UNIVERSIDAD COMPLUTENSE DE MADRID
Facultad de Informática

Título: “MOVA Tool”

Proyecto de Sistemas Informáticos

Alumnos:

Rocío Prieto Ruiz
Belén Rodríguez Alonso
Álvaro Suárez Bravo

Profesor director: Manuel García Clavel

Curso: 2006 / 2007

Madrid

INDICE

1. INTRODUCCIÓN

El presente documento presenta la herramienta MOVA Tool, que es una herramienta para modelar y validar diagramas UML.
El lenguaje Unificado de Modelado (UML) es el lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad; aún cuando todavía no es un estándar oficial, está apoyado en gran manera por el OMG (Object Management Group). Es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema de software. Como se indica posteriormente, el lenguaje UML puede resultar no tan conciso y expresivo como puede resultar un lenguaje textual. Es por ello que aparece el lenguaje de restricciones OCL, que complementa al lenguaje UML. El lenguaje OCL se explica más profundamente en la sección 1.1.

El primer objetivo de la herramienta MOVA Tool fue diseñar diagramas de clases y objetos, incorporando las restricciones OCL. Para ello, MOVA Tool incluye la herramienta ITP/OCL Tool que se explica en la sección 1.3. Ésta se basa directamente en especificaciones ecuacionales de los diagramas UML+OCL. ITP/OCL Tool está escrito completamente en Maude, un lenguaje de programación basado en la reescritura de términos que implementa la lógica ecuacional de pertenencia y la lógica de reescritura. En el presente curso se han diseñado e implementado mejoras sobre la versión previa existente, permitiendo una mejor extensión de la herramienta en futuras ampliaciones. Se han aplicado los patrones que se han creído convenientes para hacer más reutilizable el código. En la sección 4 se explican los cambios realizados y su objetivo.

El incremento de software vulnerable ha mostrado repetidamente que el diseño y la ingeniería en sistemas distribuidos desde el punto de vista de la seguridad son a menudo escasos. Esto se debe, en gran parte, a que la seguridad se integra en las últimas fases del proceso de desarrollo. Es recomendable, por tanto, incorporar el diseño de la seguridad desde las primeras fases del ciclo de vida del software. Para cubrir esta deficiencia surgen los diagramas SecureUML.

Como se indica en la sección 1.2, SecureUML es un lenguaje de modelado que define un vocabulario para modelos basados en UML con información relevante al control de acceso. Durante el curso 2006/2007 se han incorporado a la herramienta MOVA Tool los diagramas SecureUML, permitiendo de esta forma diseñar software seguro. En la sección 3 se explica la arquitectura del sistema.

Como se indica en la sección 1.4, MOVA está compuesto de 3 aplicaciones: *UML modeling*, *SecureUML modeling*, y *UML modeling con métricas*. En la sección 2 hay un manual de SecureUML modeling, y en la sección 5 se encuentran los manuales de UML modeling y UML modeling con métricas, indicando el funcionamiento general de la aplicación y cada una de las herramientas.

Para entender mejor el funcionamiento y la utilidad de MOVA, se han incluido en la sección 6 una serie de ejemplos representativos.

Por último, se incluye en la sección 7, el historial de reuniones del curso 2006/2007.

1.1. UML y OCL

El lenguaje unificado de modelado (UML) es un lenguaje de modelado visual de propósito general para especificar, visualizar, construir y documentar los distintos elementos de un sistema software. La notación utilizada se basa en diagramas de distintos tipos: casos de uso, clases, objetos, secuencia… Sin embargo, para ciertos aspectos de los modelos, estos diagramas no son tan concisos y expresivos como puede resultar un lenguaje textual. El lenguaje de restricciones sobre objetos (OCL) es un lenguaje de restricciones textual que complementa a los diagramas UML para proporcionar un mayor manejo del dominio de la representación. OCL ofrece ayuda a la hora de especificar información muy precisa sobre modelos UML.

En la sección de ejemplos se muestran algunos ejemplos que muestran la utilidad de OCL.

1.2. SecureUML

La necesidad de seguridad en sistemas distribuidos a gran escala se ha convertido en una prioridad principal para todas las organizaciones. El incremento de software vulnerable ha mostrado repetidamente que el diseño y la ingeniería de estos sistemas desde el punto de vista de la seguridad son a menudo escasos. Las características de seguridad son frecuentemente añadidas durante una fase posterior al proceso de integración causando errores y vulnerabilidades que los hacen potencialmente atacables. Una posible razón para este enfoque puede ser que la seguridad ha sido sólo recientemente considerada como una parte integral del ciclo de vida en el desarrollo de software. Las organizaciones están empezando a descubrir esta necesidad de realizar cambios en su desarrollo de programas y educar a los desarrolladores y diseñadores de software en el campo de la seguridad.

La necesidad de SecureUML

Hay varias ventajas en la integración de la seguridad durante el ciclo de vida del desarrollo de software. Para empezar, este enfoque permite que los requisitos de seguridad sean integrados en el diseño de sistemas a un alto nivel de abstracción. Este hecho facilita el desarrollo de aplicaciones seguras que evitan la violación de política de seguridad. Además, utilizando SecureUML para modelar la infraestructura de control de acceso pueden prevenirse errores durante la implementación de políticas de control de acceso y permite a la tecnología el desarrollo independiente de sistemas seguros.

Los defectos arquitectónicos más comúnmente conocidos son:

1. Uso incorrecto de criptografía
2. Incorrectas técnicas de gestión de usuarios

3. Mal diseño de autorización
4. Mecanismos ineficaces de autentificación
5. Conjunto de reglas de validación de datos incorrectas e ineficientes.

Muchos estudios han mostrado que es menos caro capturar un error en la fase de diseño que capturarlo en una posterior fase de implementación.

Los motivos por los que las políticas de seguridad no están integradas en la fase de diseño son:

1. La carencia de conocimiento. Hasta hace poco, muchos arquitectos de software no han reconocido totalmente la necesidad de diseñar software seguro.
2. Gastos. La integración de políticas de seguridad y procedimientos incrementa los costes de producción. El actual descubrimiento de vulnerabilidades seguirá acentuando la necesidad del desarrollo de software seguro. Esta tendencia ya ha comenzado y las organizaciones están empezando a ver que la seguridad en el software es una medida de fiabilidad de las aplicaciones que es al menos tan importante como el buen funcionamiento.

SecureUML ayuda a los desarrolladores:

- identificando un pobre diseño e implementación de autorización
- identificando oportunidades de evitar la autorización
- alentando al uso de control de autorización centralizado
- previniendo el uso de suposiciones indocumentadas
- siendo ideal para el control de acceso basado en roles

SecureUML está basado en un modelo extendido para el acceso controlado basado en roles (role-based access control, RBAC). RBAC carece de la capacidad de soportar las condiciones de control de acceso que se refieren al comportamiento de un sistema. SecureUML por lo tanto introduce el concepto de

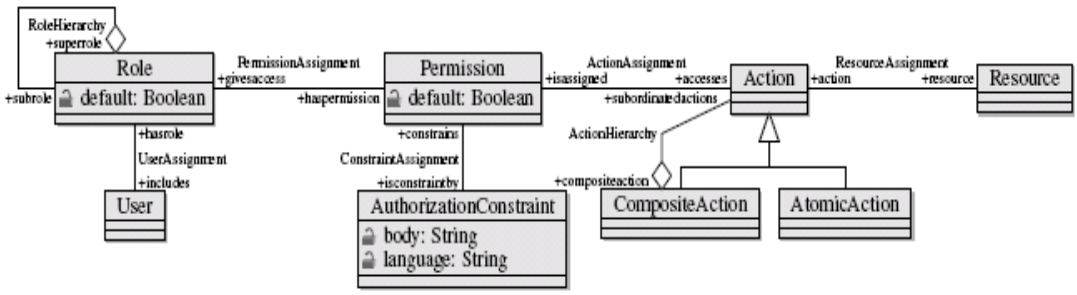
restricción de autorización. Una restricción de autorización es definida como una condición previa para conceder el acceso a una operación.

SecureUML es un lenguaje de modelado que define un vocabulario para modelos basados en UML con información relevante al control de acceso. SecureUML define un vocabulario para expresar diferentes aspectos del control de acceso, como roles, permisos a roles, y asignación de roles a usuarios. Debido a su modelo de control de acceso general y extensible, SecureUML es ideal para el análisis de negocio así como para modelos de diseño para tecnologías diferentes.

Metamodelo de SecureUML

El metamodelo de SecureUML está definido como una extensión del metamodelo UML. Los conceptos de RBAC son representados directamente como tipos de metamodelo.

A continuación se muestra el metamodelo de SecureUML.



Metamodelo de SecureUML

1.3. ITP/OCL

La herramienta ITP/OCL se basa directamente en especificaciones ecuacionales de los diagramas UML+OCL. En concreto:

1) Los diagramas de clases y objetos UML están especificados como teorías ecuacionales de pertenencia.

2) Las restricciones OCL están expresadas como términos booleanos sobre extensiones de las teorías de 1).
3) La validación de diagramas de objetos con respecto a las restricciones se reduce a comprobar si los términos booleanos correspondientes se reescriben a cierto o falso.

ITP/OCL está escrito completamente en Maude, un lenguaje de programación basado en la reescritura de términos que implementa la lógica ecuacional de pertenencia y la lógica de reescritura. Maude es un lenguaje de programación reflexivo. Esto significa que tanto el analizador sintáctico como el motor de reescritura están disponibles para el programador como operaciones: de hecho se utiliza lo primero para implementar el analizador sintáctico OCL en la herramienta, y lo segundo para implementar el motor de validación UML+OCL. El tamaño del código de esta herramienta ronda las 4.000 líneas.

La implementación de una herramienta interactiva en Maude comprende cuatro tareas principales:

- Definir un bucle continuo lectura-evaluación-impresión.
- Definir la sintaxis de las órdenes.
- Definir la interacción con el bucle continuo.
- Definir el procesamiento de las órdenes.

Maude ofrece facilidades para entrada/salida genérica a través del uso de sus "*loop objects*". Proporciona además gran flexibilidad para definir la sintaxis de las órdenes gracias a su terminal y al uso de "*bubbles*" (cualquier lista no vacía de identificadores Maude). Finalmente, el procesamiento de peticiones en una herramienta interactiva viene definido en Maude por ecuaciones actuando sobre los *"loop objects".*

Las órdenes de la herramienta ITP/OCL puede agruparse en cuatro tipos:

1. *Órdenes de creación de diagramas*: Ecuaciones de inserción de diagramas, ya sean de clases o de objetos.

2. *Órdenes de inserción de elemento (clase, atributo, relación y demás) en un diagrama:* añaden clases, atributos, relaciones, generalizaciones y asociaciones a un diagrama de clases, y objetos, valores y enlaces a un diagrama de objetos.
3. *Órdenes que fijan una restricción a un diagrama de clases:* asocian restricciones a diagramas de clases por medio de invariantes definidos por el usuario.
4. *Órdenes que validan un diagrama de objetos:* chequean las restricciones asociadas al diagrama de clases sobre los objetos del diagrama de objetos asociado.

MOVA Tool transforma las acciones del usuario en comandos de los grupos anteriormente citados. Estos comandos se ejecutan a través del intérprete Maude y devuelven el resultado.

1.4. MOVA Tool

Como se indicaba en la introducción, MOVA Tool es una herramienta para modelar y validar diagramas UML. Para ello, MOVA consta de tres aplicaciones:

1. *UML modeling*: que permite al usuario:
   - Dibujar diagramas de clases y objetos,
   - Escribir y comprobar invariantes,
   - Escribir y evaluar consultas, y
   - Definir operaciones OCL para usarlas en las invariantes y consultas.

2. *SecureUML modeling*: que permite al usuario:
   - Dibujar diagramas de seguridad,
   - Escribir y evaluar políticas de seguridad, y
   - Definir operaciones OCL para usarlas en las políticas de seguridad.

3. *UML modeling con métricas*: que permite al usuario:
    - Dibujar diagramas de clases y objetos,
    - Escribir y evaluar invariantes,
    - Escribir y evaluar consultas,
    - Escribir y evaluar métricas, y
    - Definir operaciones OCL para usarlas en las invariantes, consultas y métricas.

2. DIAGRAMAS DE SEGURIDAD

Al ejecutar la herramienta, aparece una ventana (ver Figura 1) que ofrece la opción de iniciar la herramienta en modo:

- UML modeling
- SecureUML modeling
- UML modeling con métricas.

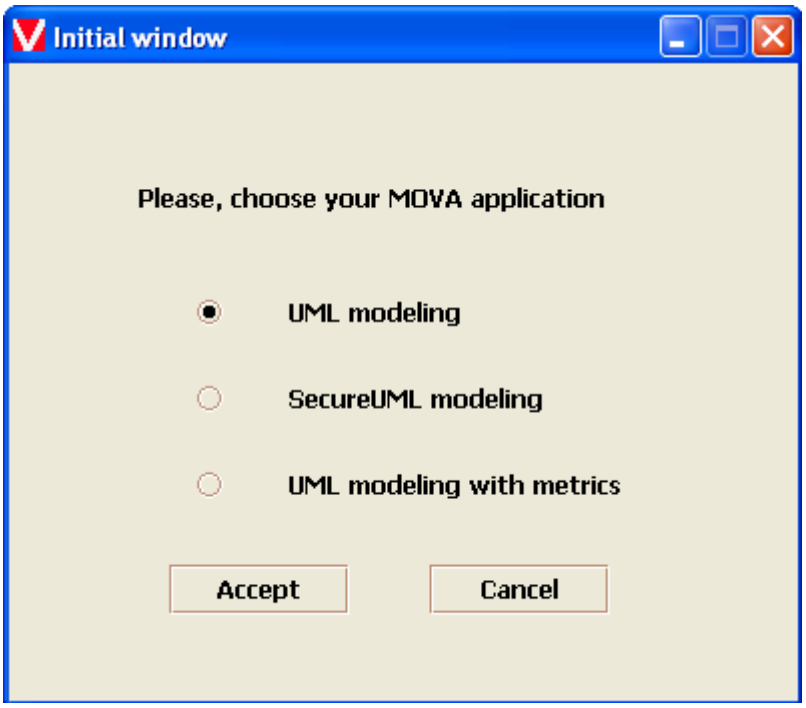


Figura 1

Si pinchamos sobre SecureUML modeling, obtenemos la siguiente pantalla (ver Figura 2), que permite al usuario:

- crear diagramas de seguridad
- definir operaciones OCL para ser usadas en políticas de seguridad, y
- evaluar dichas operaciones.

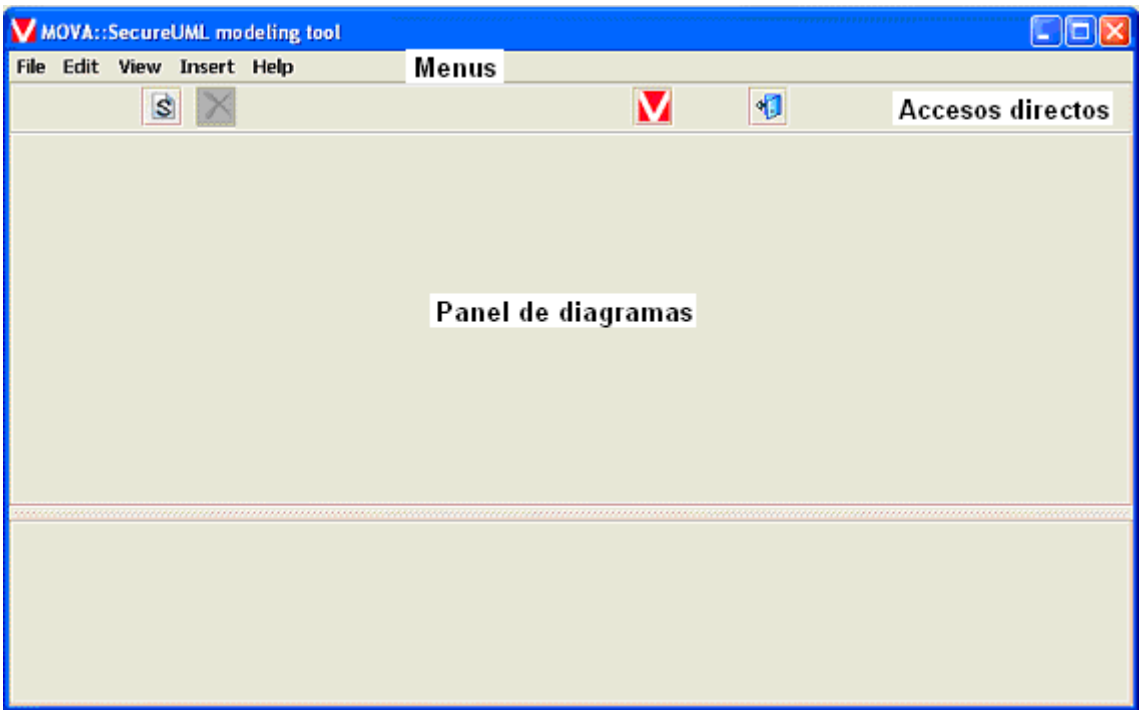


Figura 2

Se distinguen 3 bloques de elementos en la pantalla:

1- Menús de la herramienta.

2- Barra de accesos directos.

3- Panel de diagramas.

Menús de la herramienta:

- File:
  - New:
    - Security Diagram: Crea un diagrama de seguridad vacío.
  - Open:
    - Security Diagram: Abre un diagrama de seguridad.
  - Close: Cierra el proyecto en curso.
  - Page Setup: Permite configurar la impresión.
  - Print: Imprime el área imprimible del diagrama actual.
  - Export to EPS: Exporta el diagrama actual a un fichero EPS.
  - Save as:

        - Security Diagram: Almacena un diagrama de seguridad en un fichero xml.
    - Exit: Sale de la herramienta.
- Edit:
    - Zoom +: Aumenta el tamaño del diagrama.
    - Zoom -: Reduce el tamaño del diagrama.
- Insert:
    - Security Diagram:
        - Role: Inserta un rol en el diagrama.
        - User: Inserta un usuario en el diagrama.
        - Entity: Inserta una entidad en el diagrama.
        - Permission: Inserta un permiso en el diagrama.
        - Relation role-role: Inserta una relación entre roles.
        - Relation user-role: Inserta una relación entre un usuario y un rol.
        - Relation entity-entity: Inserta una relación entre entidades.
        - Relation role-permission-entity: Inserta una relación entre un rol, un permiso y una entidad.
- Help:
    - About us: Muestra una ventana (ver Figura 3) con información de la herramienta.

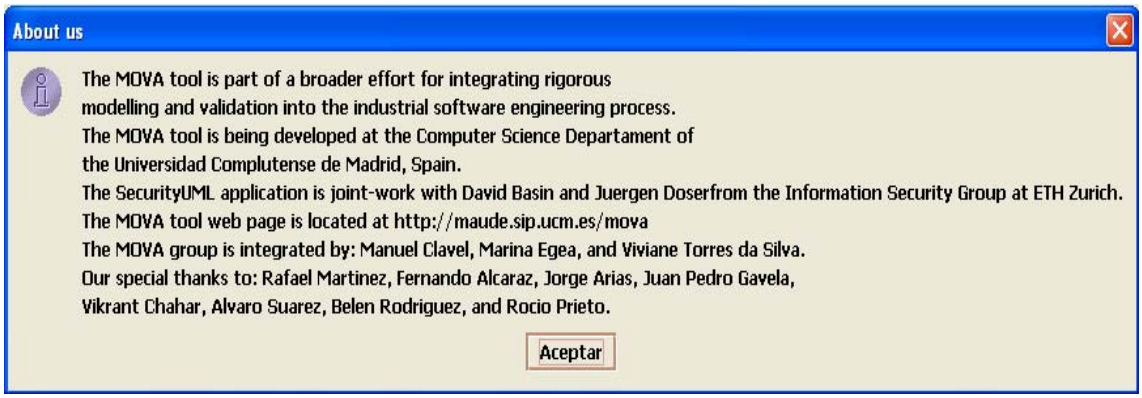


Figura 3

Accesos directos:

Figura 4

 Crea un diagrama de seguridad.

 Elimina el diagrama actual.

 Carga un diagrama de seguridad.

 Guarda el diagrama actual.

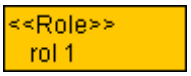
 Muestra la ayuda.

 Sale de la herramienta.

Panel de diagramas:

 Modo cursor. Permite seleccionar elementos y editarlos.

- Seleccionar elementos:
  - Selección individual: Pinchando sobre el elemento deseado con el botón izquierdo del ratón.



  - Selección múltiple: Pinchando con el botón izquierdo del ratón, fuera de cualquier elemento, y arrastrándolo hasta seleccionar todos los elementos deseados (ver Figuras 5 y 6).

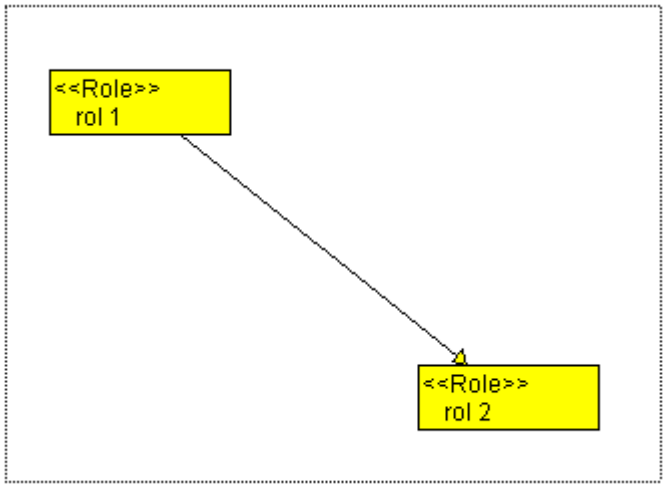


Figura 5. Mientras se arrastra

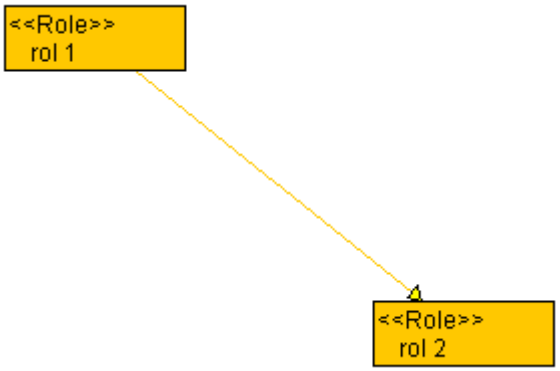


Figura 6. Tras soltar el ratón

- Editar elementos: Haciendo doble clic sobre el elemento.
  - Editar rol:

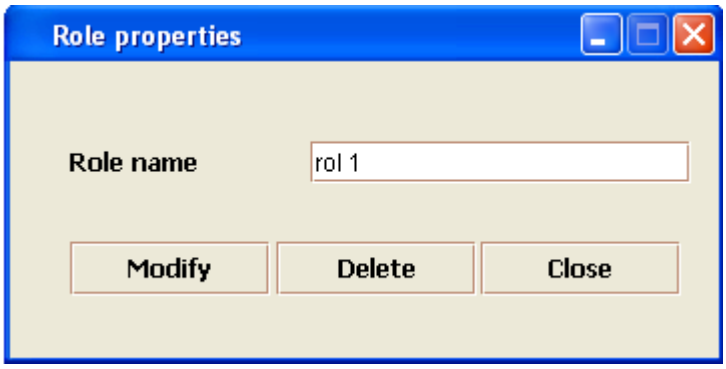


Figura 7

- Editar usuario:

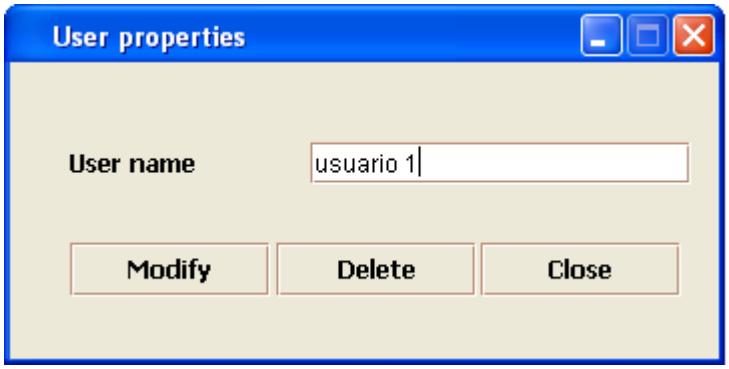


Figura 8

- Editar entidad:

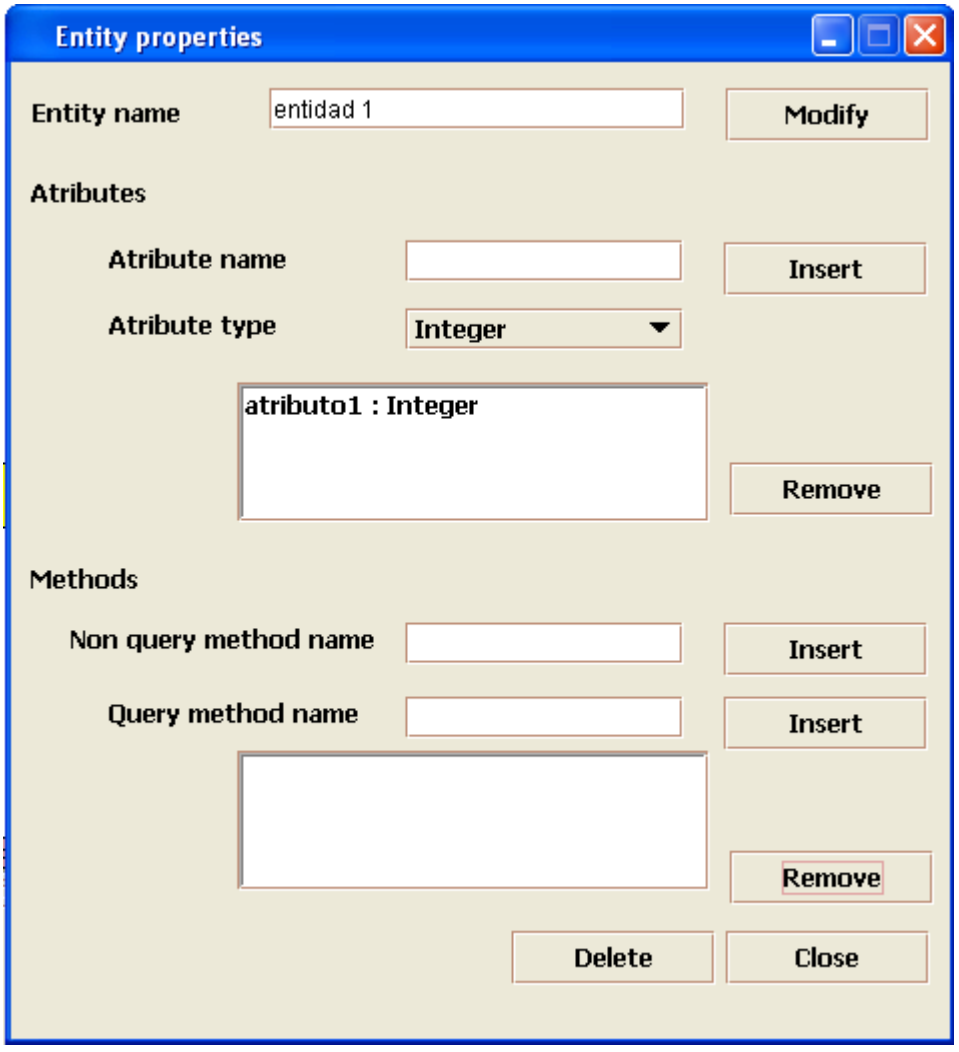


Figura 9

- Editar permiso:

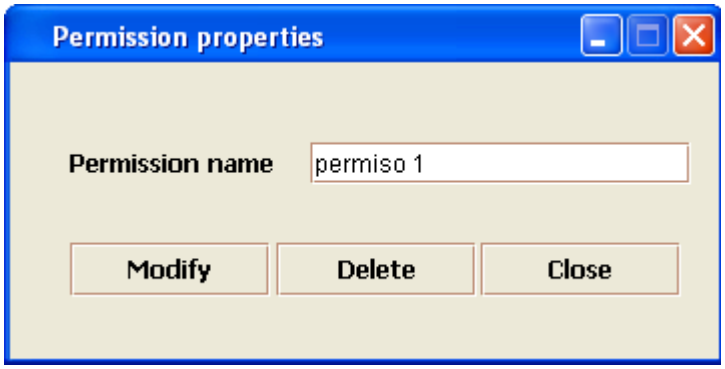


Figura 10

- Editar relación entre roles:

Figura 11

- Editar relación entre entidades:

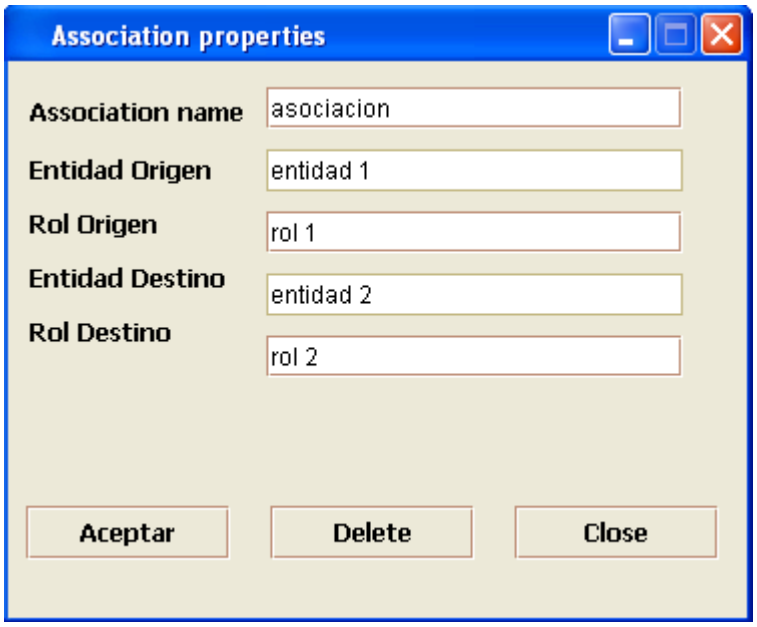


Figura 12

- Editar relación entre rol, permiso y entidad:

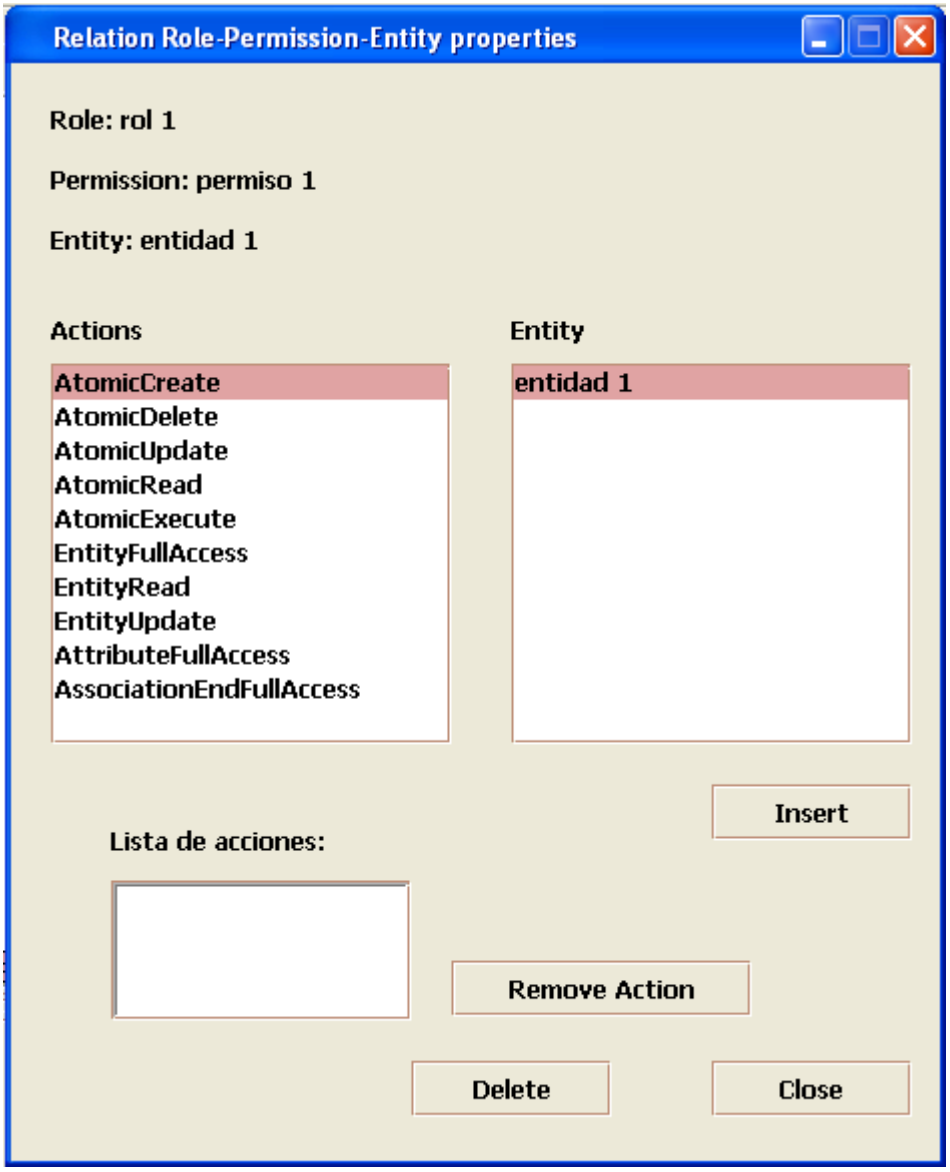
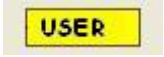


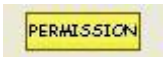
Figura 13

Crea roles en el diagrama de seguridad.



Crea usuarios en el diagrama de seguridad.

Crea entidades en el diagrama de seguridad.



Crea permisos en el diagrama de seguridad.

 Crea relaciones entre roles.

 Crea relaciones entre roles y usuarios.

 Crea relaciones entre entidades.

 Crea relaciones entre roles, entidades y permisos.

 Aumenta el tamaño del diagrama.

 Reduce el tamaño del diagrama.

 Muestra la ventana de opciones de visualización (ver Figura 14) que permite seleccionar qué características del diagrama queremos que sean visibles, y cuáles no.

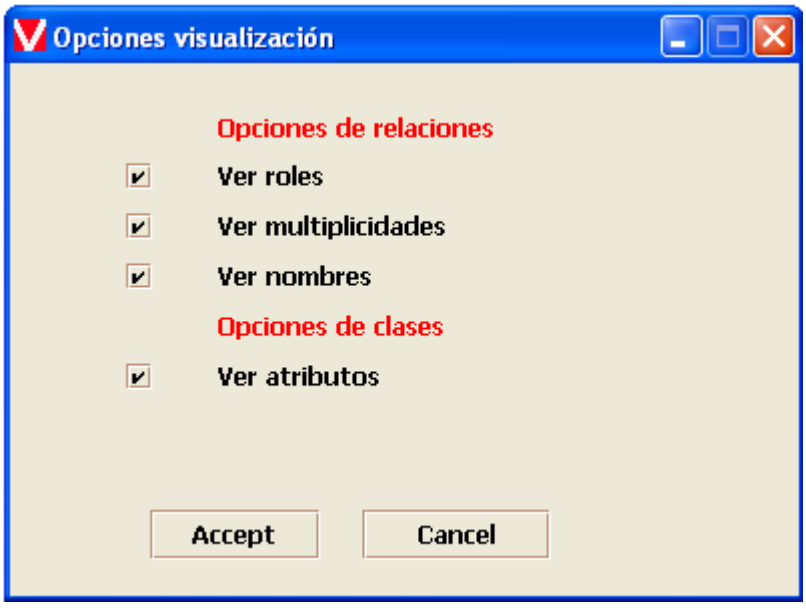


Figura 14

 Muestra una ventana (ver Figura 15) para insertar una restricción en la relación entre un rol, permiso y entidad.

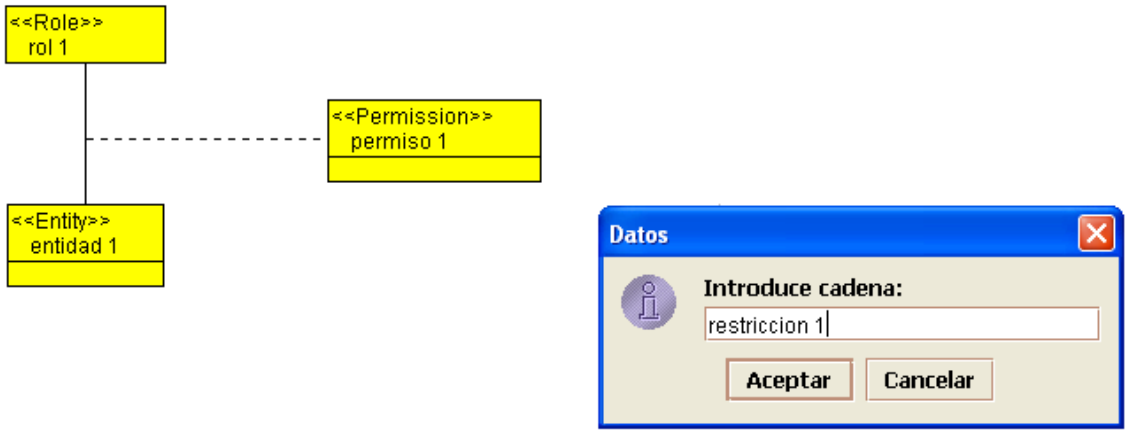


Figura 15

El resultado obtenido al insertar la restricción se muestra en la Figura 16.

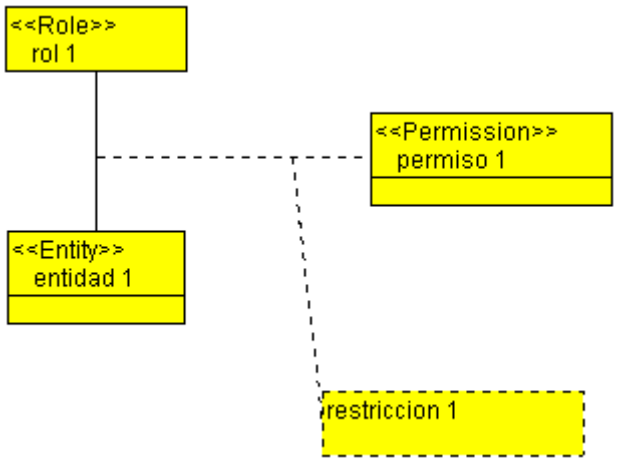


Figura 16

3. MOVA TOOL

3.1. Arquitectura del sistema

En el diseño de la arquitectura se ha intentado separar al máximo la parte gráfica de la parte del modelo. Para todo elemento que tiene una representación grafica se trabaja a un nivel lógico, actuando sobre las estructuras de datos, y una parte gráfica que ofrece la representación visual de esas estructuras de datos. Esta separación permite poder variar un nivel sin afectar al siguiente lo cual es muy útil a la hora de realizar modificaciones o mejoras en una de las partes.

Otra de las decisiones tomadas a la hora de diseñar la arquitectura ha sido seguir una estructura jerárquica basada en fachadas. Una fachada es un punto centralizado de acceso a una parte del sistema que incluye dentro de sí distintos componentes y funcionalidades a las que puede acceder el usuario, pero no de cualquier forma, sino exclusivamente a través de la fachada. La fachada maneja por tanto los distintos componentes y realiza sobre ellos las acciones que desea el usuario. Estos componentes pueden ser a su vez fachadas internas o elementos máximos de una jerarquía. Al iniciar MOVA Tool se nos da la posibilidad de acceder a tres herramientas. Cada una de ellas abrirá una ventana que actúa como fachada conteniendo los menús de la aplicación y los conjuntos de diagramas.

Con estas ideas de jerarquización y separación se ha desarrollado el código de la aplicación intentando facilitar el trabajo de ampliación o modificación que se puede realizar sobre él en un futuro.

El sistema está estructurado en los siguientes paquetes:

- Dibujables
- Elementos
- Interfaz
- Eps
- Conexión
- Persistence

- StateMova

Pasamos a explicar de forma detallada cada paquete:

- ***Paquete Dibujables***. El paquete Dibujables contienen las clases necesarias para los elementos visualizables en los distintos diagramas: clases, objetos, generalizaciones, asociaciones, usuarios, roles, etc. Como posteriormente se indica en el apartado Reingeniería, se ha realizado un rediseño en la arquitectura de este paquete aunque en la versión disponible del sistema no han sido aplicados aún estos cambios.

  El diagrama de clases (esquematizado) del paquete se muestra en la Figura 17.

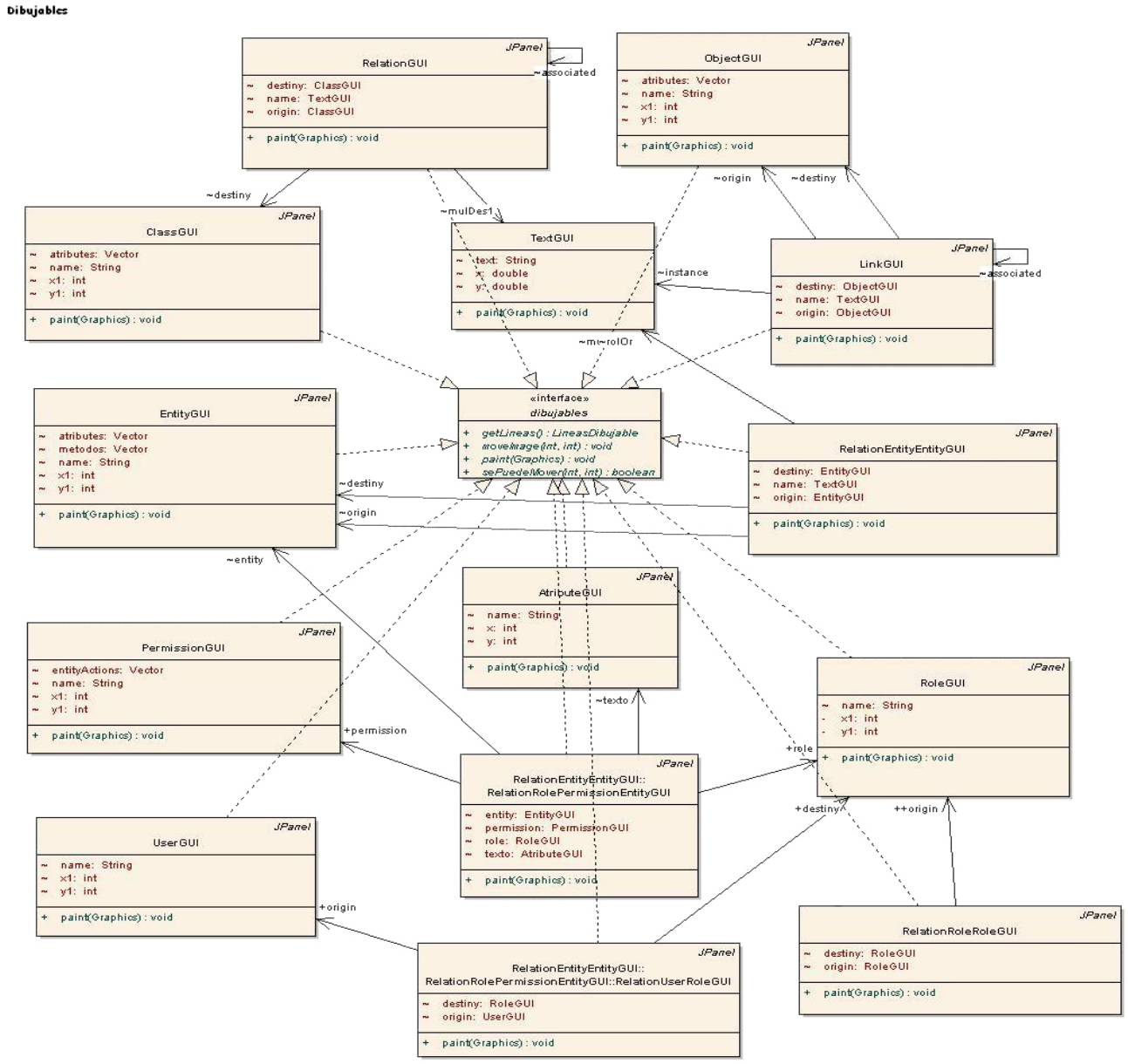


Figura 17

Las clases que contiene este paquete son:

- *Dibujables*: Es el interfaz que debe implementar todo elemento GUI que se dibuje a sí mismo. Contiene los métodos necesarios para poder pintar y mover un elemento.

- *AtributeGUI*: Representa la restricción en la relación entre un rol, un permiso y una entidad. Entre sus atributos más importantes se encuentran el nombre del atributo, y las coordenadas de posición en el tapiz.

- *ClassGUI*: Representa las clases de un diagrama de clases. Entre sus atributos más importantes se encuentran el nombre de la clase, las coordenadas de posición en el tapiz, un valor que indica si es enumerada o no, y un vector de atributos para representar los atributos de clase.

- *RelationGUI*: Representa las relaciones entre clases de un diagrama de clases. Entre sus atributos más importantes destacan el nombre de la relación, los roles y multiplicidades, las clases origen y destino, y el vector de segmentos. Tanto para el pintado como para el envío a Maude se debe diferenciar si trabajamos con una asociación clase-clase, asociación clase-relación o generalización.

- *ObjectGUI*: Representa los objetos de un diagrama de objetos. Entre sus atributos más importantes se encuentran el nombre del objeto, el nombre de la clase instanciada, las coordenadas de posición en el tapiz y un vector con atributos. A diferencia de las clases, en los objetos los atributos del vector tienen siempre un valor.

- *LinkGUI*: Representa los enlaces entre objetos de un diagrama de objetos. Sus atributos principales son el nombre, los roles, los objetos origen y destino y el vector de segmentos.

- *RoleGUI*: Representa los roles de un diagrama de seguridad. Entre sus atributos más importantes se encuentra el nombre del rol y las coordenadas de posición en el tapiz.

- *UserGUI*: Representa un tipo de usuario de un diagrama de seguridad. Entre sus atributos más importantes se encuentra el nombre del usuario y las coordenadas de posición en el tapiz.

- *EntityGUI*: Representa una entidad de un diagrama de seguridad. Entre sus atributos más importantes se encuentra el nombre de la entidad, las coordenadas de posición en el tapiz y los vectores de atributos y métodos que definen a la entidad.

- *PermissionGUI*: Representa un permiso de un diagrama de seguridad. Los permisos nos dicen las acciones que un tipo de rol puede realizar sobre una entidad. Entre sus atributos más importantes se encuentra el nombre del permiso, las coordenadas de posición en el tapiz y el vector de acciones que pueden realizarse sobre la entidad con la que está relacionada.

- *RelationEntityEntityGUI*: Representa los enlaces entre entidades de un diagrama de seguridad. Sus atributos principales son el nombre, los roles, las entidades origen y destino y el vector de segmentos.

- *RelationRolePermissionEntityGUI*: Representa a las relaciones para asignar, a un determinado rol, un permiso para realizar una acción sobre una entidad. Sus atributos principales son el nombre de la relación, el rol, el permiso y la entidad correspondiente.

- *RelationRoleRoleGUI*: Representa la relación que permite especializar un rol a partir de otro, de modo que el primero hereda las capacidades del segundo y además puede incorporar otras capacidades. Sus atributos son los roles padre e hijo.

- *RelacionUserRoleGUI*: Representa la asignación de un rol a un usuario de modo que este usuario puede realizar las acciones asignadas al rol. Los atributos son el usuario y el rol.

- *TextGUI*: Representa texto dibujable utilizado en relaciones y enlaces para almacenar el nombre, roles y multiplicidades. Son cadenas de texto con coordenadas para actualizarse, que responden a los movimientos de las relaciones. Tener esta clase es más sencillo que tener cadenas de texto corriente en las relaciones o enlaces, lo que provocaría problemas a la hora de actualizar posiciones.

Todos los elementos dibujables contienen los métodos de pintado y de envío a Maude del comando correspondiente.

- ***Paquete Elementos****:* En el paquete Elementos se han incluido algunas clases necesarias para representar las estructuras de datos para los elementos dibujables, como pueden ser atributos para clases, objetos, etc. En un rediseño propuesto para futuras versiones las clases que forman este paquete estarían incluidas en el paquete dibujables. Las clases que incluye este paquete son:

- *Atribute*: Representa los atributos de una clase, objeto, entidad, permisos, etc. Vienen definidos por tres atributos básicos: nombre, tipo y valor, y un atributo que indica si proceden de clase enumerada o no. Cuando trabajamos con clases o entidades, no es necesario que los atributos tengan valor, pero para los objetos es obligatorio.

- *Method*: Representa los métodos de una entidad. Vienen definidos por dos atributos: nombre y un atributo que indica si devuelve un tipo o no. De momento no se permiten definir los tipos que devolverá un método pero se hará en versiones posteriores.

- *Linea*: Se trata de un segmento representado mediante los puntos origen y destino. Es útil para definir el conjunto de líneas que forman el borde de un dibujable.

- *LineasDibujable*: Representa el conjunto de líneas (segmentos) que forman los bordes de un dibujable. Es útil para, en la selección múltiple de elementos, saber cual cae dentro del cuadrado de selección. Para ello se utiliza el algoritmo de Cohen-Sutherland, que nos permite saber si un dibujable o parte de éste cae dentro de un cuadrado.

- *Par*: Es una clase con dos atributos: un número entero y un nombre.

- ***Paquete Interfaz***. En el paquete Interfaz están las clases que implementan los elementos gráficos con los que se puede interactuar en la herramienta: diagramas, ventanas, menús, etc. La arquitectura actual de este paquete es el que se muestra a continuación, aunque, como se explica en el apartado Reingeniería, se han presentado rediseños con la idea de mejorar el modelo MVC.

  El diagrama de clases completo (ver Figura18) lo desglosamos en sus 4 partes principales: principal, diagramas de clases, diagramas de objetos y diagramas de seguridad.

  Vemos como principal es la fachada del sistema, incluyendo dentro de sí diagramas de clases, objetos y seguridad a la vez que funciones generales sobre ellos.

Paquete interfaz:

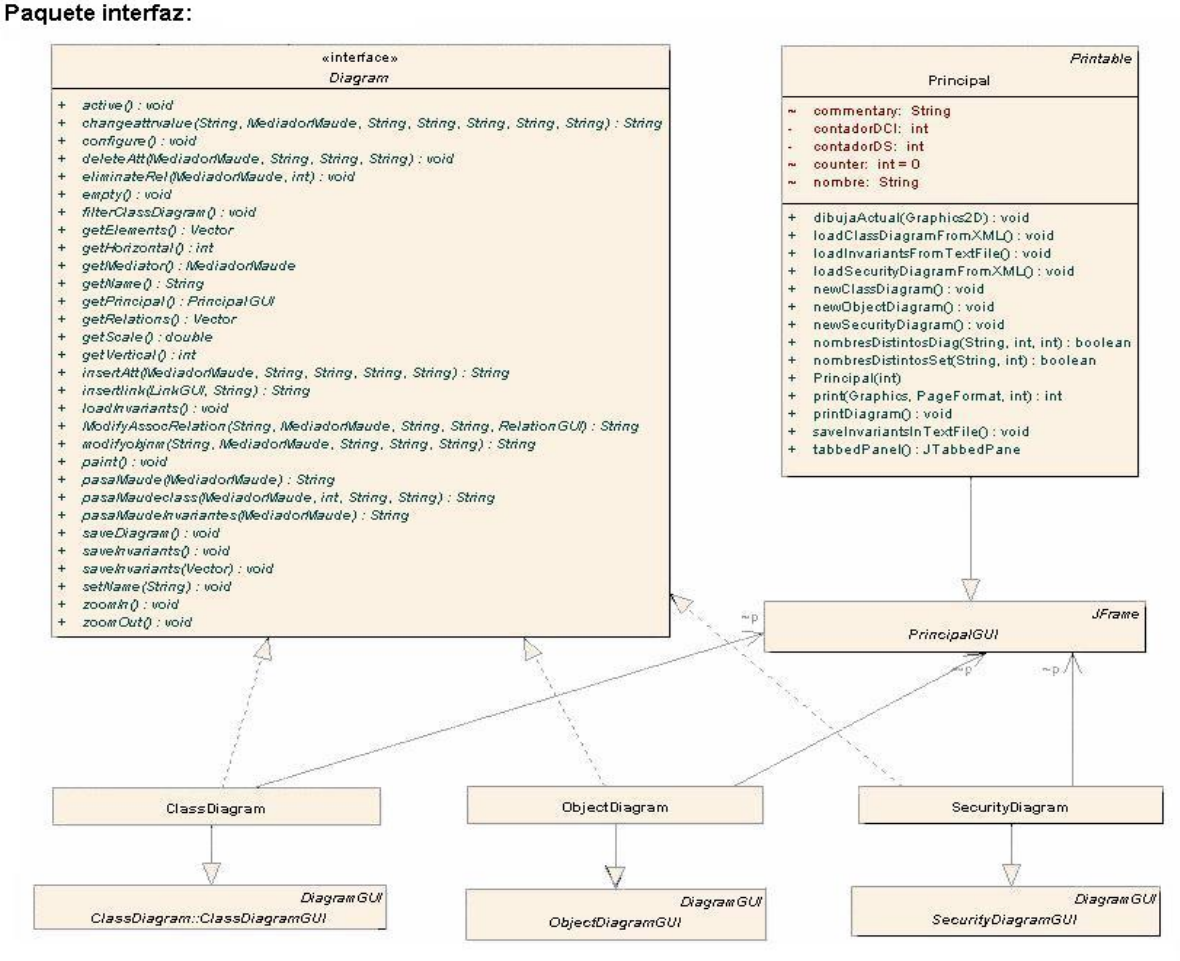


Figura 18

Los diagramas de clases, objetos y de seguridad, a su vez se descomponen en elementos (clases, objetos, roles, entidades, permisos y usuarios) y relaciones (asociaciones, enlaces,…) de manera jerarquizada. Estos elementos pueden hacer uso de elementos de un nivel jerárquico inferior como atributos o textos flotantes.

Cada diagrama tiene su correspondiente interfaz gráfico que delega las operaciones a la parte lógica. Esta parte lógica puede delegar en funciones sobre los elementos que posee para no sobrecargarse de funcionalidad. Los diagramas necesitan algunos elementos visuales secundarios, necesarios para la ejecución de la herramienta, como menús derechos, ventanas de edición o editores de invariantes (diagrama de objetos) o consultas (diagrama de objetos y de seguridad).

Las páginas siguientes muestran respectivamente el diagrama de clases referido a los diagramas de clases de la aplicación (ver Figura 19), diagramas de objetos de la aplicación (ver Figura 20) y diagramas de seguridad de la aplicación (ver Figura 21).

ClassDiagram:

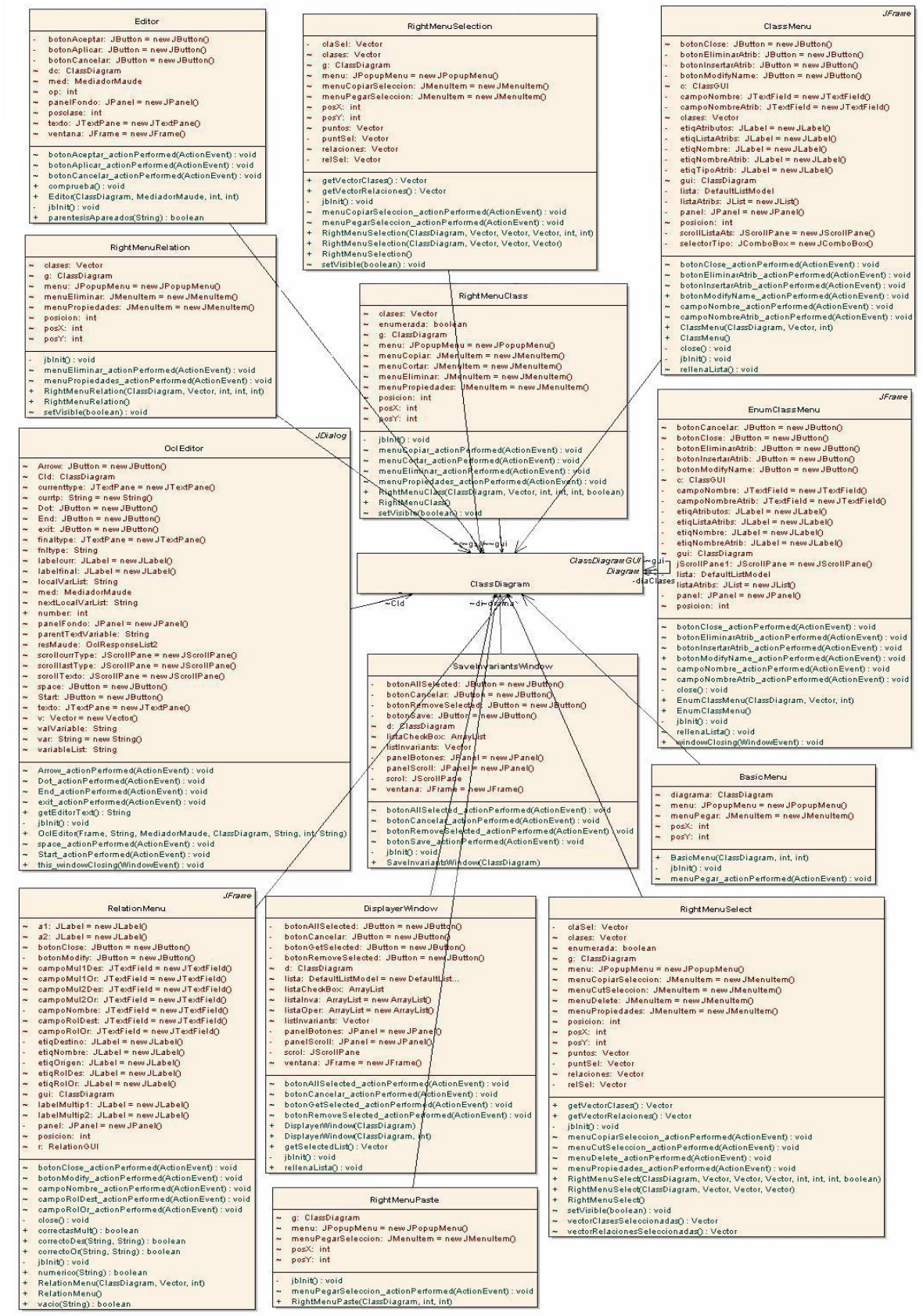


Figura 19

ObjectDiagram:

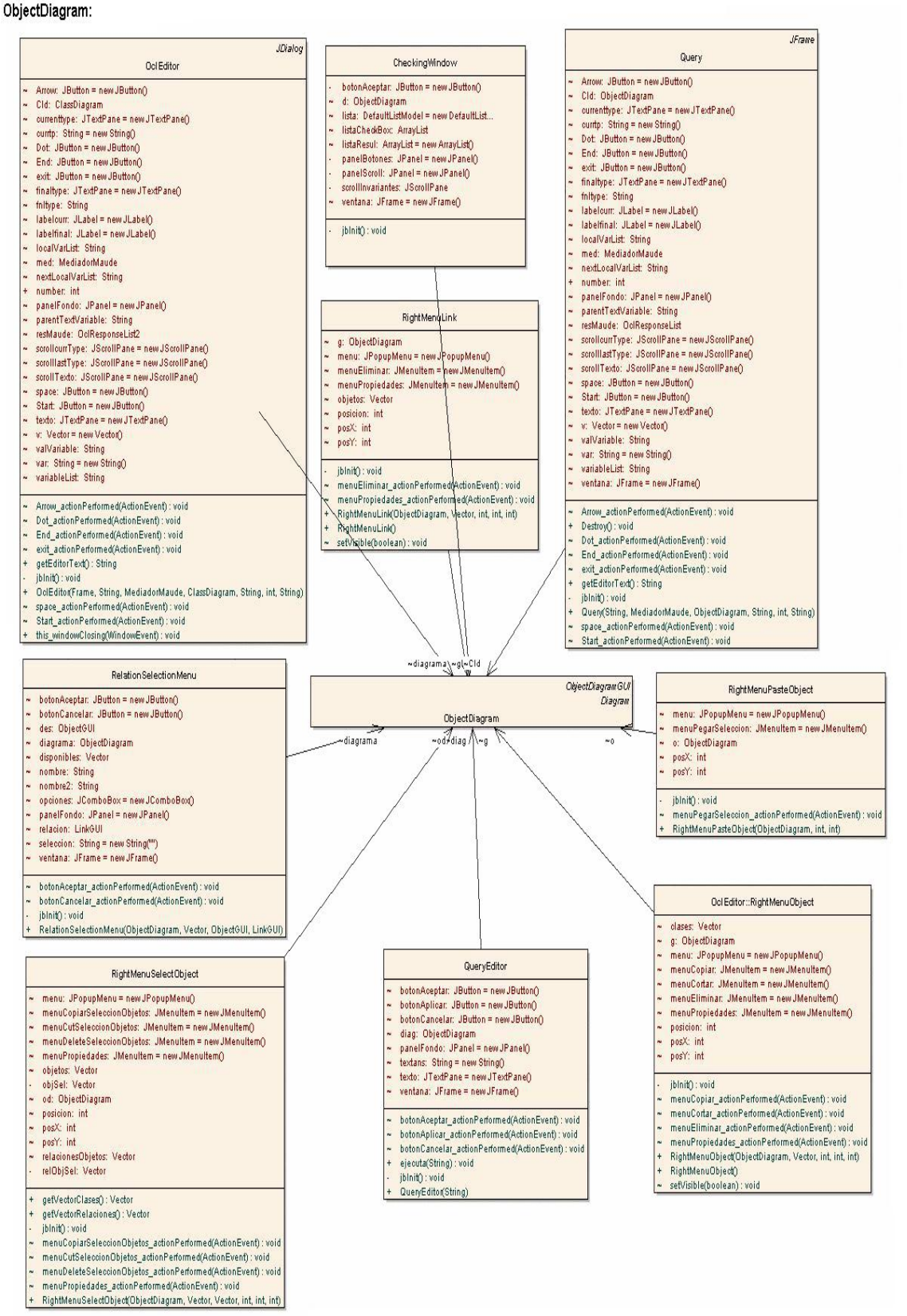


Figura 20

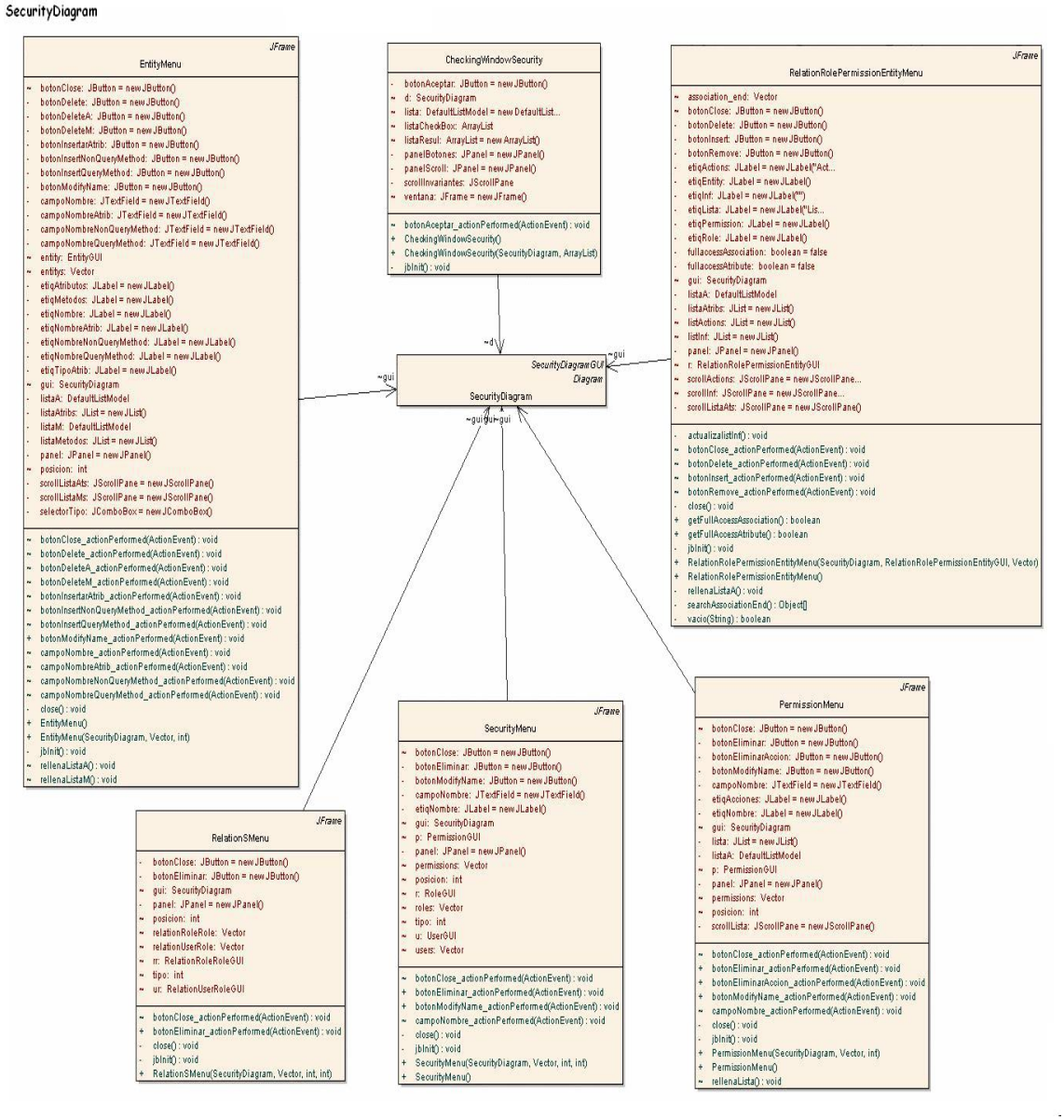


Figura 21

A continuación se explican las clases principales del paquete interfaz:

- *InitialWindow*: Esta ventana se abre al cargar la aplicación y nos permite elegir entre una de las tres herramientas posibles.

- *ClassViewOptions, ObjectViewOptions, SecurityViewOptions*: estas tres ventanas nos permiten definir las opciones visuales para los diagramas de clases, objetos y seguridad respectivamente.

- *RightMenuClass*, *RightMenuRelation*, *RightMenuObject*, *RightMenuLink*, *BasicMenu*: Son menús desplegables emergentes asociados a los elementos de los diagramas: clases, relaciones, objetos y enlaces. Aparecen al presionar el botón derecho sobre el elemento deseado. Las opciones mostradas son cortar, copiar, editar y borrar. El menú básico es desplegado sobre una zona vacía para pegar un elemento previamente cortado o pegado.

- *Theme*: Presenta una configuración visual propia de MOVA Tool basada en juegos de colores y fuentes.

- *Editor*, *DisplayerWindow*: *Editor* despliega un editor de invariantes. Los invariantes se editan y almacenan en el diagrama de clases asociado si son correctos. *DisplayerWindow* muestra todos los invariantes almacenados en el diagrama.

- *Main*: Archivo principal para lanzar MOVA Tool.

- *QueryEditor*: Despliega un editor de consultas sobre un diagrama de objetos. Permite escribirlas, ejecutarlas y observar los resultados.

- *CheckingWindow*: Ventana que muestra los resultados obtenidos de comprobar sobre un diagrama de objetos los invariantes almacenados en el diagrama de clases asociado.

- *ClassMenu*, *EnumClassMenu*, *RelationMenu*, *ObjectMenu*, *LinkMenu, EntityMenu, PermissionMenu, RelationRolePermissionEntityMenu*: Despliegan ventanas de edición referidas al elemento deseado. Dichas ventanas permiten cambiar el valor de ciertos atributos como nombre, rol, multiplicidad… Para clases se pueden añadir atributos de clase; para objetos se puede dar valores a los

atributos; a las entidades podemos añadirle atributos y métodos; y para permisos podemos quitar acciones. Las acciones se añaden en el menú de la relación role-permission-entity.

- *ClassSelectionMenu*, *RelationSelectionMenu*: Muestran menús desplegables con las posibles clases o relaciones que se pueden instanciar en un diagrama de objetos cuando se desea crear un elemento.

- *PropertiesMenu*: Muestra una ventana de configuración que permite cambiar el nombre del diagrama en uso.

- *Principal*, *PrincipalGUI*: Principal contiene todas las acciones posibles a realizar sobre la ventana principal de trabajo, esto es, acciones sobre los menús y accesos directos y manejo de las pestañas de los diagramas. Sus principales funciones son carga y almacenamiento de diagramas en ficheros, crear y trabajar con diagramas, cargar y almacenar diagramas en formato XML y EPS, e imprimir diagramas. *PrincipalGUI* contiene todos los elementos de interfaz gráfica necesarios para llevar a cabo las funciones de *Principal*. Cabe destacar el almacenamiento de los diagramas individuales dentro del marco principal. Para ello, se utilizan dos niveles gráficos de pestañas. El nivel exterior representa al conjunto del diagrama, contiene tantas pestañas como diagramas tiene el conjunto. Las pestañas son instancias de la clase MiTabbedPane con diagramas como contenido. Como los diagramas son a su vez componentes gráficas es sencillo integrarlas en el marco principal.

- MiTabbedPane: Esta clase extiende la clase Java TabbedPane. De esta forma tiene las propiedades de un TabbedPane y además se le han añadido eventos para poder reconocer cuando se realizan ciertas acciones sobre ellas y poder actualizar cierta información. Por ejemplo, se ha utilizado para disponer de un menú dinámico, de forma que cambie dependiendo de en que tipo de diagrama estamos trabajando.

- *Diagram:* Indica las operaciones obligatorias que deben implementar los diagramas de clases y objetos.

- *DiagramaGUI*: Es una generalización de ClassDiagramGUI, ObjectDiagramGUI, SecurityDiagramGUI que incluye los atributos y métodos comunes a la parte gráficas de los tres tipos de diagramas. Entre estos métodos están el zoom, la actualización del tamaño del panel, la selección múltiple, el pintado del área imprimible y el pintado de elementos. Para este último caso se ha utilizado un patrón Template Method que permite que las clases que especializan a esta clase pinten sus propios elementos.

- *ClassDiagram*, *ClassDiagramGUI*, *ObjectDiagram*, *ObjectDiagramGUI, SecurityDiagram*, *SecurityDiagramGUI*: Representan cada tipo de diagrama (clases, objetos y seguridad) junto con sus representaciones gráficas (GUI). Los diagramas permiten crear elementos del tipo correspondiente y relaciones entre ellos. Dichos elementos se pueden editar posteriormente por el usuario. Para los diagramas de clase existen además invariantes, del mismo modo que para los de objetos existen consultas. La estructura interna de los diagramas es la misma en los dos casos. Las estructuras de datos son vectores, teniendo uno para elementos y otro para relaciones. Se accede a los vectores usando un índice que se corresponde con el elemento que selecciona el usuario. Para poder calcularlo, se han implementado los métodos de evento de ratón correspondientes a clickear, presionar, arrastrar y soltar. Cada una de las clases GUI incluye un método que recorre los vectores de elementos y relaciones y llama al método de pintado de cada uno, como se explicó en los elementos del paquete dibujables. Este método es llamado desde DiagramGUI cuando es necesario. Los diagramas implementan además operaciones de inserción, eliminación, carga y almacenamiento en distintos formatos, envío a Maude, etc.

- *Load*, *Delete*, *LoadMenu*: Respectivamente sirven para cargar diagramas, borrar diagramas y cargar conjuntos.

- ***Paquete Eps***. El paquete Eps tiene el cometido específico de guardar cualquier tipo de diagrama en un archivo Eps (Encapsulated Post Script). Para ello se ha utilizado una librería de libre distribución denominada Jibble (www.jibble.org).

  La mayor ventaja que proporciona esta librería es que permite tratar los documentos EPS como elementos gráficos de Java, es decir elementos de tipo Graphics2d. Como todos los elementos gráficos de la herramienta (clases, objetos, relaciones, enlaces,…) se dibujan sobre el diagrama usando métodos Graphics2d la conversión de diagrama a formato EPS es inmediata al compartir el algoritmo de pintado.

  La librería Jibble fue adaptada en ciertos puntos para ser más eficaz en la herramienta, por lo que sus archivos pasaron de estar en la librería a ser archivos de la herramienta.

- *EpsDocument*: Representa el archivo del diagrama. Implementa las funciones usuales de un archivo: abrir, cerrar, vaciar, concatenar, limpiar…

- *EpsGraphics2D*: Es el archivo básico para guardar en formato EPS. Representa los dibujos que pueden ser almacenados en un documento, por lo que sobrescribe todos los métodos de la clase Graphics2d. Este elemento se pasa a los métodos de pintado de la herramienta para transcribir los diagramas sobre él.

- *EpsException*: Excepción propia de conversión a EPS.

El diagrama de clases se muestra en la Figura 22:

Paquete EPS

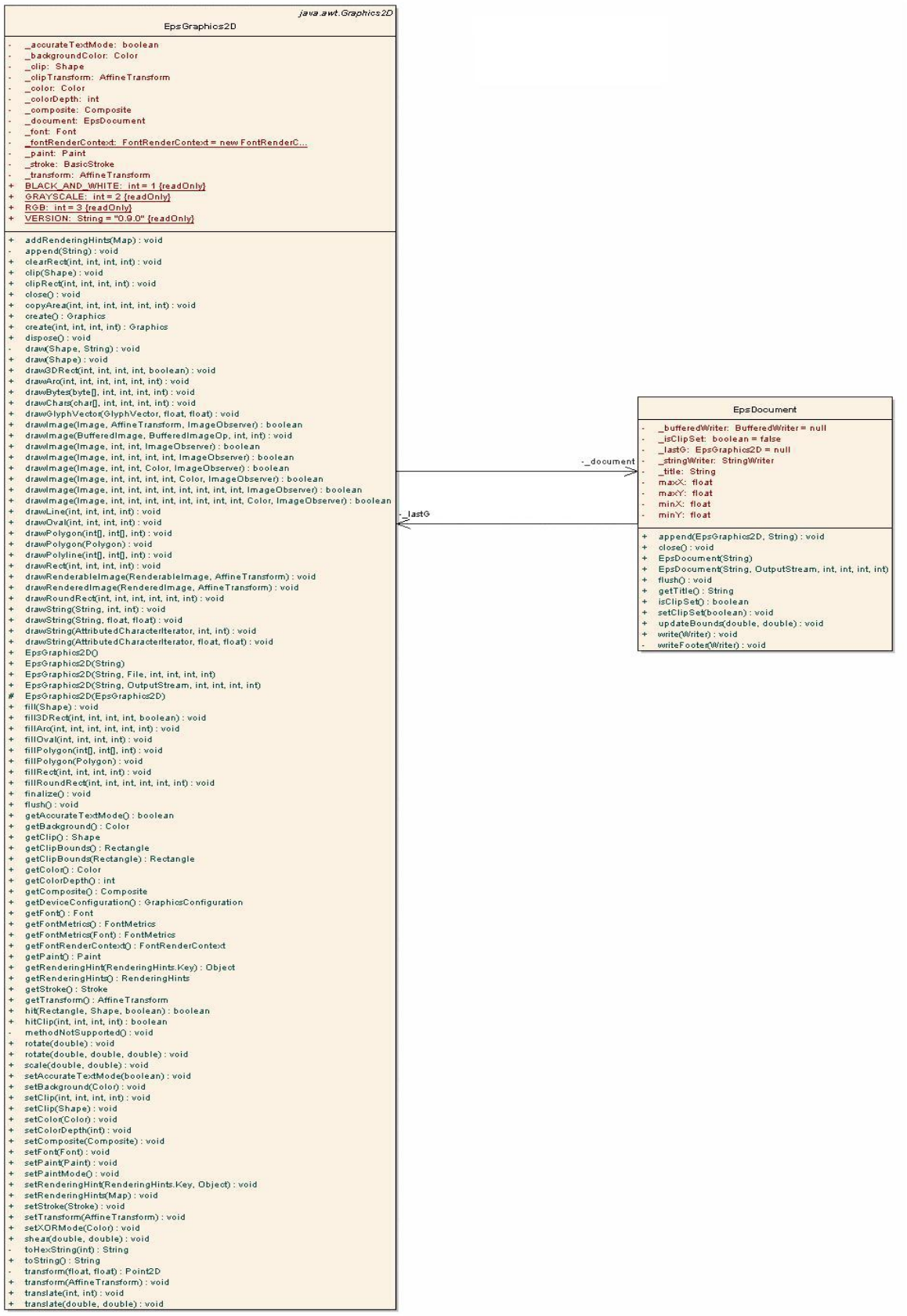


Figura 22

- ***Paquete Conexión***. El paquete Conexión realiza la comunicación entre MOVA Tool e ITP/OCL a través del intérprete Maude. Dicha comunicación se realiza por medio de comandos que manda la herramienta a un conector específico. El conector ejecuta este comando en Maude, y devuelve a la herramienta una respuesta para ser procesada. Si la respuesta es positiva, se actualizan los diagramas, y si es negativa se avisa al usuario del error.

  El paso de comandos se hace ejecutando cadenas de texto a través del terminal utilizando el comando *bash*. Éste es el método más sencillo de ejecución en los sistemas Linux aunque no el más eficiente. Tanto los diagramas como los elementos contenidos en ellos son capaces de generar los comandos de su creación utilizando la sintaxis de comandos ITP/OCL junto con datos que se consideren oportunos. Para mandar un diagrama completo a Maude, basta con mandar el comando de creación del diagrama y a continuación recorrer las estructuras de datos de elementos mandando el comando de cada elemento a Maude. En cualquier momento que un elemento sea modificado se manda el comando de dicho elemento para actualizar ITP/OCL.

  Los comandos enviados a Maude se registran en un archivo denominado *log.txt* situado en el directorio raíz de modo que el usuario pueda consultarlo si lo desea.

  El diagrama de clases del paquete se muestra en la Figura 23:

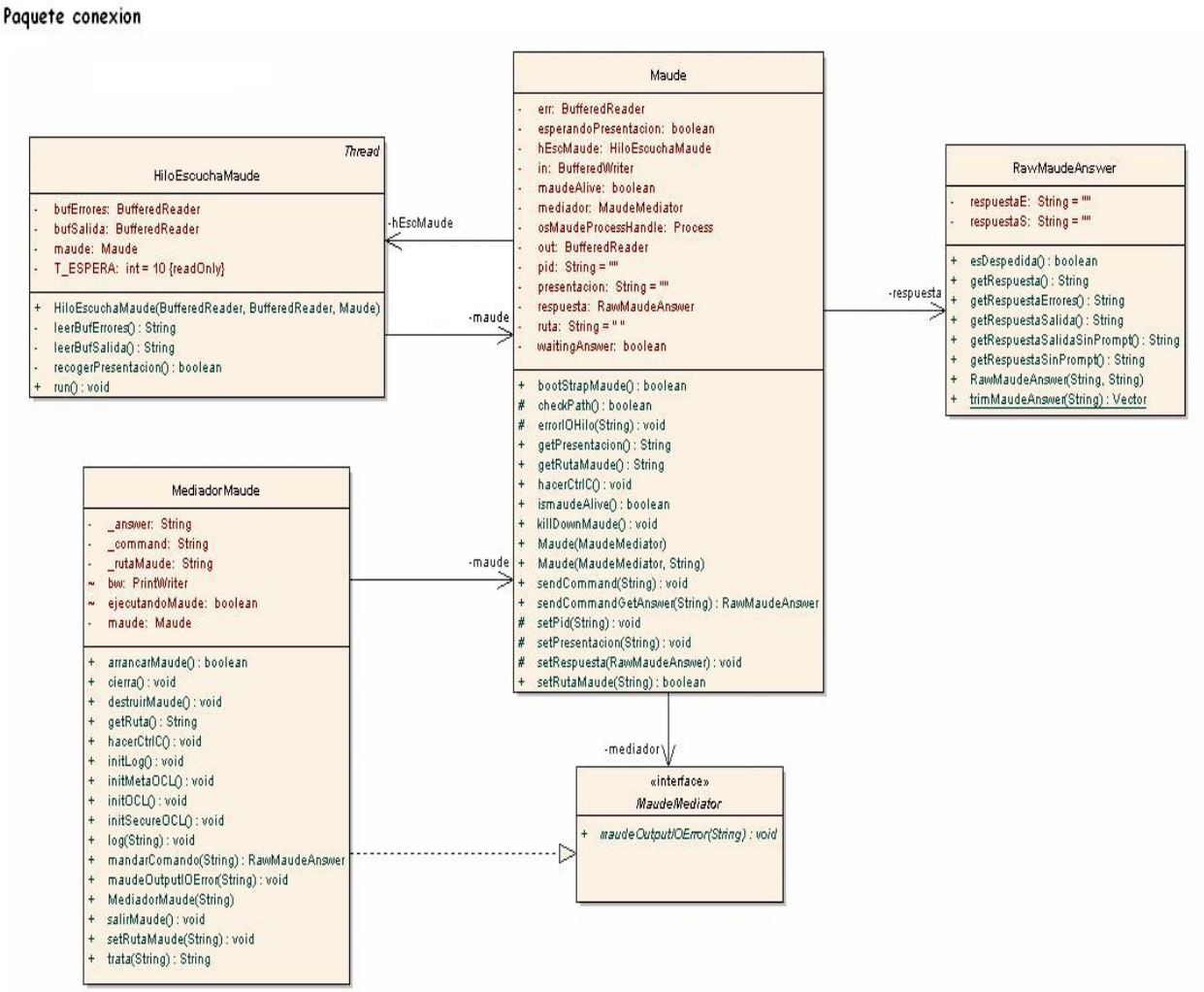


Figura 23

A continuación se explican las clases que componen el paquete:

- *Maude*: Interactúa con el intérprete Maude realizando las mismas acciones que un usuario puede ejecutar interactuando con Maude a través del terminal. Entre ellas destacan arrancar ITP/OCL y ejecutar sus comandos.

- *HiloEscuchaMaude*: Utilizado por Maude para conocer el estado del mismo y capaz de actuar cuando sucede un evento.

- *MediatorMaude*: Clase a extender con las acciones imprescindibles que debe implementar el mediador que enlace MOVA Tool con Maude.

- *Mediator*: Representa al elemento real que hará la mediación entre MOVA Tool y Maude. Este elemento debe aparecer obligatoriamente en todas las clases de la herramienta que necesiten interactuar con Maude. Está encargado de realizar todas las acciones: arrancar Maude, arrancar ITP/OCL, ejecutar comandos, devolver respuestas… Para ello usa los elementos citados previamente.

- *RawMaudeAnswer*: Cadena que se obtiene como respuesta al envío de un comando a ITP/OCL utilizando Maude. Estas cadenas deben verificarse para ver si indican éxito (Ok), error (Failed) o error de Maude (cadena vacía). En función de la respuesta la herramienta realizará una función determinada.

3.2. Reingeniería

Esta herramienta es continuación de una versión previa. Es por ello que ha sido necesario entender bien la arquitectura del sistema para poder ampliarla y mejorarla. A continuación se indican las mejoras realizadas:

- **Paquete interfaz**

Se muestran a continuación los diagrama de clases (ver Figuras 24, 25 y 26) que describían la arquitectura principal del sistema del año pasado.

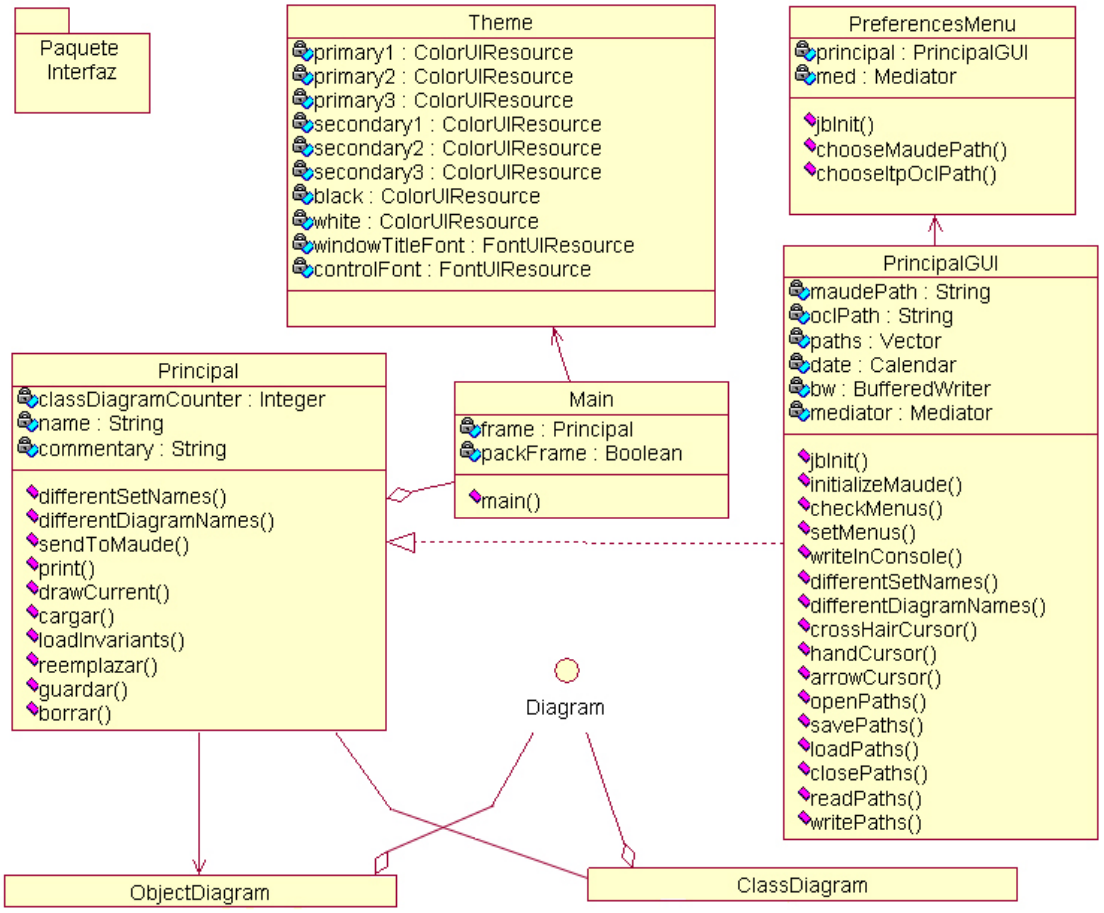


Figura 24

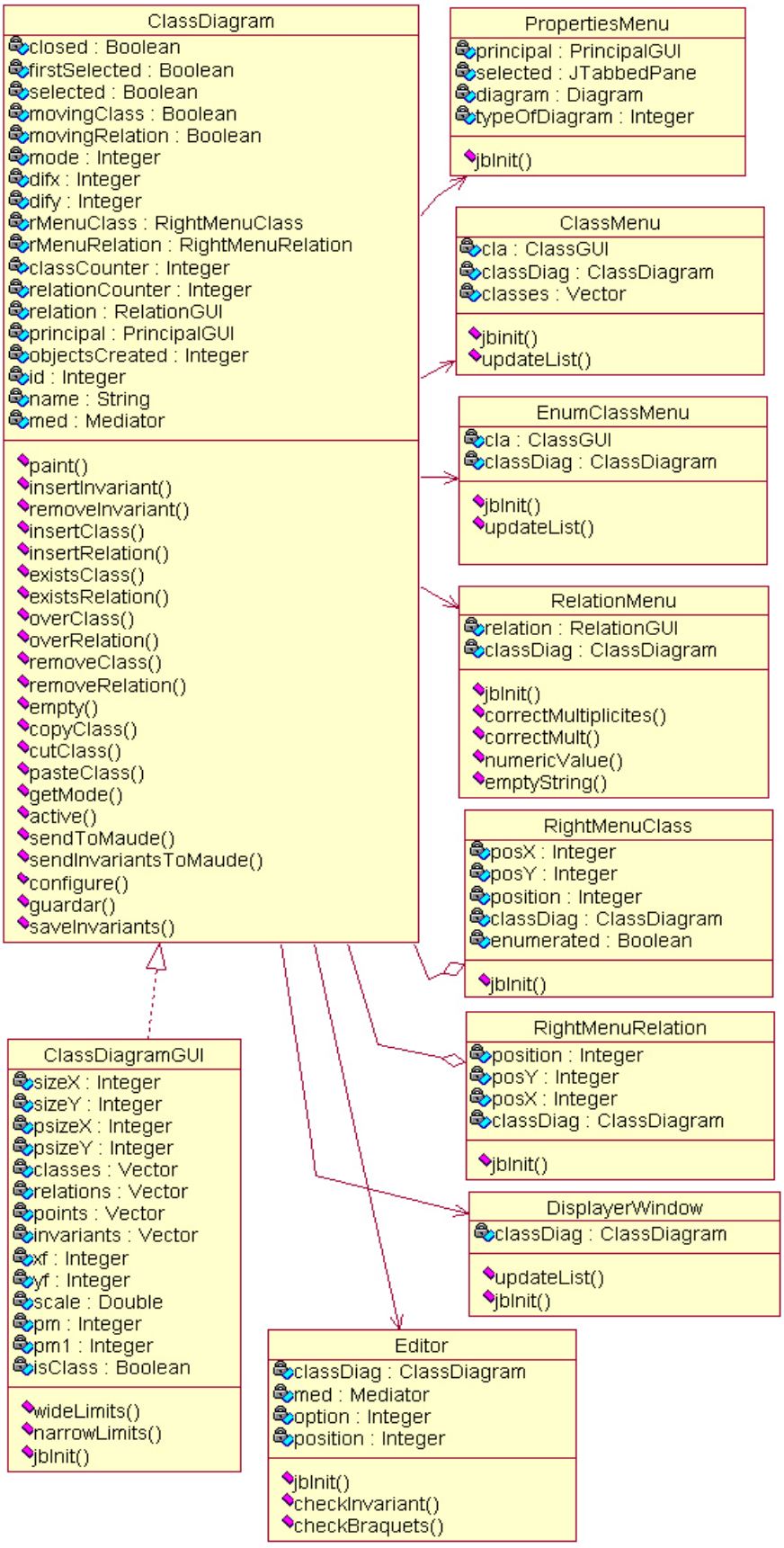

ClassDiagram
closed : Boolean
firstSelected : Boolean
selected : Boolean
movingClass : Boolean
movingRelation : Boolean
mode : Integer
difx : Integer
dify : Integer
rMenuClass : RightMenuClass
rMenuRelation : RightMenuRelation
classCounter : Integer
relationCounter : Integer
relation : RelationGUI
principal : PrincipalGUI
objectsCreated : Integer
id : Integer
name : String
med : Mediator
paint()
insertInvariant()
removeInvariant()
insertClass()
insertRelation()
existsClass()
existsRelation()
overClass()
overRelation()
removeClass()
removeRelation()
empty()
copyClass()
cutClass()
pasteClass()
getMode()
active()
sendToMaude()
sendInvariantsToMaude()
configure()
guardar()
saveInvariants()
PropertiesMenu
principal : PrincipalGUI
selected : JTabbedPane
diagram : Diagram
typeOfDiagram : Integer
jbInit()
ClassMenu
cla : ClassGUI
classDiag : ClassDiagram
classes : Vector
jbInit()
updateList()
EnumClassMenu
cla : ClassGUI
classDiag : ClassDiagram
jbInit()
updateList()
RelationMenu
relation : RelationGUI
classDiag : ClassDiagram
jbInit()
correctMultiplicites()
correctMult()
numericValue()
emptyString()
RightMenuClass
posX : Integer
posY : Integer
position : Integer
classDiag : ClassDiagram
enumerated : Boolean
jbInit()
RightMenuRelation
position : Integer
posY : Integer
posX : Integer
classDiag : ClassDiagram
jbInit()
DisplayerWindow
classDiag : ClassDiagram
updateList()
jbInit()
ClassDiagramGUI
sizeX : Integer
sizeY : Integer
psizeX : Integer
psizeY : Integer
classes : Vector
relations : Vector
points : Vector
invariants : Vector
xf : Integer
yf : Integer
scale : Double
pm : Integer
pm1 : Integer
isClass : Boolean
wideLimits()
narrowLimits()
jbInit()
Editor
classDiag : ClassDiagram
med : Mediator
option : Integer
position : Integer
jbInit()
checkInvariant()
checkBraquets()

Figura 25

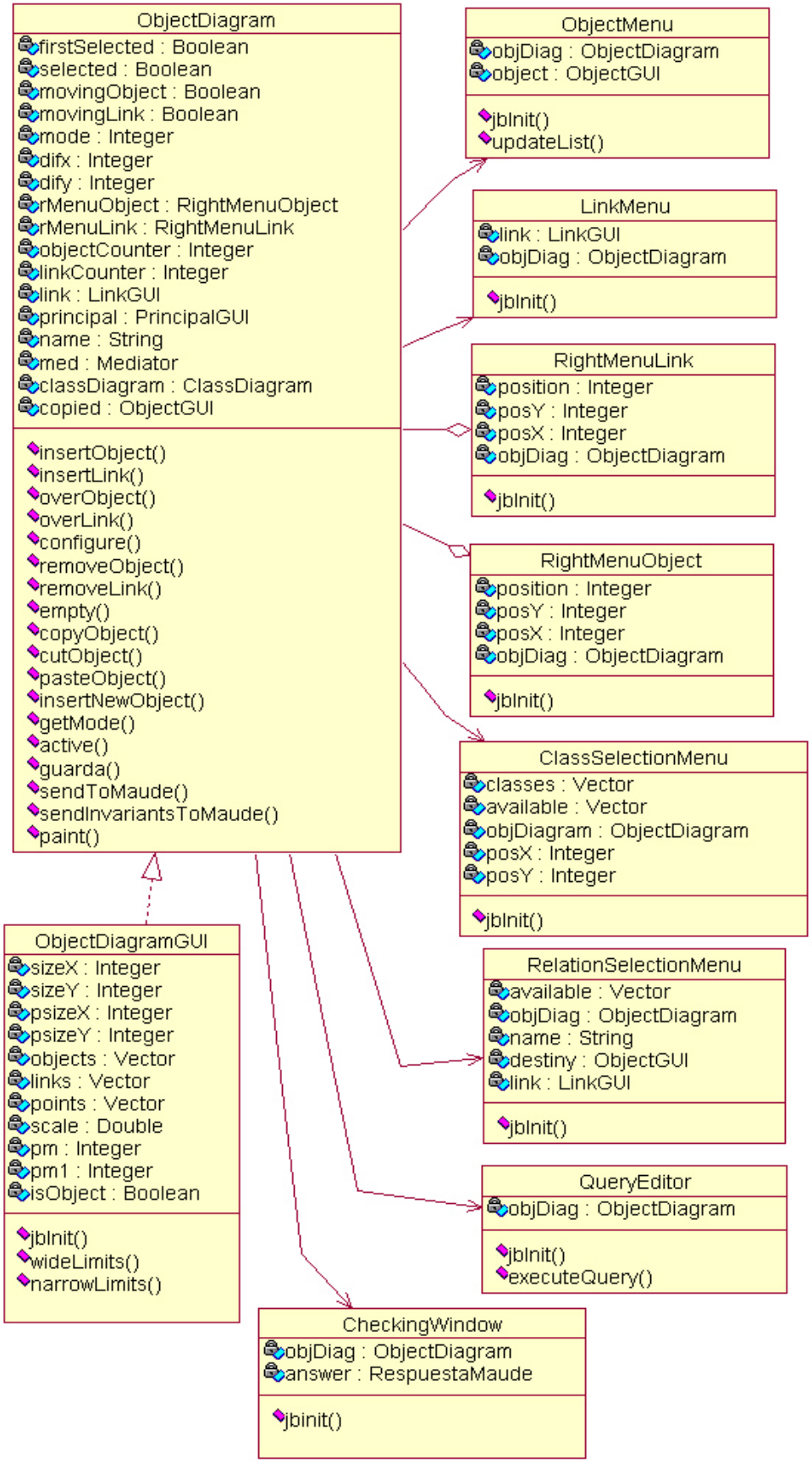

ObjectDiagram
firstSelected : Boolean
selected : Boolean
movingObject : Boolean
movingLink : Boolean
mode : Integer
difx : Integer
dify : Integer
rMenuObject : RightMenuObject
rMenuLink : RightMenuLink
objectCounter : Integer
linkCounter : Integer
link : LinkGUI
principal : PrincipalGUI
name : String
med : Mediator
classDiagram : ClassDiagram
copied : ObjectGUI
insertObject()
insertLink()
overObject()
overLink()
configure()
removeObject()
removeLink()
empty()
copyObject()
cutObject()
pasteObject()
insertNewObject()
getMode()
active()
guarda()
sendToMaude()
sendInvariantsToMaude()
paint()
ObjectMenu
objDiag : ObjectDiagram
object : ObjectGUI
jbInit()
updateList()
LinkMenu
link : LinkGUI
objDiag : ObjectDiagram
jbInit()
RightMenuLink
position : Integer
posY : Integer
posX : Integer
objDiag : ObjectDiagram
jbInit()
RightMenuObject
position : Integer
posY : Integer
posX : Integer
objDiag : ObjectDiagram
jbInit()
ClassSelectionMenu
classes : Vector
available : Vector
objDiagram : ObjectDiagram
posX : Integer
posY : Integer
jbInit()
ObjectDiagramGUI
sizeX : Integer
sizeY : Integer
psizeX : Integer
psizeY : Integer
objects : Vector
links : Vector
points : Vector
scale : Double
pm : Integer
pm1 : Integer
isObject : Boolean
jbInit()
wideLimits()
narrowLimits()
RelationSelectionMenu
available : Vector
objDiag : ObjectDiagram
name : String
destiny : ObjectGUI
link : LinkGUI
jbInit()
QueryEditor
objDiag : ObjectDiagram
jbInit()
executeQuery()
CheckingWindow
objDiag : ObjectDiagram
answer : RespuestaMaude
jbinit()

Figura 26

Como puede verse, la parte grafica para cada uno de los diagramas, ClassDiagramGUI y ObjectDiagramGUI, eran independientes entre sí, por lo que se optó por crear una clase DiagramGUI que fuera una generalización de estas clases, siguiendo un patrón Template Method. De esta forma en DiagramGUI van los métodos comunes, dejando las partes distintas como métodos abstractos que implementan las clases que heredan de DiagramGUI.
De esta forma, la parte común de la parte gráfica de los paneles no se duplicaba.
Para posteriores versiones de la aplicación se ha propuesto una nueva mejora que separe la parte gráfica de la parte lógica de la siguiente forma:

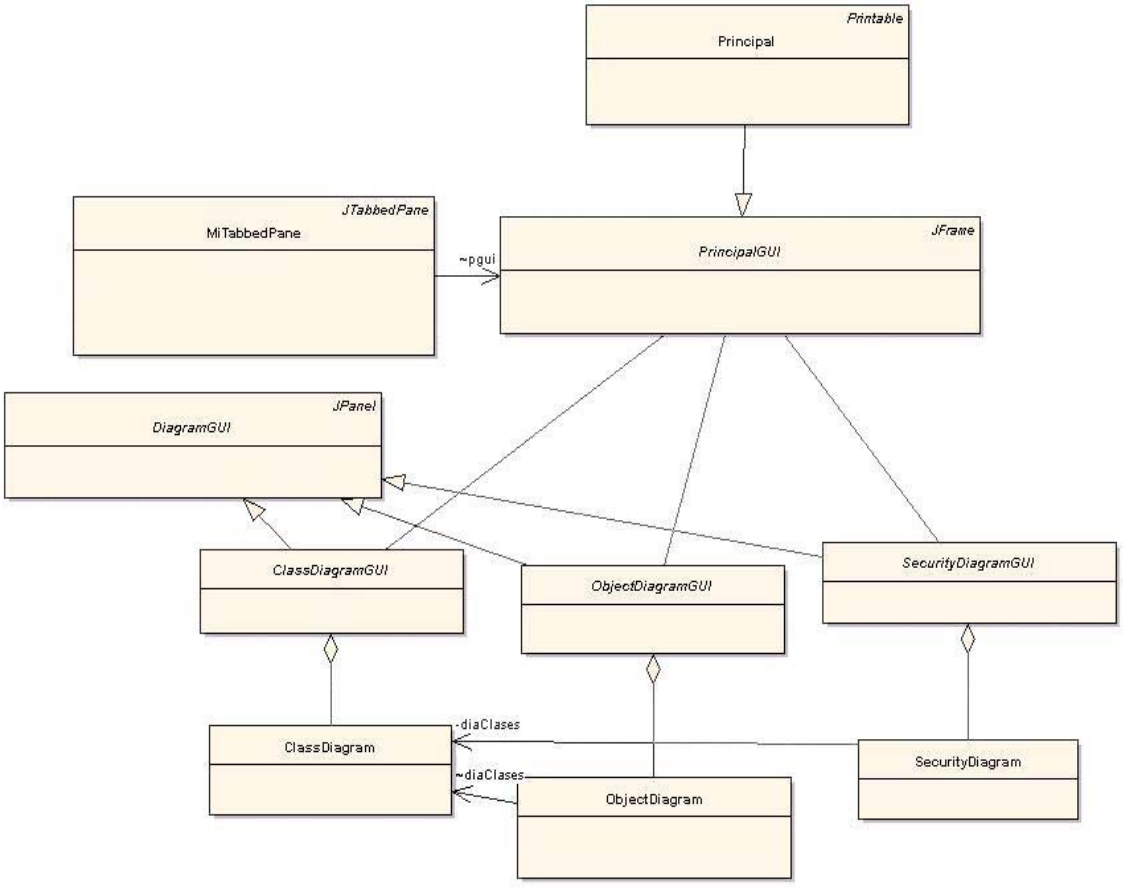
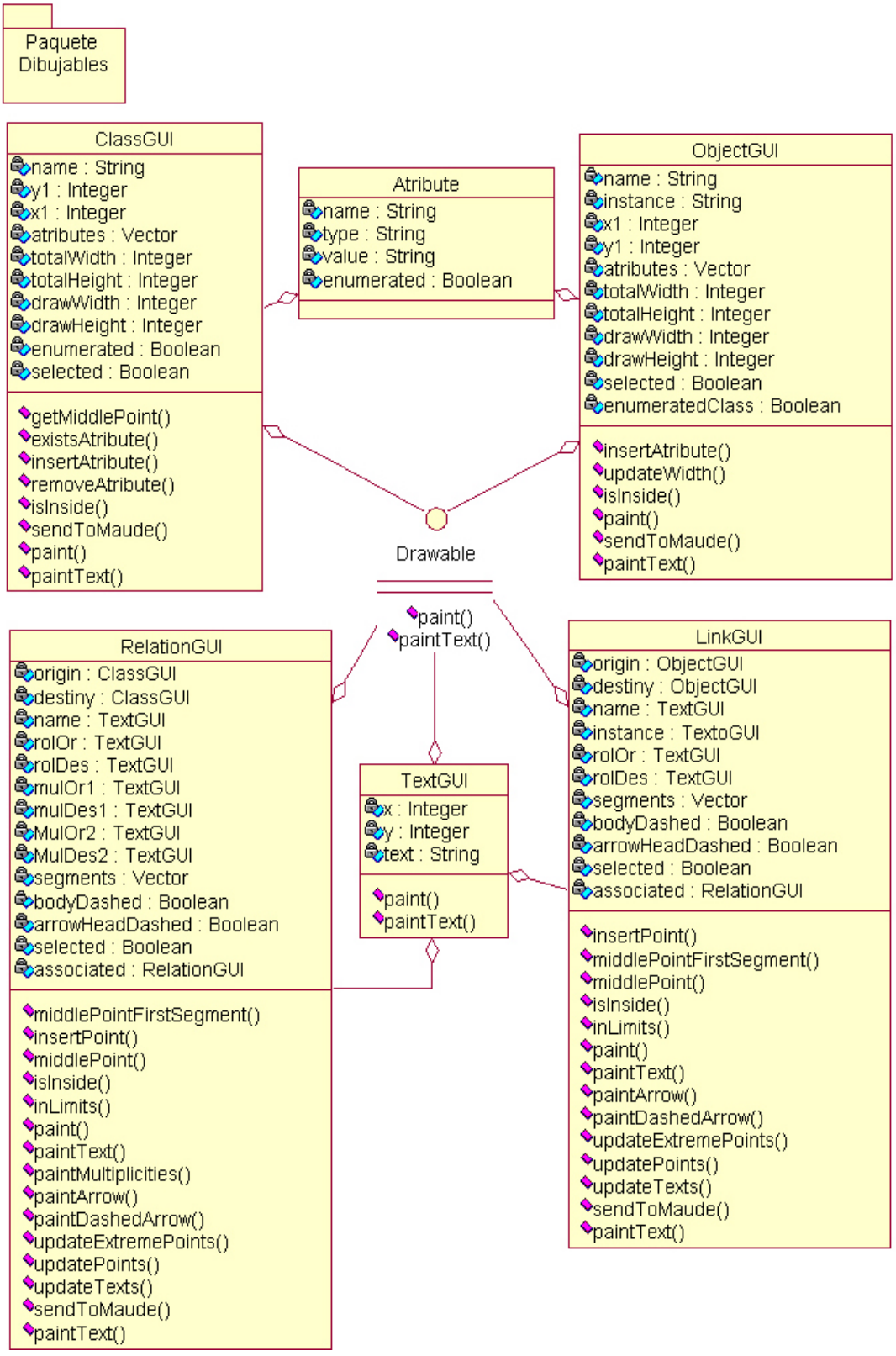


Figura 27

De esta forma ClassDiagram, ObjectDiagram y SecurityDiagram solo contendrían el modelo de los diagramas y serían utilizados por la parte gráfica, ya que anteriormente la parte gráfica contenía parte gráfica y lógica.

Otra mejora realizada sobre la versión previa ha consistido en aplicar un patrón Composite a los elementos dibujables en los distintos diagramas. En la versión previa los distintos elementos dependían entre sí a través de una interfaz teniendo que duplicar código para implementar cada uno de ellos. Por ejemplo, las distintas relaciones que, gráficamente son muy similares, en el diseño previo no tenían relación:



Como puede verse RelationGUI y LinkGUI, que implementan las relaciones entre clases y entre objetos respectivamente únicamente estaban relacionadas a

través de una interfaz. Tras aplicar el patrón Composite el diseño es el siguiente (ver Figura28):

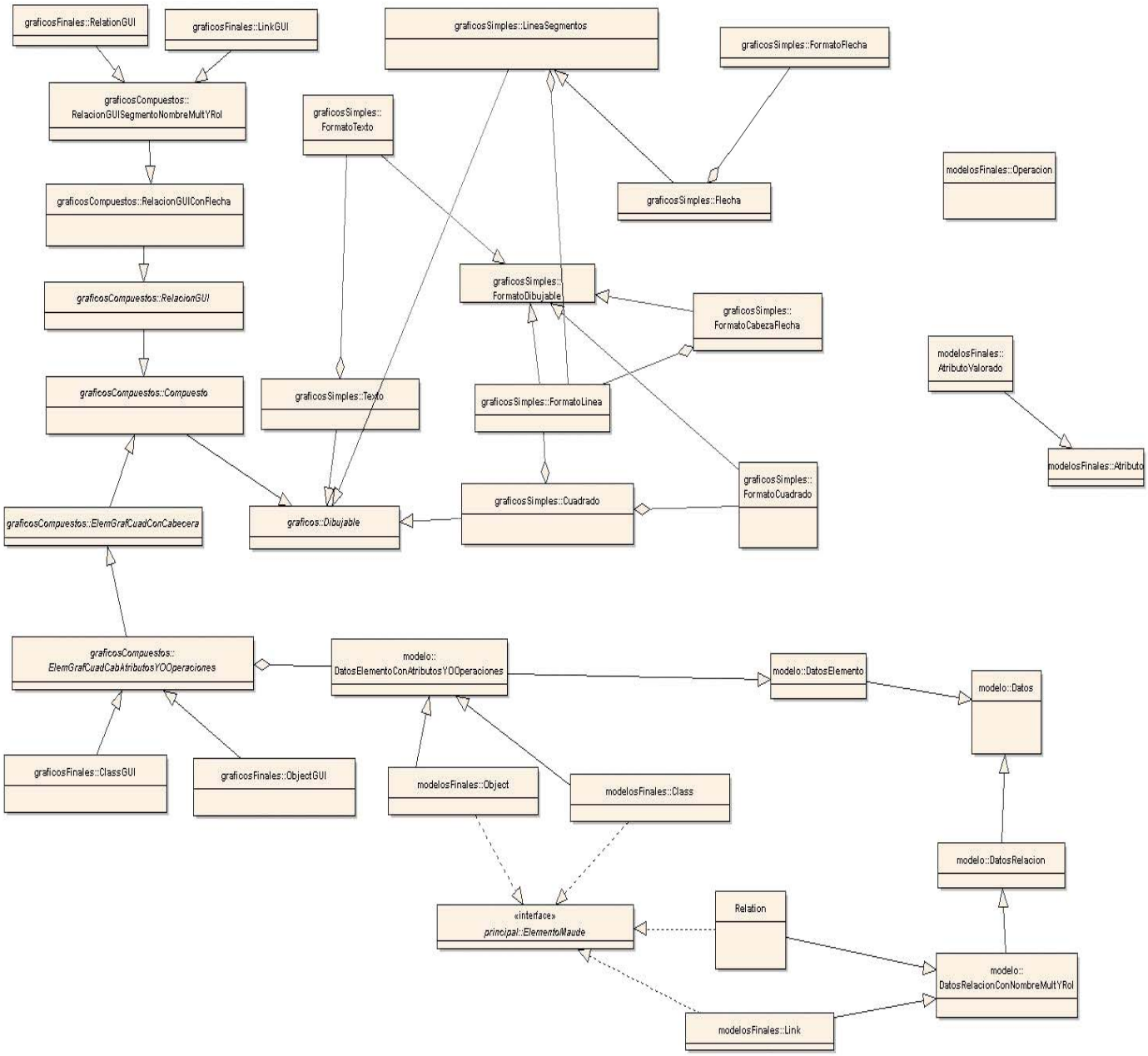


Figura 28

4. MEJORAS REALIZADAS EN LA HERRAMIENTA

Este proyecto se basa en la herramienta Visual ITP/OCL que empezó el grupo del año pasado. Su trabajo consistió en una herramienta gráfica de edición de diagramas de clases y objetos, y restricciones que ayuda la validación automática de diagramas UML con respecto a restricciones OCL.

Nuestros cambios respecto al año pasado han sido:

- Mejorar el diagrama de clases y objetos.
- Crear los diagramas de seguridad y de métricas.

- **<u>Diagrama de clases:</u>**

  - Hemos modificado el aspecto de la ventana:
    - Antes: (ver Figura 29)

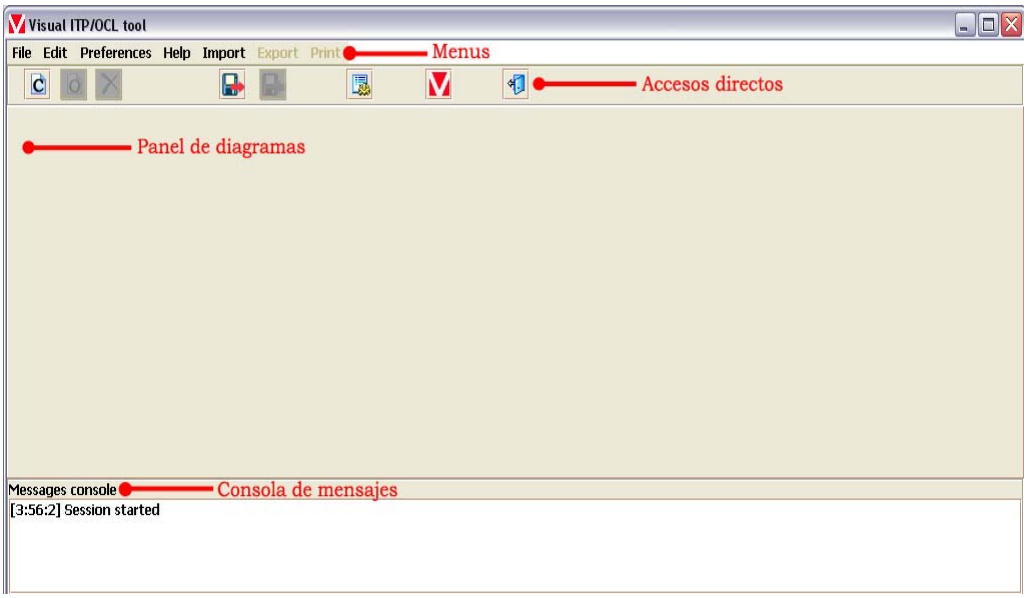


Figura 29

- Ahora: (ver Figura 30)

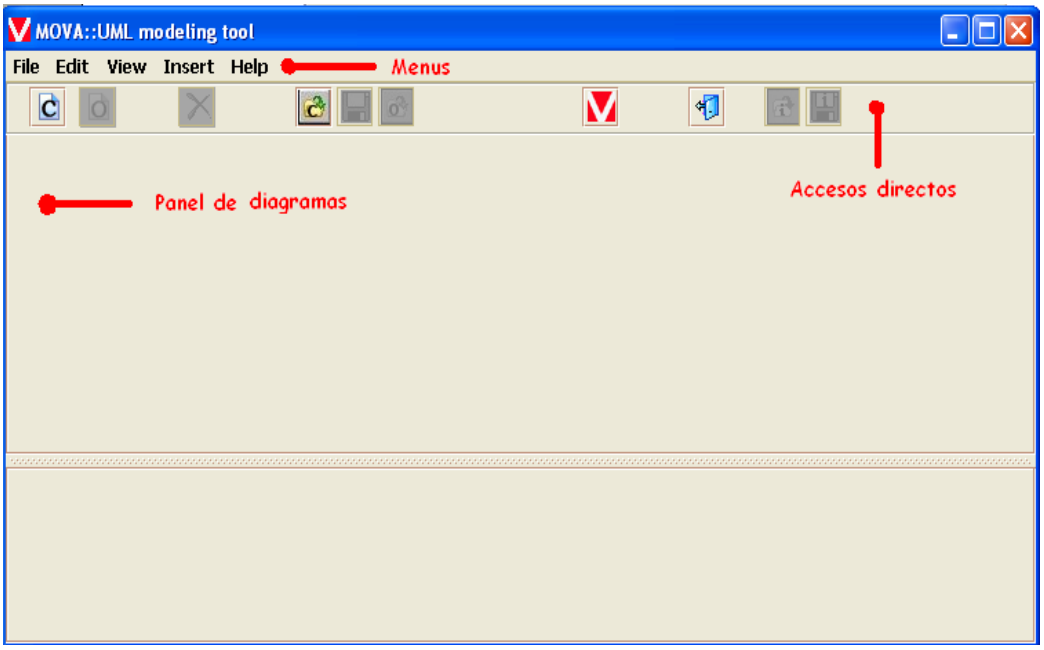


Figura 30

Los cambios en el *menú* han sido:

- Hemos eliminado los siguientes submenús:
  - Preferentes
  - Import
  - Export y
  - Print.
- Y hemos modificado:
  - El submenú *file*, donde hemos añadido:
    - Open
    - Close
    - Page setup
    - Print
    - Export to EPS
  - El submenú *edit,* donde hemos quitado las opciones que tenía, y las hemos sustituido por:
    - Zoom +
    - Zoom -

Los cambios en los *accesos directos* han sido:

- Hemos añadido:
  - Uno para cargar diagrama de clases.
  - Y otro para cargar invariantes y operaciones.

- Hemos creado la *ventana de opciones de visualización*: que es un menú (ver Figura 31) en el que podamos elegir lo que queramos que se haga visible en el diagrama de clases: atributos, roles, multiplicidades, o nombre de las relaciones.

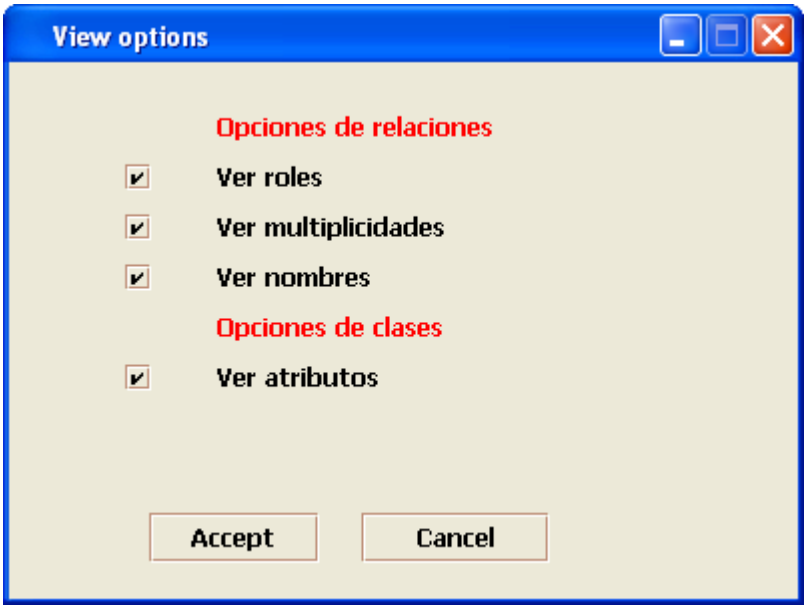


Figura 31

- Hemos añadido la opción de *copiar*, *pegar*, y *mover* elementos.
- Hemos modificado la ventana *DisplayerWindow:* de manera que ahora se puedan seleccionar algunos invariantes, para poder ser luego evaluados. (ver Figura 32)

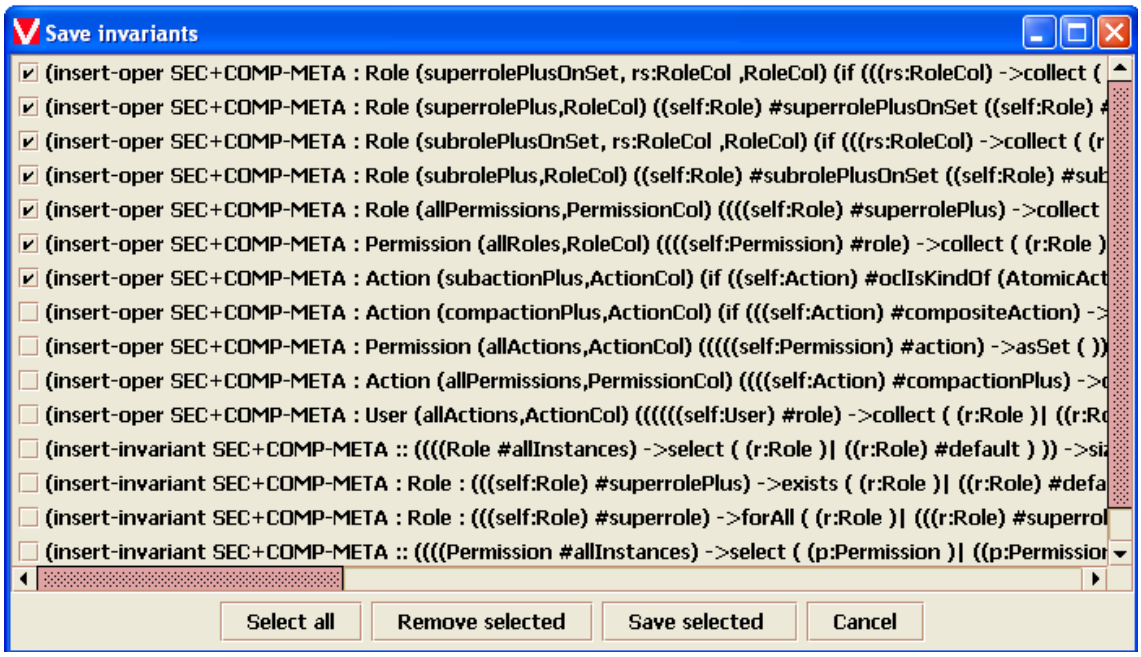


Figura 32

- Hemos modificado la *ventana de propiedades de clases*: hemos añadido un botón para modificar el nombre de la clase, y hemos quitado los botones de aplicar y aceptar. (ver Figura 33)

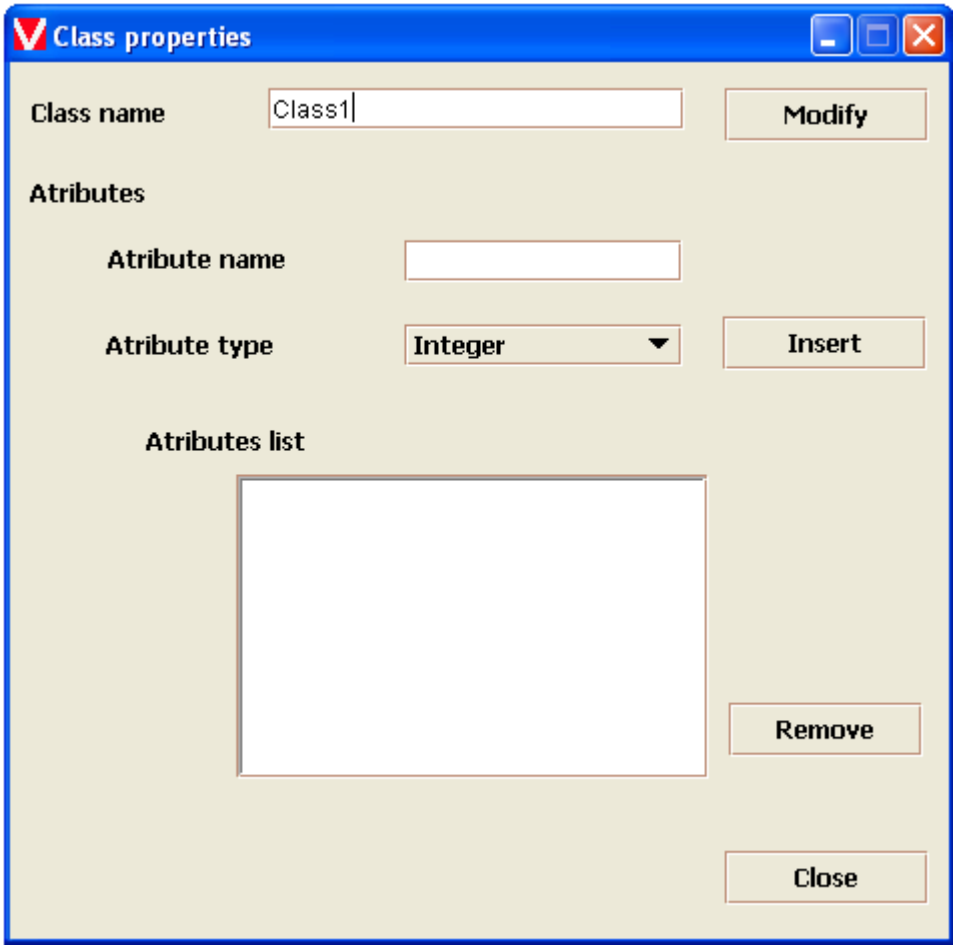


Figura 33

- **<u>Diagrama de objetos:</u>**

  - Hemos creado la *ventana de opciones de visualización*: que es un menú similar al del diagrama de clases (ver Figura 34) para poder seleccionar lo que queremos que se haga visible en el diagrama de objetos: atributos, roles o nombres de relaciones.

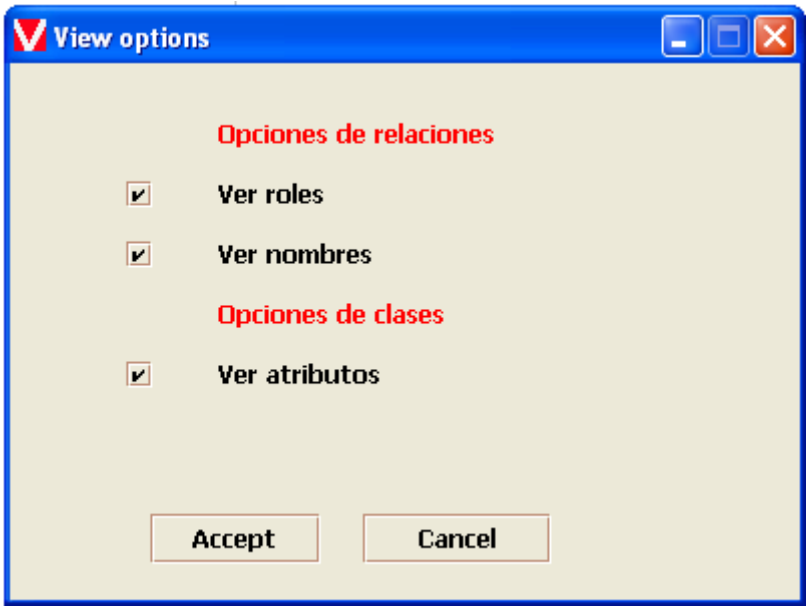


Figura 34

- Hemos añadido la opción de *copiar*, *pegar*, y *mover* elementos.
- Hemos modificado la ventana *CheckingWindow:* de manera que se muestran los resultados de comprobar los invariantes almacenados.

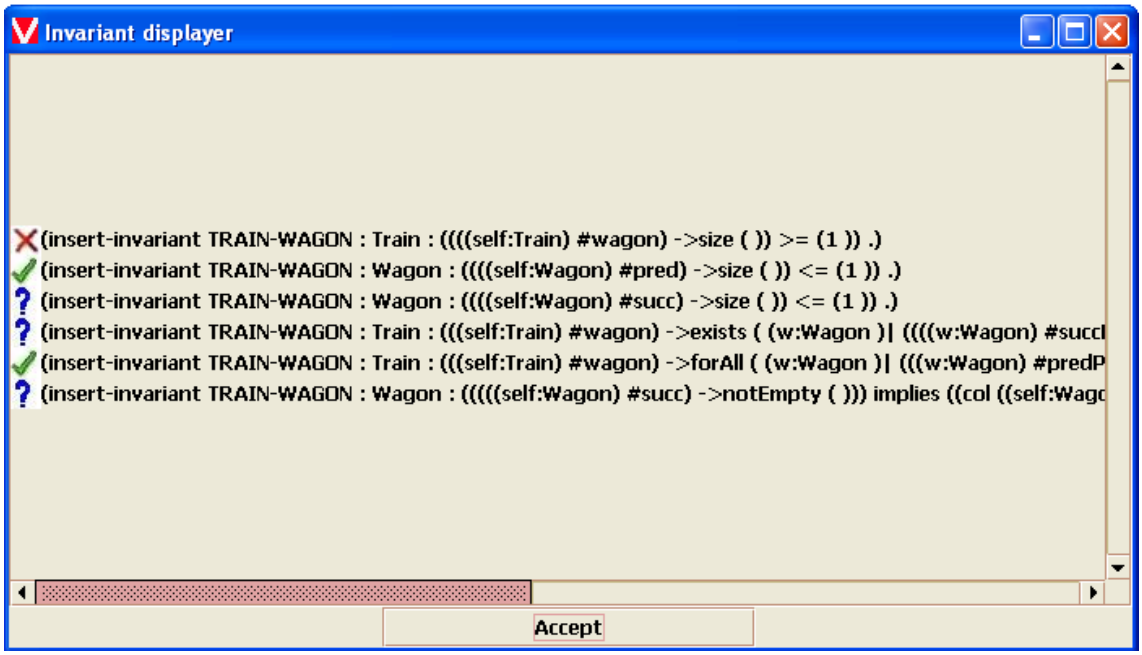


Figura 35

- Hemos modificado la *ventana propiedades de objetos*: hemos añadido un botón para modificar el nombre del objeto, y hemos quitado los botones de aplicar y aceptar.

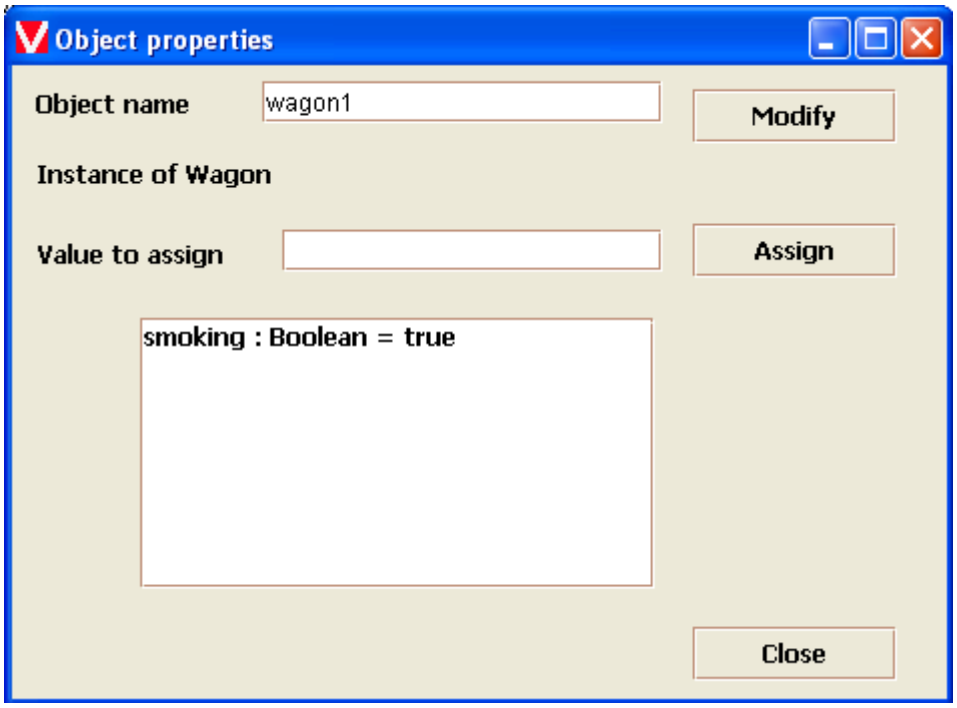


Figura 36

- **Diagrama de métricas:**

Hemos creado el diagrama de métricas que añade a UML modeling la posibilidad de evaluar métricas sobre el diagrama de clases. Ambas aplicaciones tienen interfaces similares, siendo la única diferencia que la opción de insertar métricas está activa en el diagrama de métricas.

En MOVA, las métricas son consultas en OCL sobre la instancia del metamodelo correspondiente al diagrama de clases. El editor puede ayudar al usuario escribiendo estas métricas: en este caso los patrones ofrecidos por el editor, y el resultado de su evaluación, corresponden a:

- Las clases, atributos, roles, generalizaciones, y las asociaciones incluidas en el metamodelo de MOVA.
- Los objetos y links incluidos en la instancia del metamodelo de MOVA que corresponden al diagrama de clases; esta instancia es automáticamente generada por la aplicación.

5. MANUAL DE USUARIO

UML MODELING

UML modeling permite al usuario:

- Crear diagramas de clases y objetos
- Escribir y evaluar invariantes
- Escribir y evaluar consultas, y
- Definir operaciones para ser usadas en las invariantes y consultas.

Se distinguen 3 bloques de elementos en la pantalla (ver Figura 37):

1- Menús de la herramienta.

2- Barra de accesos directos.

3- Panel de diagramas.

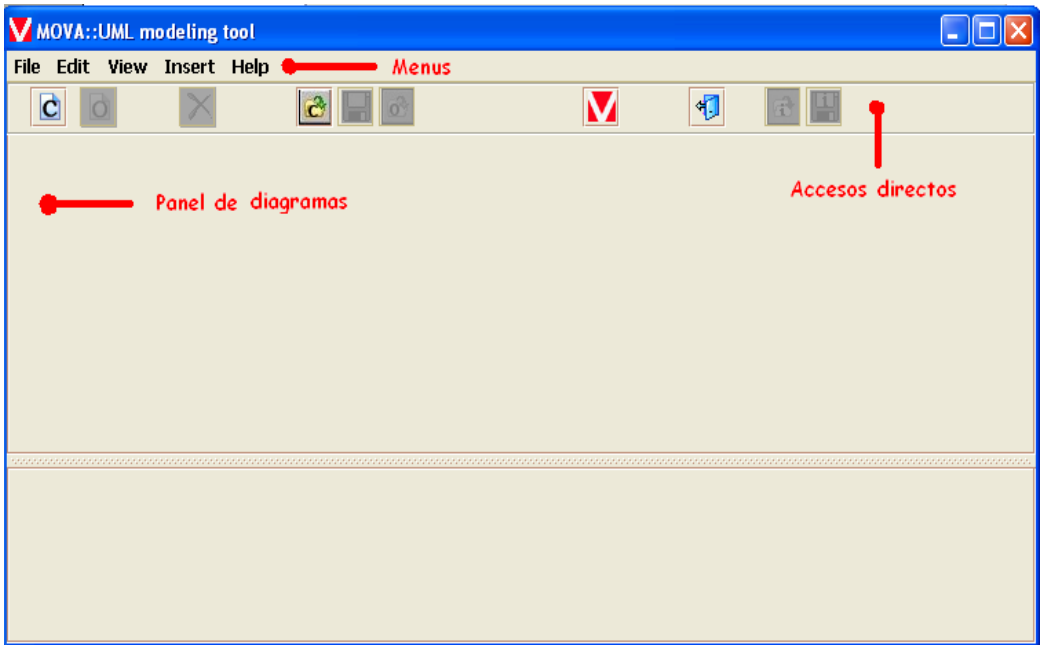


Figura 37

Menús de la herramienta:

- File:
  - New:
    - Class Diagram: Crea un diagrama de clases vacío.
    - Object Diagram: Crea un diagrama de objetos vacío.
  - Open:
    - Class Diagram: Abre un diagrama de clases.
    - Object Diagram: Abre un diagrama de objetos.
  - Load:
    - Invariants, operations: Carga las invariantes y operaciones en el diagrama.
  - Close: Cierra el diagrama en curso.
  - Page Setup: Permite configurar la impresión.
  - Print: Imprime el área imprimible del diagrama actual.
  - Export to EPS: Exporta el diagrama actual a un fichero EPS.
  - Save as:
    - Class Diagram: Almacena un diagrama en un fichero xml.
    - Invariants and operations: Almacena las invariantes y operaciones de un diagrama en un fichero.
  - Exit: Sale de la herramienta.

- Edit:
  - Zoom +: Aumenta el tamaño del diagrama.
  - Zoom -: Reduce el tamaño del diagrama.

- View:
  - Invariants: Muestra una ventana con las invariantes del diagrama.
  - Class Diagram: Muestra una ventana con opciones de visualización del diagrama.
  - Object Diagram: Muestra una ventana con opciones de visualización del diagrama.

- Insert:
    - Class Diagram:
        - Class: Inserta una clase en el diagrama.
        - Enum: Inserta una clase enumerada en el diagrama.
        - Association class-class: Inserta una asociación entre clases.
        - Association class-association: Inserta una clase de asociación.
        - Generalization: Inserta una generalización.
        - Invariant: Abre el editor de invariantes, para insertar una invariante.
        - Operation: Abre el editor de operaciones, para insertar una operación.
    - Object Diagram:
        - Object: Inserta un objeto en el diagrama.
        - Link: Inserta un link entre objetos.
        - Query: Muestra el editor de consultas.

- Help:
    - About us: Muestra una ventana con información de la herramienta.

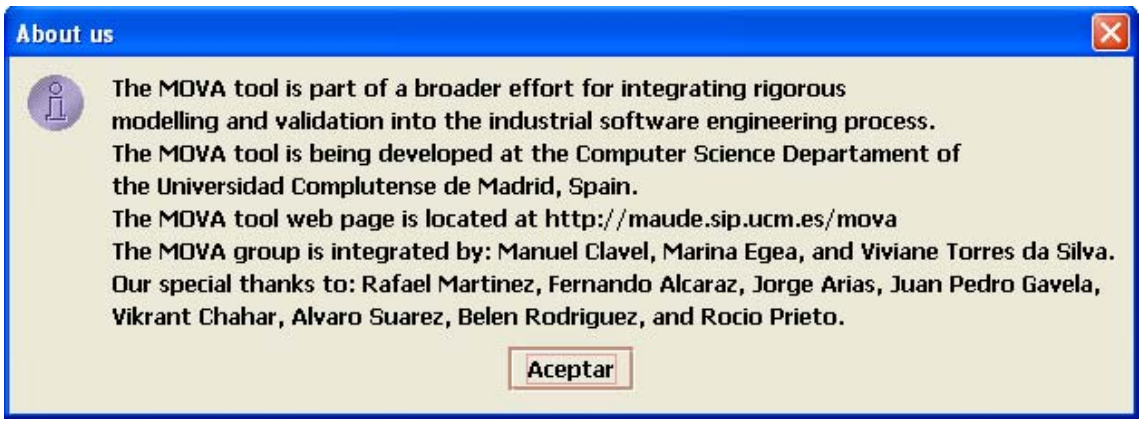


Figura 38

Accesos directos:

Figura 39

Crea un diagrama de clases.

Crea un diagrama de objetos.

Elimina el diagrama actual.

Carga un diagrama de clases.

Guarda el diagrama actual.

Carga un diagrama de objetos.

Muestra la ayuda.

Sale de la herramienta.

Carga invariantes.

Guarda todos los invariantes del diagrama.

Panel de diagramas:

El panel de diagramas es la parte central de la ventana, que contiene todos los diagramas de la aplicación.

Diagrama de clases:

Modo cursor. Permite:

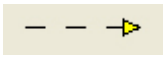
- Seleccionar elementos:
  - Selección individual: Pinchando sobre el elemento deseado con el botón izquierdo del ratón.
  - Selección múltiple: Pinchando con el botón izquierdo del ratón, fuera de cualquier elemento, y arrastrándolo hasta seleccionar todos los elementos deseados.
- Editar elementos: Haciendo doble clic sobre el elemento.
- Desplegar el menú de un elemento: Pinchando con el botón derecho sobre el elemento.

 Crea clases en el diagrama de clases.

 Crea clases enumeradas en el diagrama de clases.

 Crea relaciones de asociación entre clases.

 Crea relaciones de generalización entre clases

 Crea relaciones de clase de asociación entre una clase y una asociación.

 Aumenta el tamaño del diagrama.

 Reduce el tamaño del diagrama.

 Muestra la ventana de opciones de visualización (ver Figura 40): que permite seleccionar qué características del diagrama queremos que sean visibles, y cuáles no.

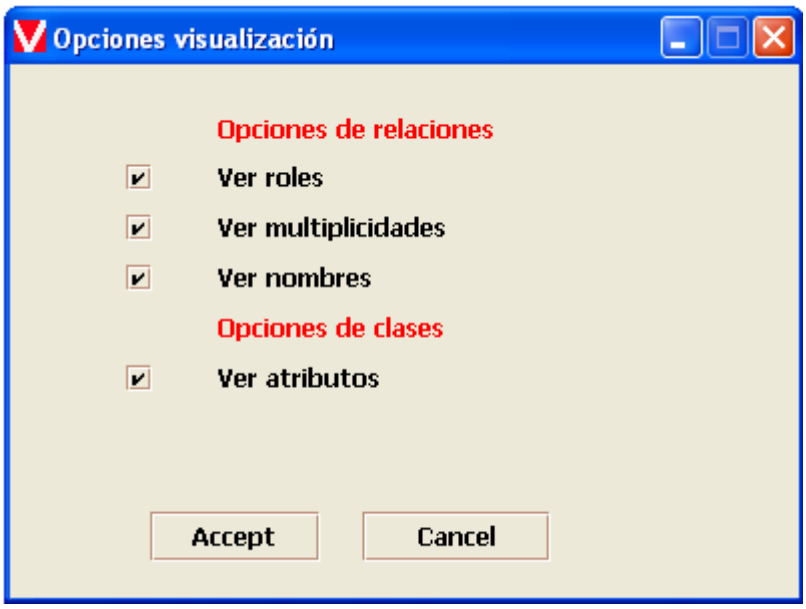


Figura 40

 Muestra la ventana (ver Figura 41) para editar invariantes:

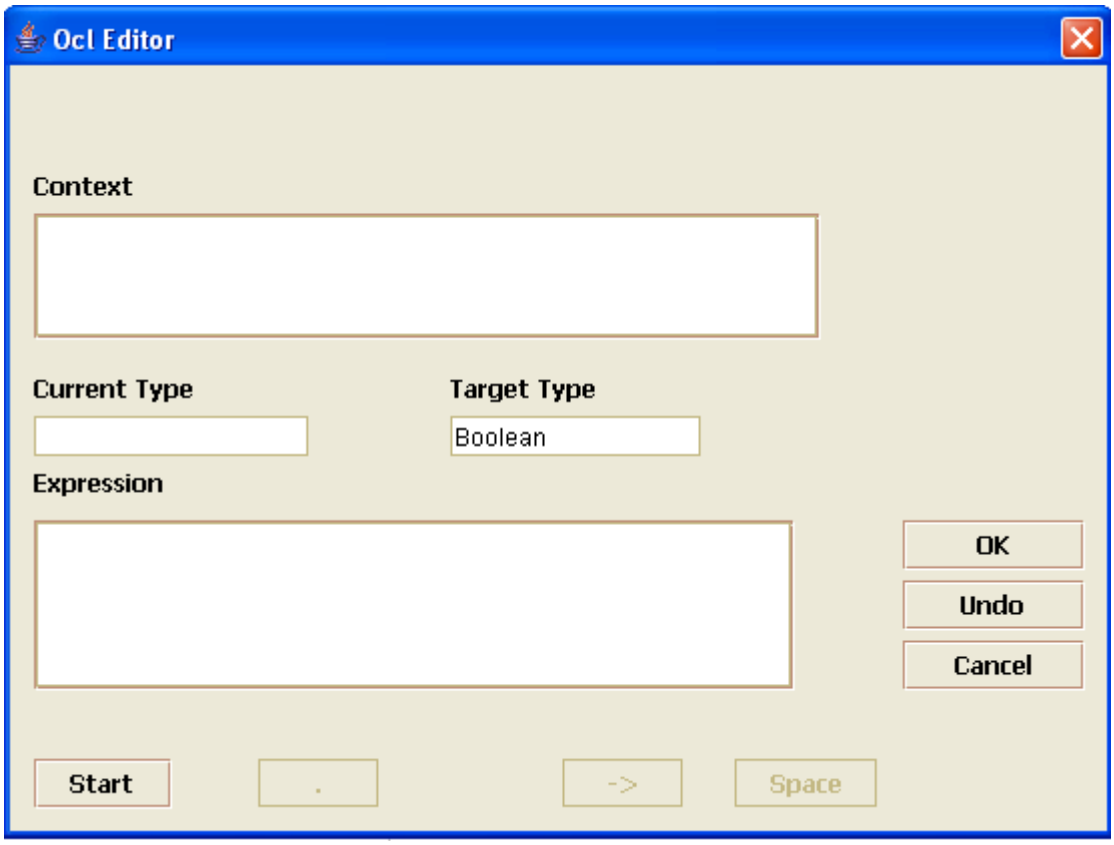


Figura 41

 Muestra la ventana (ver Figura 42) con todos los invariantes y operaciones insertados en el diagrama:

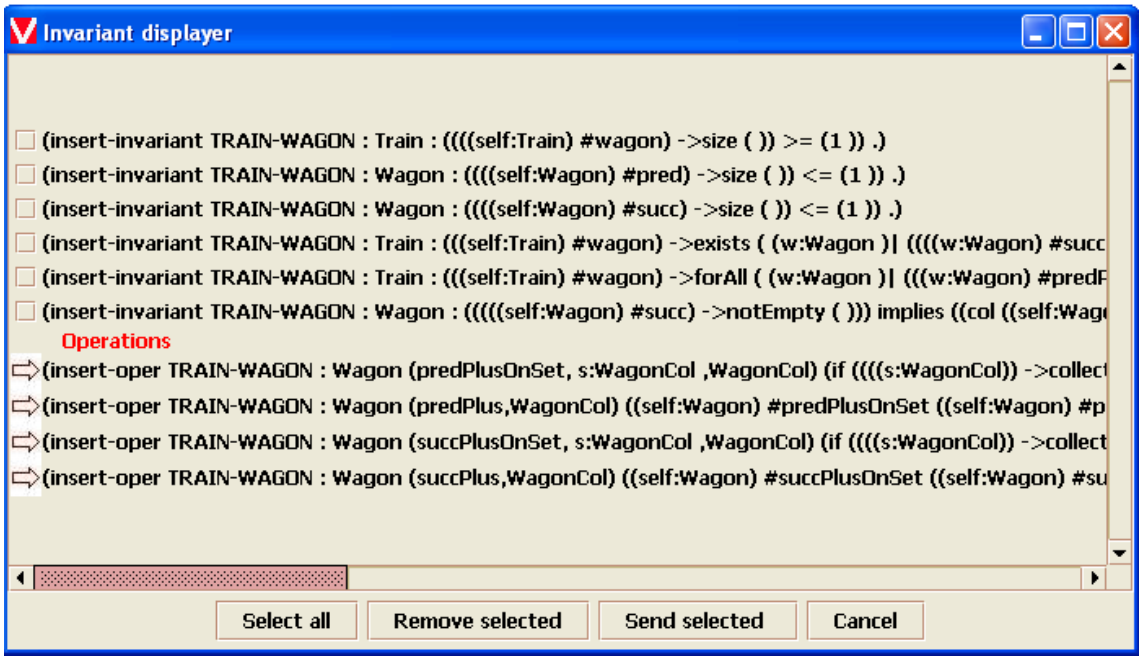


Figura 42

Los invariantes, además, se pueden seleccionar para poderse chequear en el diagrama de objetos.

Diagrama de objetos:

 Modo cursor.

Crea objetos en el diagrama de objetos.

Crea relaciones entre objetos en el diagrama de objetos.

 Aumenta el tamaño del diagrama.

 Reduce el tamaño del diagrama.

 Muestra la ventana de opciones de visualización.

 Muestra los resultados obtenidos (ver Figura 43) de comprobar sobre un diagrama de objetos los invariantes seleccionados en el diagrama de clases asociado.

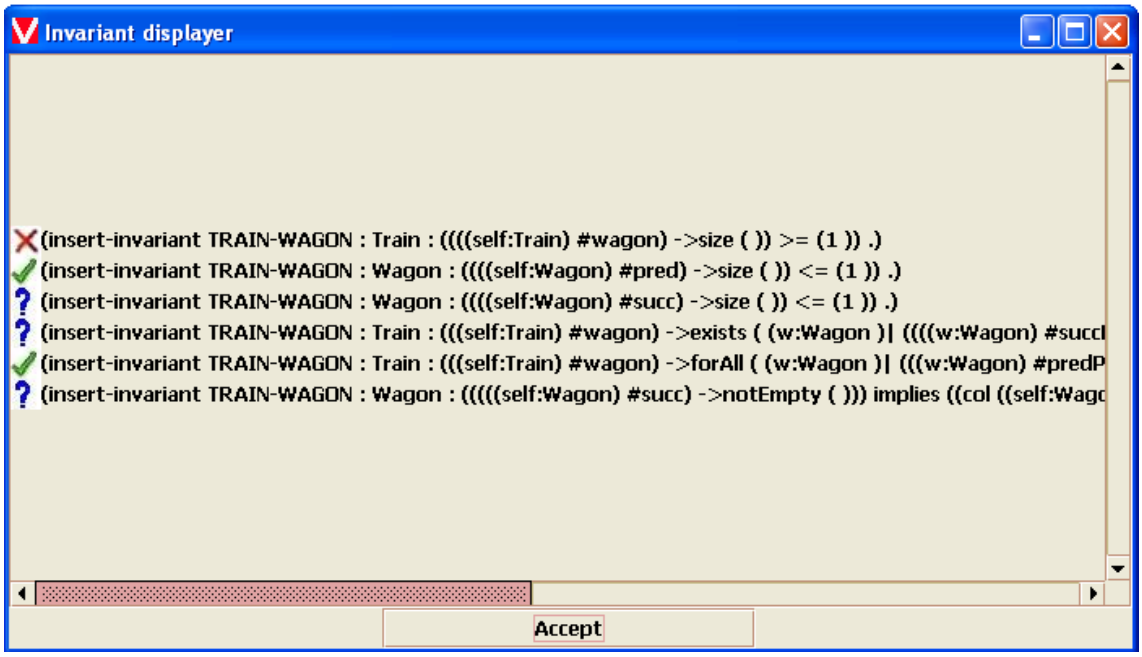


Figura 43

 Muestra la ventana (ver Figura 44) para introducir consultas:

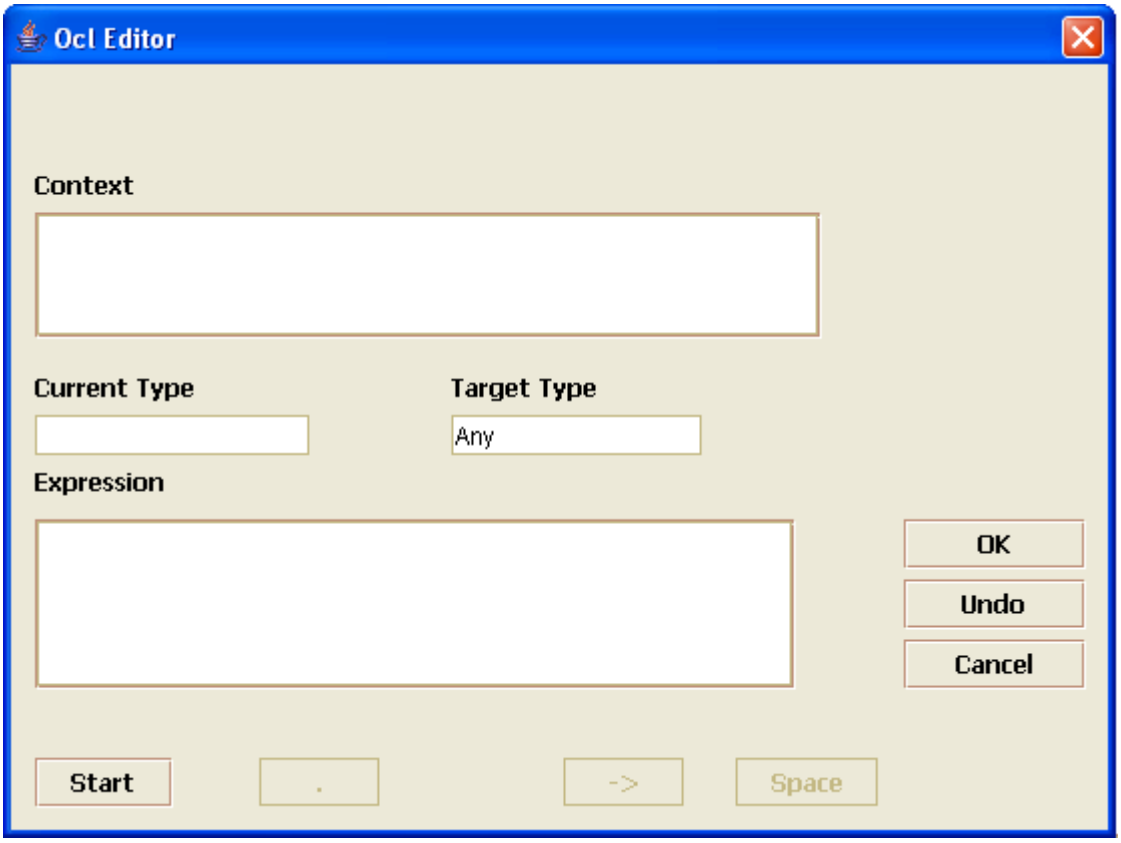

Ocl Editor
Context
Current Type
Target Type
Any
Expression
OK
Undo
Cancel
Start
.
->
Space

Figura 44

6. EJEMPLOS

EJEMPLO COMPANY MODEL:

En este ejemplo (ver Figura 45), hemos modelado una compañía con empleados, departamentos y proyectos. El departamento puede tener empleados trabajando en él y puede controlar un número de proyectos.

Un empleado puede trabajar en uno o más departamentos y puede participar en varios proyectos.

El control del proyecto es llevado a cabo por el departamento. Cada proyecto puede contar con varios empleados que trabajen en él.

El ejemplo se ha tomado de [Buettner, F. USE quick tour].

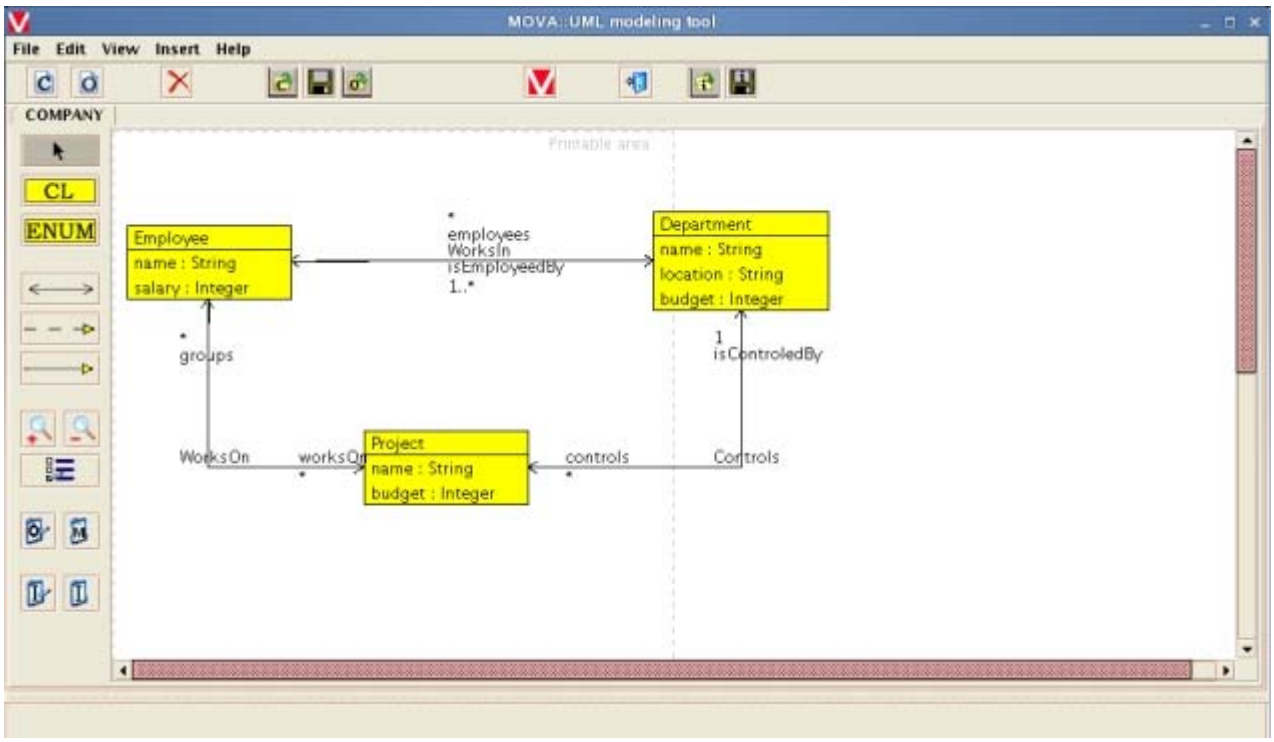


Figura 45

Las siguientes invariantes OCL añaden precisión al modelo “Company”.

El número de empleados trabajando en un departamento debe ser mayor o igual al número de proyectos controlados por el departamento.

context Department inv notTooManyProjects:
self.employees->size() >= self.controls->size()

Los empleados cobran más según trabajen en más proyectos.

Employee.allInstances()->forAll(e1, e2 |
(e1.worksOn->size() > e2.worksOn->size()) implies e1.salary > e2.salary)

El presupuesto de un proyecto no debe exceder el presupuesto del departamento que controla el proyecto.

context Project inv budgetControl:
self.budget <= self.isControledBy.budget

Los empleados que trabajen en un proyecto deben también trabajar en el control del departamento.

context Project inv departmentParticipation:
self.isControledBy.employee->includesAll(self.employee)

Todas estas invariantes se insertan fácilmente en MOVA usando el editor OCL (ver Figura 46):

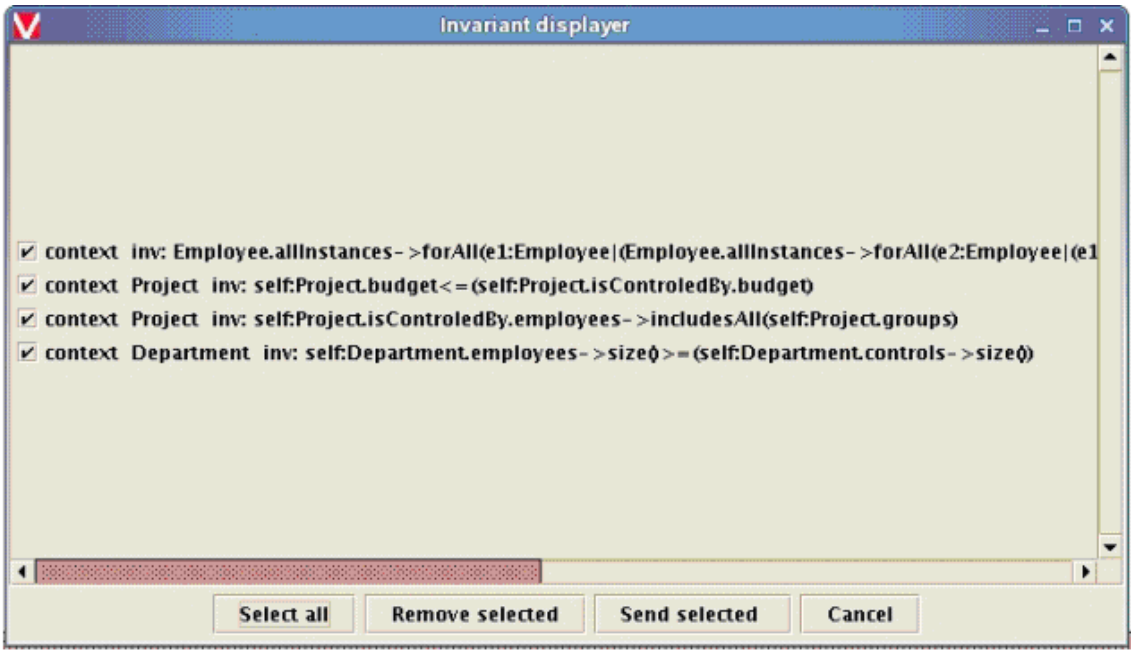


Figura 46

Consideramos ahora la siguiente instancia del modelo Company (ver Figura 47):

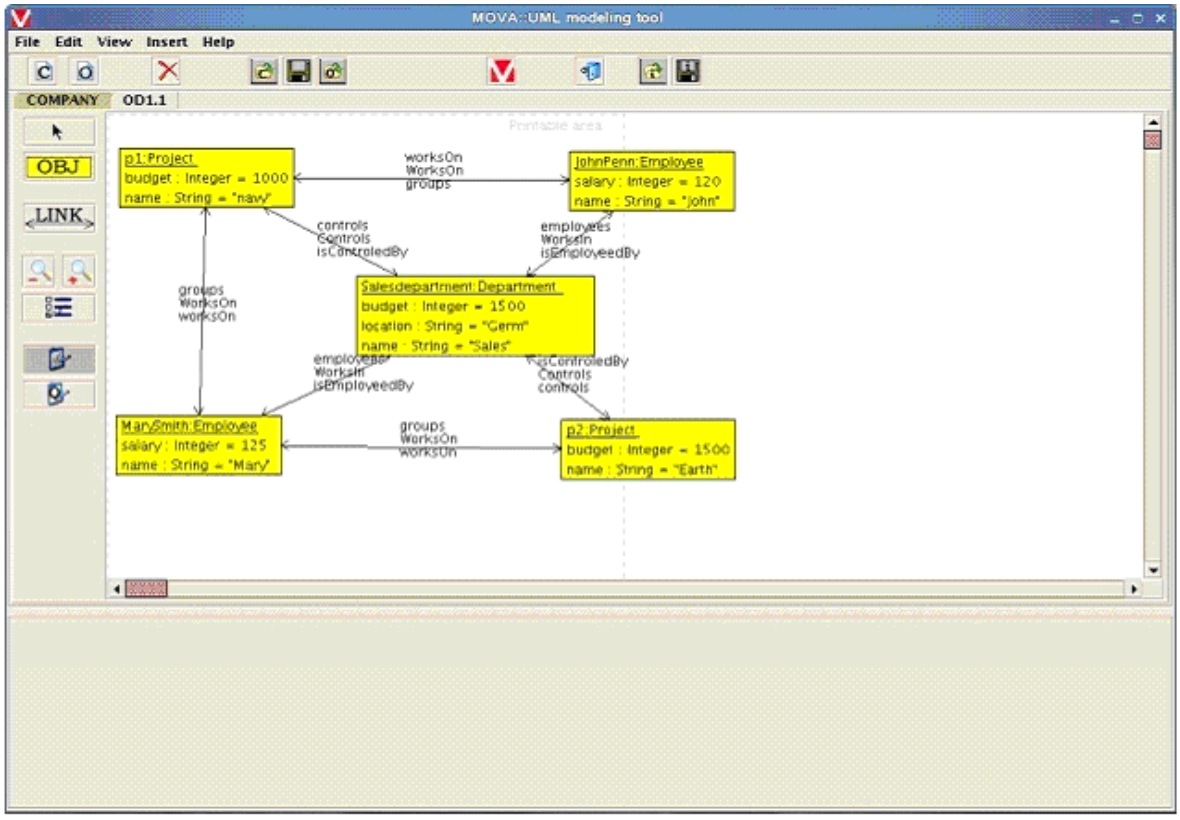


Figura 47

Este diagrama de objetos cumple claramente todas las invariantes añadidas al modelo. Se comprueba automáticamente usando MOVA.

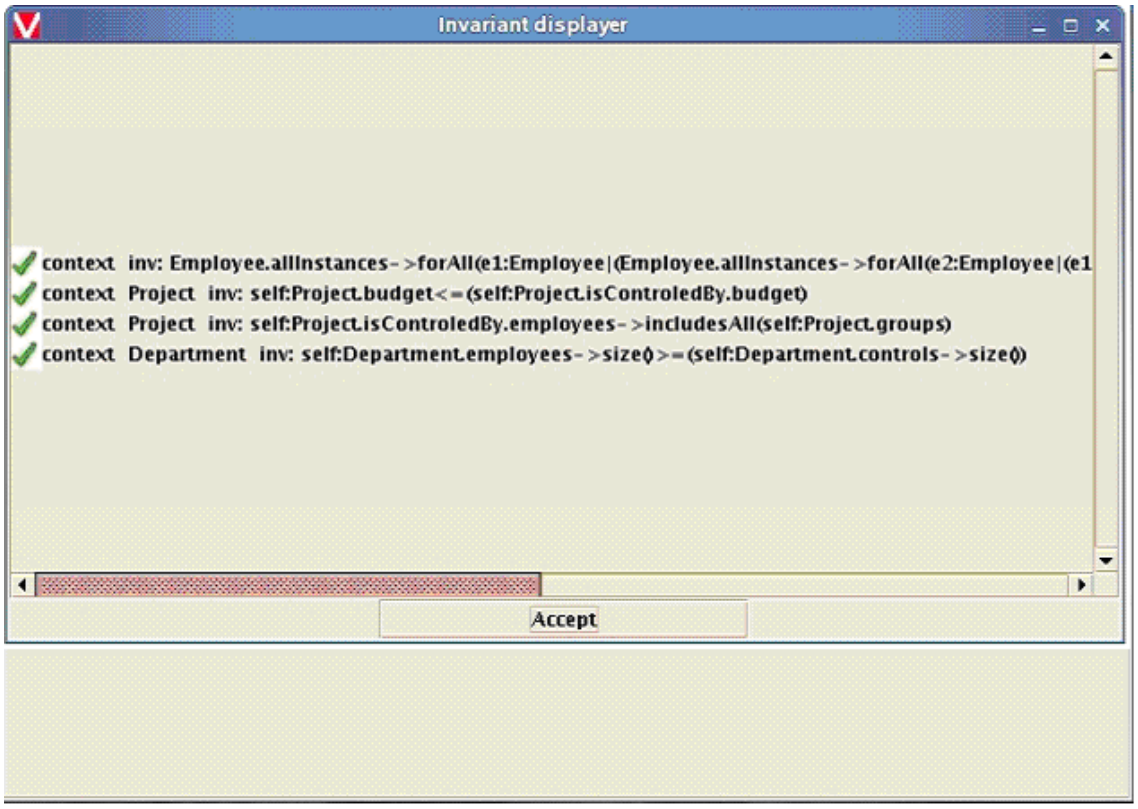


Figura 48

EJEMPLO THE FLIGHT MODEL:

En este ejemplo, tomado de *[Warmer, J., Kleppe, A. The Object Constraint Language, Second Edition: Getting Your Models Ready for MDA., Addison-Wesley, 2003]*, la asociación entre la clase Flight y la clase Person indicando que un cierto grupo de personas son los pasajeros de un vuelo, tiene la multiplicidad al lado de la clase Person:

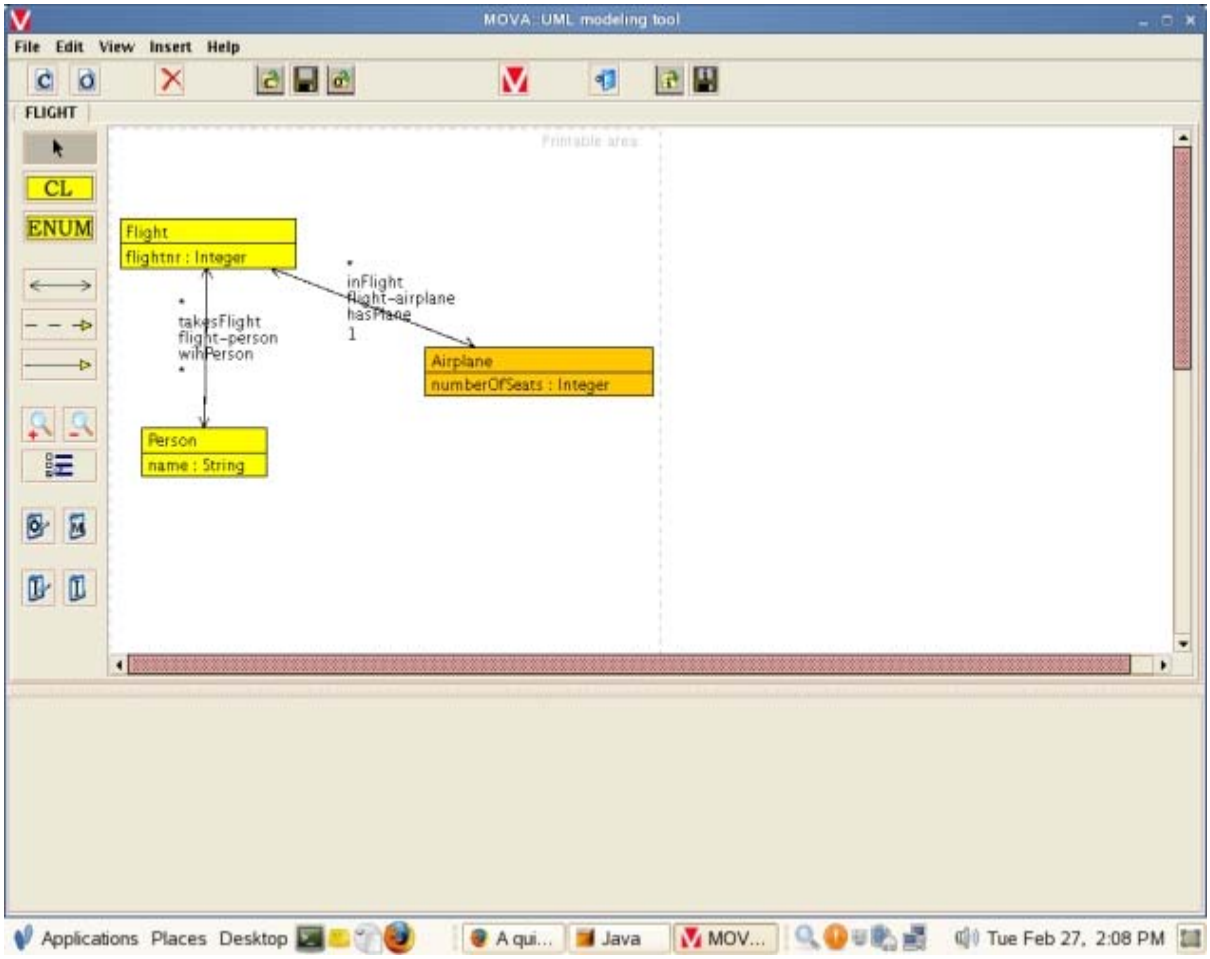


Figura 49

El número de pasajeros del vuelo está restringido por el número de asientos del avión. Sin embargo es imposible expresar esta restricción en el diagrama. En este ejemplo la manera correcta de especificar la multiplicidad es añadir al diagrama la correspondiente restricción OCL.

context Flight inv:

self.with Person->size() <= self.hasPlane.numberOfSeats

Esta invariante se inserta fácilmente usando el editor OCL de la herramienta MOVA (ver Figura 50).

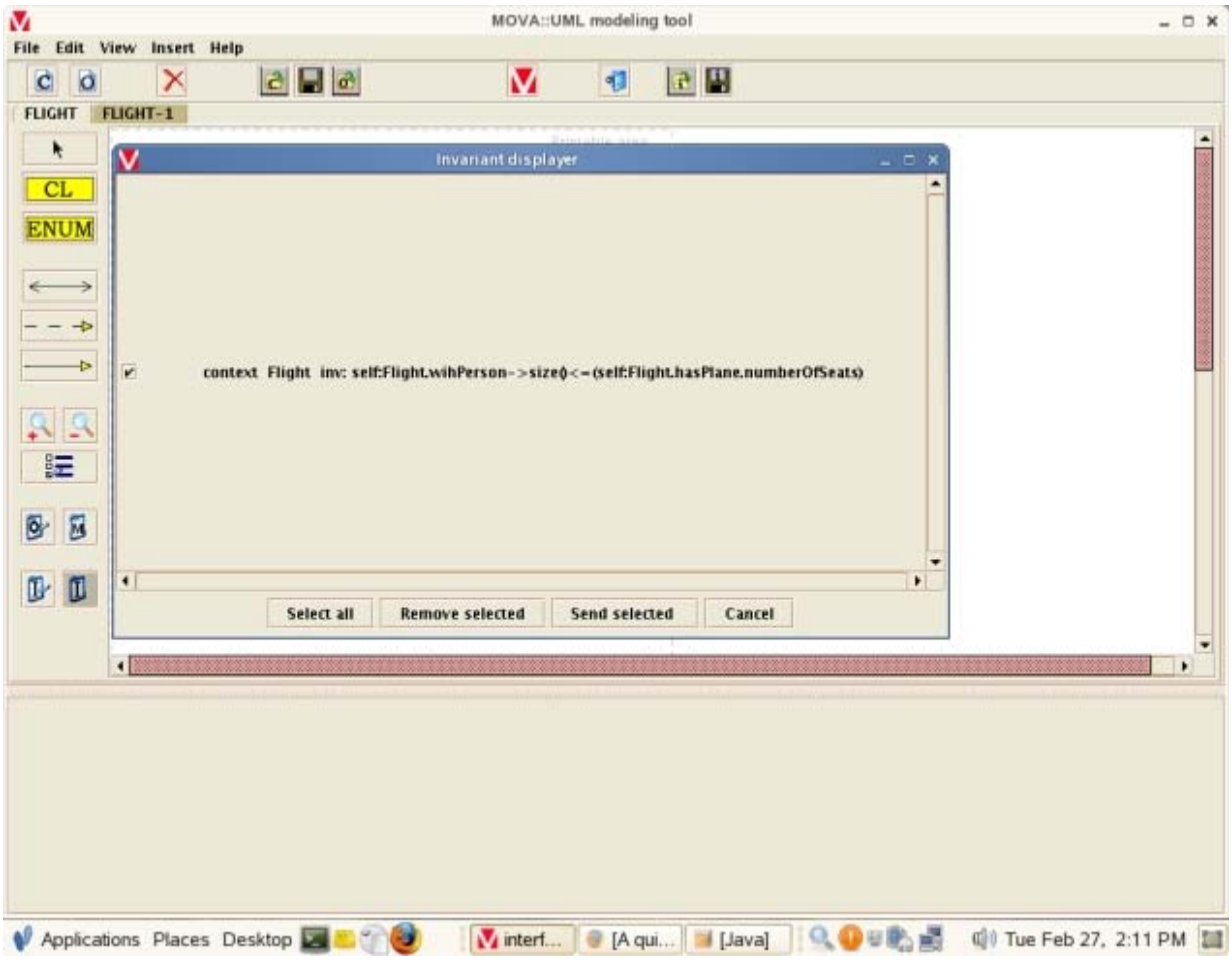


Figura 50

Consideramos ahora la siguiente instancia del modelo Flight (ver Figura 51):

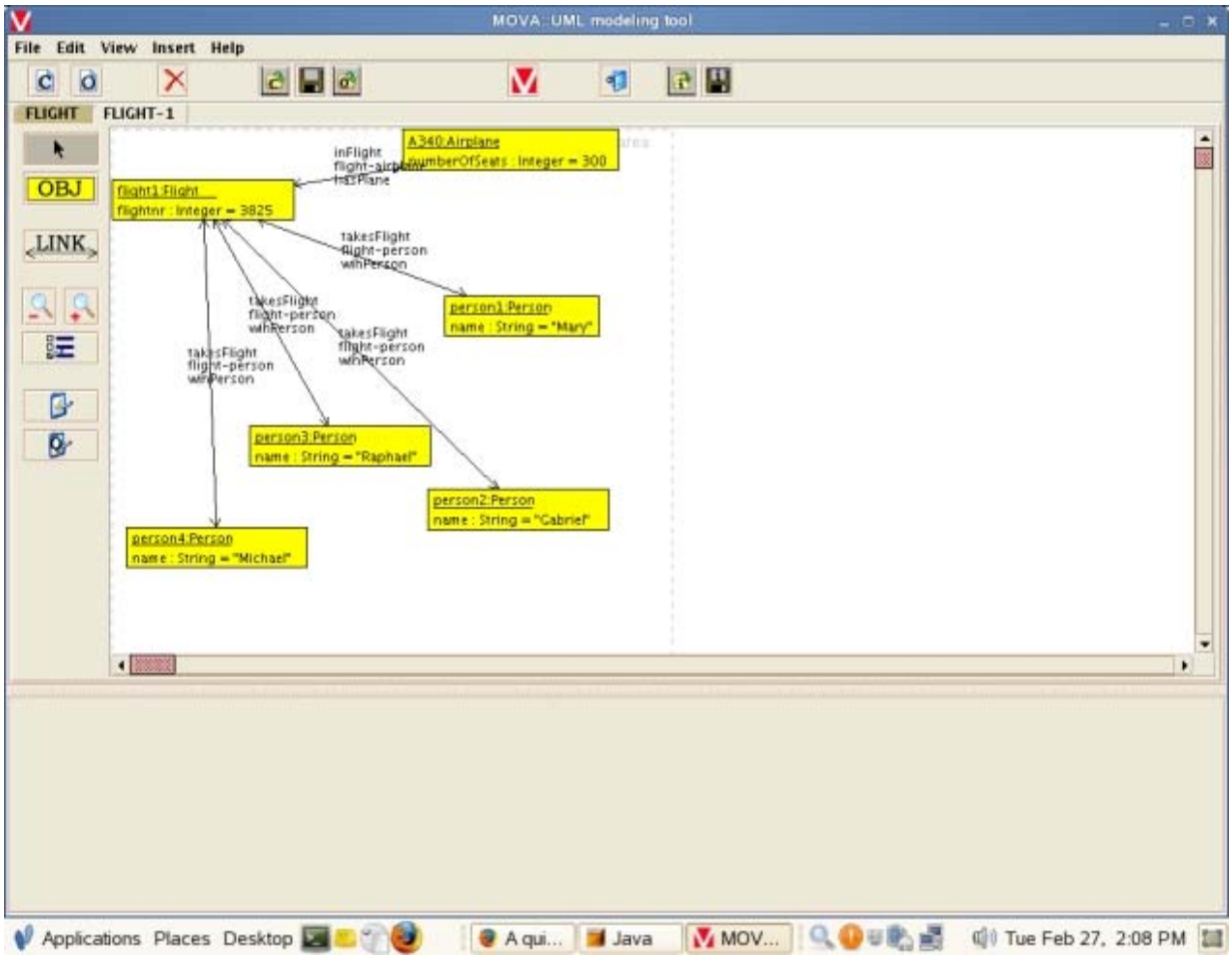


Figura 51

La invariante claramente está cumplida por esta instancia, ya que el número de asientos disponibles en el avión es de 300 y el número de pasajeros que reservaron el vuelo es 4. (Ver Figura 52)

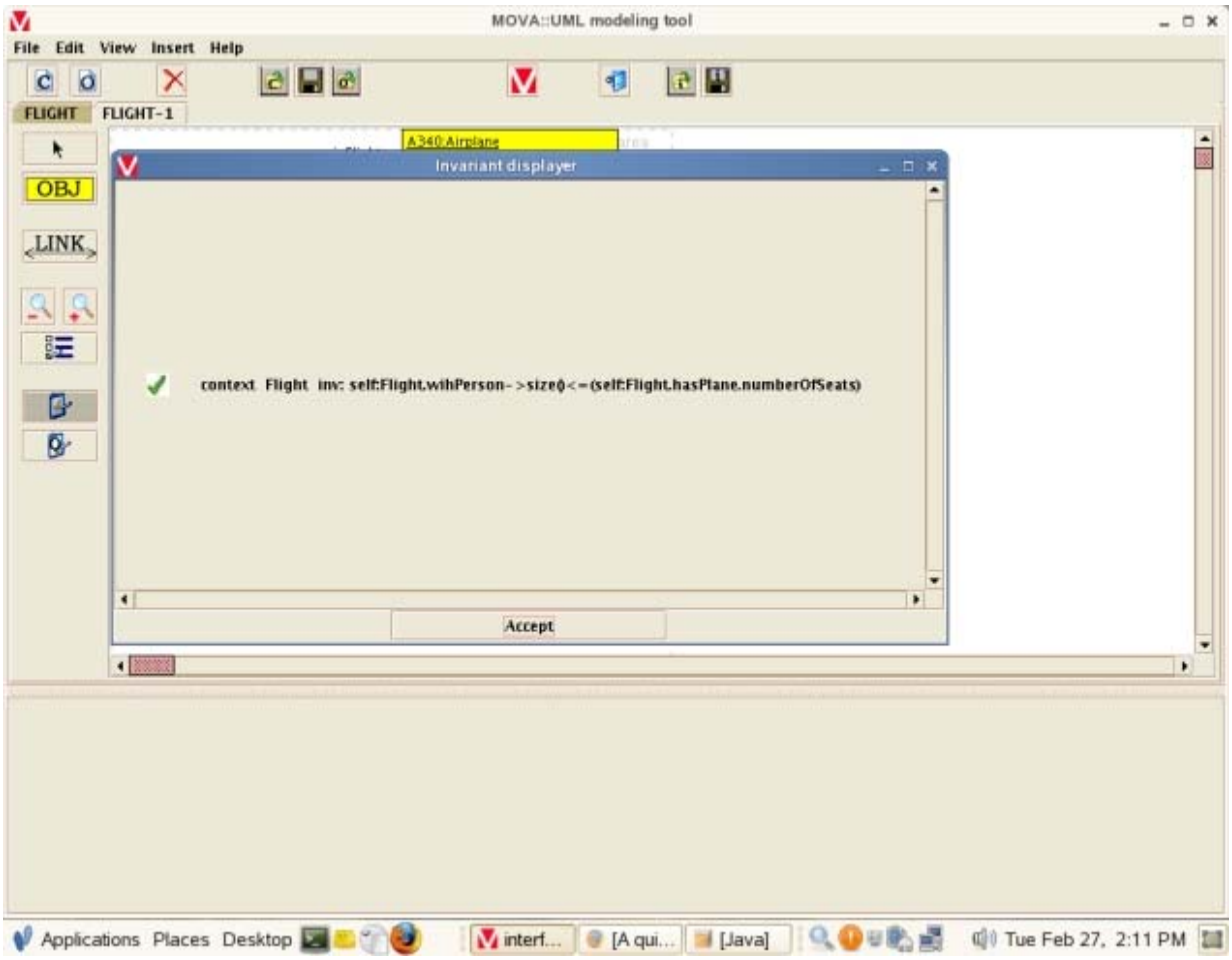

MOVA::UML modeling tool
File Edit View Insert Help
FLIGHT FLIGHT-1
A340:Airplane
OBJ
LINK
Invariant displayer
context Flight inv: self:Flight.wihPerson->size()<=(self:Flight.hasPlane.numberOfSeats)
Accept
Applications Places Desktop
interf...
[A qui...
[Java]
Tue Feb 27, 2:11 PM

Figura 52

EJEMPLO THE COACH-COMPANY MODEL:

En este ejemplo, tomado de *[Boronat, A., Ramos I., Carsí, J.Algebraic Semantics of Generic OCL Queries in the Eclipse Modeling Framework., 2006]*, modelamos una compañía de autobuses (ver Figura 53). Un autobús tiene un número específico de asientos y puede ser contratado para viajes normales o privados. En viajes normales, los billetes son comprados individualmente. En viajes privados, el autobús es alquilado entero.

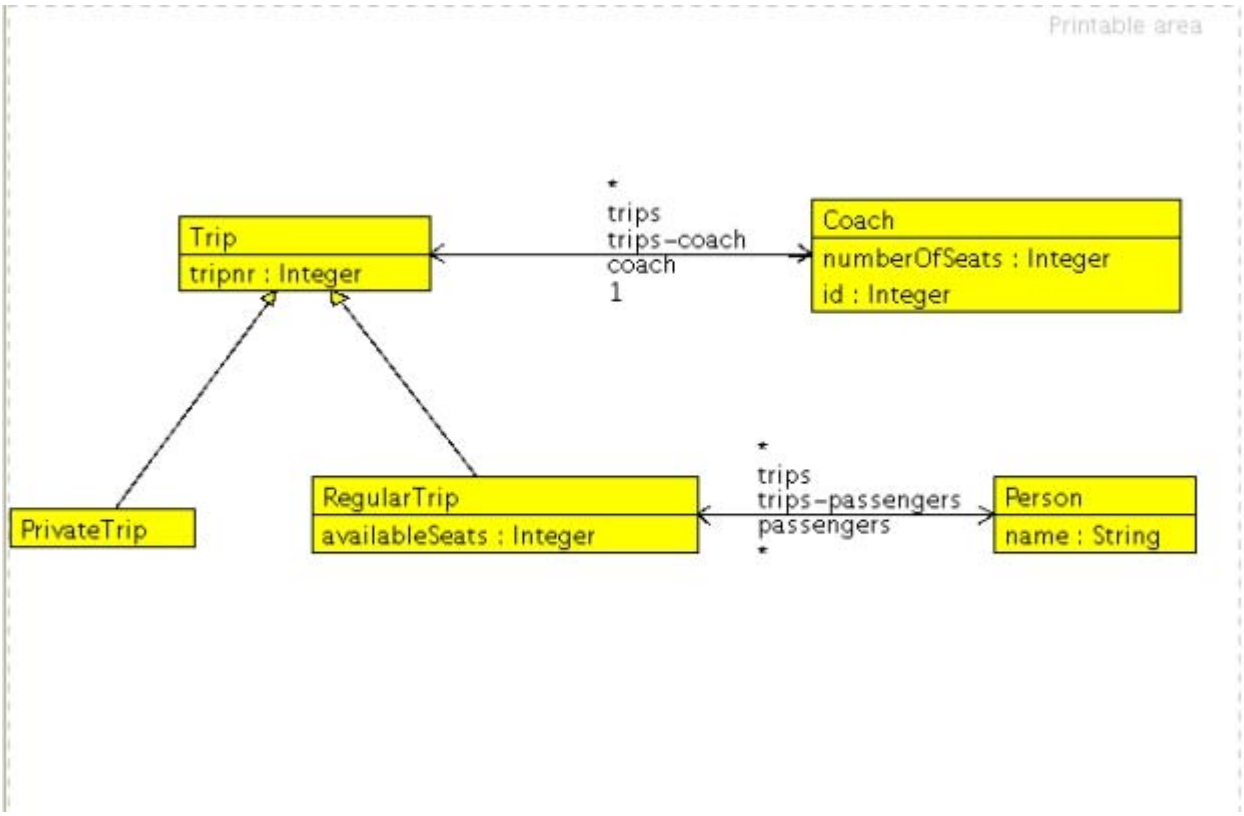


Figura 53

Usando restricciones OCL podemos precisar más exactamente nuestro modelo. En particular, indicamos que el “overbooking” no está permitido en un viaje normal.

context Coach inv:

self.trips->select(r | r.oclIsTypeOf(RegularTrip))->forAll(t |

t.oclAsType(RegularTrip).passengers->size() <= t.coach.numberOfSeats)

Esta invariante puede ser insertada en MOVA usando el editor OCL (ver Figura 54):

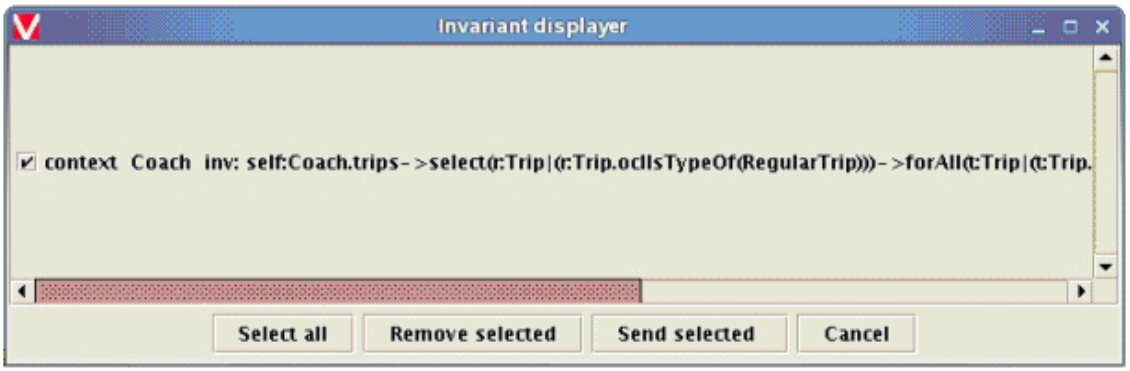


Figura 54

La instancia COACH-1 satisface la restricción sobre el “overbooking”.

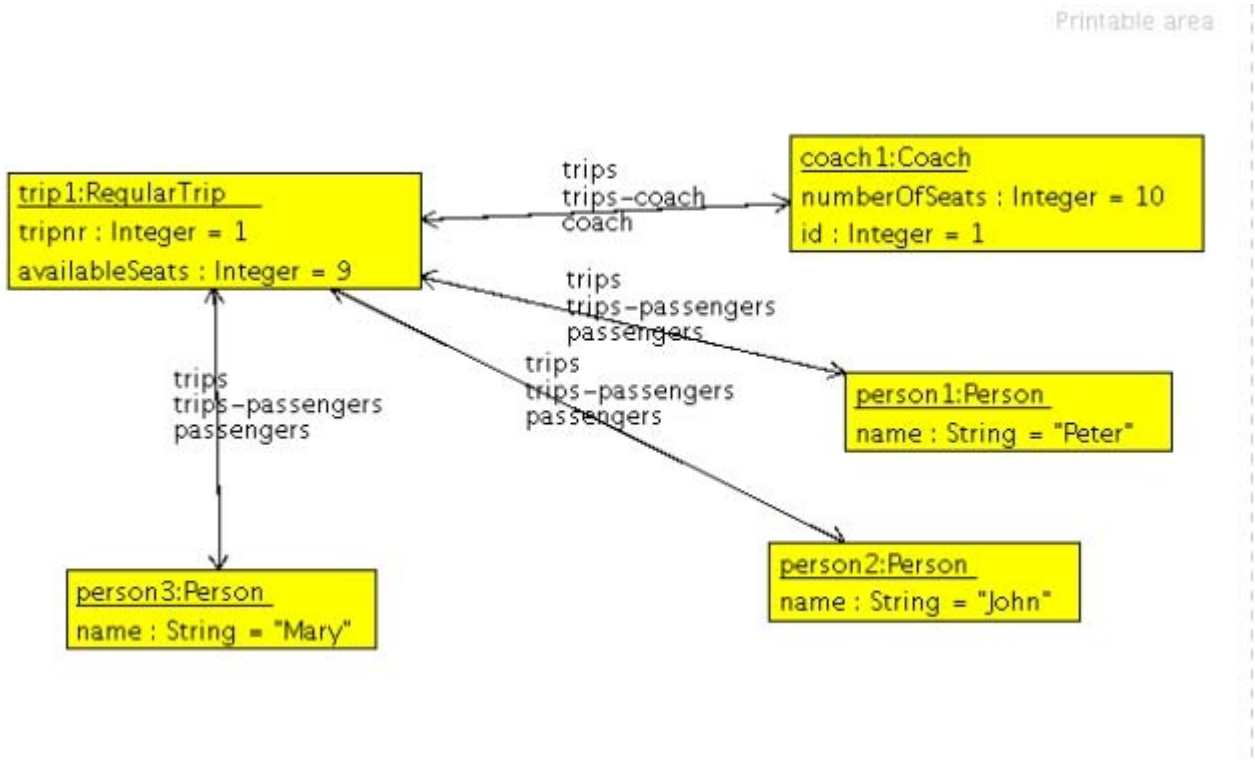


Figura 55

Esto puede comprobarse automáticamente usando MOVA. (Ver Figura 56)

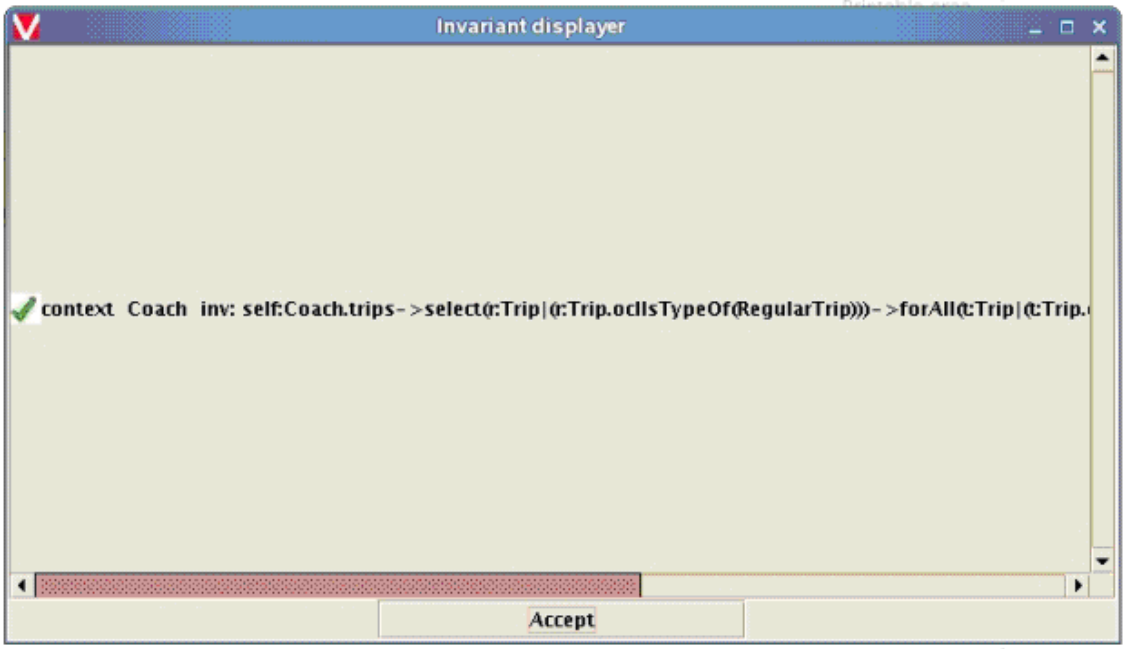

Invariant displayer
context Coach inv: self:Coach.trips->select(t:Trip|(t:Trip.oclIsTypeOf(RegularTrip)))->forAll(t:Trip|(t:Trip.
Accept

Figura 56

EJEMPLO THE TRAIN-WAGON MODEL:

Consideramos el siguiente diagrama de clases TRAIN-WAGON (ver Figura 57). Un tren puede tener varios vagones y éstos a su vez pueden estar conectados a otros vagones. Hay dos tipos de vagones: de fumadores o no fumadores.

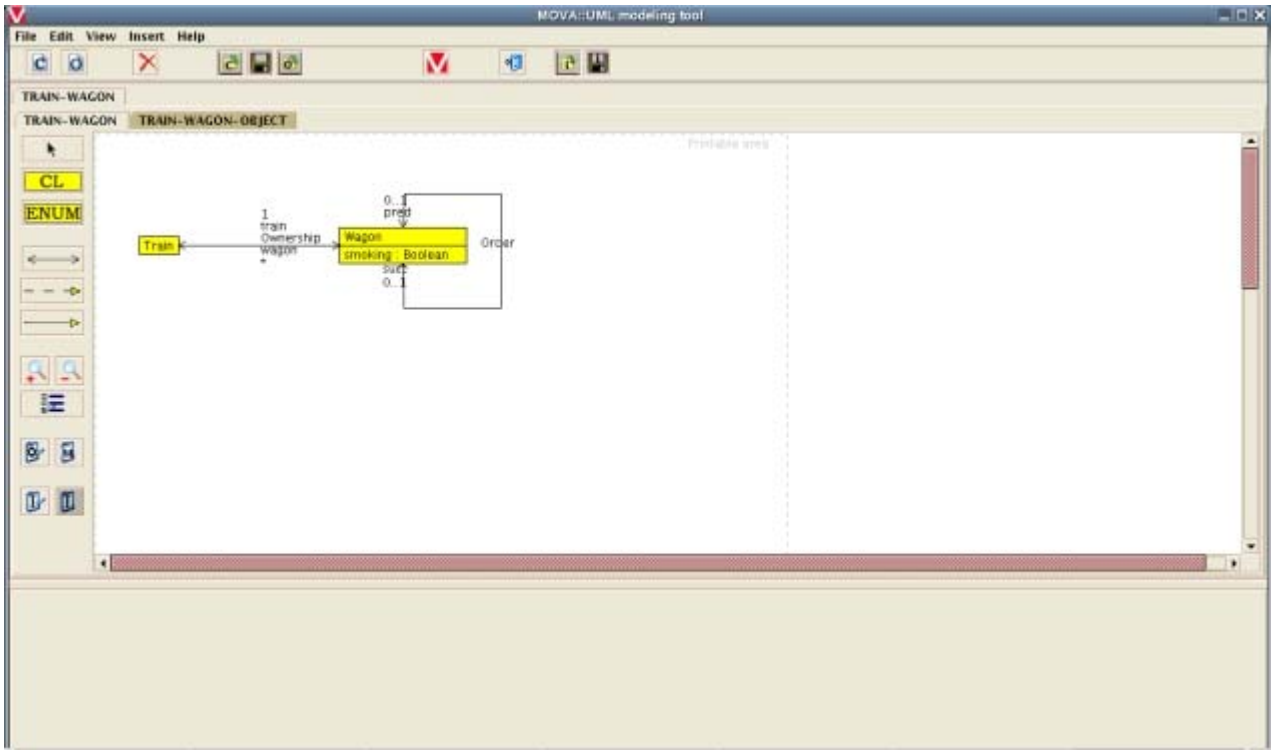


Figura 57

Consideremos las siguientes restricciones sobre el diagrama de clases TRAIN-WAGON:

Todos los trenes deben contener al menos un vagón.

context Train inv atLeastOneWagon:
self.wagon->size() >= 1

Un vagón y su sucesor deben pertenecer al mismo tren.

context Wagon inv belongToTheSameTrain:
self.succ->notEmpty() implies self.succ->forAll(w | w.train = self.train)

Todos los trenes deben tener el mismo número de vagones.

context Train inv sameNumberOfWagons:
Train.allInstances->forAll(t1 | (self.wagon->size() = t1.wagon->size())

No existen dos vagones distintos unidos formando un ciclo.

context Wagon inv notInCyclicWay:
(Wagon.allInstances)->forAll(w2 |
self <> w2 implies not ((self.succ)->includes(w2) and
(w2.succ)->includes(self))

La declaración de operaciones auxiliares es útil cuando la definición de la invariante se vuelve más compleja. Además estas operaciones están disponibles para que el usuario pueda seguir reutilizándolas.

La operación predPlusOnSet construye el cierre transitivo de los predecesores de un conjunto de vagones.

context Wagon
body: predPlusOnSet(s:Set(Wagon)):Set(Wagon) =
if s->collect(s1| s1.pred)->exists(w| s->excludes(w))
then(self.predPlusOnSet(s)->union(s->collect(s2|s2.pred)
else s fi

La operación predPlus consigue el cierre transitivo de los predecesores de un vagón aprovechándose de la operación predPlusOnSet.

context Wagon

body: predPlus : Set(Wagon) = self.predPlusOnSet(self.pred)

Una vez definidas estas operaciones, podemos definir una invariante estableciendo que no pueden existir dos vagones diferentes ligados el uno con el otro de manera cíclica.

context Train inv noCycles:
self.wagon->forAll(w|w.predPlus->excludes(w:Wagon))

Estas invariantes y operaciones pueden añadirse fácilmente al diagrama de clases TRAIN-WAGON usando el editor OCL. (Ver Figura 58)

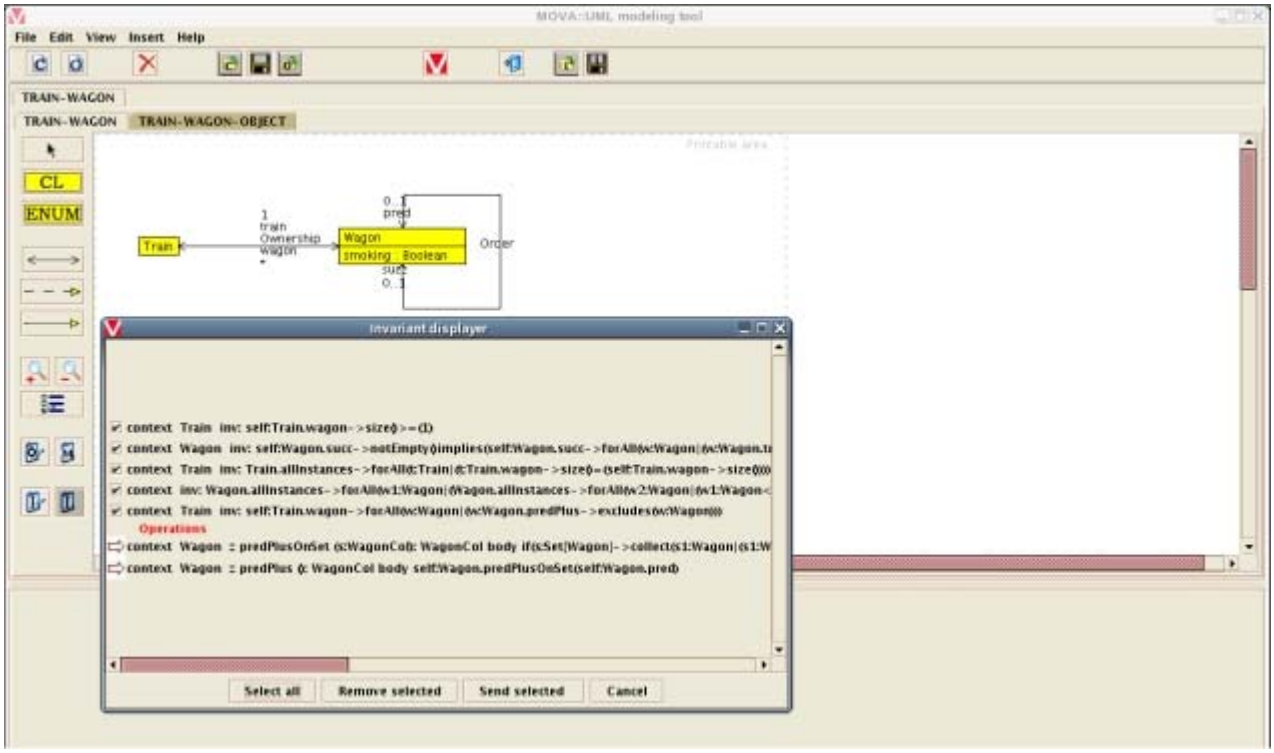


Figura 58

Consideramos ahora la siguiente instancia del diagrama de clases TRAIN-WAGON. En el ejemplo se muestra que train1 posee cinco vagones (w1, w2, w3,

w4, w5) unidos por links. Los vagones w2 y w4 están también unidos por un link formando un ciclo.

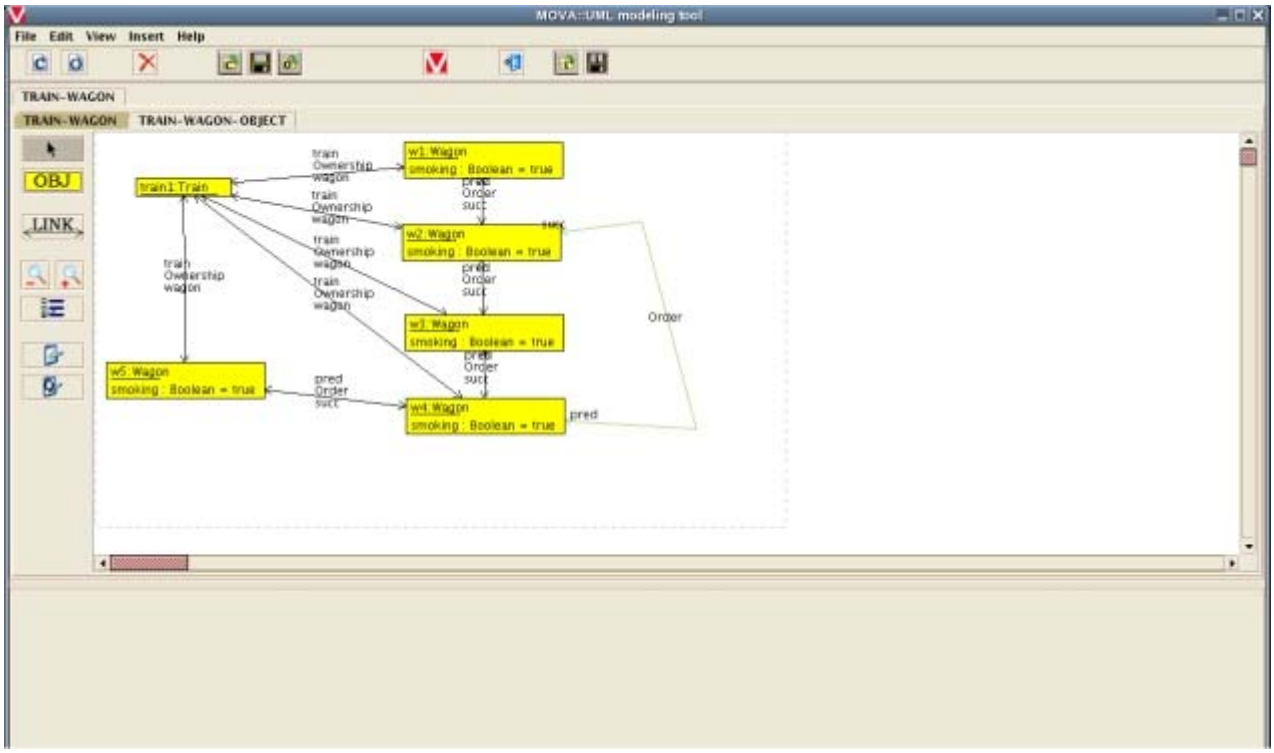


Figura 59

En este escenario, las invariantes atLeastOneWagon, notInCyclicWay, belongToTheSameTrain y sameNumberOfWagons se satisfacen. Sin embargo la invariante noCycles no se mantiene. (Ver Figura 60)

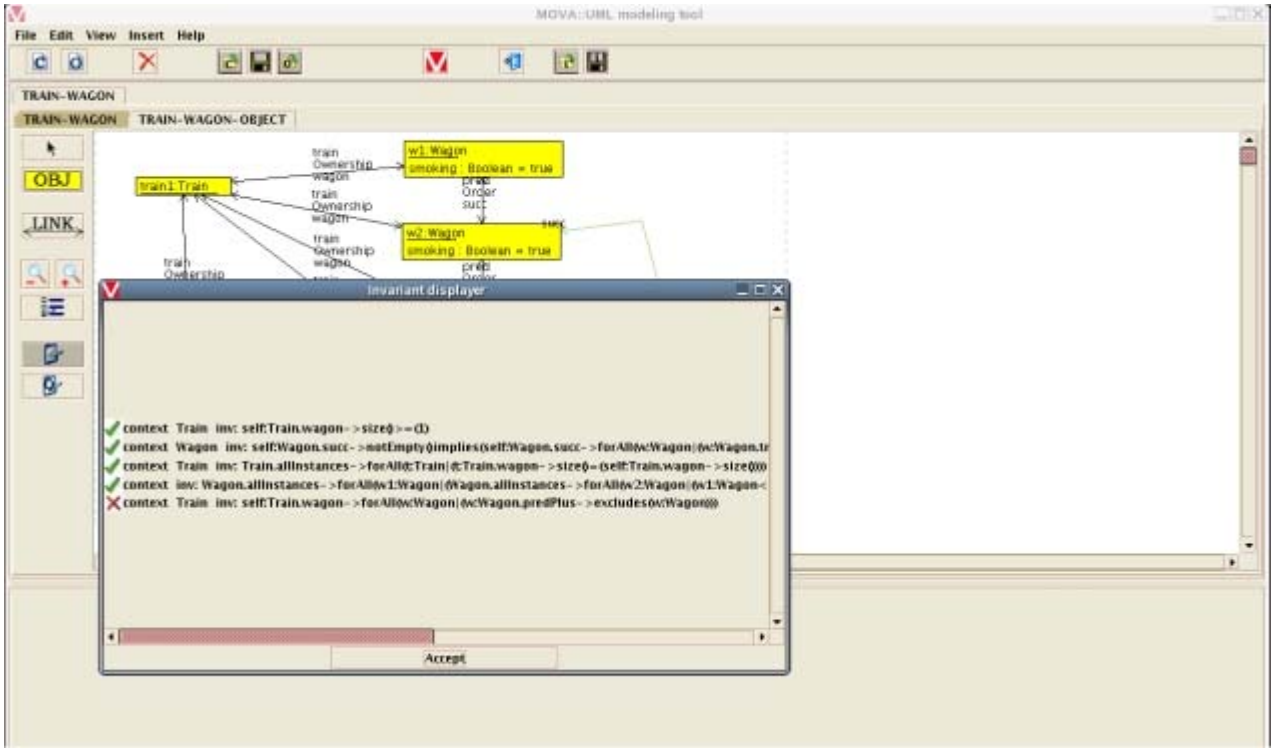

MOVA: UML modeling tool
File Edit View Insert Help
TRAIN-WAGON
TRAIN-WAGON
TRAIN-WAGON-OBJECT
OBJ
LINK
train1:Train
w1:Wagon
smoking : Boolean = true
w2:Wagon
smoking : Boolean = true
train
Ownership
wagon
pred
Order
succ
Invariant displayer
context Train inv: self:Train.wagon->size()>=(1)
context Wagon inv: self:Wagon.succ->notEmpty()implies(self:Wagon.succ->forAll(wc:Wagon|(wc:Wagon.tr
context Train inv: Train.allInstances->forAll(t:Train|(t:Train.wagon->size()=(self:Train.wagon->size()))
context inv: Wagon.allInstances->forAll(w1:Wagon|(Wagon.allInstances->forAll(w2:Wagon|(w1:Wagon=
context Train inv: self:Train.wagon->forAll(wc:Wagon|(wc:Wagon.predPlus->excludes(wc:Wagon)))
Accept

Figura 60

7. HISTORIAL DE REUNIONES

17-octubre-2006... 30-octubre-2006 ⇨

- Instalar NetBeans 5.0 para Windows.
- Conectar con el CVS, y bajar el proyecto.
- Mirar el código, y entenderlo.
- Crear un menú con distintas opciones, en el que podamos elegir lo que queramos que se haga visible en el diagrama de clases: atributos, roles, multiplicidades, o nombre de las relaciones.

31-octubre-2006… 6-noviembre-2006 ⇨

- Crear un menú similar al anterior para poder seleccionar lo que queremos que se haga visible en el diagrama de objetos: atributos, roles o nombres de relaciones.
- Seleccionar un conjunto de elementos de un diagrama para poder pegarlos en otros diagramas.
- Imprimir diagrama.

6-noviembre-2006… 13-noviembre-2006 ⇨

- Márgenes imprimir.
- Copiar, pegar elementos de un diagrama de clases en otros diagramas.
- Modificar las ventanas de invariantes DisplayerWindow, y CheckingWindow.

13-noviembre-2006… 20-noviembre-2006 ⇨

- Poner los nombres de clases, métodos… en inglés.
- Terminar lo de copiar-pegar.
- Terminar ventanas invariantes.
- Meter todos los cambios hasta el momento en visual-ocl.

20-noviembre-2006… 27-noviembre-2006 ⇨

- Opción imprimir diagrama de objetos.

- Mandar comandos a Maude cada vez que se vayan a copiar y pegar elementos.
- Ventanas invariantes: poner barras desplazamiento.

27-noviembre-2006… 4-diciembre-2006 ⇨

- Ventana invariantes: añadir dos botones:
    - Select all, que selecciona todos los invariantes
    - Remove selected, que elimina los invariantes seleccionados.
- Mover elementos del diagrama:
    - No permitir que se muevan elementos fuera del área visible del panel.
    - Evitar que las relaciones se deformen.
    - Arreglar que se puedan mover correctamente los elementos tras cargar un diagrama.
- Copiar – pegar elemento del diagrama.
- Ojear “Role-Based Access Control: RBAC”.

4-diciembre-2006…11-diciembre-2006 ⇨

- Ventana invariantes:
    - Mantener seleccionados los invariantes, hasta que el usuario indique lo contrario.
    - No chequear las operaciones, distinguir entre invariante y operación. En la ventana DisplayerWindow que se vean las operaciones, pero que no se puedan seleccionar. Y en la ventana ChekingWindow que no se vean.
- Mover elementos del diagrama:
    - Permitir que se pueda mover elementos fuera del área visible del panel para que este pueda aumentar su tamaño.
- Ventana propiedades de clases y objetos
    - Cambiar la ventana de propiedades para que no salga el mensaje de error debido al nombre:
        - Añadir un botón que sirva para modificar el nombre.

- Quitar los botones de aplicar y aceptar. El botón cancelar sencillamente oculta la ventana.

11-diciembre-2006…18-diciembre-2006 ⇨

- Ventana invariantes:
  - Guardar las invariantes seleccionadas por el usuario en un archivo de texto.
- Menú opciones de visualización:
  - Permitir que se puedan modificar las opciones de visualización de los diagramas de objetos desde el menú principal.
- Cambiar el resto de menús de propiedades de clases, objetos y relaciones.
- Hacer documento explicando como se guarda la información de los diagramas.
- Cortar, copiar, pegar y eliminar elementos de diagramas de clases y diagramas de objetos.
- Evento clic del botón derecho del ratón (opciones cortar, copiar, pegar y propiedades).

18-diciembre-2006…8-enero-2007 ⇨

- Crear diagrama de seguridad con roles, entidades, usuarios y permisos.
- Realizar un rediseño de las clases ClassDiagramGUI, ObjectDiagramGUI y SecurityDiagramGUI para reutilizar código común.
- Permitir cambiar propiedades de un diagrama haciendo doble clic sobre su etiqueta.

8-enero-2007…15-enero-2007 ⇨

- Diagrama de seguridad:
  - Poner estereotipos en los roles, usuarios, entidades y permisos.
  - Hacer icono para la relación entre un rol y un usuario.

- Al crear un rol, usuario, entidad o permiso, pedir antes el nombre de éstos.
- Permitir crear relaciones entre:
  - dos roles,
  - un usuario y un rol,
  - dos entidades.

15-enero-2007…22-enero-2007 ⇨

- Diagrama de seguridad:
  - Hacer una ventana de propiedades para las entidades, al hacer doble clic sobre ella. Esta ventana tiene que contener:
    - Nombre de la entidad con opción de poder modificarse.
    - Opción de insertar atributos.
    - Opción de insertar métodos.
  - Asociarle al diagrama de seguridad, el diagrama de clases SEC+COMP_META
  - Permitir crear relaciones entre un rol, permiso y entidad, y asignarle permisos a un rol para realizar acciones sobre una entidad.

22-enero-2007…5-febrero-2007 ⇨

- Hacer una ventana inicial que pregunte con qué tipo de herramienta queremos trabajar:
  - UML modeling
  - SECURE modeling
- Terminar las relaciones Role-Permission-Entity.

5-febrero-2007…19-febrero-2007 ⇨

- Diagrama de clases: últimos cambios para la publicación de la herramienta:
  - Añadir en el menú insert, un JMenuItem para la asociación que faltaba.

- Añadir en el menú file, la opción “close” que cierra el diagrama actual.
- Hacer que funcione el load invariants del menú “file”.
- Añadir en el menú “view” un JMenuItem para que se vean los invariantes.
- Añadir en el submenú “saveAs”, JMenuItem para guardar diagrama e invariantes y operaciones.
- Desactivar el botón eliminar diagrama cuando no hay ninguno creado.

19- febrero -2007…26- febrero -2007 ⇨

- Hacer diagramas, para ver qué patrones y técnicas de IS se pueden aplicar a la herramienta.
- Guardar y cargar roles, entidades, usuarios y permisos.

26- febrero -2007…5- marzo -2007 ⇨

- Paquete dibujables: Pensar cómo aplicar patrón para que sea más sencillo crear nuevos elementos.
- Guardar y cargar relaciones role-role, entity-entity, role-user y role-permission-entity.

5- marzo -2007…12- marzo -2007 ⇨

- Diagrama de clases: Invariantes: Distinguir un vector de invariantes y otro de multiplicidades, para que en la ventana que muestra los invariantes aparezcan separados.
- Diagrama de seguridad: Hacer los iconos de relaciones, y los botones de guardar y abrir diagrama.
- Diagrama de métricas: En el menú, añadir opción de insertar métricas.
- Hacer diagrama de clases, y ver lo común y no común de los 3 tipos de diagramas, para ver cómo se pueden reorganizar extrayendo parte grafica y lógica.

- Realizar el nuevo diseño para el paquete dibujables usando un patrón Composite.
- Guardar y cargar atributos y métodos.

12- marzo -2007…19- marzo -2007 ⇨

- Diagrama de seguridad:
  - Hacer los iconos de relaciones, al ser seleccionados.
  - Hacer el menú “insert”.
- Diagrama de métricas:
  - Añadir al menú “insert”, insertar métricas
- Empezar implementación del nuevo paquete dibujables.
- Guardar y cargar acciones.

19- marzo -2007…26- marzo -2007 ⇨

- Diagrama de seguridad:
  - Quitar botón check all invariants.
  - Quitar el envío de comando a Maude al insertar, borrar o modificar un elemento. Los comandos se mandan al pinchar en el botón Query.
- Continuar implementación del nuevo paquete dibujables.

26- marzo -2007 – 16- abril -2007 ⇨

- Diagrama de seguridad:
  - Al pinchar sobre el icono  y sobre un permiso que pertenezca a una relación con un rol, y una entidad, debe aparecer un editor de texto, de manera que se añada la cadena introducida como restricción de la relación.
- Memoria proyecto.

16- abril -2007 – 4 - mayo -2007 ⇨

- Diagrama de seguridad:

    - El atributo creado en la reunión anterior, debe poder moverse.
- Memoria proyecto.

8. BIBLIOGRAFÍA

- *Design Authorization Systems Using SecureUML*

Rudolph Araujo & Shanit Gupta, Foundstone Professional Services

February 2005

http://www.foundstone.com/us/resources/whitepapers/wp_secureuml.pdf

- *MOVA 0.3.1 user manual*

MOVA group

February 2007

http://maude.sip.ucm.es/~clavel/pubs/user-manual-0.3.1.pdf

Los alumnos Rocío Prieto Ruiz, Belén Rodríguez Alonso, y Álvaro Suárez Bravo autorizan a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria como el código, la documentación y/o el prototipo desarrollado.

Firmado:

Rocío Prieto Belén Rodríguez Álvaro Suárez