Herramientas CASE para la validación de modelos UML a través de invariantes OCL

Pablo Pesce and Claudia Pons

LIFIA - Laboratorio de Investigación y Formación en Informática Avanzada
Universidad Nacional de La Plata, Buenos Aires, Argentina.
{cpons, ppesce}@info.unlp.edu.ar

Resumen

A partir de la estandarización del lenguaje UML, han surgido en el mercado numerosas herramientas de modelado. Entre todas ellas es posible identificar funcionalidades que se repiten, mas agregados que las distinguen. En general, estas herramientas están construidas de manera monolítica, haciendo impracticable la combinación de funcionalidades o el desarrollo de una herramienta que provea una capacidad adicional y que pueda ser usada en el contexto de cualquiera de ellas. Con el advenimiento de los ambientes de desarrollo integrados (IDE) como eclipse, el concepto de herramienta CASE ha dado un cambio significativo. La posibilidad de generar plug-in's que enriquezcan el ambiente con una funcionalidad adicional hace posible la combinación por parte del usuario de distintos plugins que le permitan formar el ambiente de desarrollo que mejor se ajuste a los requerimientos de cada proyecto en particular. Se describe aqui la arquitectura de la herramienta ePLATERO, construida en base a este enfoque y en particular se detalla uno de los plug-in's que fueron desarrollados en el contexto de este proyecto, y que provee la funcionalidad necesaria para extender la plataforma eclipse con la capacidad de asociar modelos de distintos metamodelos sobre los proyectos de la misma.

Palabras claves: UML, OCL, Proceso de desarrollo, Lenguajes de modelación, eclipse.

1 Introducción

A partir de la estandarización del lenguaje UML[4], han surgido en el mercado numerosas herramientas de modelado basadas en UML, por ejemplo Rational Rose[9], Together[10], ArgoUML[7], OCLE[1]. En general, es posible identificar un conjunto de funcionalidades básicas que deberían estar presentes en cualquiera de ellas, por ejemplo:

- Edición de modelos.
- Persistencia de los modelos.
- Evaluación de restricciones sobre los modelos, tales como restricciones de integridad, consistencia, propiedades sintácticas y semánticas.
- Evolución de modelos a través del proceso de desarrollo iterativo e incremental.
- Generación de código a partir de los modelos.
- Sincronización entre el código y los modelos.

Sin embargo, debido a la complejidad de cada una de estas funcionalidades, las herramientas incluyen sólo algunas de ellas. Además, resulta impracticable la combinación de las funcionalidades de dos herramientas distintas, ya que cada una usa su propia representación para los modelos UML.

Recientemente, el eXtensible Metadata Interchange (XMI)[5] surgió como una solución para este problema ya que provee un formato standard para expresar modelos UML, permitiendo de esta forma que diferentes herramientas cooperen mediante el intercambio de modelos expresados en XMI.

Por lo tanto la tendencia actual en el área de construcción de herramientas de modelado consiste en crear pequeñas herramientas que puedan ser combinadas para obtener distintas funcionalidades, de acuerdo a los requerimientos de cada proyecto de desarrollo de software.

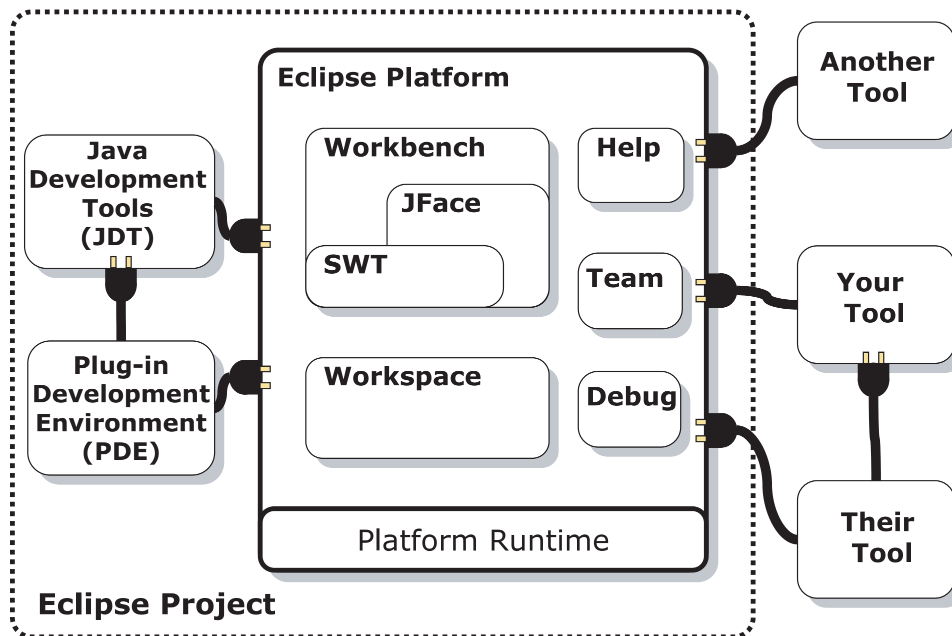


Figure 1: Arquitectura de Eclipse

Un grupo configurable de pequeñas herramientas es mas útil que una herramienta monolítica con un conjunto fijo de funciones. Además de esta manera se evita la duplicación de funcionalidad entre las distintas herramientas.

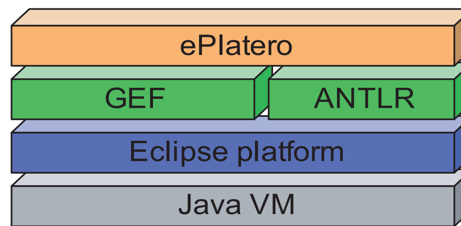
Por otra parte, la plataforma Eclipse es un ambiente de desarrollo integrado (IDE) de código abierto de propósito general, construida sobre un mecanismo para descubrir, integrar, y ejecutar módulos llamados plug-in. En la plataforma Eclipse, un plug-in es la unidad más pequeña de una función que puede ser desarrollada y entregada de manera separada. El modelo de interconexión es simple: un plug-in declara uno o más puntos de extensión, y a su vez puede contribuir con extensiones a uno o mas puntos de extensión de otros plug-ins. Excepto por un pequeño núcleo llamado Platform Runtime, toda la funcionalidad de la plataforma Eclipse esta situada en plug-ins (ver Figura 1).

Estas características hacen de Eclipse el ambiente ideal para el desarrollo de herramientas altamente integradas. Como un primer esfuerzo por estudiar mecanismos que permitan agregar capacidades de validación formales a las herramientas CASE basadas en modelos UML, nació la herramienta *Pampero*[3], concebida como un plug-in para eclipse. La misma provee la capacidad de edición de diagramas UML, escritura de expresiones OCL a nivel de metamodelo, y evaluación de dichas expresiones detectando invariantes que fallan y marcando directamente sobre los diagramas estas fallas encontradas. De la experiencia adquirida en el desarrollo de esta herramienta se presenta aquí la propuesta de arquitectura de ePLATERO (**e**clipse **Pl**ugin **A**ssuring **T**raceability in an **E**nvironment with **R**efinement **O**rientation), como la evolución natural de *Pampero*. La arquitectura fue completamente rediseñada para que la herramienta se forme como el uso coordinado de una serie de plug-in's, cada uno de los cuales ofrece una funcionalidad específica y con la posibilidad de reusar desacopladamente los mismos en otras herramientas y en otros contextos. En las sección 2 se describe en detalle la arquitectura de ePLATERO. En la sección 3 se describe uno de los plug-in's desarrolados en el contexto de este proyecto, que define la minima funcionalidad necesaria para lograr la fácil comunicación entre los modelos de distintas herramientas y es propuesto como un plug-in que podría formar parte de la plataforma eclipse, beneficiando a tanto a los desarrolladores de plug-in's, como a los usuarios de herramientas basadas en metamodelos. Finalmente, en la sección 4 se enumeran las conclusiones y trabajo futuro.

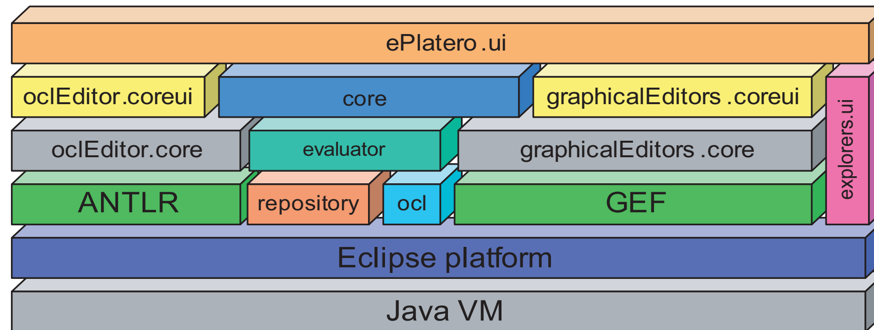
2 Arquitectura de ePLATERO

La arquitectura del plug-in ePLATERO se basa en una serie de plug-in's con funcionalidades altamente desacopladas. Desde un nivel abstracto, puede ser visto como una herramienta que extiende la plataforma eclipse y que tiene como requisitos a los plug-in's GEF[2] y ANTLR[6], externos al proyecto ePLATERO. (Figura 2(a)).

En un nivel mas detallado, el plug-in puede ser visto como un componente que agrupa la funcionalidad de otras componentes individuales, lo cual se logra separando funcionalidades reusables en plug-in's independientes, con un estructura de dependencias como muestra la figura 2(b). El principal objetivo de este esquema,



(a) Plug-in ePLATERO



(b) ePLATERO en detalle

Figure 2: Abstract View

fue buscar que cada plug-in fuese una unidad de código reusable y extensible. aún fuera del contexto de ePLATERO.

Si bien existen posibles variantes en cuanto a esta arquitectura, se decidió que este esquema permitía poder separar las funcionalidades específicas en plug-in's que eran más fácilmente reusables en otros contextos. Para lograr esto se decidió contar con plug-in's en las capas superiores que actúan como plug-in's integradores.

Una situación adicional por la cual las dependencias entre los plug-ins es uno de los puntos mas delicados del desarrollo, es que si no se logra una adecuada separación, puede verse afectada no solo la performance de ePLATERO, sino la de todo el ambiente integrado (el eclipse IDE). Se debe tener en cuenta que el plug-in desarrollado es solo uno de muchos otros que van a estar conviviendo en el ambiente de desarrollo eclipse, por lo cual, pequeños tiempos de carga de cada plug-in individual pueden afectar la performance total en el arranque de la plataforma.

La separación en plug-in's individuales tiene varias razones y ventajas:

1. Por un lado, dado que los plug-in's se cargan de manera lazy, si no se usa cierta funcionalidad, como por ejemplo la validación, el plug-in nunca se activa. Este es uno de los principios de diseño claves en al arquitectura de eclipse, y que se logró luego de frustrados intentos en los cuales el tiempo de carga de la aplicación era extremadamente largo, dado el costo de tener que activar todas las extensiones[8].

2. Por otro lado, los plug-ins pueden ser usados de manera independiente con otras finalidades.

Otro de los principios de diseño fuertemente usados en la plataforma eclipse, al que se le dio mucha importancia en ePLATERO, es el de separar la funcionalidad no-UI (sin presentacion gráfica) de la UI. Esto trae ventajas, ya que si solo se requiere usar cierta capacidad de un plug-in, sin requerir la parte gráfica de la herramienta, la misma no se activará.

3 Core plug-in

Uno de los objetivos de ePLATERO es brindarle al usuario la capacidad de crear elementos de modelación que son instancias de algún meta-modelo. El core plug-in es el que reutiliza la funcionalidad de otra componente llamada *models*, registrando a través de una *extensión* un repositorio particular.

El plug-in *models* extiende la plataforma eclipse para permitir que en cada proyecto sea posible asociar modelos, independientemente de la forma en que se representen. Ademas se provee la capacidad de notificación de eventos en el momento mas adecuado siguiendo una filosofía de carga *lazy*. La relación con los demás componentes se muestra en la figura 3. La funcionalidad genérica para asociar modelos con proyectos independientemente del meta-modelo del que sean instancia está definida en el plug-in *models*. El plug-in *core* extiende este plug-in proporcionando un *ModelAdapter* que lo vincula con el meta-modelo de ePLATERO.

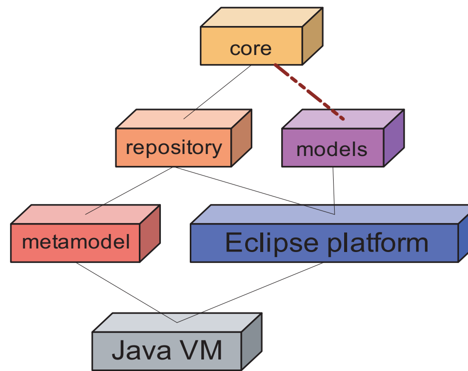


Figure 3: Plug-in ePLATERO.core

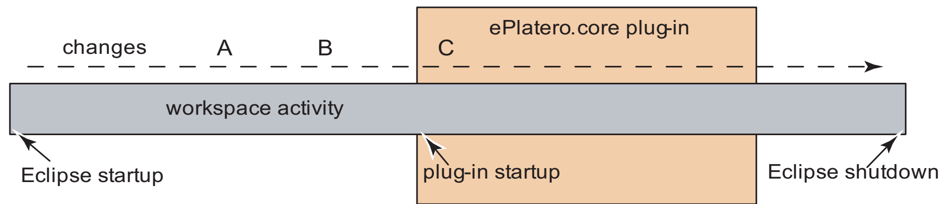


Figure 4: Ciclo de vida del core plug-in

3.1 El ciclo de vida del core plug-in

En general, todo plug-in tiene asociada una clase cuya única instancia permite acceder a información del plug-in. La clase *CorePlugin* es un singleton que permite entre otras cosas controlar el ciclo de vida del plug-in.

Los plug-in's se cargan dinamicamente. Hasta que su funcionalidad no es requerida, ningún objeto es mantenido en memoria. Cuando algún otro plug-in requiere funcionalidad del *core* ocurre el startup del plug-in. En la figura 4 se muestra la relación entre el ciclo de vida de la plataforma y el *core* plug-in.

La clase *CorePlugin* reimplementa el método startup() el cual es ejecutado automaticamente por la plataforma *eclipse* la primera vez que se activa el plug-in. La activación la realiza el *ClassLoader* en el instante en que se requiere el acceso a alguna clase interna dentro del plug-in como se muestra en la figura 5.

Como se muestra en la figura 6, la clase *CorePlugin* hereda de su superclases los métodos startup() y shutdown() de los cuales reimplementa el primero. También da acceso al workspace de eclipse, lo cual evita tener que referenciar directamente a la clase *ResourcesPlugin* que es quien realmente posee una referencia al workspace. La clase *Workspace* es importante para el core plug-in ya que en ella se registran listeners sobre distintos eventos que ocurren en los recursos, como cambios en proyectos, archivos, etc. Una de las responsabilidades del core plug-in, es brindar una vision del workspace orientada a modelos llamada *WorkspaceModel*. Para mantener un sincronismo entre ambos workspaces es de gran ayuda los métodos antes mencionados.

Para entender la arquitectura del core plug-in, es necesario conocer ciertos detalles sobre la plataforma eclipse que se tratan de describir brevemente a continuación y que se pueden apreciar en la figura 7.

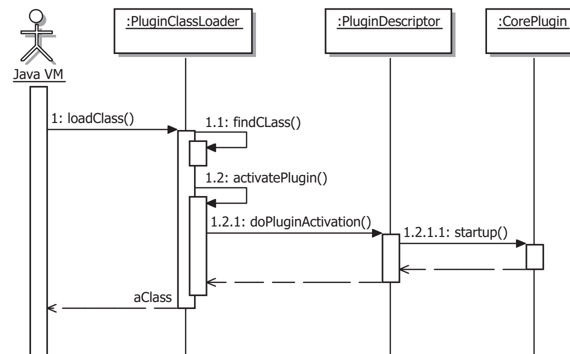


Figure 5: Activación de un Plug-in

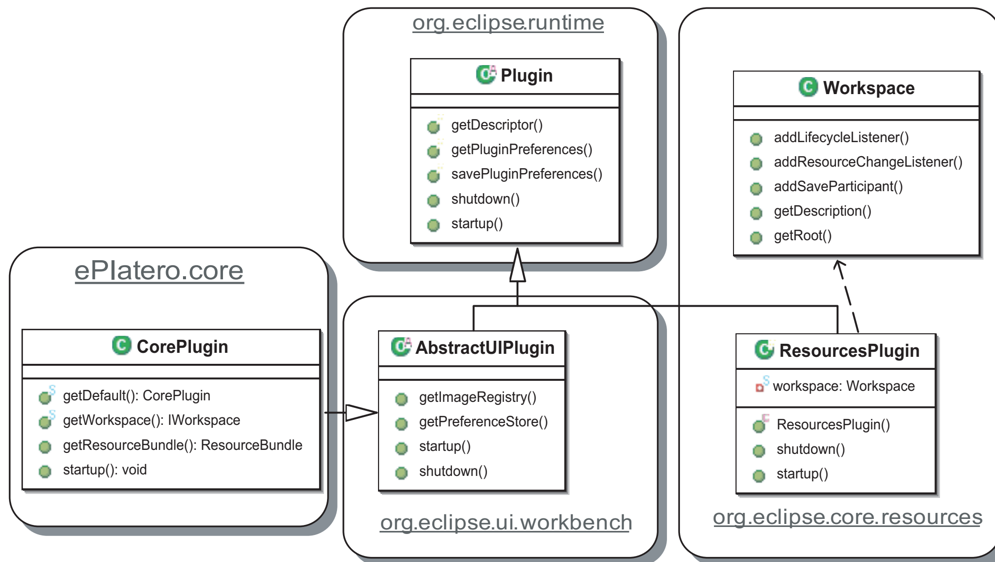


Figure 6: Clase CorePlugin

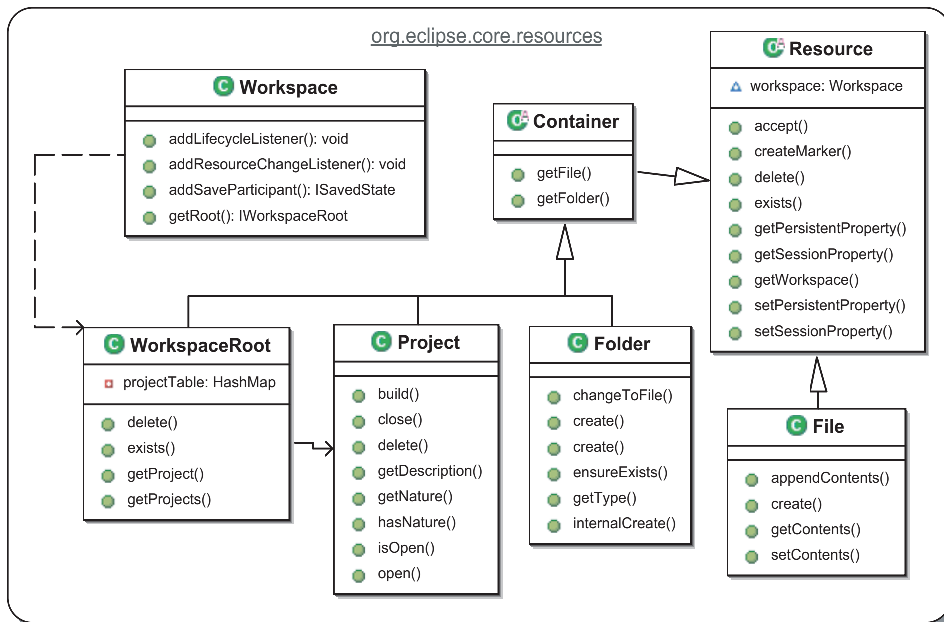


Figure 7: Plug-in org.eclipse.core.resources

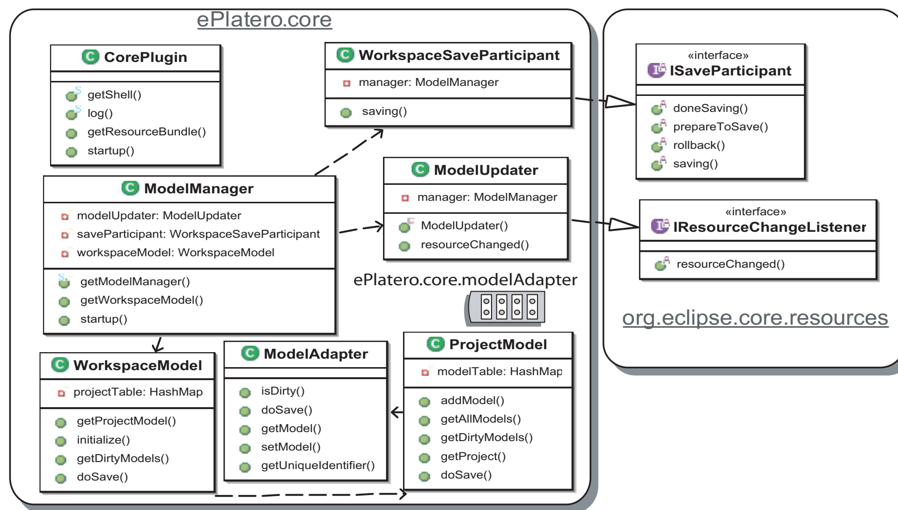


Figure 8: Arquitectura del core plug-in

Eclipse es una plataforma basada fuertemente en recursos. Al arrancar eclipse es posible configurar un directorio del sistema, el cual funcionara como area de trabajo o *Workspace*. Dentro del area de trabajo el usuario organiza sus trabajos en carpetas separadas llamadas proyectos. En cada proyecto, es posible crear tanto archivos como carpetas. Esto es lo que ocurre físicamente en el sistemas de archivo de disco. Internamente eclipse mantiene un conjunto de objetos sincronizada con esta estructura. Es así como tenemos la clase *Project* que se mapea con los proyectos en el workspace. Las carpetas y los archivos en el workspace se representan en eclipse como instancias de las clases *Folder* y *File*. En general estas clases no se referencian directamente sino que eclipse provee las interfaces *IProject*, *IFolder* o *IFile*. De esta forma es posible acceder y operar sobre los recursos del workspace.

3.2 Soporte de modelos

Uno de los objetivos del core plug-in es dar soporte para mantener en memoria y permitir acceder a instancias de algún metamodelo. La forma en que se resuelve esto es dando un conjunto de clases que dan una visión del workspace orientada a modelos, donde cada proyecto puede tener asociados uno o mas modelos de distintos metamodelos. En la figura 8 se pueden apreciar las clases que permiten asociar modelos con proyectos.

Basicamente el *WorkspaceModel* contiene tantos *ProjectModel* como proyectos *abiertos* haya en el workspace de eclipse. Cada *ProjectModel* puede a su vez referenciar tantos modelos como sea necesario. La clase *ModelAdapter* es la que se encarga de hacer la adaptación con los distintos modelos. Como se verá mas adelante, independientemente de la forma en que se represente el metamodelo (ya sea a través de un repositorio o directamente como una librería jar), se deberá registrar el mismo a través de una clase que respete la interface *IModelAdapter* comunicando la misma al *core* plug-in a través de un punto de extensión llamado ePlatero.core.modelAdapter (ver Figura 8).

3.3 Guardando los cambios

Una de las responsabilidades del core plug-in, además de darnos un lugar para hacer residir los modelos, es la capacidad de que los mismos estén disponibles para cualquier otro plug-in que los requiera, aun después de cerrar y reabrir el eclipse. Desde el punto de vista del desarrollador que desea usar la funcionalidad del core plug-in, al implementar el *IModelAdapter* adecuado para su meta-modelo o repositorio, deberá implementar las operaciones doSave() y doLoad(). El core se encarga de invocar en el momento adecuado estas operaciones sobre el adapter del modelo.

A continuación se analizarán las dos operaciones en detalle. Se comenzará describiendo las acciones que dan origen al envío del mensaje doSave() sobre el modelo.

Una situación que puede darse es que se intente cerrar el eclipse habiendo hecho cambios sobre el modelo y no habiéndolos guardado. Es común si trabajamos con un editor en eclipse y dado a lo que se conoce como *open-save-close lifecycle model*, que si modificamos un recurso e intentamos cerrar el eclipse, nos aparezca una lista con los recursos que han sido modificados y no han sido guardados, permitiendonos guardarlos antes de cerrar el eclipse. La serie de eventos que llevan a la apertura de esta ventana se pueden apreciar

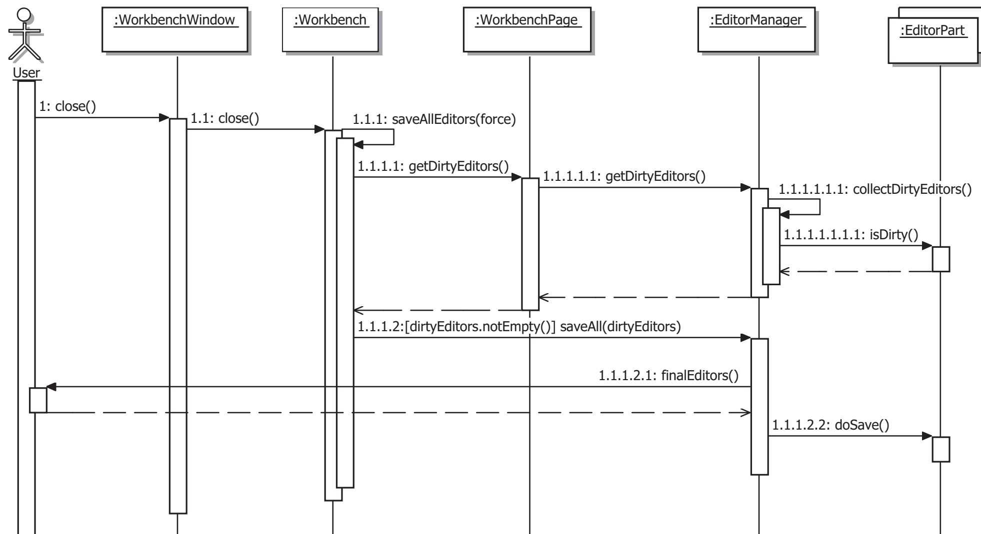


Figure 9: Cierre de Eclipse

en el diagrama de secuencia de la figura 9. El método isDirty() en los editores es el que debe retornar si el modelo tiene un estado que ha cambiado respecto del que tenia inicialmente al abrir el editor.

El problema para reutilizar este mecanismo en el core plug-in, es que el mismo solo funciona para instancias de la clase *IEditorPart* y dado que se quiere que un modelo pueda ser compartido por multiples editores, no se tiene un editor especifico sobre el modelo que este abierto en todo momento y que sea responsable de guardar los cambios.

Sin embargo el mecanismo que se implementó, tiene mucho en común con lo anterior. En principio la interface *IModelAdapter* exige la implementación del método idDirty() con el mismo significado que para los editores, es decir, poder consultar si el modelo ha sufrido cambios que pueden perderse si no son guardados.

Para lograr que cuando se cierre el eclipse se chequee por esa propiedad sobre los modelos se definió la clase *WorkspaceSaveParticipant*. Esta clase se registra en el startup() del plug-in, de modo de ser notificada por el workspace, del evento de cierre de la plataforma. Como un *ISaveParticipant* esta clase es notificada de tres tipos de eventos: SNAPSHOT, PROJECT_SAVE y FULL_SAVE. Actualmente solo se responde al evento FULL_SAVE que ocurre cuando el usuario da la orden de cerrar el eclipse. En este caso, el *WorkspaceSaveParticipant* recibe el método saving() (Figura 10(a)). Es aquí donde se recolectan todos los modelos que estén en un estado dirty y solo aquellos que el usuario desea guardar recibirán el método doSave() el cual tiene por objetivo dejar la información disponibles para ser recuperada como se explica mas adelante. En el diagrama de secuencia de la figura 10(b) se muestran las acciones descriptas anteriormente.

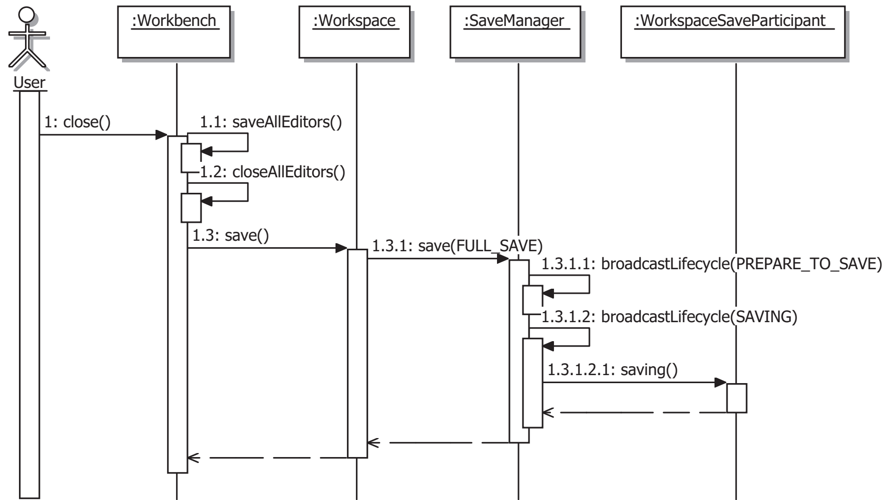
Durante el método doSave() del *ProjectModel* se debe guardar la información necesaria para poder volver a recuperar el proyecto. Esto se logra, guardando para cada modelo, la información necesaria para restaurarlo, todo en un archivo XML dentro de la carpeta del proyecto. Se utiliza la clase *XMLMemento* de eclipse para facilitar la generación del archivo XML. Cada *IModelAdapter* es responsable de guardan en el archivo XML la informacion que necesita para restaurar el modelo particular que adapta. Esto lo debe hacer implementando el método save(IMemento) escribiendo en el memento la información de recuperación. Por ejemplo, un *ModelAdapter* particular podría guardar el nombre del archivo dentro del proyecto donde se guardo el modelo. En la figura 11 se muestran las acciones que ocurren durante el doSave() del *ProjectModel*.

3.4 Recuperación de modelos

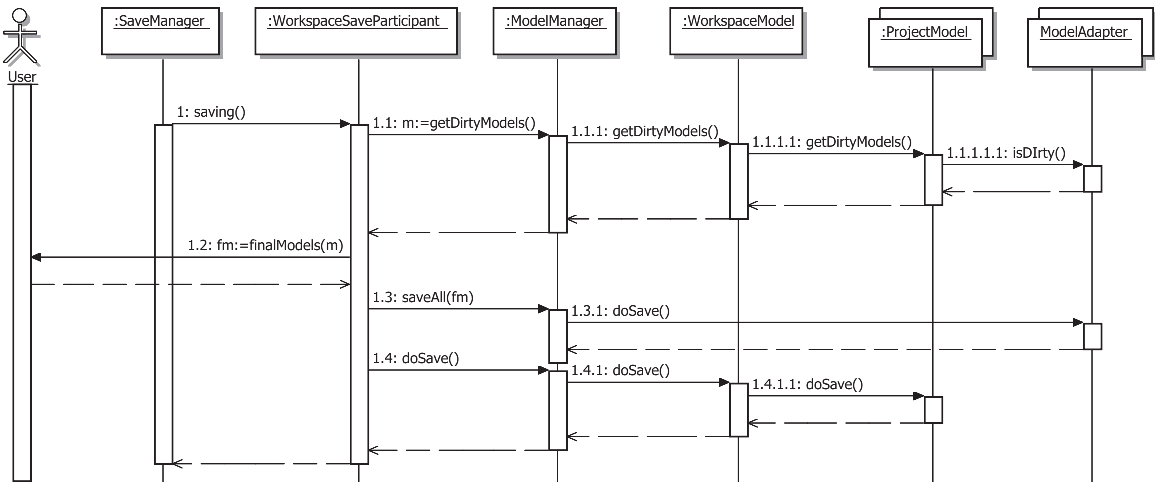
Cuando algún plug-in desea acceder a alguno de los modelos generados previamente, puede hacerlo a través del core plug-in. Por una cuestión de eficiencia los modelos no se recuperan de disco hasta que no sean realmente requeridos. Todo plug-in que quiera usar las facilidades del core plug-in, deberá navegar por sus instancias, comenzando generalmente por el *WorkspaceModel* el cual puede ser accedido mediante el singleton *ModelManager*.

Por ejemplo, la siguiente secuencia recupera el adapter de un modelo especifico dentro de un proyecto.

```
IWorkspaceModel workspaceModel = ModelManager.getModelManager().getWorkspaceModel();
IProjectModel projectModel =  workspaceModel.getProjectModel(anEclipseProject);
IModelAdapter modelAdapter =  projectModel.getModelAdapter(aModelAdapterID);
```



(a) Protocolo de notificación a un *SaveParticipant*



(b) Acciones del core plug-in durante el `saving()`

Figure 10: Acciones durante el cierre del core plug-in

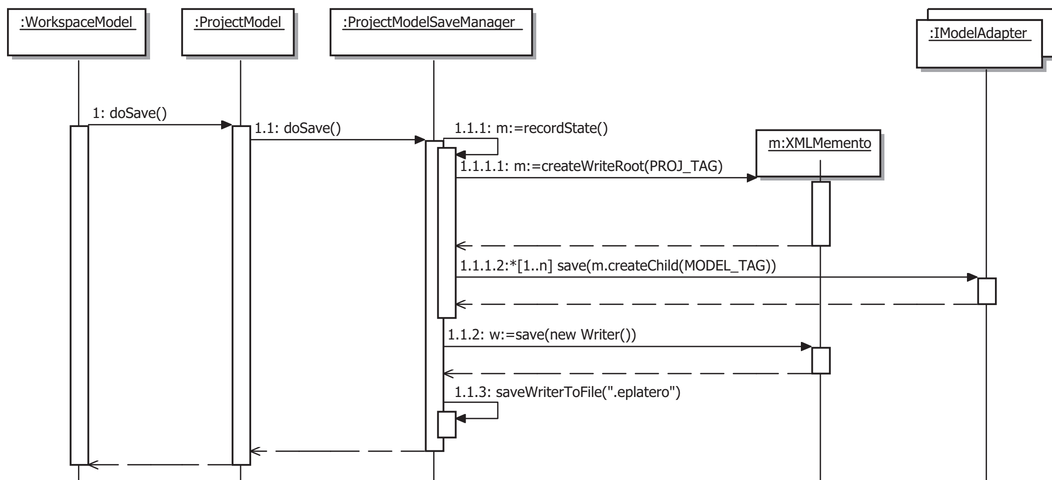


Figure 11: Memorizando el estado del core

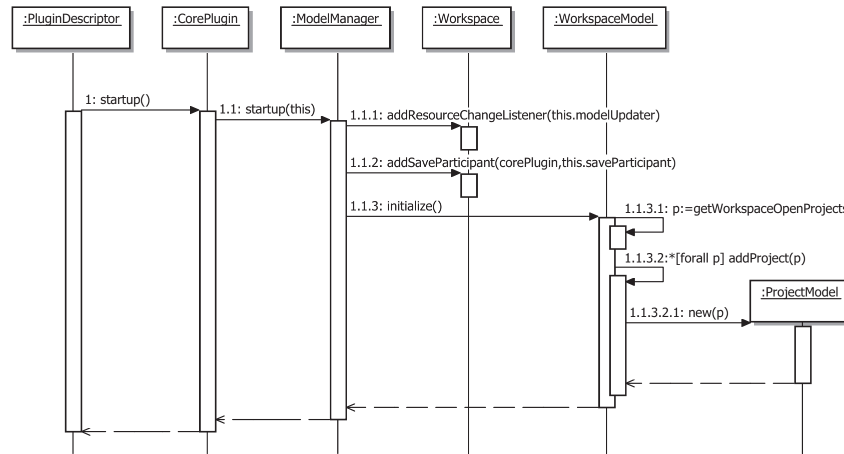


Figure 12: Activación del Core Plug-in

Es de esperarse que quien acceda al modelAdapter pueda realizar el casting a un tipo más especifico que *IModelAdapter*.

Como todo plug-in, el core se carga dinamicamente. Al activarse, se ejecuta automáticamente el método `startup`. Durante el mismo se realizan las siguientes acciones:

- se registra un *SaveParticipant* de modo de recibir la notificación de eventos como por ejemplo, el de cierre de la plataforma, y de esa forma poder salvar el estado actual del core plug-in de modo de poder restaurarlo en la proxima ejecución de la aplicación.
- se registra un *ResourceChangeListener* para lograr un sincronismo con el workspace de eclipse.
- se inicializa el *WorkspaceModel*, creando para cada proyecto *abierto* del workspace de eclipse, un *ProjectModel*

En el diagrama de la figura 12 se aprecia la secuencia de operaciones durante este proceso.

Otra situación a tener en cuenta durante el `startup()` es que existe la posibilidad de que hayan ocurridos eventos de interés para el estado del *WorkspaceModel* pero que no hayan sido notificados dado que el core plug-in no estaba activo. Durante este instante se reciben esos cambios, por lo que se debe actuar en consecuencia para restablecer el estado anterior.

Durante la vida del *WorkspaceModel* pueden ocurrir cambios en los recursos del workspace de eclipse. La clase *ModelUpdater* se registra como un *ResourceChangeListener* de modo de recibir los que se describen a continuacion.

- PRE_CLOSE Este evento se envía antes de que se cierre un proyecto. Debe garantizarse que el modelo asociado haya sido guardado. También se deben cerrar todos los editores asociados con recursos dentro del proyecto Por ultimo se elimina el modelo asociado al proyecto del *WorkspaceModel*.
- PRE_DELETE En caso de que no se haga automáticamente se deberían borrar los archivos que guardan el modelo. Al igual que en el caso anterior, se destruye la representación en memoria del modelo en el *WorkspaceModel*.
- POST_CHANGE Este evento permite notificarse de cambios en algún recurso como markers, apertura de proyectos, movimiento de recursos, etc.

Para lograr que la información de cada proyecto se recupere solo si se necesita, se definen en la implementación de *ProjectModel* una serie de estados en los que puede estar el proyecto. Estos estados determinan el comportamiento del *ProjectModel* respecto de que información debe ser recuperada desde el archivo con meta-información sobre el proyecto.

Cuando se intenta acceder a un modelo en particular, en ese instante y en función del estado se pueden realizar las siguientes acciones:

- *NotLoadedProjectModel* En este estado, la información sobre los modelos que contiene el proyecto no ha sido recuperada desde disco. Lo que es peor aún, es que el adapter sobre el modelo debe ser instanciado por el core plug-in, sin poder hacer referencia a la clase, ya que la misma se provee a través de un punto de extensión. La instanciación de esta clase se logra usando los mecanismos de conexión de plug-in's de eclipse.

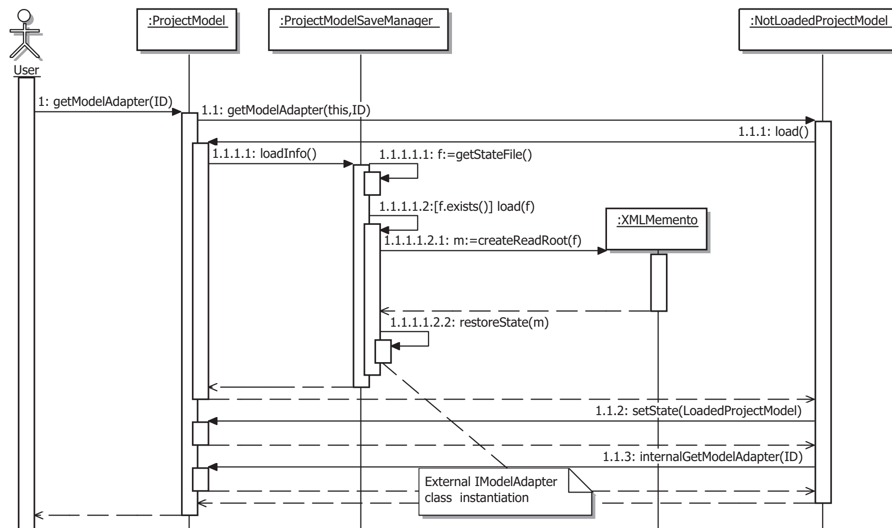


Figure 13: Recuperación Lazy de un *ProjectModel*

- *LoadedProjectModel* Estado de operación normal. Se accede directamente a través del adapter sobre el modelo, que es una instancia en memoria.

En la figura 13 se puede apreciar la primer etapa de la restauración del estado de un *ProjectModel*. Consiste en la lectura a través de un *XMLMemento* del archivo en el cual el proyecto había guardado su estado. El mismo contiene algo como lo que muestra el siguiente esquema:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2  <projectModel>
3     <model IMemento.internal.id=
4                  "ar.edu.unlp.info.sol.ePlatero.core.tests.SimpleModel">
5          <id>ePlatero.nature</id>
6          <filename>IntegerModel.tst</filename>
7      </model>
8  </projectModel>
```

Donde en la linea 4 el core plug-in guardó el nombre de la clase que cumplía el rol de *IModelAdapter* para ese modelo. En el método restoreState() es donde se procede a la instanciación de esta clase, la cual no es conocida por el core, de modo que no la puede instanciar de la forma tradicional, sino que puede hacer esto a través del método getExecutableExtension() sobre la extensión coincidente de todas las que haya sobre el punto de extensión *modelAdapter* (Figura 14). Una vez instanciado se le da su estado previo a través de submemento adecuado del memento global del proyecto.

Al crear los *ModelAdapter's* y como implementación por defecto de *ModelAdapter*, se establece el estado del mismo como *NotSavedState* que le indica que no ha sido guardada en disco. Estando en este estado, y al recibir el método load(memento) se espera que el adapter, se cargue de manera *lazy*, es decir, que solo se prepara para saber a donde ir a buscar el modelo cuando se lo pida, pero sin cargarlo. Por ello el adapter cambia de estado, pasando a un estado llamado *NotLoadedState*. En el momento en que el usuario requiere realmente el modelo, es donde se recupera desde disco, antes de retornarlo. (Figura 15).

4 Conclusiones

Lograr una buena separación de funcionalidades en plug-in's que sean reusables y extensibles no es una tarea sencilla, pero que trae grandes ventajas a la hora de lograr que la herramienta sea fácilmente adaptable a cambios o a nuevas exigencias. Creemos que con la arquitectura propuesta cumplimos con estas exigencias dando no solo un entorno para la validación de modelos UML a través de invariantes OCL, sino que como agregado se logra la rápida adaptabilidad tanto a otras notaciones gráficas basadas en metamodelos, como así también a otros esquemas de verificación y validación.

Eclipse define la noción de *View* y *Editor* dándole al primero un significado persistente en cada acción y asociándole solo a los editores un ciclo *open-save-close*. Este esquema no parece resultar adecuado para nuestro caso, ya que deseamos que cada editor de diagramas, que pueden haber sido generados por distintos desarrolladores, trabajen sobre un modelo compartido, pero sin obligar al usuario de que este modelo

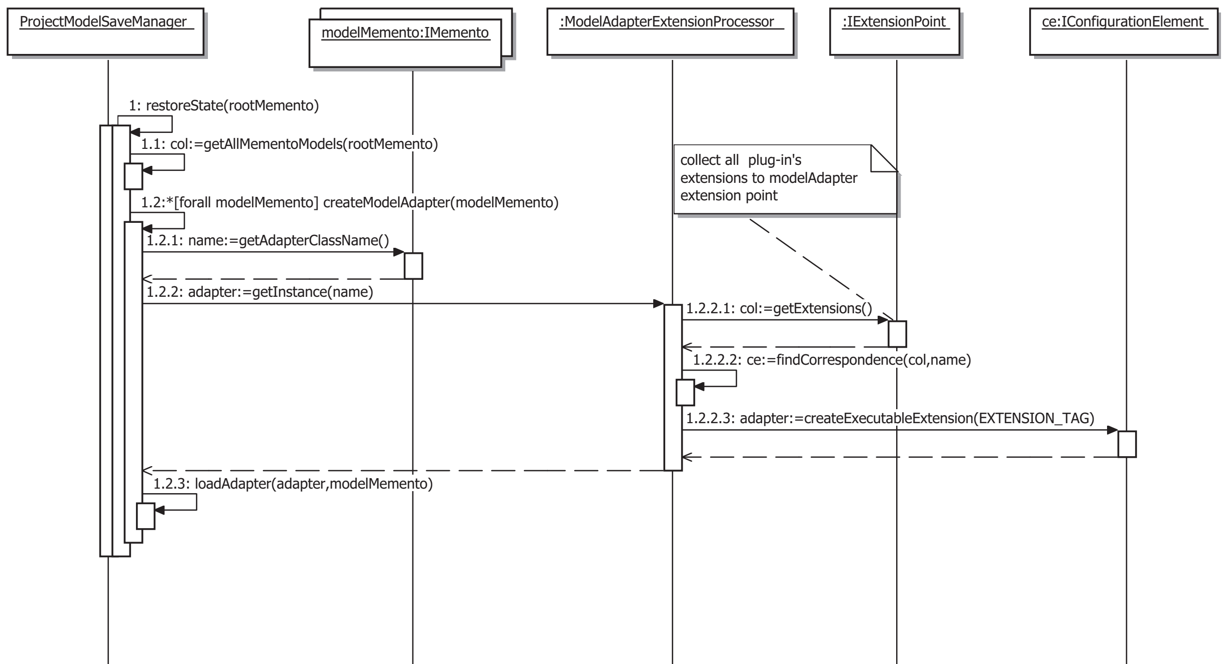


Figure 14: Instanciación de un *ModelAdapter*

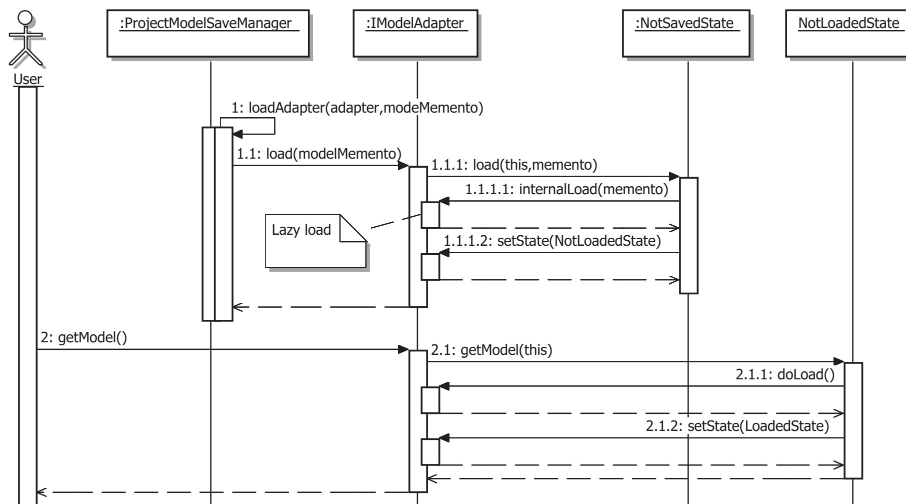


Figure 15: Carga Lazy de un *ModelAdapter*

compartido sea persistente con cada cambio, sino que se desea darle la usuario la capacidad de decidir si los cambios globales deben ser guardados o no. El core plug-in, es el resultado de intentar modificar el comportamiento de la plataforma adecuándolo a las necesidades anteriormente descriptas.

En los últimos años los servicios existentes en las diferentes herramientas de modelado de software, tales como generación de código, validación de modelos, no han podido ser compartidos, ya que cada herramienta usa su propia representación interna para los modelos. Esto llevó a que los desarrolladores de herramientas se vieran forzados a re implementar una y otra vez funcionalidad existente en otras herramientas, ante la imposibilidad de reusarla.

El XMI surgió como un medio para solucionar esta falta de integración entre las herramientas. Sin embargo XMI por si solo no es suficiente, además es necesario definir la arquitectura de la herramienta de manera que en lugar de ser una unidad monolítica, sea un ensamblado de pequeñas herramientas intercambiables, las cuales además puedan ser usadas independientemente o reusadas dentro de otros contextos.

Las ventajas son claras:

- Se reducen de los tiempos y costos para la construcción de nuevas herramientas como consecuencia de la posibilidad de reusar funcionalidad ya implementada por otras herramientas;
- Las herramientas son mas fácilmente modificables y mantenibles debido a su modularidad y simplicidad.
- La funcionalidad es altamente configurable como consecuencia de la posibilidad de combinar las distintas herramientas de varias maneras.

References

[1] Dan CHIOREAN. *OCLE 2.0 - Object Constraint Language Environment*. "BABES-BOLYAI" University. Computer Science Research Laboratory, 2003.
http://lci.cs.ubbcluj.ro/OCLE.

[2] Randy Hudson. *Graphical Editing Framework*. eclipse project, 2003.
http://www.eclipse.org/gef/.

[3] LIFIA. *PAMPERO: **P**recise **A**ssistant for the **M**odeling **P**rocess in an **E**nvironment with **R**efinement **O**rientation*. Universidad Nacional de La Plata, Buenos Aires, Argentina, 2003.
http://sol.info.unlp.edu.ar/eclipse.

[4] Object Management Group (OMG). *The Unified Modeling Language (UML) Specification*. Version 1.5 formal/03-03-01, March 2003.
http://www.omg.org/uml.

[5] Object Management Group (OMG). *XML Metadata Interchange (XMI)*. v2.0 formal/03-05-02, May 2003.
http://www.omg.org/uml.

[6] Terence Parr. *ANTLR*. Developed by jGuru.com, 1989-2003.
http://www.ANTLR.org.

[7] Alejandro Ramirez, Philippe Vanpeperstraete, Andreas Rueckert, Kunle Odutola, and Jeremy Bennett. *ArgoUML*. , 2002.
http://argouml.tigris.org/.

[8] Sherry Shavor, Jim D´Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy. *The Java Developer Guide to Eclipse*. Addison-Wesley, 2003.

[9] Rational Software. *Rational Rose Professional*. , v2002.
http://www.rational.com.

[10] Borland technologies. *Borland Together*. , 2003.
http://www.borland.com/together/index.html.