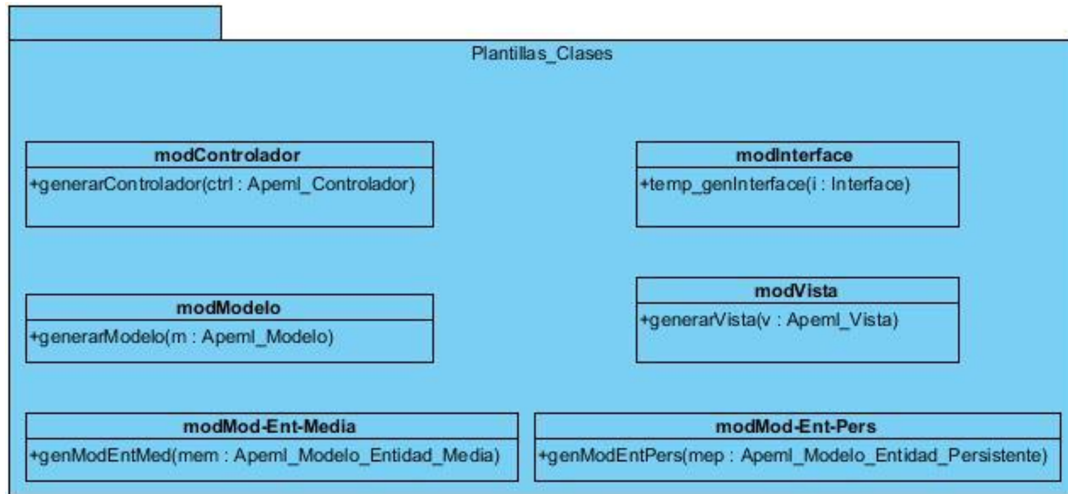

Figura 8: Paquete de plantillas para cada tipo de clases

Tabla 8: Descripción de las clases del diseño

Nombre	Descripción
AcceleoGeneratePHPGeneratorAction	Es la clase que permite actualizar cada acción realizada a través de la herramienta, así como ejecutar las mismas.
GenerateAll	Contiene los métodos encargados de ofrecer los datos necesarios para la generación de código.
Principal	Contiene los métodos necesarios para realizar la validación del metamodelo e iniciar la generación de código accediendo a las plantillas implementadas.
principal	Es la primera plantilla que será ejecutada y será la encargada de llamar a las plantillas correspondientes para generar las clases del diagrama.
Plantillas_Clases	Este paquete contiene las clases con los métodos necesarios para definir la estructura del código a generar para cada tipo de clase del diagrama.
modAtributos	Contiene los métodos necesarios para la declaración de los atributos de cada una de las clases del diagrama.

Nombre	Descripción
modMetodos	Contiene las operaciones necesarias para la declaración de los métodos de cada una de las clases del diagrama.
modRelaciones	Contiene el método necesario para definir como representar cada tipo de relación del diagrama en el código generado para cada una de las clases implicadas.
modUtiles	Contiene un conjunto de métodos comunes y consultas que utilizarán lo implementado en ServiciosJava.
ServiciosJava	Contiene los métodos que serán utilizados para realizar un conjunto de acciones complejas de hacer a través de plantillas.

2.9. Diagrama de Interacción

Según (Rumbaugh *et al.*, 2007, p 34) «*la vista de interacción describen el comportamiento mostrando el intercambio de secuencias de mensajes entre las partes de un sistema. Esta vista se muestra mediante dos diagramas que se centran en aspectos distintos: el diagramas de secuencia y el de colaboración*». Los diagramas de secuencia muestran la secuencia de mensajes entre objetos durante un escenario concreto, mientras que los diagramas de colaboración son útiles para la identificación de objetos y la interacción entre estos. A continuación se presenta el diagrama de colaboración que muestra la interacción de los objetos para generar código de las clases modelo entidad persistente en particular y guardar el mismo en el propio workspace.

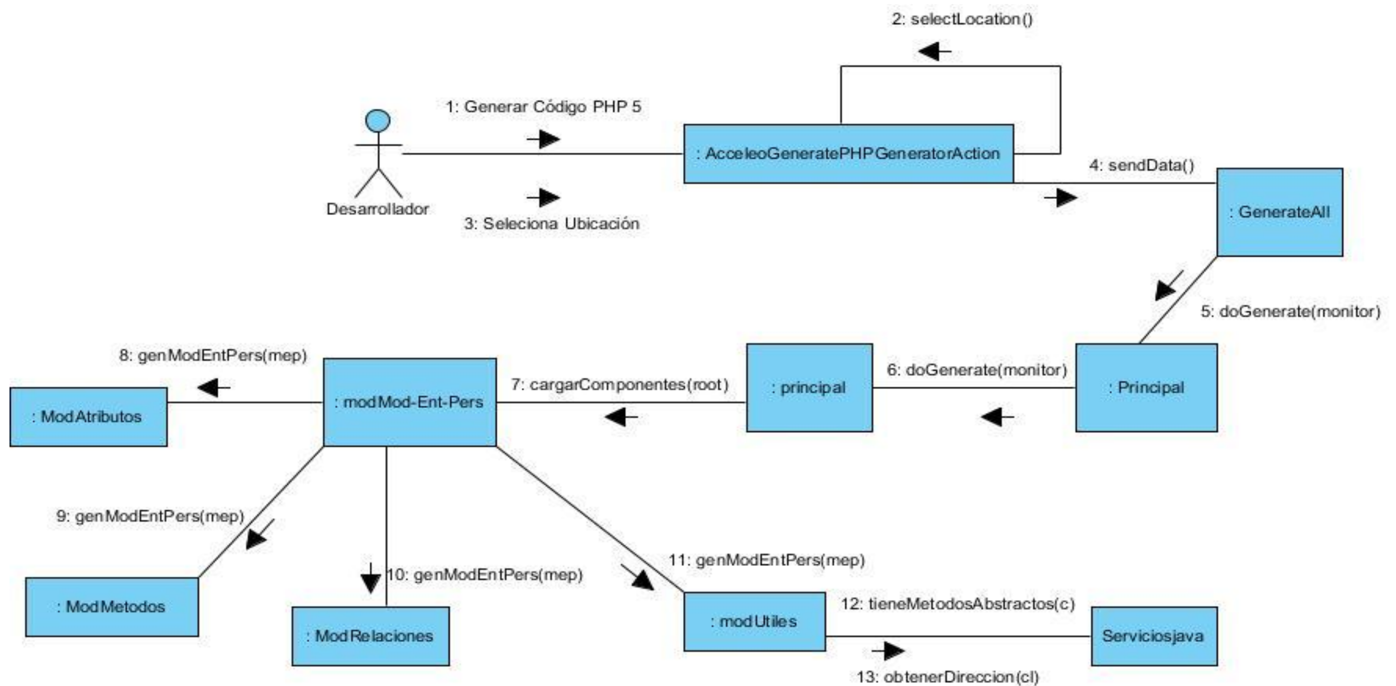

Figura 9: Diagrama de colaboración

Conclusiones parciales

- La caracterización del proceso de generación de código actual para el dominio de las aplicaciones educativas modeladas con ApEM-L, permitió constatar que las herramientas de modelado utilizadas no prestan soporte al metamodelo de ApEM-L 2.0 por lo que la consistencia y completitud del código generado son media y baja respectivamente.
- El análisis del flujo de información correspondiente al proceso de generación de código así como del entorno objeto de informatización permitió identificar cinco requisitos funcionales y ocho no funcionales dirigidos a resolver las necesidades del cliente.
- Para separar en la solución propuesta las responsabilidades asociadas a los datos de la aplicación, la interfaz de usuario y el control del flujo de datos siendo este el enfoque

mayormente utilizado por las herramientas de generación de código se utilizó el patrón arquitectónico MVC.

CAPÍTULO 3. IMPLEMENTACIÓN Y PRUEBAS DEL PLUGIN PARA LA GENERACIÓN DE CÓDIGO PHP 5

En el presente capítulo se analizará el modelo de Implementación como artefacto poniendo en práctica el diseño de la solución. Se comenzará a implementar el sistema en términos de componentes y se describirán las reglas de transformación que serán usadas en las plantillas para generar código PHP 5 correspondiente al diagrama de clases del diseño de ApEM-L 2.0. Además, se describirán las pruebas a realizar, con el objetivo de comprobar las funcionalidades de la herramienta en los diferentes escenarios, para de esta forma verificar en todos los casos que los resultados de las pruebas sean los esperados.

3.1. Modelo de Implementación

(Jacobson *et al.*, 2000) plantea que el modelo de implementación describe cómo los elementos del modelo del diseño, como las clases, se implementan en términos de componentes, por ejemplo en ficheros de código fuente, ejecutables, entre otros. Este modelo describe también cómo se organizan los componentes de acuerdo a los mecanismos de estructuración y modularización disponibles en el entorno de implementación y en el lenguaje de programación utilizado, y cómo dependen los componentes unos de otros.

Diagrama de Componentes

Según (Rumbaugh *et al.*, 1999) un diagrama de componentes modela la vista de implementación donde se muestra el empaquetado físico de las partes reutilizables del sistema. Esta vista expone la implementación de los elementos del diseño (tales como clases) mediante componentes, así como sus interfaces y dependencias entre componentes, siendo los componentes las piezas reutilizables de alto nivel a partir de las cuales se pueden construir los sistemas.

Los autores del Manual de Referencia del Lenguaje Unificado de Modelado en su primera edición mencionan cinco estereotipos estándar que define UML define para ser aplicados a los componentes:

- **<< executable >>:** Especifica un componente que se puede ejecutar en un nodo.
- **<< library>>:** Especifica una biblioteca de objetos estática o dinámica.

- **<< table>>:** Especifica un componente que representa una tabla de una base de datos.
- **<< file>>:** Especifica un documento que contiene código fuente o datos.
- **<< document>>:** Especifica un componente que representa un documento.

A continuación se presenta el diagrama de componentes correspondiente al generador automático de código:

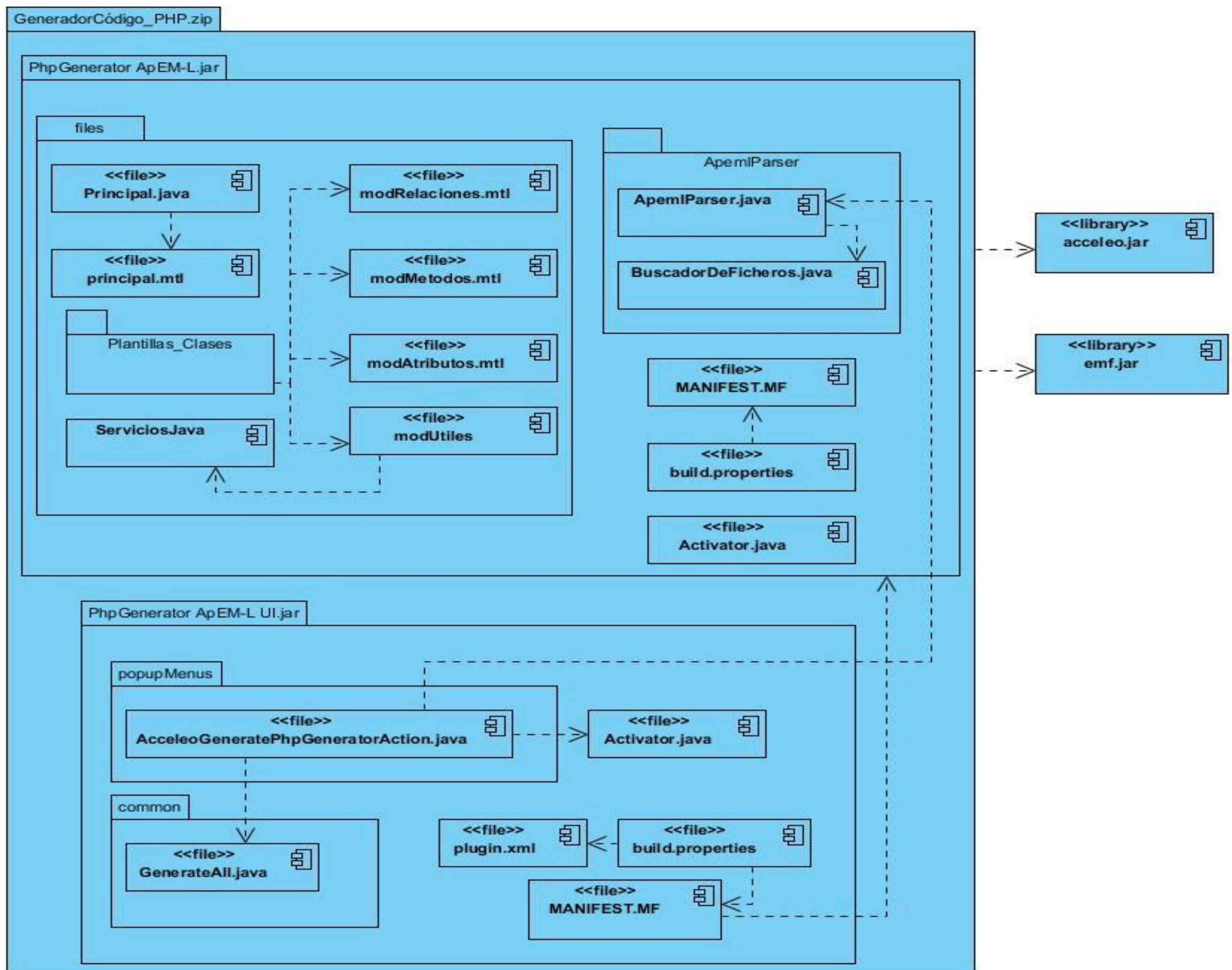

Figura 10: Diagrama de componentes

Descripción de componentes

GeneradorCódigo_PHP.zip: Representa el instalador para eclipse que realizará la generación automática de código PHP 5 a partir de los diagramas de clases de ApEM-L 2.0 dado que el mismo contiene los plugin necesarios para llevar a cabo este proceso.

PHPGenerator ApEM-L.jar: "PHPGenerator ApEM-L" representa el plugin que realiza transformaciones de modelo a texto generando código fuente PHP 5. Para cumplir con este objetivo el plugin agrupa un conjunto de componentes asociados al proceso de generación de código, las cuales son:

***Activator.java*:** Su función está dirigida a controlar el ciclo de vida del plugin.

***Principal.java*:** Se encarga de realizar las principales funcionalidades en el proceso de transformación del diagrama de clases del diseño al código fuente usando las librerías acceleo.jar y emf.jar.

***ServiciosJava.java*:** Se encarga de realizar un conjunto de funcionalidades que serían difíciles realizar usando plantillas.

***principal.mtl*:** Se encarga de determinar las plantillas a usar para generar código de cada tipo de clase del diagrama.

ApemlParser.java: Tiene los métodos necesarios para a validar en cada regeneración que no existan violaciones del metamodelo.

BuscadorDeFicheros.java: Tiene los métodos necesarios para buscar los ficheros con extensión php a partir de una dirección especificada.

Plantillas_Clases: En este paquete se encuentran los archivos mtl que se encargan del acceso a todos los tipos de clases del diagrama así como definir la estructura del código a generar para cada una de las clases en lenguaje PHP 5.

***modAtributos.mtl*:** Se encarga del acceso a los atributos de cada clase del diagrama así como definir la declaración de los mismos.

modMetodos.mtl: Se encarga del acceso a los métodos de cada clase del diagrama así como definir la declaración de los mismos.

***modRelaciones.mtl*:** Se encarga del acceso a las relaciones entre clases del diagrama así como definir como representar cada tipo de relación en el código generado para cada clase implicada en la relación.

modUtiles.mtl: Contiene un conjunto de métodos comunes a utilizar en las demás plantillas además de consultas utilizando lo implementado en la clase ServiciosJava.

PHPGenerator ApEM-L UI.jar: "PHPGenerator ApEM-L UI" representa el plugin encargado de la interfaz de usuario de "PHPGenerator ApEM-L". Agrupa un conjunto de componentes tales como: Activator.java, GenerateAll.java y AcceleoGeneratePHPGeneratorAction.java así como las clases asociadas a la configuración del plugin como build.properties, plugin.xml y MANIFEST.MF.

GenerateAll.java: Es la clase referente al control de las acciones que se llevan a cabo en la interfaz.

AcceleoGeneratePHPGeneratorAction.java: Se encarga de ejecutar las acciones definidas en la herramienta, así como de gestionar toda la interfaz de usuario y realizar las validaciones necesarias.

MANIFEST.MF: Este fichero es generado dentro del directorio META-INF cuando es creado un plugin. Es un manifiesto del paquete OSGi en donde se describen las dependencias que posee el plugin con otros plugins en tiempo de ejecución y sus atributos como son: el nombre que posee el plugin, el proveedor, la versión del mismo entre otros.

plugin.xml: se encuentra en la misma raíz del proyecto y consiste en un archivo de formato XML que describe los puntos de extensión que el propio plugin ha definido, y las acciones a ejecutar por el mismo.

build.properties: contiene cadenas de caracteres externalizadas que son referenciadas en el plugin.xml.

3.2. Reglas para la generación automática de código

Como resultado de la tendencia actual al desarrollo de aplicaciones web educativas, se determinaron las reglas necesarias para realizar la transformación a código PHP 5, siendo este uno de los lenguajes más utilizados. A partir del metamodelo de ApEM-L 2.0, en particular el paquete de vista estática, se proponen un conjunto de reglas de transformación que permiten relacionar cada instancia del metamodelo con el código fuente que puede obtenerse a partir de éste.

El principal elemento del diagrama de clases es la clase. Por tal motivo, las reglas de transformación se definen a partir de este elemento. En el metamodelo de ApEM-L existen 6 conceptos que son catalogados como clases: vista, modelo, controlador, modelo entidad – media, modelo entidad – persistente y HLL, las que heredan de la metaclase Class de UML. Teniendo en cuenta estos elementos, a continuación se presentan las reglas de transformación implementadas por la herramienta.

R1_ABS- Regla de clases abstractas: Toda instancia c de las metaclases Apeml_Controlador, Apeml_Modelo, Apeml_Modelo_Entidad_Media, Apeml_Modelo_Entidad_Persistente o Apeml_Vista cuyo atributo isAbstract sea verdadero, serán clases abstractas. En el caso de que una clase tenga algún método abstracto, la clase pasará a ser abstracta automáticamente.

∀ c ∈ Apeml_Controlador OR Apeml_Modelo OR Apeml_Modelo_Entidad_Media OR Apeml_Modelo_Entidad_Persistente OR Apeml_Vista (c.isAbstract ⇒ 'abstract')

R2_DEFC- Regla de definición de clase: Toda instancia c de las metaclases Apeml_Controlador, Apeml_Modelo, Apeml_Modelo_Entidad_Media, Apeml_Modelo_Entidad_Persistente o Apeml_Vista permitirán el acceso al nombre de la misma a través del atributo name.

∀ c ∈ Apeml_Controlador OR Apeml_Modelo OR Apeml_Modelo_Entidad_Media OR Apeml_Modelo_Entidad_Persistente OR Apeml_Vista ⇒ 'class' c.name '{' '}'

R3_ATTRP- Regla de atributos propios de la clase: Los atributos propios de una clase se obtienen a través de la consulta *ownedAttribute* definida para cada clase c. Por otra parte los datos de un atributo tales como la multiplicidad, visibilidad, tipo y nombre se puede obtener a través de los atributos upper, visibility, type y name respectivamente de una instancia p de la superClase Property.

∀ c ∈ Apeml_Controlador OR Apeml_Modelo OR Apeml_Modelo_Entidad_Media OR Apeml_Modelo_Entidad_Persistente OR Apeml_Vista ∀ p ∈ Property (p | c. *ownedAttribute* ⇒

//si la multiplicidad es 1, el valor del atributo es un elemento

p.upper=1 ⇒ (p.visibility '$'p.name = p. obtenerValor() ';')

//si la multiplicidad es -1 o mayor que 1 el valor que toma el atributo es un conjunto de elementos

(p.upper = -1 or p.upper > 1) ⇒ (p.visibility '$' p.name = 'array' (p. obtenerValor ()) ';')

R4_OP- Regla de operaciones propias de la clase: Las operaciones de una clase se obtienen a través de la consulta *ownedOperation* definida para cada clase c. Por otra parte los datos de una operación y los

parámetros de la misma se obtienen de la relación entre las metaclases Operation y Parameter. La visibilidad y nombre de la operación o está dado por sus atributos *visibility y name* respectivamente; además cuenta con la consulta *ownedParameter* la cual constituye una instancia de la metaclase Parameter y su función es devolver el tipo y nombre de los parámetros de cada operación a través de los atributos type y name respectivamente.

∀ c ∈ ApemI_Controlador OR ApemI_Modelo OR ApemI_Modelo_Entidad_Media OR ApemI_Modelo_Entidad_Persistente OR ApemI_Vista ∀ o ∈ Operation (o | c. *ownedOperation* ⇒
//visibilidad 'function' nombre (parámetros)

o.visibility 'function' o.name '(' ∀ ownedParameter∈Parameter // '$' tipo parámetro nombre parámetro

'$'o.ownedparameter.type.name o.ownedparameter.name ') {' '}'

R5_STA- Regla de atributos y métodos estáticos: Todas las instancias de la metaclase Property u Operation cuyo atributo isStatic sea verdadero, serán métodos o atributos estáticos.

∀p∈Property (p.*isStatic* ⇒ 'static')
∀o∈Operation (o.*isStatic* ⇒ 'static')

R6_DEP- Regla de dependencia de clases: El atributo clientDependency de una instancia c de la metaclase Class devolverá las relaciones de dependencia que existe entre c y otras clases.

∀ c ∈ Class(*c.clientDependency.supplier* ⇒ // require_once ('nombre. extension');
'require_once' '(" *c. clientDependency.supplier.name* '.php' *");'*

R7_GEN- Regla de clase general: Si la clase c tiene una relación de generalización con otra clase el atributo superClass de c permite obtener la clase general de la relación.

∀ c ∈ Class(c. *superC* ⇒ // require_once (‘nombre. extension’);
‘require_once’ ‘(‘’ *c. superClass.name* ‘.php’ *‘’);’*
c.name ‘extends’ c.superClass.name)

Sin embargo la generalización múltiple no es soportada por PHP (al no ser que sea entre interfaces) por lo que solo se añaden las referencias a las clases generales.

R8_ASC- Regla de asociaciones de clases: Para cada instancia c de la metaclase Class el método *getAssociations()* devuelve las relaciones de asociación que tiene c con otras clases**.** La metaclase Association a través del atributo name y memberEnd accede al nombre y los extremos de la relación respectivamente además la instancia p de la metaclase Property brinda las propiedades de la relación.

∀ c ∈ Class ∀a∈Association ∀p∈Property (*a |c.getAssociations()* ⇒ p = a.memberEnd
// require_once (‘nombre. extension’);
‘require_once’ ‘(‘’ *p. type.name* ‘.php’ *‘’);’*

R9_DINF- Regla de definición de clases interfaces: Para cada instancia i de la metaclase Interface el atributo name devolverá el nombre de la misma.

∀ I ∈ Interface ⇒ ‘interface’ i.name ‘{’ ‘}’

R10_INFREA- Regla de realización de interfaces: La instancia c de la metaclase Class provee el acceso a las interfaces que implementa a través del método getImplementedInterfaces() y el nombre de la interfaz está dado por el atributo name.

∀ c ∈ Class ∀ I ∈ Interface (c. getImplementedInterfaces()⇒ ‘implements’ i.name)

R11_AUNI- Regla de asociación unidireccional: En caso de ser la asociación unidireccional solo se creará en la clase origen la referencia a la clase destino y no viceversa.

R12_STER- Regla de estereotipos: A las clases ApemI_Modelo_Entidad_Persistente dadas sus responsabilidades se le definirán los métodos Insertar, Modificar, Eliminar y Mostrar

R13_SG- Regla de set y get: Para cada atributo se podrá definir las funciones set y get correspondientes, por lo tanto luego de que el usuario decida si desea agregar algunas de estas funciones, las mismas serán implementadas de la siguiente manera:

```
//public function set_Nombreatributo ($Nombreatributo) {
                    $this->Nombreatributo= '$'Nombreatributo}
   'public function set_'p.name('$'p.name){ '$this->'p.name '=$'p.name}

// public function get_Nombreatributo () {
                    return $this->Nombreatributo}
   'public function get_'p.name() { 'return $this->'p.name}
```

R14_CD- Regla de constructor y destructor: Para cada instancia c de la superClase Class se podrá definir un constructor, en este caso se adicionará en el código un destructor, ambas funciones serán implementadas de la siguiente manera:

```
// public function NombreClase($Nombreatributo){
              $this->Nombreatributo= '$'Nombreatributo}
      'public function' c.name('$'p.name)
            { '$this->'p.name '=$'p.name}

// public function __destruct () { }
        'public function _destruct() { }'
```

A continuación se muestra a través de un ejemplo cómo las reglas de transformación antes expuestas generan las sentencias de código fuente PHP 5 a partir del diagrama de clases del diseño presentado en la Figura 11.

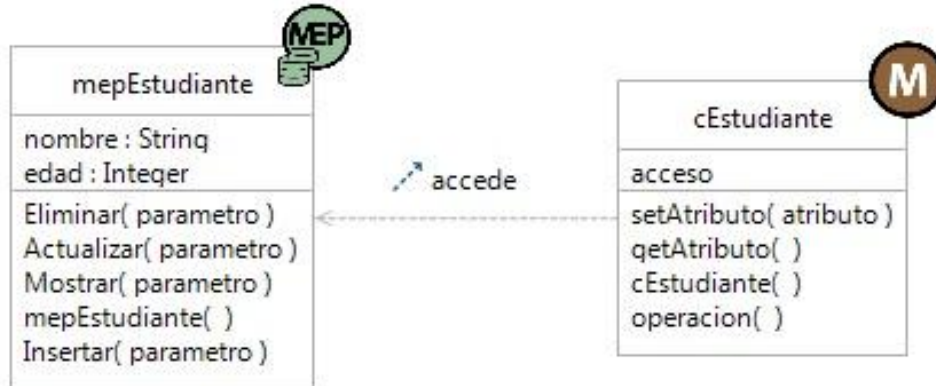

Figura 11: Diagrama de clases modelado con ApEM-L

El plugin genera los ficheros correspondiente a cada una de las clases representadas en el diagrama: mepEstudiante.php y cEstudiante.php. A continuación se presenta el código generado para la clase cEstudiante así como las reglas de transformación aplicadas para generar cada una de las sentencias de código contenidas en el fichero.

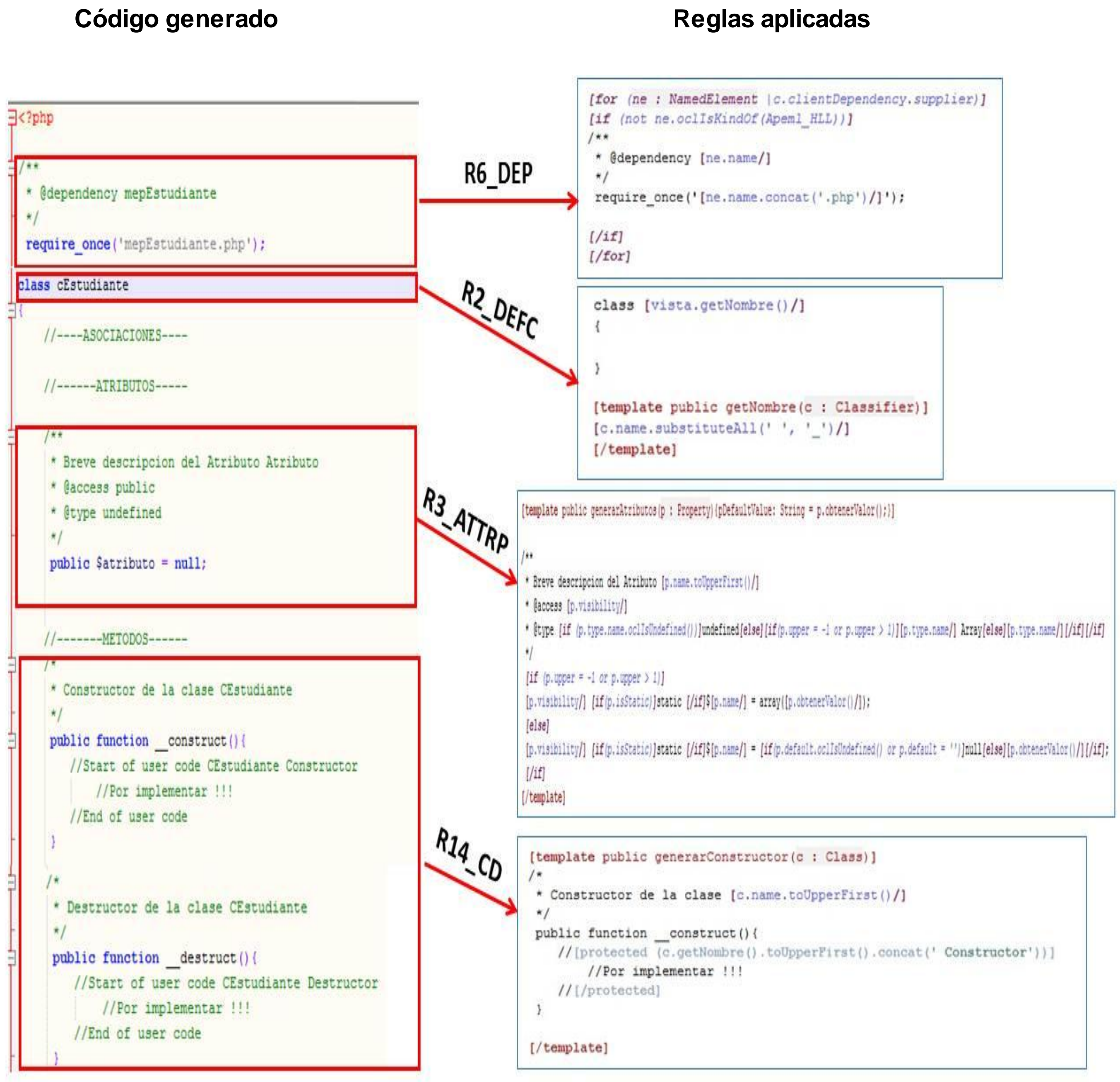

Figura 12: Aplicación de las reglas de transformación para la obtención del código fuente

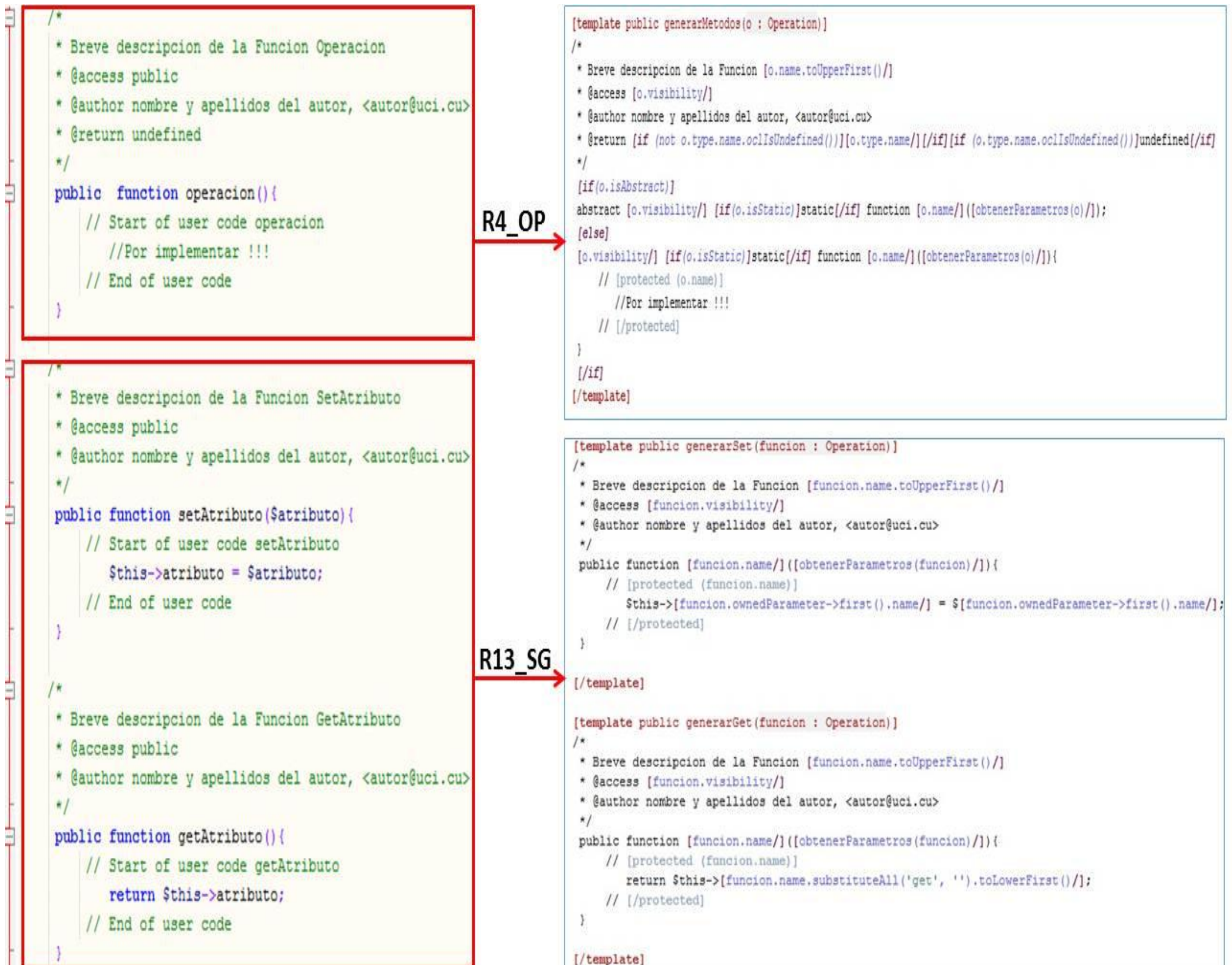

Figura 13: Aplicación de las reglas de transformación para la obtención del código fuente

3.3. Plantillas para la generación de código

Como ya se comentó anteriormente Acceleo utiliza el lenguaje MOFM2T para realizar las transformaciones de modelo a texto, además de permitir el uso de funciones externas de Java. Según (Pérez, 2012) este lenguaje estándar genera los artefactos de texto a través del uso de plantillas, las cuales están parametrizadas con elementos del modelo a transformar. Dentro de estas plantillas la principal forma de extraer la información del modelo es a través de consultas sobre las entidades del metamodelo. Esta información es convertida en fragmentos de texto usando expresiones del lenguaje para la manipulación de cadenas de texto.

Teniendo en cuenta lo comentado hasta ahora se ilustrará a través de la siguiente figura un ejemplo del funcionamiento práctico de las plantillas:

```
[template public temp_genInterface(i : Interface)]
[file (i.getFullPathFile().trim(), false)]

<?php

    /**
     * Breve descripción de la Interfaz [i.name.toUpperFirst()/]
     * @comment[for (cmt : Comment | i.ownedComment)] [cmt.body/][/for]
     * @author nombre y apellidos del autor, <autor@uci.cu>
     */
    interface [i.name/][if (not i.getGenerals()->isEmpty())] extends [for (aInterface : Interface | i.getGe
    {
        //-------METODOS------
        [for (fun : Operation |i.ownedOperation)]
        /*
         * Breve descripción de la Función [fun.name/]
         * @access [fun.visibility/]
         * @author nombre y apellidos del autor, <autor@uci.cu>
         * @return [if (not fun.type.name.oclIsUndefined())][fun.type.name/][/if][if (fun.type.name.oclIsUr
         */
        [if(fun.isStatic)]abstract[/if] [fun.visibility/] [if(fun.isStatic)]static[/if] function [fun.name/

        [/for]
    }
?>
[/file]
[/template]
```

Figura 14: Plantilla para la generación de clases interfaces

En la figura anterior se presenta un ejemplo de plantilla donde todo lo que se encuentra dentro de los caracteres especiales [ ] son elementos o funciones para extraer datos del parámetro de entrada de la plantilla. El resto del texto será imprimido en el fichero de salida tal y como está.

3.4. Pruebas de Software

Las pruebas del software son un elemento crítico para la garantía de la calidad del software y representa una revisión final de las especificaciones, del diseño y la codificación.
Existen diferentes clasificaciones para las pruebas de software, cada una encaminada a probar un aspecto específico del sistema. Entre ellas se encuentran las pruebas de **Caja Negra** y pruebas de **Caja Blanca**. En el primero de los casos, las pruebas son realizadas sobre la interfaz del software, teniendo en cuenta los datos de entrada y estudiando si la respuesta del sistema es o no la esperada. (Juristo *et al.*, 2006) El segundo caso se basan en el examen minucioso de los detalles procedimentales, donde se comprueban los caminos lógicos del subsistema, al generar casos de prueba que ejerciten las estructuras condicionales y los bucles.(Pressman, 2005)

Para comprobar y verificar la funcionalidad correcta del plugin, se realizarán **Pruebas de Desarrollador** puesto que es el primer nivel de prueba y estas son diseñadas e implementadas por el propio equipo de desarrollo. Como técnica de evaluación dinámica se utilizó las pruebas de **Caja Negra**, las cuales fijan su atención en la validación de las funciones. Durante la aplicación de esta técnica se analiza cada funcionalidad implementada para ejercitar los requisitos funcionales establecidos y verificar que los resultados esperados ocurran cuando se usen datos válidos y se desplieguen los mensajes apropiados de error. Se decidió además realizar pruebas de **Caja Blanca** para la detección de errores en el código fuente del plugin desarrollado, garantizando así la calidad de la solución propuesta.

Aplicación de las pruebas de Caja Negra

Para confeccionar los casos de prueba de Caja Negra se utilizó el método de partición de equivalencia, este método se basa en la evaluación de clases de equivalencia, las cuales representan un conjunto de estados válidos o no válidos para condiciones de entrada. (Pressman, 2002)

Este tipo de pruebas son aplicadas mediante los casos de prueba que constituyen un conjunto de entradas, condiciones de ejecución y resultados esperados desarrollados para cumplir un objetivo en particular o una función esperada.

A continuación se presentan los casos de pruebas "Validar fichero de entrada" y "Guardar Archivo" con el objetivo de comprobar que el plugin funcione de forma correcta, donde:

V: indica válido, **I:** indica inválido, **NA**: que no es necesario proporcionar un valor del dato en este caso, ya que es irrelevante.

Caso de prueba: Validar fichero de entrada

Tabla 9: Descripción de las variables

No	Nombre del campo	Clasificación	Valor nulo	Descripción
1	Fichero del diagrama	Campo de Texto	no	Es el campo donde se va a mostrar el nombre del fichero del diagrama que se está cargando en el workspace.

Tabla 10: Matriz de Datos

Escenario	Descripción	Fichero del diagrama	Respuesta del sistema	Flujo central
1.1 Se carga un diagrama de clases con la extensión correcta	Se carga en el workspace un diagrama de clases con la extensión correcta.	V(diagrama_diseño.apeml)	La herramienta habilita las opciones del plugin.	1- El especialista carga el diagrama de clases en el workspace. 2-Da clic derecho encima del diagrama 3- Selecciona la opción Generar
1.2 Se carga un diagrama de clases con la extensión incorrecta	Se carga en el workspace un diagrama de clases que no tenga la extensión .apeml	Se debe buscar un fichero con extensión .uml u otra cualquiera y cargarlo en el workspace.	La herramienta no habilita las opciones del plugin.	
1.2 Fichero no válido	El fichero del diagrama de clases cargado en el workspace tiene	Se debe buscar un fichero de diagrama de clases de ApEM-L y se debe borrar información en su interior o	La herramienta muestra el mensaje: "Error en el formato del	

Escenario	Descripción	Fichero del diagrama	Respuesta del sistema	Flujo central
	problemas en el formato	agregar valores que no correspondan a la estructura de un XML.	fichero" y no procede con la generación	Código/ PHP 5

Caso de prueba: Guardar Archivo

Tabla 11: Descripción de las Variables

No	Nombre del campo	Clasificación	Valor nulo	Descripción
1	Dirección	campo de texto	no	Recoge la dirección destino de los archivos generados

Tabla 12: Matriz de datos

Escenario	Descripción	Dirección	Respuesta del sistema	Flujo central
1.1 Guardar archivo en otra ubicación con campos vacíos	El usuario deberá dejar campos vacíos	V()	La herramienta se queda esperando una dirección física o un nombre para crear la carpeta en una dirección por defecto	1- El especialista selecciona el diagrama de clase y da clic en la opción generar código del menú. 2- Luego selecciona el lenguaje PHP5 3- Da clic en el botón: Otra ubicación 4- Selecciona la dirección física
1.2 Guardar archivo en otra ubicación con datos válidos	El usuario deberá seleccionar una ubicación física correcta	V(home/dir)	La herramienta procede a guardar los archivos en la dirección seleccionada y si no existe la carpeta la crea	
1.3 Guardar archivo en otra ubicación con datos inválidos	El usuario deberá llenar el campo con una ubicación física que no existe	I(no existe/dir)	La herramienta muestra un mensaje de error al crear el fichero	

Después de realizar las pruebas de Caja Negra mediante los casos de prueba antes descritos, se comprobó el correcto funcionamiento del plugin y la correcta validación de los campos. Cada dificultad detectada en el desarrollo del plugin y resueltas a raíz del trabajo continuo de los desarrolladores, fueron

recogidas en la planilla de No Conformidades (NC). En la siguiente tabla se muestra un resumen de las dificultades encontradas:

PD: Pendiente **RA**: Resuelta

Tabla 13: Dificultades encontradas con las pruebas de caja negra

Fecha	Versión	Caso de Prueba	Cantidad de NC	Cantidad de NC PD	Cantidad de NC RA
20-5-2014	1.0	Validar fichero de entrada	1	1	-
20-5-2014	1.1	Guardar Archivo	1	1	-
27-5-2014	1.1	Validar fichero de entrada	1	-	1
27-5-2014	1.1	Guardar Archivo	1	-	1

Para más información referente a las dificultades encontradas en el proceso de desarrollo del plugin ver **Anexo 2**

Aplicación de las pruebas de Caja Blanca

Dentro de las pruebas de caja blanca, se ejecutaron casos de pruebas tanto de forma manual, como automática, para garantizar la veracidad de los resultados obtenidos.

Para realizar el proceso de prueba de forma manual, se decidió por el equipo de desarrollo utilizar la técnica de camino básico, y de forma automática con el uso de la herramienta EclEmma, la cual brinda un conjunto de librerías que se integran fácilmente al IDE de desarrollo seleccionado: Eclipse.

La selección de la técnica del camino básico está basada en las ventajas que ofrece con respecto a otras técnicas, dado que el número mínimo requerido de pruebas se conoce por adelantado y por tanto el proceso de prueba se puede planear y supervisar en mayor detalle con respecto a las restantes técnicas. Por otra parte, esta técnica asegura que los casos de prueba diseñados permitan que todas las sentencias

del programa sean ejecutadas al menos una vez y que las condiciones sean probadas tanto para su valor verdadero como falso.
Para la aplicación de la técnica de camino básico se realizaron los siguientes pasos:

- ✓ Representar el programa en un grafo de flujo: se utiliza para representar el flujo de control lógico de un programa.
- ✓ Calcular la complejidad ciclomática: el valor calculado define el número de rutas independientes en el conjunto básico de un programa, y proporciona un límite superior para el número de pruebas que deben aplicarse, lo cual asegura que todas las instrucciones se hayan ejecutado por lo menos una vez.
- ✓ Determinar el conjunto básico de rutas independientes: es cualquier ruta del programa que ingresa por lo menos un nuevo conjunto de instrucciones de procesamiento o una nueva condición.
- ✓ Derivar los casos de prueba que fuerzan la ejecución de cada camino.

A continuación se muestra el caso de prueba guiado por los pasos anteriores al método **run,** el cual se localiza en la clase **ApemlParser**.
Este método es el encargado de correr el Parser para la validación del código regenerado a partir del metamodelo. Se selecciona este método teniendo en cuenta la importancia que representa para el resultado y la implementación del plugin.
A continuación, en la Figura 15 se muestra el código fuente referente al método descrito anteriormente y en la Figura 16 y 17 se aplica la técnica de Camino Básico al mismo.

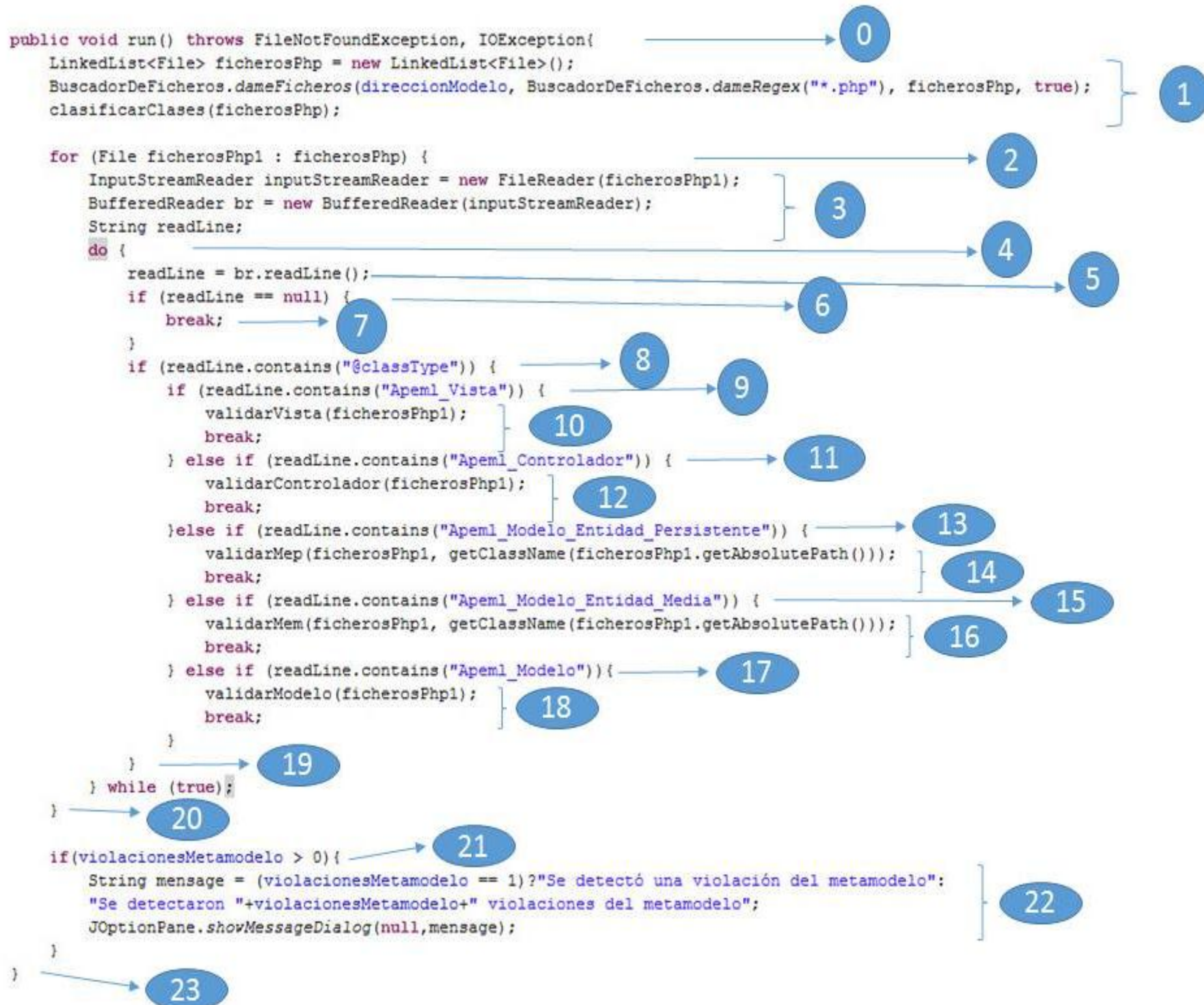

Figura 15: Código fuente del método run

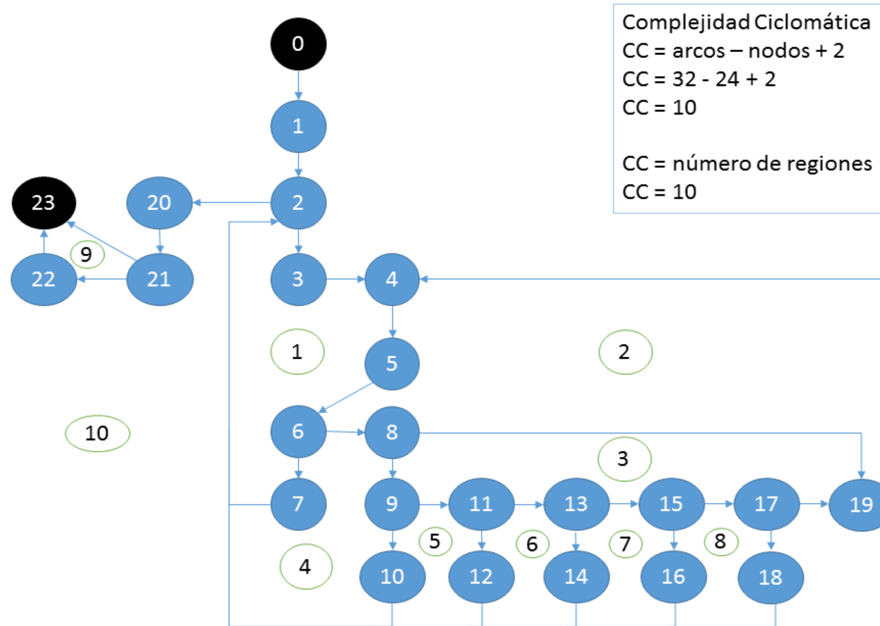

Figura 16: Gráfico de flujo y complejidad ciclomática correspondiente al método run

Conjunto de rutas independientes

Ruta 1: 0-1-2-20-21-22-23
Ruta 2: 0-1-2-20-21-23
Ruta 3: 0-1-2-3-4-5-6-7-2-20-21-22-23
Ruta 4: 0-1-2-3-4-5-6-8-9-10-2-20-21-23
Ruta 5: 0-1-2-3-4-5-6-8-19-4-5-6-7-2-20-21-23
Ruta 6: 0-1-2-3-4-5-6-8-9-11-12-2-20-21-23
Ruta 7: 0-1-2-3-4-5-6-8-9-11-13-14-2-20-21-23
Ruta 8: 0-1-2-3-4-5-6-8-9-11-13-15-16-2-20-21-23
Ruta 9: 0-1-2-3-4-5-6-8-9-11-13-15-17-18-2-20-21-23
Ruta 10: 0-1-2-3-4-5-6-8-9-11-13-15-17-19-4-5-6-7-2-20-21-23

Figura 17: Rutas linealmente independientes correspondientes al método run

Casos de prueba diseñados para cubrir los caminos independientes presentados:

Caso de prueba del camino 1:

Entrada: -

Resultado Esperado: No existen ficheros y por lo tanto no ocurre ninguna validación.

Objetivo: Una vez verificado que no existen ficheros a validar, se verifica si existen violaciones del metamodelo, mostrándolas.

Caso de prueba del camino 2:

Entrada: -

Resultado Esperado: No se realizan las validaciones.

Objetivo: Verificar si existen violaciones del metamodelo, mostrándolas.

Caso de prueba del camino 3:

Entrada: -

Resultado Esperado: Se encuentra el final de un archivo y se procede a mostrar las violaciones existentes.

Objetivo: Terminar el ciclo una vez analizado el último archivo existente.

Caso de prueba del camino 4:

Entrada: -

Resultado Esperado: Se encuentra una clase de tipo Apeml_Vista y se procede a verificar posibles violaciones del metamodelo de ApEM-L 2.0.

Objetivo: Analizar violaciones del metamodelo en las clases Apeml_Vista y mostrarlas.

Caso de prueba del camino 5:

Entrada: -

Resultado Esperado: No se analiza la línea actual del fichero que se está analizando.

Objetivo: Si la línea que se analiza no posee la marca @ClassType, se procede a analizar la siguiente línea del fichero.

Caso de prueba del camino 6:

Entrada: -

Resultado Esperado: Se encuentra una clase de tipo Apeml_Controlador y se procede a verificar posibles violaciones del metamodelo de ApEM-L 2.0.
Objetivo: Analizar violaciones del metamodelo en las clases Apeml_Controlador y mostrarlas.

Caso de prueba del camino 7:
Entrada: -
Resultado Esperado: Se encuentra una clase de tipo Apeml_Modelo_Entidad_Persistente y se procede a verificar posibles violaciones del metamodelo de ApEM-L 2.0.
Objetivo: Analizar violaciones del metamodelo en las clases Apeml_Modelo_Entidad_Persistente y mostrarlas.

Caso de prueba del camino 8:
Entrada: -
Resultado Esperado: Se encuentra una clase de tipo Apeml_Modelo_Entidad_Media y se procede a verificar posibles violaciones del metamodelo de ApEM-L 2.0.
Objetivo: Analizar violaciones del metamodelo en las clases Apeml_Modelo_Entidad_Media y mostrarlas.

Caso de prueba del camino 9:
Entrada: -
Resultado Esperado: Se encuentra una clase de tipo Apeml_Modelo y se procede a verificar posibles violaciones del metamodelo de ApEM-L 2.0.
Objetivo: Analizar violaciones del metamodelo en las clases Apeml_Modelo y mostrarlas.

Caso de prueba del camino 10:
Entrada: -
Resultado Esperado: Se itera por cada uno de los ficheros seleccionados ejecutando la validación del mismo en dependencia del tipo de clase que represente.
Objetivo: Analizar violaciones del metamodelo de ApEM-L 2.0, en todos los ficheros y mostrarlas.

Una vez realizado el procedimiento de forma manual, se procede a utilizar la herramienta EclEmma al método descrito anteriormente para comprobar los resultados. EclEmma es una herramienta que permite

realizar pruebas de Cobertura de Sentencias, las cuales describen casos de prueba suficientes para que cada sentencia en el programa se ejecute, al menos, una vez.

En las pruebas automáticas se utilizaron los mismos casos de pruebas que de la forma manual, para comprobar los resultados. En cada caso de la forma automática, quedará marcada la cobertura del código de la siguiente manera:

- En color verde, las líneas de código que han sido ejecutadas.
- En color amarillo, las líneas de código que han sido ejecutadas pero no totalmente.
- En color rojo, las líneas de código que no han sido ejecutadas.

En las Figuras 18 y 19 se muestran los resultados de los casos de pruebas 1 y 2 aplicados de forma automática

```
public void run() throws FileNotFoundException, IOException{

    LinkedList<File> ficherosPhp = new LinkedList<File>();
    BuscadorDeFicheros.dameFicheros(direccionModelo,
    BuscadorDeFicheros.dameRegex("*.php"), ficherosPhp, true);
    clasificarClases(ficherosPhp);

    for (File ficherosPhp1 : ficherosPhp) {
        InputStreamReader inputStreamReader = new FileReader(ficherosPhp1);
        BufferedReader br = new BufferedReader(inputStreamReader);
        String readLine;
        do {
            readLine = br.readLine();
            if (readLine == null) {
                break;
            }
            if (readLine.contains("@classType")) {
                if (readLine.contains("Apeml_Vista")) {
                    validarVista(ficherosPhp1);
                    break;
                } else if (readLine.contains("Apeml_Controlador")) {
                    validarControlador(ficherosPhp1);
                    break;
                }else if (readLine.contains("Apeml_Modelo_Entidad_Persistente")) {
                    validarMep(ficherosPhp1, getClassName(ficherosPhp1.getAbsolutePath()));
                    break;
                } else if (readLine.contains("Apeml_Modelo_Entidad_Media")) {
                    validarMem(ficherosPhp1, getClassName(ficherosPhp1.getAbsolutePath()));
                    break;
                } else if (readLine.contains("Apeml_Modelo")){
                   validarModelo(ficherosPhp1);
                    break;
                }
            }
        } while (true);
    }

    if(violacionesMetamodelo > 0){
        String mensage = (violacionesMetamodelo == 1)?
        "Se detectó una violación del metamodelo":
        "Se detectaron "+violacionesMetamodelo+" violaciones del metamodelo";
        JOptionPane.showMessageDialog(null,mensage);
    }

}
```

Figura 18: Resultado del Caso de Prueba 1

```
public void run() throws FileNotFoundException, IOException{

    LinkedList<File> ficherosPhp = new LinkedList<File>();
    BuscadorDeFicheros.dameFicheros(direccionModelo,
    BuscadorDeFicheros.dameRegex("*.php"), ficherosPhp, true);
    clasificarClases(ficherosPhp);

    for (File ficherosPhp1 : ficherosPhp) {
        InputStreamReader inputStreamReader = new FileReader(ficherosPhp1);
        BufferedReader br = new BufferedReader(inputStreamReader);
        String readLine;
        do {
            readLine = br.readLine();
            if (readLine == null) {
                break;
            }
            if (readLine.contains("@classType")) {
                if (readLine.contains("Apeml_Vista")) {
                    validarVista(ficherosPhp1);
                    break;
                } else if (readLine.contains("Apeml_Controlador")) {
                    validarControlador(ficherosPhp1);
                    break;
                }else if (readLine.contains("Apeml_Modelo_Entidad_Persistente")) {
                    validarMep(ficherosPhp1, getClassName(ficherosPhp1.getAbsolutePath()));
                    break;
                } else if (readLine.contains("Apeml_Modelo_Entidad_Media")) {
                    validarMem(ficherosPhp1, getClassName(ficherosPhp1.getAbsolutePath()));
                    break;
                } else if (readLine.contains("Apeml_Modelo")){
                   validarModelo(ficherosPhp1);
                    break;
                }
            }
        } while (true);
    }

    if(violacionesMetamodelo > 0){
        String mensaje = (violacionesMetamodelo == 1)?
        "Se detectó una violación del metamodelo":
        "Se detectaron "+violacionesMetamodelo+" violaciones del metamodelo";
        JOptionPane.showMessageDialog(null,mensaje);
    }

}
```

Figura 19: Resultado del Caso de Prueba 2

Conclusiones parciales

- Las reglas de transformación definidas permitieron describir cómo serán traducidos por el plugin cada uno de los componentes del diagrama de clases del diseño para obtener en estructuras de código PHP 5.
- La técnica de partición de equivalencia permitió elaborar los casos de pruebas necesarios para comprobar el correcto funcionamiento de los requisitos funcionales del plugin. Este tipo de pruebas arrojó como resultados dos no conformidades: una en la validación del fichero de entrada y otra al guardar los archivos de código, las cuales fueron corregidas garantizando así la calidad de la solución propuesta.

- La técnica del camino básico posibilitó probar las sentencias de código del plugin desarrollado. Estas pruebas fueron automatizadas utilizando la herramienta EclEmma la cual permitió comprobar los resultados obtenidos.

CAPÍTULO 4. APLICACIÓN DEL PLUGIN A UN CASO DE ESTUDIO

Según (Guardo, 2009, p 10) «*La hipótesis científica es la esencia y el corazón de la investigación [..],* lo que indica que todo el proceso investigativo está vinculado a la comprobación o refutación de la misma. Su demostración puede realizarse de diferentes formas teórica y/o práctica, en dependencia del alcance de la investigación y puede estar dirigida a constatar su pertinencia a partir de fundamentos teóricos, por criterio de especialistas o expertos, la comprobación de la viabilidad de aplicación o validación a través del experimento».

Por lo antes mencionado se hace inminente probar el generador automático de código desarrollado aplicando el mismo a un caso de estudio que permita comprobar si los estereotipos restrictivos definidos por ApEM-L 2.0 se encuentran correctamente reflejados en el código generado mejorando así la consistencia y completitud del mismo; además de evidenciarse una mejora en la calidad de la documentación del código con respecto a la documentación provista por las herramientas CASE usadas en la universidad hasta el momento para el modelado de aplicaciones educativas.

4.1. Caso de estudio: Proyecto "A Jugar"

Para realizar la validación de la hipótesis se utilizará un diagrama de clases modelado con ApEM-L para el proyecto productivo "A Jugar" presentado por el propio creador del lenguaje en (Ciudad, 2007). "A Jugar" formaba parte de los proyectos de la antigua Facultad 9 y su objetivo fundamental era el desarrollo de un producto educativo interactivo para desarrollar el aprendizaje de los niños en edades preescolares.

Según (Ciudad, 2007) es muy importante y de una gran relevancia el desarrollo de un exhaustivo modelo conceptual de la aplicación que permita conocer y relacionar los conceptos que sustentan el funcionamiento de la aplicación educativa. En la Figura 20 se presenta el diagrama de clases correspondiente al Caso de Uso "Presentar Software", modelado en la herramienta Visual Paradigm, en el cual se utilizan todos los elementos que incorpora ApEM-L al lenguaje base UML en aras de representar todos los conceptos lógicos y estructurales necesarios en este tipo de aplicaciones.

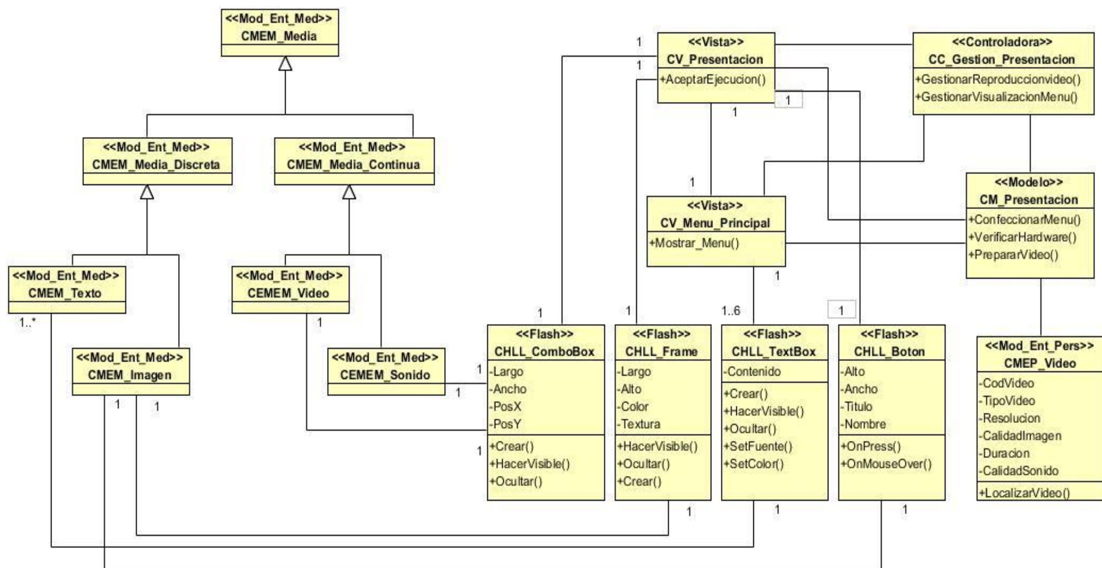

Figura 20: Diagrama de Clases del Caso de Uso “Presentar Software” [Tomado de (Ciudad, 2007)]

Sin embargo, la herramientaa CASE utilizada para el modelado no comprende los estereotipos aplicados a las clases debido a que no valida las restricciones referentes a dichos estereotipos. Por lo antes descrito, es posible cometer errores en el modelado y arrastrar los mismos en la generación de código fuente a partir de los diagramas de clases realizados en Visual Paradigm, Rational Rose, ArgoUML o cualquier otra herramienta CASE ya que estas no trabajan con el metamodelo de ApEM-L y por lo tanto es propenso a permitir violaciones del metamodelo.

A continuación se demostrará a partir el código generado para la clase Modelo Entidad – Persistente la implicación que tiene el uso de una herramienta de modelado que trabaje o no con el metamodelo de ApEM-L:

Código generado por la herramienta Visual Paradigm:

```php
<?php
require_once(realpath(dirname(__FILE__)) . '/CM_Presentacion.php');

/**
 * @access public
 */
class CMEP_Video {
    private $_codVideo;
    private $_tipoVideo;
    private $_resolucion;
    private $_calidadImagen;
    private $_duracion;
    private $_calidadSonido;
    /**
     * @AssociationType CM_Presentacion
     */
    public $_unnamed_CM_Presentacion_;

    /**
     * @access public
     */
    public function LocalizarVideo() {
        // Not yet implemented
    }
}
?>
```

Figura 21: Código generado por Visual Paradigm para la clase CMEP_Video

Código generado por la solución propuesta:

A continuación se muestra un fragmento de código generado por el plugin que evidencia la representación de los estereotipos; para ver el código completo de la clase CMEP_Video remitirse al **Anexo 3**

```
 * @author nombre y apellidos del autor, <autor@uci.cu>
 * @return undefined
 */
 public  function localizarVideo(){
     // Start of user code localizarVideo
        //Por implementar !!!
     // End of user code
 }


//Start of user code Insertar

/**
 * Función encargada de insertar los datos pasados por parametros
 * en la base de datos
 * @author PhpGenerator
 */
public function Insertar($valor1,$valor2){
    $consulta = "INSERT INTO nombre_tabla (columna1,columna2) VALUES ('$valor1','$valor2')";
    return $this->conexion->query($consulta);
}

///End of user code

//Start of user code Eliminar

/**
 * Función encargada de eliminar los datos pasados por parametros
 * en la base de datos
 * @author PhpGenerator
 */
public function Eliminar($valor1,$valor2){
    $consulta = "DELETE From nombre_tabla WHERE columna1 ='$valor1' and columna2 = '$valor2'";
    return $this->conexion->query($consulta);
}

//End of user code
```

Figura 22: Fragmento de código generado por el plugin desarrollado para la clase CMEP_Video

A continuación se muestra en la Tabla 14 una comparación del código generado por Visual Paradigm y el plugin desarrollado en cuanto a: la comprensión de los estereotipos del lenguaje ApEM-L 2.0, la validación de las relaciones entre cada uno de los tipos de clases del diagrama y la validación del código regenerado en función del metamodelo.

Tabla 14: Comparación de los resultados de la aplicación de ambas herramientas

Conceptos	Visual Paradigm	Plugin desarrollado
Comprensión de estereotipos del lenguaje	No comprende los estereotipos por lo que no se garantiza la completitud del código.	Realiza una adecuada codificación de las clases estereotipadas.
Validación de las relaciones entre clases	No realiza las validaciones entre clases estereotipadas implicando errores en el diagrama y código generado.	Valida las relaciones entre clases estereotipadas.
Validación del código regenerado en función del metamodelo	No realiza la validación del código según el metamodelo.	Realiza un análisis sintáctico del código, documentando las violaciones encontradas.

A partir del análisis anterior se evidencia la mejoría de los conceptos tratados entre el plugin desarrollado y Visual Paradigm por lo que sería prudente realizar una comparación entre las variables de la investigación en cuanto al valor tomado por sus indicadores: (ver Tabla 15)

Tabla 15: Comparación de los valores tomados por las variables de la investigación al utilizar ambas herramientas

Variables	Dimensiones	Indicadores	Visual Paradigm	Plugin desarrollado
Consistencia	Grado de correspondencia entre el código y el diagrama de clases del diseño	Cantidad de clases y relaciones del diagrama que han sido codificadas	Alta	Alta
		Representación de los estereotipos	Baja	Alta
Completitud	Auto-documentación	Calidad de la documentación del código	Media	Alta

Variables	Dimensiones	Indicadores	Visual Paradigm	Plugin desarrollado
	Cantidad de código generado en función del enfoque	Transformación a código de los componentes del diagrama	Baja	Alta

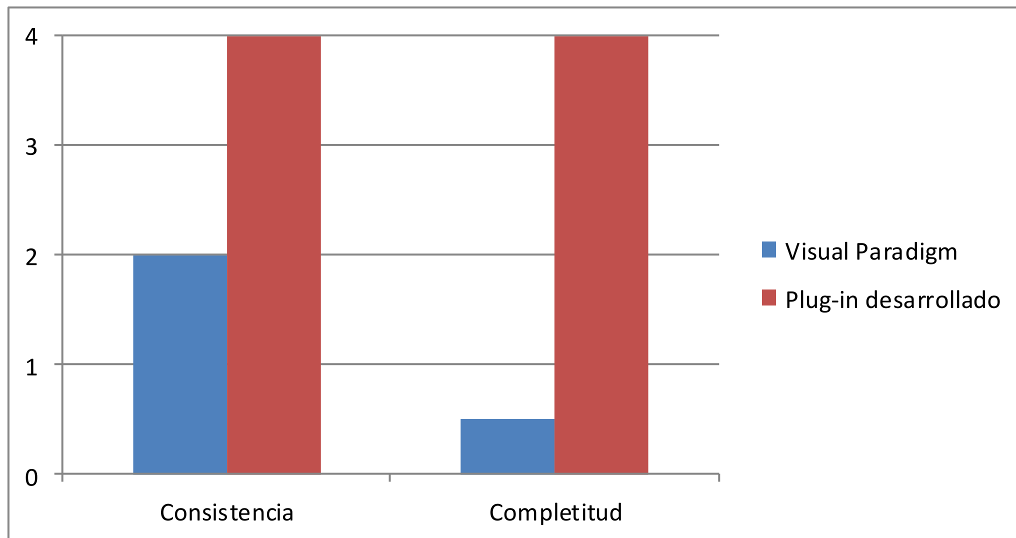

Figura 23: Comparación de las variables de la investigación

Conclusiones parciales

Del análisis de los valores obtenidos para cada una de las variables de la investigación se pudo determinar que:

- El experimento realizado logró demostrar el incremento de los indicadores: grado de correspondencia entre el código y el diagrama de clases del diseño, auto-documentación y cantidad de código generado en función del enfoque al utilizar el plugin desarrollado en lugar de Visual Paradigm.
- Se comprobó la hipótesis presentada a inicios de la investigación pues se elevaron los grados de consistencia y completitud del código generado.

CONCLUSIONES FINALES

- En la actualidad existe una gran variedad de herramientas para el modelado que brindan soporte a la generación automática de código fuente a través de la definición de plantillas, las cuales implementan un conjunto de reglas de transformación basadas en el metamodelo del lenguaje de modelado al que le de soporte la herramienta.
- A partir de la caracterización del proceso de generación de código en los proyectos productivos que utilizaron ApEM-L para el modelado de aplicaciones educativas, se evidenció que la inexistencia de una herramienta CASE que brinde soporte al metamodelo del lenguaje implica una afectación en la consistencia y completitud del código generado.
- El proceso de desarrollo del plugin estuvo guiado por la metodología OpenUP, generando los artefactos necesarios para la comprensión y mantenimiento de la solución. Además se implementaron un conjunto de reglas de transformación en las plantillas, que serán las que guíen la transformación de modelo a texto.
- El experimento a través del caso de estudio permitió apreciar la transformación favorable que sufren la consistencia y completitud del código generado automáticamente, al utilizar una herramienta que comprenda los estereotipos definidos por el lenguaje de modelado.

RECOMENDACIONES

Al concluir la presente investigación se hace necesario el siguiente conjunto de recomendaciones para guiar el desarrollo de investigaciones futuras en el mismo ámbito y mejorar los resultados científicos obtenidos:

1. Valorar la incorporación de otros lenguajes utilizados por la comunidad cubana en el desarrollo de aplicaciones educativas, tales como ActionScript, JavaScript, entre otros.
2. Valorar extender la herramienta para incorporar la generación de código a partir de diagramas de comportamiento que contribuyan a elevar la completitud del código generado.
3. Sugerir a la dirección del centro FORTES y a la Vicerrectoría de Producción de la UCI, introducir la nueva herramienta en los proyectos productivos de software educativo que utilicen ApEM-L, en una fase de prueba que permitan obtener elementos para la mejora de la solución presentada.

REFERENCIAS BIBLIOGRÁFICAS

BALDUINO, R. Introduction to OpenUP (Open Unified Process). 2007, nº Disponible en: http://www.eclipse.org/epf/general/OpenUP.pdf.

BELL, R. Code Generation from Object Models. *Embedded Systems Programming*, 1998, vol. 11, nº 3, p. 74-88.

BERNER, S.; GLINZ, M.*, et al.* A Classification of Stereotypes for Object-Oriented Modeling Languages. 1999, nº

CAMPOALEGRE, L. *Herramienta de Generación de Código Mediante Sistema Experto.* UNIVERSIDAD DE LAS CIENCIAS INFORMÁTICAS, 2008.

CIUDAD, F. A. *ApEM – L como una nueva solución a la modelación de aplicaciones educativas multimedia en la UCI.* Tesis de Maestría, Universidad de las Ciencias Informáticas, 2007.

---. Utilización del Patrón Modelo – Vista – Controlador (MVC) en el diseño de software educativos. En *II TALLER DE SOFTWARE EDUCATIVO E HIPERMEDIA. Universidad de las Ciencias Informáticas. 2006*.

CIUDAD, F. A. y HERRERA, Y. *ApEM-L 1.5: Una nueva versión del Lenguaje de modelado Orientado a Objetos para aplicaciones educativas y multimedia.* La Habana: publicado el: julio-diciembre de 2008, última actualización: julio-diciembre. vol. 2, 52-63 p.

---. DoMet COMO PROPUESTA PARA LA MODELACIÓN DE ENTORNOS ORGANIZACIONALES COMPLEJOS Y DIFUSOS. *UCIENCIA* 2006, nº

---. FUNDAMENTOS ACTUALES DE LA INGENIERÍA DEL SOFTWARE PARA LAS APLICACIONES EDUCATIVAS. En *UCIENCIA*, *UCI,* 2012.

DOMINGO, I.; RIUS, G.*, et al.* Una revisión sobre el estado del arte en herramientas de modelado basado en UML. En *6th International Conference on Industrial Engineering and Industrial Management.*, *XVI Congreso de Ingeniería de Organización,* July 18-20.2012.

ECHEMENDIA, R. M. y HERVIS, Y. G. *"Diseño e implementación de un plugin de Eclipse que agilice el desarrollo de aplicaciones Web que utilicen la arquitectura única Dalas".* Universidad de las Ciencias Informáticas, 2010.

ENGELS, G. y SAUER, S. *UML-based Behavior. Specification of Interactive Multimedia Applications.* 2000,

FIGUEROA, R. G.; SOLÍS, C. J., *et al. METODOLOGÍAS TRADICIONALES VS. METODOLOGÍAS ÁGILES.* Universidad Técnica Particular de Loja.Escuela de Ciencias en Computación, 2011,

FUENTES, L. y VALLECILLO, A. Una Introducción a los Perfiles UML. 2003, nº

FUNDIBEQ. *Diagrama de Flujo* Disponible en: http://www.fundibeq.org/opencms/export/sites/default/PWF/downloads/gallery/methodology/tools/diagrama_de_flujo.pdf. .

GARCÍA, F. J. *SOFTWARE EDUCATIVO: EVOLUCIÓN Y TENDENCIAS* España: Departamento de Informática y Automática. Universidad de Salamanca, 2002, vol. 14, 19-29 p. ISBN 0214-3402.

GÉNOVA, G. *Basic modeling concepts in model-driven software engineering.* Universidade do Minho Braga: 2013,

GIMSON, L. *Metodologías ágiles y desarrollo basado en conocimiento.* UNIVERSIDAD NACIONAL DE LA PLATA, 2012.

GOMEZ, J. *Softonic* (Eclipse SDK). Disponible en: http://eclipse-sdk.softonic.com/.

GUARDO, M. E. *LOS COMPONENTES DEL DISEÑO TEÓRICO DE LA INVESTIGACIÓN CIENTÍFICA. UNA REFLEXIÓN PRAXIOLÓGICA.* Matanzas: I.S.C.F. Manuel Fajardo. Facultad Cultura Física, 2009, vol. XIV,

GUILLÉN, P. R. *CAPITULO I EL DESARROLLO DEL SOFTWARE.* 2001, Disponible en: www.itlalaguna.edu.mx/Academico/Carreras/sistemas/Analisis%20y%20dise%C3%B1o%20orientado%20a%20objetos/cap1.pdf.

HERRERA, Y. *Tesis en opción al título de Máster en Informática Aplicada.* Universidad de las Ciencias Informáticas UCI, 2014.

HERRINGTON, J. *Code Generation in Action.* Manning Publications Co., 2003. ISBN 1-930110-97-9.

JACOBSON, I.; BOOCH, G., *et al. El Proceso Unificado de Desarrollo de Software. La guía completa del proceso unificado escrita por sus creadores.* Madrid: Pearson Educacion, 2000. ISBN 84-7829-036-2.

JURISTO, N.; MORENO, A. M., *et al. TÉCNICAS DE EVALUACIÓN DE SOFTWARE.* 2006,

KLEPPE, A. G.; WARMER, J., *et al. MDA Explained: The Model Driven Architecture: Practice and Promise.* Boston,USA: Addison-Wesley, 2003. ISBN 0-321-19442-X.

KOCH, N.; KNAPP, A., *et al. An Approach Based on Standards.* En *UML-BASED WEB ENGINEERING.*

LARMAN, C. *UML y Patrones. Introducción al análisis y diseño orientado a objetos.* 1ra ed. México: Prentice Hall, 1999. ISBN 970-17-0261-1.

MOLINA, P. J. *Especificación de Interfaz de usuario: de los requisitos a la generación automática.* Universidad Politécnica de Valencia, 2003.

MORENO, A. M. *Verificación dinámica de modelos UML.* Universidad de Málaga Escuela Técnica Superior de Ingeniería Informática, 2013.

MUÑETON, A.; ZAPATA, C. M., *et al.* Reglas para la generación automática de código definidas sobre metamodelos simplificados de los diagramas de clases, secuencias y máquina de estados de UML 2.0. *Dyna*, noviembre 2007, vol. 74, nº 153, p. 267-283.

MURILLO, F. *Herramientas Case.COLECCION CULTURA INFORMATICA*
Instituto Nacional de Estadística e Informática, 1999.

PÉREZ, A. *Generación de código ADA para aplicaciones embebidas y de tiempo real.* Universidad de Cantabria, 2012.

PONS, C.; GIANDINI, R., *et al. Desarrollo de software dirigido por modelos. Conceptos teóricos y su aplicación práctica.* Universidad Nacional de La Plata: Mc Graw Hill, 2010. ISBN 978-950-34-0630-4.

PRESSMAN, R. S. *Ingeniería del Software. Un enfoque práctico.* 6ta ed.: Mc Graw Hill, 2005. ISBN 970-10-5473-3.

---. *Ingeniería del Software. Un enfoque práctico.* 5ta ed. Madrid: Mc Graw Hill, 2002. ISBN 84-481-3214-9.

Programación en Java Disponible en: http://es.wikibooks.org/wiki/Programación_en_Java/Características_del_lenguaje.

REYNA, R. *Scribd .Capitulo I HERRAMIENTAS CASE* [Consultado el: 2014-02-18 Disponible en: http://es.scribd.com/doc/3062020/Capitulo-I-HERRAMIENTAS-CASE.

RIBA, N. A. *Un enfoque MDA para el desarrollo de aplicaciones basadas en un modelo de componentes orientados a servicios.* Universidad Autónoma Metropolitana, 2007.

RINCÓN, M.; AGUILAR, J., *et al. Generación automática de código a partir de máquinas de estado finito.* México: Instituto Politécnico Nacional
2011, vol. 14, Disponible en: http://www.redalyc.org/articulo.oa?id=61520767007. ISBN 1405-5546.

RUMBAUGH, J.; JACOBSON, I., *et al. El Lenguaje Unificado de Modelado. MANUAL DE REFERENCIA.* 2da ed. Madrid: PEARSON EDUCACIÓN, S.A, 2007. ISBN 978-84-7829-087-1.

---. *El Lenguaje unificado de modelado. Manual de referencia.* 1ra ed. Madrid: Pearson Educación, 1999. ISBN 84-7829-037-0.

SAMPIERI, R. H.; FERNANDEZ, C., *et al. Metodología de la Investigación Científica.* México: McGraw-Hill, 2006.

SELIC, B. *The pragmatics of model-driven development.* 2003, vol. 20, 19-25 p.

SOMMERVILLE, I. *Ingeniería del Software.* 7ma ed. Madrid: Pearson Educación S.A, 2005. ISBN 84-7829-074-5.

---. *Software Engineering.* 8va ed.: Pearson Education, 2006. ISBN 13:978-0-321-31379-9.

TEDESCHI, N. ¿Qué es un Patrón de Diseño? *Microsoft Developer Network* nº [Consultado el: 08/04/2014]. Disponible en: http://msdn.microsoft.com/es-es/library/bb972240.aspx.

TRUJILLO, J. L. y ESPINOZA, A. D. Conceptos fundamentales de Ingeniería dirigida por Modelos y Modelos de Dominio Específico. *Revista de Investigación de Sistemas e Informática*, julio-diciembre 2010, vol. 7, nº 2,

ZUKOWSKI, J. *Programación Java 2.* Madrid: SYBEX Inc, 2003. ISBN 84-415-1559-X.

ANEXOS

Anexo 1: Operacionalización de Variables.

1) **<u>Variable Independiente</u>: Herramienta CASE para el modelado**

***Definición**:* Según (Sommerville, 2006) una herramienta CASE es "el software utilizado para apoyar las actividades del proceso de software, tales como la ingeniería de requerimientos, diseño, implementación y pruebas". Por tanto, se entiende como una herramienta CASE para el modelado al "software utilizado para modelar un sistema", que incluye además funciones automatizadas tales como la generación de código.

Tabla 16: Operacionalización de la variable independiente de la investigación

Variable	Dimensión	Indicador	Escala de valores
Herramienta Case para el modelado		Uso del meta-modelo de ApEM-L 2.0.	**Sí**: Trabaja con el meta-modelo de ApEM-L 2.0.
			No: Trabaja con el meta-modelo de cualquier otro lenguaje de modelado.
	Capacidad de codificación.	Grado de transformación a código fuente usando ApEM-L 2.0.	**Alto**: codifica todos los componentes del diagrama.
			Bajo: no codifica todos los componentes del diagrama.

2) **<u>Variable Dependiente:</u> Consistencia y completitud del código generado**

***Definición de consistencia**:* Es un estado en el que no existen contradicciones entre el modelo y el código generado.

***Definición de completitud**:* Se refiere al grado en que se ha conseguido la total implementación de los componentes del diagrama.

Dimensiones:

Grado de correspondencia: Se refiere al nivel de sincronización que existe entre los diagramas y el código generado de un determinado sistema.

Auto-Documentación: El grado en que el código fuente proporciona documentación significativa.

Tabla 17: Operacionalización de las variables dependientes de la investigación

Variable	Dimensión	Indicador	Escala de valores
Consistencia	Grado de correspondencia entre el código y el diagrama de clases del diseño.	Cantidad de clases y relaciones del diagraman que han sido codificadas	**Alta**: se codifican todas las clases y relaciones del diagrama.
			Baja: faltan algunas clases o relaciones del diagrama por codificar.
		Representación de los estereotipos	**Alta:** los estereotipos se ven adecuadamente reflejados en el código.
			Baja: los estereotipos no se ven adecuadamente reflejados en el código.
Completitud	Auto-Documentación.	Calidad de la documentación del código.	**Alta:** Se realiza una exhaustiva documentación del código.
			Media: Se realiza poca documentación del código.
			Baja: No se realiza documentación del código.
	Cantidad de código	Transformación a	**Alta:** Se generó código de todas las clases con sus atributos, métodos,

	generado en función del enfoque.	código de los componentes del diagrama.	relaciones y estereotipos restrictivos definidos.
			Baja: No se generó el código de todas las clases con sus atributos, métodos, relaciones y estereotipos restrictivos definidos.

Combinación de valores:

Alta y Baja → Media

Alta y Alta → Alta

Baja y Baja → Baja

Baja y Alta →Media

Media y Alta→ Alta

Media y Baja→ Baja

Anexo 2: Registro de defectos y dificultades detectados.

Tabla 18: Listado de las No conformidades detectadas con las pruebas de Caja Negra

Elemento	No	No Conformidad	Etapa de detección	Estado	Respuesta del equipo de desarrollo
Formato	1	A la hora de realizar la validación del formato detecta si tiene algún error pero no muestra el mensaje correspondiente al usuario.	Prueba	PD 20-05-2014 RA 27-05-2014	NC corregida
Dirección	3	Cuando se va a guardar el código no muestra el mensaje de error en caso de que los datos sean incorrectos.	Prueba	PD 20-05-2014 RA 27-05-2014	NC corregida

Anexo 3: Código completo generado por el plugin para la clase CMEP_ Video.

```
<?php

require_once('Coneccion.php');
//Start of user code referencias
    //Referencias definidas por el usuario
//End of user code

//Start of user code constantes
    //Constantes definidas por el usuario
//End of user code


/**
 * Breve descripción de la Vista CModEP_Video
 * @comment
 *
 * @access public
 * @author nombre y apellidos del autor, <autor@uci.cu>
 * @classType Apeml_Modelo_Entidad_Persistente
 */
class CModEP_Video
{
    //----ASOCIACIONES----

    //------ATRIBUTOS-----

    /**
     * Breve descripción del Atributo CodVideo
     * @access public
     * @type undefined
     */
     public $CodVideo = null;


    /**
     * Breve descripción del Atributo TipoVideo
     * @access public
     * @type undefined
```

Figura 24: Primer fragmento de código generado por el plugin

```
public $TipoVideo = null;


/**
 * Breve descripción del Atributo Resolucion
 * @access public
 * @type undefined
 */
public $Resolucion = null;


/**
 * Breve descripción del Atributo CalidadImagen
 * @access public
 * @type undefined
 */
public $CalidadImagen = null;


/**
 * Breve descripción del Atributo Duracion
 * @access public
 * @type undefined
 */
public $Duracion = null;


/**
 * Breve descripción del Atributo CalidadSonido
 * @access public
 * @type undefined
 */
public $CalidadSonido = null;


//-------METODOS------

/**
```

Figura 25: Segundo fragmento de código generado por el plugin

```
 * @author nombre y apellidos del autor, <autor@uci.cu>
 * @return undefined
 */
 public  function localizarVideo(){
     // Start of user code localizarVideo
        //Por implementar !!!
     // End of user code
 }


//Start of user code Insertar

/**
 * Función encargada de insertar los datos pasados por parametros
 * en la base de datos
 * @author PhpGenerator
 */
public function Insertar($valor1,$valor2){
    $consulta = "INSERT INTO nombre_tabla (columna1,columna2) VALUES ('$valor1','$valor2')";
    return $this->conexion->query($consulta);
}

///End of user code

//Start of user code Eliminar

/**
 * Función encargada de eliminar los datos pasados por parametros
 * en la base de datos
 * @author PhpGenerator
 */
public function Eliminar($valor1,$valor2){
    $consulta = "DELETE From nombre_tabla WHERE columna1 ='$valor1' and columna2 = '$valor2'";
    return $this->conexion->query($consulta);
}

//End of user code
```

Figura 26: Tercer fragmento de código generado con el plugin

```
//Start of user code Actualizar

/**
 * Función encargada de actualizar los datos
 * en la base de datos
 * @author PhpGenerator
 */
public function Actualizar($valor1 ,$valor2){
    $consulta = "UPDATE nombre_tabla SET columna='$valor1' WHERE columna2 = '$valor2'";
    $this->conexion->query($consulta);
}

//End of user code

//Start of user code Mostrar

/**
 * Función encargada de mostrar los datos
 * @author PhpGenerator
 */
public function Mostrar($valor1 ,$valor2){
    $consulta = "SELECT elementos FROM nombre_tabla WHERE columna1 = '$valor1' and columna2 = '$valor2'";
    $resultado = $this->conexion->queryAsObject($consulta);
    return $resultado;
}

//End of user code
```

Figura 27: Cuarto fragmento de código generado con el plugin

Anexo 4: Entrevista dirigida a los profesores y especialistas informáticos que han usado el lenguaje ApEM-L en el proceso de desarrollo de aplicaciones educativas.

Cro. (a): Actualmente existe una investigación sobre el Lenguaje para la Modelación de Aplicaciones Educativas (ApEM-L) en su versión 1.0. La entrevista que se presenta a continuación tiene como objetivo identificar los problemas existentes en la generación de código y la importancia de automatizar este proceso.

Muchas Gracias, Colectivo de Autores.

a. ¿Qué herramienta utiliza para el modelado de las aplicaciones educativas?

b. ¿Cómo se lleva a cabo la generación de código?

c. ¿Los nuevos estereotipos definidos por el lenguaje se reflejan correctamente en el código generado por la herramienta?

d. ¿Qué por ciento de error presenta el código generado?

e. ¿Qué ventajas le proporcionaría contar con una herramienta que genere el código a partir de los diagramas?

GLOSARIO DE TÉRMINOS

Metamodelo: Un metamodelo es un modelo que define el lenguaje para expresar un modelo.

Metodología: Se refiere a los métodos de investigación en una ciencia. Se entiende como la parte del proceso de investigación que permite sistematizar los métodos y las técnicas para llevarla a cabo.

Pruebas: Son un elemento importante para garantizar la calidad de los productos. Su principal objetivo es encontrar la mayor cantidad de errores en el software. No aseguran la ausencia de defectos en el software, sólo demuestran la presencia de errores en el mismo.

Software: Es un término genérico que designa al conjunto de programas de distintos tipos (sistema operativo y aplicaciones diversas) que hacen posible operar con el ordenador.

Ingeniería de Software: Es una disciplina dentro del área de la informática que ofrece métodos y técnicas para desarrollar y mantener software de calidad.

Tecnología: Tecnología es el conjunto de conocimientos técnicos, ordenados científicamente, que permiten construir objetos y máquinas para adaptar el medio y satisfacer las necesidades de las personas.

UML: Lenguaje Unificado de Modelado (Unified Modeling Language). Notación gráfica utilizada para describir sistemas de software.

IDE: Entorno Integrado de Desarrollo, es un programa compuesto por un conjunto de herramientas para un programador. Puede dedicarse exclusivamente para un lenguaje de programación o bien para varios.

Plugin: Es una aplicación que se relaciona con otra para aportarle una función nueva y generalmente muy específica. Esta aplicación adicional es ejecutada por la aplicación principal e interactúan por medio de la API.