UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Título: Un metalenguaje de programación orientado al diseño de interfaces gráficas
Autores: Carlos Ariel Santana / Cintia Vanesa Coni
Director: Prof. Dra. Claudia Fabiana Pons
Carrera: Licenciatura en Sistemas

Resumen

En la actualidad, la mayoría de las herramientas para generar código a partir de un modelo no abarcan la creación de interfaces gráficas, y si lo hacen, sólo crean la típica interfaz tipo CRUD asociando un control a cada propiedad del modelo.
Las interfaces de usuario cumplen un rol fundamental en la usabilidad y el éxito de las aplicaciones. Las mismas han evolucionado enormemente en los últimos años, y en la actualidad se cuenta con interfaces sofisticadas, que intentan reducir las distancias tecnológicas entre el usuario y las computadoras. Diseñar interfaces que satisfagan correctamente las necesidades del usuario no es sencillo, y por otro lado, posee costos de investigación y aprendizaje de cada tecnología a utilizar.
La presente tesina propone una solución basada en MDA para la generación automática de aplicaciones a partir de un modelo de entrada definido en un meta-lenguaje específico. Este meta-lenguaje permite definir de manera completa la apariencia y el comportamiento de las interfaces de la aplicación. Dicho modelo de entrada, podrá luego ser transformado a código ejecutable en diversos lenguajes y plataformas obteniendo aplicaciones listas para ser utilizadas.

Palabras Claves

Arquitectura dirigida por modelos
Transformaciones de modelos
Interfaz de usuario

Trabajos Realizados

Hemos desarrollado un lenguaje específico de dominio, GuiDSL, enfocado a la definición de interfaces de usuario. Para esto hemos utilizado el framework Xtext, el cual facilita la especificación del DSL y las transformaciones.

Conclusiones

A diferencia de la mayoría de las herramientas basadas en MDE, GuiDSL permite definir una aplicación completa, incluyendo la apariencia y comportamiento de sus interfaces de usuario. Esta aplicación definida en GuiDSL, podrá ser transformada a múltiples lenguajes y plataformas, obteniendo aplicaciones funcionales completas, sin necesidad de adaptar o modificar el código generado. GuiDSL podrá ser extendido fácilmente, permitiendo acompañar los avances tecnológicos que surgen constantemente y adaptarse ante nuevas necesidades. Es así como GuiDSL se destaca y realiza su aporte al mundo MDA.

Trabajos Futuros

Desarrollo de un editor gráfico para la definición del modelo y el diseño de sus interfaces.
Ampliación del DSL incorporando controles de usuario, extendiendo el lenguaje y otras facilidades de diseño.
Incorporación de nuevas transformaciones para otros lenguajes.

Fecha de la presentación: NOVIEMBRE 2011

Tesina de Licenciatura

Un metalenguaje de programación

orientado al diseño de interfaces gráficas.

Carlos Ariel Santana / Cintia Vanesa Coni

Director: Prof. Dra. Claudia Fabiana Pons

Noviembre de 2011

Agradecimientos

Agradecimientos de Cintia

A mis viejos, que me dieron la posibilidad de estudiar.

Y a Ariel, mi amigo, por haber confiado en mí y darme la oportunidad de compartir esta experiencia, de quien aprendí muchísimo, de su generosidad y perseverancia, y a quien admiro profundamente.

Amo mi profesión, disfruto a diario de lo que hago, y estoy feliz de recibirme!

Agradecimientos de Ariel

A Vale, mi familia y amigos que me acompañaron en todos estos años de estudio y, en especial, a mi amiga Cintia por compartir este proceso final junto a mí.

Agradecimientos compartidos

A Claudia, por ayudarnos, motivarnos y guiarnos.

A nuestro amigo Ariel P., por acompañarnos incondicionalmente y compartirnos toda su experiencia.

A los docentes de la Facultad de Informática que tanto nos enseñaron.

Índice

Índice de figuras

Lista de Bloques de código

Parte I
Introducción

En la parte I se presenta el objetivo de la tesina, los motivos que dan origen a la herramienta, exponiendo la importancia de las interfaces gráficas, lo costoso de su implementación y los inconvenientes más comunes. Se presenta el paradigma de la ingeniería dirigida por modelos, las iniciativas más importantes, y se describen brevemente algunas herramientas MDA. Se plantean los importantes beneficios de este paradigma, y los inconvenientes que hacen que aún no se encuentre masivamente aplicado en la ingeniería de software.

Se presenta también un resumen con la estructura global de la tesina.

1. Introducción

Las interfaces gráficas de usuario constituyen hoy en día uno de los factores más relevantes a la hora del diseño y el desarrollo de un sistema para garantizar la aceptación de los usuarios. Un buen diseño de interfaz requiere tiempo, esfuerzo y conocimiento acerca de buenas prácticas de diseño centrado en el usuario, accesibilidad y facilidad de uso, cuestiones en las cuales muchas veces los programadores de sistemas no se encuentran familiarizados o no se sienten interesados.

Debido a la diversidad de plataformas en la cuales operan los sistemas, como consola de texto, interfaz gráfica de escritorio, interfaz web y las más recientes, interfaces para dispositivos móviles, surge constantemente la necesidad de adaptación o migración de plataformas o lenguajes para un mismo sistema [UJN]. Esto conlleva a invertir nuevamente los costos de diseñar la nueva interfaz de usuario.

En la actualidad, existen diversas herramientas para generar código a partir de un modelo, pero no abarcan la creación de interfaces gráficas, y si lo hacen, solo crean la típica interfaz tipo CRUD (Create-Read-Update-Delete), asociando un control a cada propiedad de las entidades del modelo. Estas herramientas no ofrecen la posibilidad de definir en el modelo ninguna característica de presentación ni de comportamiento de las interfaces de usuario, por ejemplo, definir el color y posición de un botón, o indicar que acción realizar al perder foco un control.

1.1. Objetivo

El objetivo de la tesina es proponer una herramienta basada en MDA (Model Driven Arquitecture) para la generación automática de aplicaciones junto con sus interfaces de usuario, en diversos lenguajes y plataformas, minimizando así los costos de adaptación o cambios de tecnología.

Los principales aspectos de la solución incluyen:

- Un metalenguaje que permite definir de manera íntegra la aplicación a generar. Además de permitir definir las entidades del modelo y sus características, permite también incorporar código y la definición de las interfaces gráficas de la aplicación especificando su apariencia y comportamiento.
- Generación completa del código, resultando una aplicación lista para ser utilizada.
- Reutilización del modelo de entrada: el mismo modelo podrá generarse en diversos lenguajes y plataformas sin necesidad de adaptación alguna.
- Posibilidad de extender las transformaciones libremente cuando sea necesario incorporar otro lenguaje, *framework* o plataforma. También podrá extenderse el metalenguaje para incorporar nuevas características como sentencias, tipos de datos o controles visuales; aportando así máxima flexibilidad.

1.2. Organización de la tesina

La presente tesina se organiza en 4 partes:

Parte I

En la parte I se presentan los motivos que dan origen a la herramienta, exponiendo la importancia de las interfaces gráficas, lo costoso de su implementación y los inconvenientes más comunes. Luego, se presenta el paradigma de la ingeniería dirigida por modelos, las iniciativas más importantes, y se describen brevemente algunas herramientas MDA. Se plantean los importantes beneficios de este paradigma, y los inconvenientes que hacen que aún no se encuentre masivamente aplicado en la ingeniería de software.

Parte II

En esta parte se analiza la arquitectura de las interfaces de usuario, desde su implementación a bajo nivel (comunicación del hardware con el sistema operativo), hasta los *frameworks* de alto nivel más recientes. También se analizan los detalles de implementación de acuerdo a las distintas plataformas donde se ejecutan las aplicaciones (escritorio, web, dispositivos móviles, etc).

Luego del análisis, se concluye en cuáles son las características comunes a todas las plataformas, tecnologías y lenguajes, las cuales constituirán un gran aporte al diseño del lenguaje de la herramienta.

Parte III

En la parte III se presenta la herramienta GuiDSL: sus características y beneficios principales, se describe su lenguaje, desde el punto de vista sintáctico y semántico, y se especifica cada uno de los elementos del lenguaje. Luego se explica como GuiDSL utiliza las transformaciones para obtener el código fuente de la aplicación. Se muestra cómo utilizar la herramienta con un ejemplo sencillo, abarcando todos los pasos del proceso: desde la creación del modelo de la aplicación, hasta ejecutar el código generado. Se presentan los detalles de implementación más importantes de la herramienta. Se explican las herramientas utilizadas para su desarrollo, cómo se implementan los chequeos del lenguaje y la arquitectura de las transformaciones. Finalmente se muestra cómo es posible extender la herramienta, tanto el lenguaje como las transformaciones, y se explica conceptualmente cómo hacerlo.

Parte IV

En la parte IV se realiza un resumen de los contenidos más importantes vistos a lo largo de la tesina, se presentan trabajos relacionados, y conclusiones y aportes de la herramienta desarrollada. Finalmente se exponen posibles líneas de trabajo sobre la investigación y la herramienta, para quienes estén interesados en contribuir al área de estudio de la tesina.

2. Motivación

En este capítulo se detallarán los motivos que dan origen a la herramienta, exponiendo los problemas más comunes a la hora de implementar las interfaces de usuario, y destacando la gran importancia de éstas en el éxito de una aplicación.

2.1. Importancia de la interfaz gráfica en el proceso de desarrollo

El usuario sólo interactúa con la interfaz de las aplicaciones, e ignora cuestiones de diseño o implementación. En una aplicación que aún no dispone de una interfaz para interactuar, o la misma no ofrece los medios correctos para accederla, el proceso de desarrollo, su complejidad y la calidad de su diseño, se ve inutilizado. La interfaz es la única parte de la aplicación que permite explotar su funcionalidad, es el único medio del usuario para poder utilizarla y debe ofrecer los medios correctos para ello. Si la aplicación tiene una característica que no puede ser accedida por el usuario, simplemente es como si no la tuviera.

Por esto, la importancia de un óptimo diseño de las interfaces, accesibilidad, usabilidad y simpleza, son determinantes para el éxito o fracaso de una aplicación. Jakob Nielsen, definió la usabilidad en el 2003 como *"un atributo de calidad que mide lo fáciles de usar que son las interfaces web"* [UJN] [USA]. Además describe que se debe lograr la mayor satisfacción y mejor experiencia de uso posible con el mínimo esfuerzo por parte del usuario. Para ello, se requiere conocer técnicas multidisciplinarias donde cada decisión tomada debe estar basada en las necesidades, objetivos, expectativas, motivaciones y capacidades de los usuarios.

Por otro lado, lograr una correcta combinación de estas características, sumada la necesidad de cambio constante de tecnología, hacen que esta tarea requiera tiempo, esfuerzo y experiencia.

2.2. Inconvenientes más comunes en el desarrollo de interfaces gráficas

Los inconvenientes más comunes en el desarrollo de las interfaces gráficas son: la necesidad de portabilidad, la dificultad del diseño y el costo de implementación.

2.2.1. Necesidad de portabilidad

La industria del software tiene una característica especial que la diferencia de las otras industrias: surgen constantemente nuevas tecnologías que rápidamente llegan a ser populares. Muchas compañías necesitan adoptar estas nuevas tecnologías por varias razones:

- Los clientes las exigen, por ejemplo servicios web.
- Incursión en nuevos mercados, por ejemplo versiones para dispositivos móviles.
- La empresa que desarrolla la herramienta deja de soportar las viejas tecnologías y se centra en las nuevas.
- La migración de una aplicación de un lenguaje “obsoleto” a uno “moderno”.
- Necesidad de integración o interacción con otras aplicaciones implementadas con tecnología no compatible, por ejemplo, utilizar XML para el intercambio de información entre plataformas heterogéneas.

Las nuevas tecnologías ofrecen beneficios concretos para las compañías y muchas de ellas no pueden quedarse atrás, por lo tanto, tienen que adoptarlas rápidamente. El software ya existente puede seguir sin cambios, pero necesitará comunicarse con las nuevas aplicaciones que serán construidas usando una nueva tecnología.

Por otro lado, ante la necesidad de un cambio de tecnología o plataforma, todo el trabajo y tiempo invertido en el diseño e implementación de una interfaz adecuada para una aplicación, queda inutilizado, ya que la misma debe ser reimplementada desde el comienzo, dado que su diseño se encuentra altamente acoplado al lenguaje de programación.

Las actuales herramientas de generación de código no contemplan la generación de interfaces gráficas, con la importancia que éstas tienen sobre la percepción del usuario. Sólo crea interfaces básicas que permiten manipular los datos del modelo (Alta/Modificación/Baja/Consulta) y no abarcan los aspectos más importantes como son la interacción de los controles y su lógica de presentación, la aplicación de las reglas de negocio, interacción con otras aplicaciones, manipulación de archivos, etc. Estos aspectos deben ser programados manualmente.

2.2.2. Dificultad para el diseño visual de la interfaz

La necesidad de diseñar interfaces intuitivas y atractivas para el usuario, requiere que el programador deba sumergirse en cuestiones meramente de diseño visual como manejar estilos, incorporar librerías externas de controles y centrar el diseño en el usuario. Los programadores no son, en general, diseñadores gráficos y no están familiarizados con estas cuestiones, las cuales hacen más difícil la capacidad de abstracción hacia cuestiones funcionales.

2.2.3. Costo tecnológico de implementación de las interfaces

Durante el proceso de implementación de las interfaces, el programador se ve obligado a enfocarse en aspectos tecnológicos particulares de un lenguaje de programación: debe aprender a definir controles, eventos y lógica en un lenguaje en particular. Muchas veces se requiere el uso de librerías de controles externas, las cuales ofrecen controles y funcionalidad más sofisticada, pero que requieren tiempo para aprender su funcionamiento y utilización. Por otro lado, en la actualidad, y sobre todo para plataformas web, los IDEs no cuentan con editores gráficos de interfaces suficientemente útiles, por lo cual el programador debe implementar la interfaz sin ninguna herramienta visual. Todas estas cuestiones hacen que la implementación de interfaces gráficas no sea una tarea sencilla, requiere invertir tiempo en investigación y poseer experiencia con herramientas cada vez más complejas.

2.2.4. Desestimación de costos en la planificación

El esfuerzo que demanda el diseño e implementación de la interfaz es, en general, representativo dentro de todo el proceso de desarrollo de la aplicación. En muchas ocasiones se minimiza esta etapa del desarrollo sin tener en cuenta su rol clave en el éxito de la aplicación. Planificaciones imprecisas sobre los costos de tiempo y recursos hacen peligrar el correcto diseño de la interfaz, o deja fuera cuestiones fundamentales que hacen a la buena experiencia del uso de la aplicación.

2.3. Conclusión

La interfaz de usuario es la "cara visible" de la aplicación, y muchas veces, es determinante para el éxito y aceptación de ésta.

Se destacó lo costoso que es diseñar interfaces que satisfagan correctamente las necesidades del usuario, y por otro lado, los costos de investigación y aprendizaje de cada tecnología a utilizar.

Sería de suma utilidad, contar con una herramienta que permita minimizar estos inconvenientes y ayude a agilizar y mejorar el proceso de implementación de las interfaces gráficas. Esta mejora, permitiría a los programadores abstraerse de cuestiones técnicas y enfocarse en cuestiones de diseño centrado en el usuario, permitiendo mayor usabilidad de la aplicación y mejor aceptación por parte de los usuarios.

Los cambios de tecnología o plataforma deberían ser soportados con costos mínimos de implementación, sin que ello implique desestimar el esfuerzo invertido anteriormente.

En el siguiente capítulo se analizará el paradigma MDE (Model Driven Engeenering), el cual soluciona algunos de los inconvenientes planteados, y ofrece una solución integral, basada en transformaciones de modelos, que permite generar la aplicación en múltiples plataformas o tecnologías a partir de un modelo de entrada.

3. Ingeniería de software dirigida por modelos

La abstracción es una de las principales técnicas con la que la mente humana se enfrenta a la complejidad. Ocultando lo que es irrelevante, una aplicación compleja se puede reducir a algo comprensible y manejable. Cuando se trata de software, se deben abstraer los detalles tecnológicos de implementación y tratar con los conceptos del dominio de la forma más directa posible. Así, el modelo de una aplicación expone una visión minimizada y útil de la misma, permitiendo a los usuarios, analistas y programadores concentrarse en las necesidades del dominio de la aplicación.

La ingeniería de software dirigida por modelos, MDE [MDE], se basa en la importancia de los modelos en el proceso de desarrollo de software, combinando:

- Un lenguaje específico del dominio, que permita definir el dominio de la aplicación, con su estructura y comportamiento, sin cuestiones tecnológicas o de implementación.
- Transformaciones sobre el modelo, que permiten obtener distintos tipos de resultados, como puede ser documentación, código ejecutable u otras representaciones del modelo de entrada.

MDE mejora la productividad y la calidad del software generado debido a que se reduce el salto semántico entre el dominio del problema y el de la solución, permitiendo que los programadores se enfoquen en los modelos, y se abstraigan de los detalles técnicos de las plataformas.

Como se observa en la figura 1, las iniciativas más reconocidas del paradigma MDE son MDA (Model Driven Arquitectura) de la OMG (Object Managment Group) y las Software Factories de Microsoft.

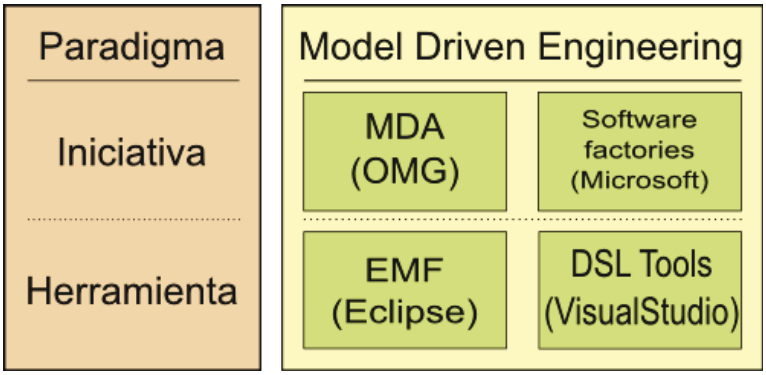

Figura 1: Paradigma MDE

3.1. La iniciativa MDA

La arquitectura dirigida por modelos, MDA, definida por la OMG, es la iniciativa MDE más extendida y conocida, la cual establece un conjunto de reglas para separar la especificación de una aplicación de su implementación. MDA propone la definición y el uso de modelos a diferentes niveles de abstracción y reglas de transformación, posibilitando la generación automática de código a partir de los modelos [MDA].

Aunque MDA no es un estándar en sí mismo, establece el uso de otros estándares de la OMG:

- MOF (Meta Object Facility) es usado para definir el metamodelo.
- UML (Unified Modeling Language), JMI (Java Metadata Interface) o XMI (XML Metadata Interchange) son usados para definir el modelo de la aplicación.
- QVT (Query/View/Transformation) es usado para especificar las transformaciones.

Estas tecnologías definen una forma estándar de almacenar e intercambiar modelos, sean de negocio o de diseño, y sus transformaciones.

Para llevar a cabo este proceso, MDA define tres niveles de abstracción de la aplicación a desarrollar:

- Modelo independiente de la plataforma, PIM (Platform Independent Model): es el modelo con alto nivel de abstracción e independiente de cualquier tecnología de implementación.
- Modelo específico de la plataforma, PSM (Platform Specific Model): es el modelo que se especifica en el PIM pero en términos de una implementación en alguna tecnología específica.

- Generación de código: es el proceso por el cual se transforma un PSM en código ejecutable, a través de técnicas formalmente definidas y una herramienta de transformación automática.

Como se observa en la figura 2 un PIM es transformado a uno o más PSM, los cuales son luego transformados a código ejecutable para la plataforma asociada a cada PSM. Así, a partir de un único modelo, es posible obtener código para múltiples tecnologías o plataformas.

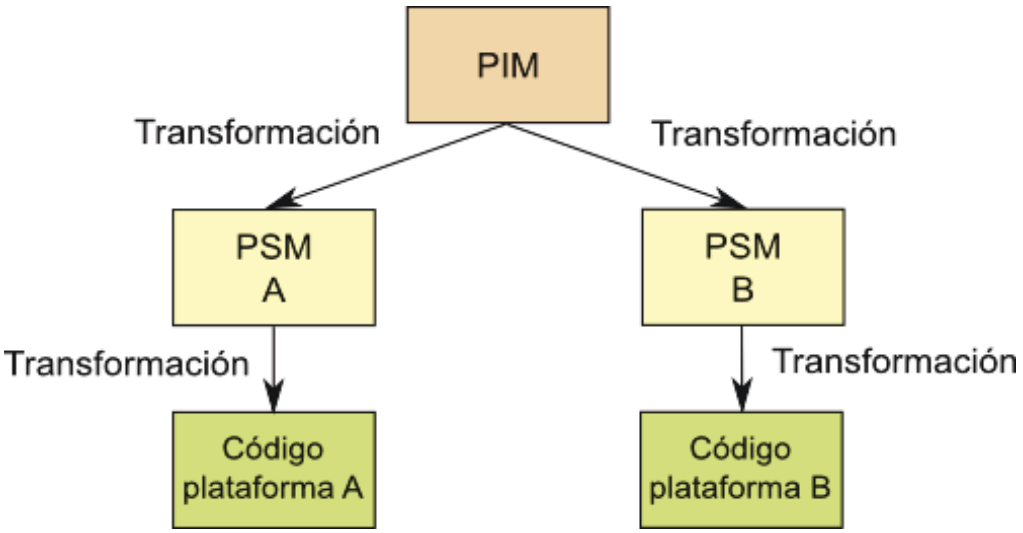

Figura 2: Transformaciones PIM a PSM

Cada PSM permite generar código para una plataforma, tecnología y lenguaje de programación específico. Ante la necesidad de implementar la aplicación en una nueva tecnología, se debe desarrollar un nuevo PSM (probablemente reutilizando partes de otro existente). Los costos de desarrollar un PSM desde cero, pueden ser altos en comparación a la utilización de las metodologías tradicionales de desarrollo de software. Sin embargo, se debe tener en cuenta que este costo se ve amortizado en la sucesivas generaciones de código que utilicen ese PSM. Es decir, la inversión de esfuerzo inicialmente es alta para la escritura del PSM, pero luego, el PSM podrá ser reutilizado sucesivas veces, sin que se requiera ningún tipo de ajuste o adaptación al mismo.

MDA ofrece múltiples ventajas respecto del desarrollo tradicional de software: acelera el desarrollo de aplicaciones, simplifica la integración entre distintas tecnologías y reduce el costo de la migración de aplicaciones a nuevas plataformas.

3.1.1. Arquitectura 4 capas de modelado

La arquitectura 4 capas de modelado que establece MDA está orientada a estandarizar conceptos relacionados al modelado, desde los más abstractos a los más concretos. Los niveles definidos en esta arquitectura se denominan comúnmente M3, M2, M1, M0 [PONS].

- El nivel M3 (el meta-metamodelo) es el nivel más abstracto, donde se encuentra el MOF. Un meta-metamodelo es un modelo que define el lenguaje para representar un metamodelo.
- En el nivel M2 (el metamodelo) los elementos son lenguajes de modelado, por ejemplo UML. Los conceptos a este nivel podrían ser Clase, Atributo, Asociación.
- En el nivel M1 (el modelo de la aplicación) los elementos son modelos de una aplicación, por ejemplo, clases como "Equipo", "Jugador" o "Estadio", atributos como "Nombre", relaciones entre estas entidades.
- El nivel M0 (instancias de la aplicación) modela a la aplicación real. Sus elementos son datos, por ejemplo "Lionel Messi", que juega en el equipo "Barcelona".

En la figura 3 se observan las 4 capas de modelado, donde cada capa hace uso de la capa anterior.

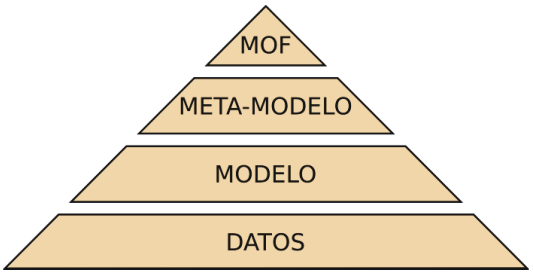

Figura 3: Capas de la arquitectura OMG

3.1.2. La implementación de Eclipse

El EMF (Eclipse Modeling Framework) [EMF], es un *framework* de modelado para Eclipse, el cual comenzó como una implementación de la especificación de MOF, pero luego evolucionó en base a la experiencia adquirida en la construcción de un conjunto de herramientas. Permite la generación automática de código para construir herramientas y otras aplicaciones a partir de modelos de datos estructurados. Soporta la lectura y escritura de serializaciones MOF, y está basado en un meta-metamodelo llamado Ecore. EMF, al igual que MOF, respeta el estándar XMI facilitando de este modo el intercambio de modelos entre diferentes herramientas compatibles con MOF. Un modelo Ecore se puede generar a partir de diagramas de clases UML, o esquemas XML. A través de un asistente se transforma el modelo de entrada en un modelo Ecore y un modelo generador. Este último permite generar el código de implementación Java correspondiente.

EMF permite usar un modelo como el punto de partida para la generación de código, e iterativamente refinar el modelo y regenerar el código, hasta obtener el código requerido. También prevé la posibilidad que el programador necesite modificar ese código, es decir, se contempla la posibilidad de que el usuario edite las clases generadas, para agregar o modificar métodos y variables de instancia. Siempre se puede regenerar desde el modelo cuando se necesite, y las partes agregadas serán preservadas durante la regeneración.

En los últimos años EMF se usó para implementar una gran cantidad de herramientas lo que permitió mejorar la eficiencia del código generado. Actualmente el uso de EMF para desarrollar herramientas está muy extendido [PONS].

Correspondencia con la arquitectura 4 capas

Nivel M3 y M2: Ecore

En EMF los modelos se especifican usando un meta-metamodelo llamado Ecore, el cual es una implementación de eMOF (Essential MOF). Ecore en sí, es un modelo EMF y su propio metamodelo. La figura 4, extraída de [PONS] muestra las clases más importantes del meta-metamodelo Ecore.

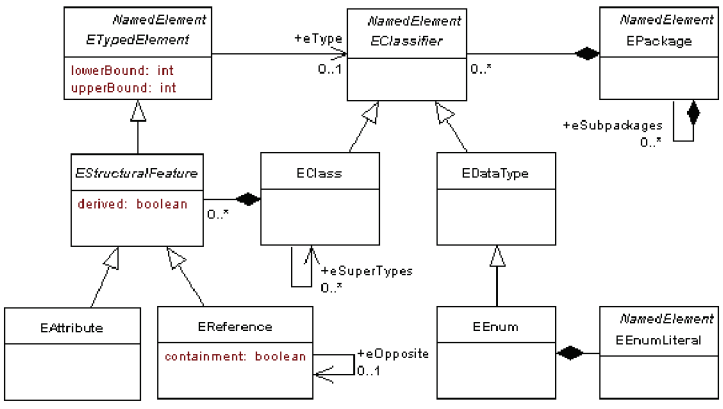

Figura 4: Meta-metamodelo Ecore

Nivel M1: el modelo

El modelo puede especificarse con un editor gráfico para metamodelos Ecore, o de cualquiera de las siguientes formas: como un documento XML, como un diagrama de clases UML o como interfaces Java con anotaciones. EMF provee asistentes para interpretar ese metamodelo y convertirlo en un modelo EMF, es decir, en una instancia del meta-metamodelo Ecore.

Los modelos en UML ofrecen mayor ventaja, ya que es un lenguaje conocido, con herramientas amigables que proveen una mayor funcionalidad. Sólo hay que tener en cuenta que la herramienta permita exportar un modelo Ecore que pueda ser usado como entrada para el *framework* EMF. La opción más reciente para la especificación de un metamodelo es utilizar el editor gráfico para Ecore.

Nivel M0: la aplicación

A partir de un modelo representado como instancias de Ecore, EMF puede generar el código para ese modelo. Se genera un *plugin* que contiene el código Java de la implementación del modelo, es decir, un conjunto de clases Java que permiten crear instancias de ese modelo, hacer consultas, actualizar, persistir, validar y controlar los cambios producidos en esas instancias. Por cada clase del modelo, se generan dos elementos en Java: una interfaz y la clase que la implementa. Todas las interfaces generadas extienden directa o indirectamente a la interfaz EObject, la base de todos los objetos en EMF. EObject y su correspondiente implementación EObjectImpl proveen la base para los mecanismos de notificación y persistencia.

3.2. La iniciativa Microsoft

Microsoft ofrece un enfoque distinto sobre el paradigma MDE y la generación de código: la utilización de DSL (Domain Specific Language). Un DSL es un lenguaje diseñado para realizar tareas específicas para un dominio concreto, así, cada DSL da soporte intrínsecamente a la generación de código para ese dominio específico, "mapeando" los modelos definidos en el DSL al código requerido. Los DSLs ofrecen la ventaja que el mismo entorno de modelado permite chequear y validar el modelo creado para la semántica del dominio, lo cual no es posible utilizando UML *profiles*.

Las herramientas que presenta Microsoft son denominadas "DSL Tools" y utilizan técnicas de desarrollo específicas del dominio para crear e implementar DSLs y forman parte de Visual Studio a partir de la versión 2008 [MSDSL].

En la creación de un DSL hay 3 actividades principales:

- Definir el modelo de dominio constituido por clases y relaciones. Se refiere al metamodelo o sintaxis abstracta del DSL. Este modelo de dominio está compuesto por una jerarquía de clases y relaciones.
- Definir la notación, es decir las *shapes* (formas) que representan a los elementos del dominio (rectángulos, círculos, etc.). Existen 5 diferentes formas: geométricas, compartimentos, imágenes, puertos y carriles.
- Definir la conexión entre los elementos del modelo de dominio con las *shapes*.

Es posible definir distintos tipos de validaciones sobre los modelos, que impidan a los usuarios crear modelos no válidos, distinguiéndose así dos tipos de restricciones dependiendo de la interacción con el usuario:

- Usando el sistema de validaciones (*soft-constraints*): estas validaciones se realizan cuando se disparan determinados eventos. Cuando una restricción no se cumple, aparecerá un mensaje en la pestaña de errores, y se permitirá que el usuario siga trabajando con el modelo aunque no se verifiquen todas las restricciones.
- Usando el sistema gráfico en tiempo de ejecución (*hard-constraints*): estas validaciones se realizan constantemente, en tiempo de ejecución (por ejemplo, cuando se intenta conectar dos *shape* mediante un *connector*).

La propuesta de Microsoft posee un robusto soporte para especificar y evaluar restricciones sobre los modelos; genera mensajes de errores útiles y comprensibles; cuenta con mapeo visual muy amigable para definir la conexión entre la sintaxis abstracta y la concreta. Como contrapartida, se puede decir que las DSL Tools no son una herramienta de código abierto; los editores generados necesitan de Visual Studio para funcionar; y que se dificulta la portabilidad de proyectos debido a que usa un lenguaje de metamodelado propio que no cumple con el estándar [PONS].

Comparación de la arquitectura de EMF y MS-DSL

En la figura 5 se puede observar la correspondencia de la iniciativa de Microsoft con la iniciativa de Eclipse [OZGUR]. El equivalente de un metamodelo para la iniciativa Microsoft se llama modelo de dominio. Este modelo de dominio está compuesto por una jerarquía de clases y relaciones.

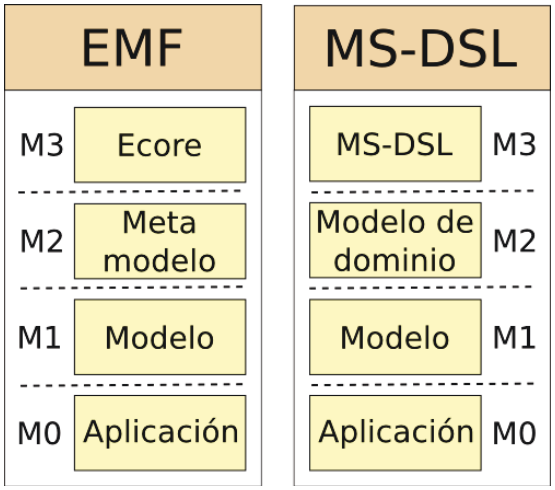

Figura 5: Comparación capas EMF y MS-DSL

3.3. Herramientas MDE

En la actualidad, la mayoría de las herramientas automáticas para generar código a partir de un modelo se centran en la modelización del diseño conceptual (diagrama de clases), requerimientos funcionales (casos de uso) y en la capa de acceso a datos. Estas herramientas generan código basadas en dos principios:

- *Parte gráfica*: generan automáticamente las interfaces gráficas tipo CRUD y de forma fija: un control por cada propiedad de la entidad. Esto en general no es suficiente, ya que se requieren interfaces donde interactúan varias entidades.
- *Parte lógica de negocio:* la "lógica" del negocio puede definirse utilizando motores de reglas, como OCL (Object Constraint Language), los cuales permiten definir restricciones invariantes y precondiciones. Pero toda la "magia" del trabajo del programador no puede realizarse.

En conclusión, las herramientas de generación basadas en MDA generan código que no abarcan los aspectos más importantes como son la aplicación de las reglas de negocio, interacción con otras aplicaciones, la interacción de los controles y su lógica de presentación, manipulación de archivos, etc. Estos aspectos deben ser programados manualmente.

AndroMDA

AndroMDA [ANDROMDA] es un *framework* de código abierto. Toma como entrada modelos descritos en herramientas CASE como UML, y usa XDoclet como tecnología de marcado para el acceso a datos desde las clases Java. Admite como entrada archivos XMI, y como herramienta de modelado utiliza UML. Es posible generar código para cualquier lenguaje de programación, y permite también insertar bloques de código propio.

Posee una herramienta para generar *templates* de proyectos para J2EE: androMDAApp, la cual es un agregado en Maven. También ofrece una herramienta para generar un archivo XMI, y derivar luego el modelo UML, a partir de una base de datos existente (Schema2XMI).

En la actualidad AndroMDA es utilizado mayormente para desarrollos utilizando tecnologías J2EE. Es posible generar código para Hibernate, EJB, Spring, Web Services y Struts. Para la capa de presentación, AndroMDA ofrece dos tecnologías: JSP o Struts. Acepta diagramas de actividad UML para especificar el flujo de navegación y generar los componentes web (Struts o JSF).

AndroMDA ofrece también generación de código en .NET. Utiliza NHibernate y objetos para la capa de acceso a datos y entidades de dominio o servicios web (ASMX en .NET) para la capa de negocio. Para la capa de presentación, AndroMDA no ofrece ninguna transformación.

Como desventaja se puede decir que aunque se basa en MDA, no basta sólo con los modelos para llegar a un despliegue, es necesario que el programador intervenga el código, por lo cual, requiere que éste tenga un buen conocimiento de la plataforma.

ArcStyler

ArcStyler [ARCSTYLER] es una herramienta de iO-Software basada en el uso de transformaciones que permiten generar aplicaciones de múltiples capas codificadas en J2EE y .NET a partir de diagramas UML y la especificación de los procesos del negocio. Permite extender las capacidades de transformación, generando nuevos PSMs a partir de UML, cuyo objetivo sea cualquier plataforma o lenguaje. Utiliza MOF para soportar estándares como UML y XMI, y además JMI para el acceso al repositorio de modelos. También incluye herramientas relacionadas con el modelado del negocio y el modelado de requisitos, por lo que cubre todo el ciclo de vida. Con ArcStyler se requiere de etapas intermedias para poder llegar al código ejecutable. Permite generar código para diferentes plataformas.

OptimalJ

OptimalJ [OPTIMALJ] es una herramienta de Compuware que utiliza MOF para soportar estándares como UML y XMI. Permite generar aplicaciones J2EE completas a partir de un PIM. Implementa completamente la especificación MDA. Está desarrollado en Java, lo que le hace portable a cualquier plataforma para su ejecución. Del proceso de desarrollo con OptimalJ se puede destacar:

- Generación automática a partir del PIM de los modelos PSM de la capa de presentación (web), capa de negocio EJB y base de datos, estableciendo la conexión entre las tres capas.
- Distinción entre bloques libres y protegidos en el código para impedir la modificación del código generado.
- La interfaz web generada proporciona una navegación por defecto para cada objeto de negocio, que permite el mantenimiento de los datos asociados a las clases del modelo del dominio. Esa interfaz es muy pobre pero existe la posibilidad de crear un patrón de presentación que se ajuste a necesidades concretas, o bien manualmente modificar el código de la aplicación.

Gracias a la clara separación entre el PIM y los PSM, OptimalJ puede considerarse como una herramienta que refleja fielmente el proceso MDA. A partir de un modelo de entidades permite crear de forma sencilla y en poco tiempo, una aplicación básica para la plataforma J2EE, generando código de buena calidad.

Acceleo

Acceleo [ACCELEO] es un generador de código abierto creado y desarrollado por Obeo. Está totalmente integrado con Eclipse y viene con éste desde la versión 3.6 Helios. Ofrece un conjunto de herramientas de suma utilidad para el programador: editor con remarcado de sintaxis, API para depuración, trazabilidad y *profiler*. Utiliza *plugins* como módulos para realizar la generación de código. Los módulos pueden adaptarse modificando sus scripts para personalizar la generación de código. Los modelos son archivos con extensión XMI. Existen diversos módulos para generar código en .NET, utilizando CSharp y NHibernate; en Java utilizando Hibernate y Struts, y otros que se encuentran en desarrollo. También es posible generar aplicaciones para dispositivos móviles que utilicen Android.

En el *apéndice A* se presenta un cuadro comparativo de las herramientas descritas.

3.3.1. Herramientas de modelado de interfaces gráficas

Existen en la actualidad diversas herramientas (WebML, OO-H, OOWS, UWE, WebSA) para el modelado de interfaces gráficas basadas en el paradigma MDE, la mayoría de ellas sólo orientadas a aplicaciones web.

UWE

UWE (UML-based Web Engineering) describe un proceso basado en MDE, orientado a objetos, para la especificación de aplicaciones web. UWE está totalmente basado en estándares, lo cual facilita su extensibilidad y reusabilidad. Utiliza UML para la definición de los modelos, XMI para el intercambio de modelos, MOF para el meta-modelado, el lenguaje de transformación QVT y XML, entre otros.

UWE utiliza UML estándar en todos los casos que es posible, pero para definir conceptos propios del dominio web, utiliza UML *profiles*, los cuales incorporan estereotipos, definición de etiquetas y restricciones. Estos *profiles* junto con la utilización de reglas OCL y herramientas CASE (*plugins* para ArgoUML y MagicDraw), permiten modelar cuestiones de navegación, presentación y lógica de negocio. UWE permite además incorporar chequeos semánticos, y generación semi-automática de código, basada en ATL y transformaciones gráficas.

UWE requiere de la separación conceptual de la aplicación en análisis de requerimientos, procesos, presentación y navegación. Tal como se observa en la figura 6, estos enfoques generan modelos que son refinados en sucesivas iteraciones en el proceso de desarrollo que define UWE .

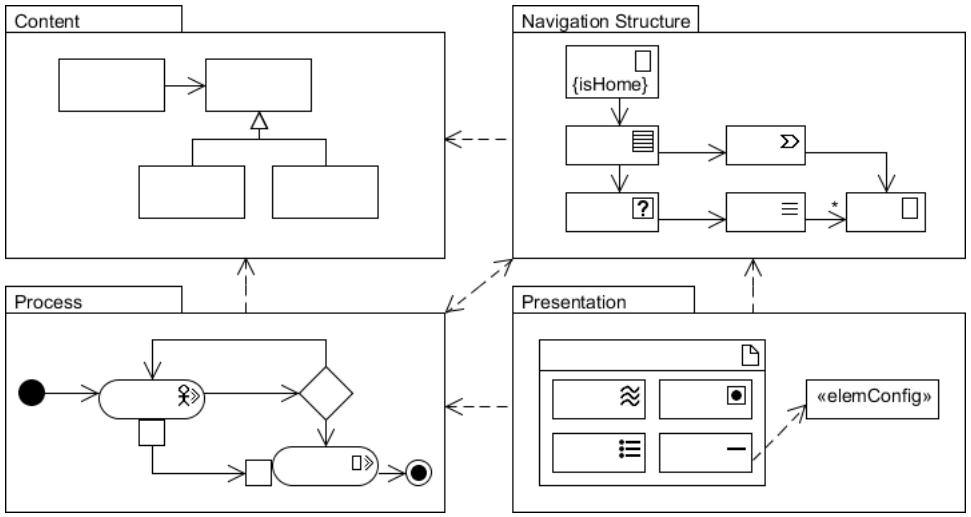

Figura 6: Tipos de modelos en UWE

Generación de código con UWE

UWE4JSF es un *plugin* para Eclipse que permite la generación automática de aplicaciones web en la plataforma JSF (JavaServer Faces). Tomando como entrada un modelo UWE generado con UML *profiles*, y utilizando puramente tecnologías de EMF, este *plugin* genera una aplicación completa JSF.

El modelo concreto de presentación es el responsable de definir cómo los componentes del PIM son transformados al PSM. Este mecanismo puede utilizarse también para incorporar el uso de otras librerías JSF, que mejoren la generación de aplicaciones RIA (Rich Internet Applications).

WebML

WebML [WEBML] es un lenguaje para especificar aplicaciones web a alto nivel. Provee especificaciones gráficas formales para el proceso de definición de la aplicación, y soporta sintaxis XML, la cual permite la generación automática de la aplicación a través de herramientas que generen software basadas en XML.

Para definir una aplicación web con WebML es necesario contemplar las siguientes 4 perspectivas de la aplicación:

- Modelo estructural: define el modelo de entidades, con sus relaciones. Para esto es posible utilizar distintas herramientas de modelado gráficas, como modelos clásicos de entidad-relación, diagrama de clases UML o modelos orientados a objetos.
- Modelo de hipertexto: define las vistas de la aplicación. Posee dos submodelos: un modelo de composición, el cual especifica qué páginas componen el hipertexto y qué contenido conforma la página; y un modelo navegacional, que indica cómo el usuario navegará el sitio.
- Modelo de presentación: define la apariencia de las páginas utilizando una sintaxis XML abstracta, sin tener en cuenta cuestiones de implementación (cómo tipo de *renderizado* o dispositivo físico a utilizar).
- Modelo de personalización: permite modelar usuarios y grupos, para luego personalizar las distintas vistas de acuerdo al usuario.

El proceso de desarrollo con WebML se compone de fases que son realizadas de manera iterativa e incremental. En cada iteración se obtiene un prototipo parcial de la aplicación, que permite su testeo y evaluación desde el inicio del desarrollo.

Generación de código con WebML

WebRatio es una implementación de WebML y de su proceso de diseño. Los requisitos se expresan a través de un modelo de alto nivel y el código de la aplicación se genera automáticamente, con reglas que se pueden ampliar y personalizar por completo. El resultado es una aplicación Java estándar, que no hace uso de entornos de ejecución o componentes propietarios. WebRatio permite que los analistas de proceso y de la aplicación trabajen juntos a los diseñadores de aplicaciones y a los programadores, optimizando la colaboración dentro del equipo de trabajo, observar la figura 7 [WEBRATIO].

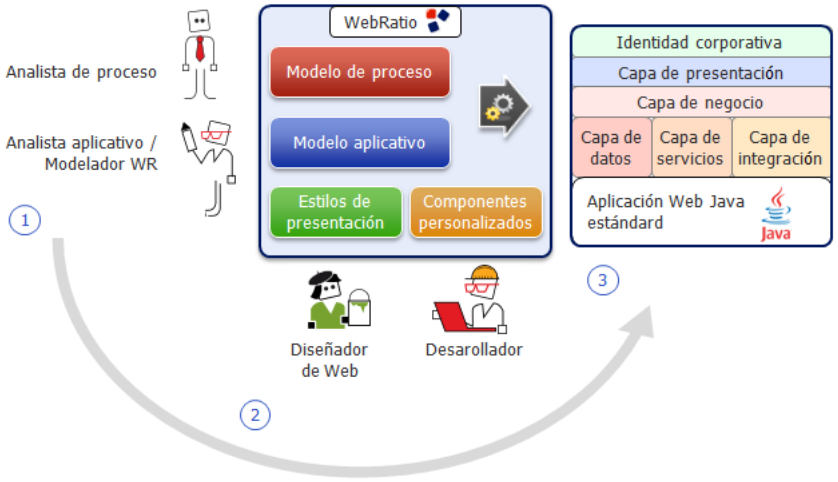

Figura 7: Los tres pasos del proceso de WebRatio extraído de [WEBRATIO]

3.4. Conclusión

El paradigma MDE es una buena propuesta para el desarrollo de aplicaciones, basándose en un modelo abstracto, independiente de la implementación, aporta gran flexibilidad dado lo cambiante que son hoy en día las plataformas y las nuevas tecnologías que surgen casi a diario.

La OMG ha creado la iniciativa MDA, la cual representa fielmente el paradigma MDE, modelando todas las capas de abstracción necesarias: meta-metamodelo, metamodelo, PIM y PSM. Por otro lado, las herramientas que implementan MDA se encuentran en proceso de expansión y maduración, pero aún, poseen algunas de las siguientes limitaciones, las cuales son importantes a la hora de decidir incorporarlas al desarrollo de una aplicación:

- En general no son multiplataforma o multilenguaje, es decir, generan código para una única plataforma o lenguaje.
- No permiten definir comportamiento en el PIM, o si lo hacen, es de manera muy limitada.
- No generan código 100 % listo para ejecutar. El programador debe modificar o completar el código generado, lo cual, para las herramientas que no permiten definir regiones protegidas de código, esto deberá repetirse en cada iteración de la generación. Esto "empaña" el concepto básico de MDE sobre los modelos, y la bidireccionalidad modelo - código. Por otro lado, ante la necesidad cambiar de implementación, se deberá replicar y mantener manualmente este código adicional.
- Respecto de las interfaces gráficas, sólo generan interfaces tipo CRUD para las entidades del modelo. No es posible definir eventos o apariencia de manera personalizada.

El desafío de la presente tesina es crear una herramienta que solucione estas limitaciones, y se enfoque en el diseño de las interfaces de usuario y la posibilidad de definir el comportamiento "extra", el cual no es posible representar en el modelo. Es decir, poder definir en el PIM todas las características necesarias para representar la aplicación en su totalidad.

Parte II
Implementación de interfaces de usuario

En esta parte se analiza la arquitectura de las interfaces de usuario, desde su implementación a bajo nivel (comunicación del hardware con el sistema operativo), hasta los frameworks de alto nivel más recientes. También se analizan los detalles de implementación de acuerdo a las distintas plataformas donde se ejecutan las aplicaciones (escritorio, web, dispositivos móviles, etc).

Luego del análisis, se concluye en cuáles son las características comunes a todas las plataformas, tecnologías o lenguajes, las cuales constituirán luego, un gran aporte al diseño del lenguaje de la herramienta.

4. Interfaces de usuario

4.1. Definición

La etimología de la palabra interfaz está compuesta por dos vocablos: Inter proviene del latín inter, y significa, "entre" o "en medio", y Faz proviene del latín faces, y significa "superficie, vista o lado de una cosa". Por lo tanto una traducción literal del concepto de interfaz atendiendo a su etimología, podría ser "superficie, vista, o lado mediador". En el contexto de la informática, la interfaz de usuario, es el espacio que media la relación de un usuario y una computadora, permitiendo la comunicación entre ambos.

Existen diversos tipos de interfaces de acuerdo a cómo esté implementada, los elementos que utilice o el medio en que opere. En la actualidad, las más utilizadas son las interfaces gráficas de usuario (GUI, Graphical User Interface), que surgen de la necesidad de facilitar la interacción con las computadoras, utilizando imágenes, botones, menúes y otros elementos visuales con los que el usuario puede interactuar más naturalmente, sin necesidad de recordar comandos y configuraciones.

En las siguientes secciones se describirán los principales tipos de interfaces y sus características, y se analizarán cómo son implementadas a bajo nivel, y el software que utilizan a más alto nivel, como librerías de controles o *frameworks*.

4.2. Arquitectura

La implementación de interfaces de usuario se compone de una arquitectura en capas, donde las capas inferiores manejan cuestiones de bajo nivel, asociadas al hardware, y a medida que se asciende, cada capa abstrae aspectos de implementación.

Independientemente del contexto de la aplicación (escritorio, web, etc), se pueden diferenciar los siguientes niveles en la arquitectura de las interfaces de usuario:

Implementación de interfaces a bajo nivel

Cada sistema operativo implementa la interfaz de usuario a través de librerías que utilizan primitivas nativas para controlar los componentes de hardware, como monitores, teclados o *mouses*. Estas librerías, generalmente escritas en C, son las responsables de "dibujar" los elementos de la interfaz (líneas rectas o curvas, mapas de bits o colores) en los distintos dispositivos de salida a través de sus controladores.

Librerías gráficas

Las librerías gráficas ofrecen herramientas para diseñar interfaces de usuario, tales como controles, manejo de eventos y rutinas asociadas a las operaciones más comunes de interfaces de usuario.

Existen librerías gráficas que utilizan las librerías del sistema operativo para mostrar los controles, adoptando la misma apariencia. Otras librerías gráficas, utilizan directamente primitivas nativas del sistema operativo, tales como puntos y líneas, para "dibujar" los controles, aportando así una apariencia independiente a la del sistema operativo y facilitando su portabilidad.

De acuerdo a cómo estén implementadas cada una de estas librerías, será determinante la *performance* del *renderizado* de las interfaces, ya que utilizar directamente las primitivas nativas del sistema operativo será más eficiente al evitar interactuar con las librerías de éste.

Otro aspecto determinante, será la necesidad o no, de conservar la misma apariencia del sistema operativo.

Bindings entre librerías gráficas y lenguajes de programación

Los *bindings* son librerías que permiten que una librería gráfica pueda ser utilizada por un lenguaje de programación, haciendo posible la portabilidad de la librería a distintos lenguajes. Por ejemplo, la librería de controles multiplataforma GTK, permite ser utilizada a través de sus *bindings*, por lenguajes de programación de alto nivel como Java, Perl, Ruby o Python.

Lenguajes de programación

Cada lenguaje de programación implementa las interfaces de usuario haciendo uso de alguna librería gráfica, la cual podrá estar acoplada al lenguaje de programación, o utilizar librerías de controles independientes. En los capítulos venideros, se realizará un análisis de cómo implementan las interfaces de usuario los lenguajes más populares, analizando características, manejo de eventos, herencia y desarrollo de controles personalizados.

Frameworks para interfaces gráficas

Existen numerosos *frameworks* que abarcan el desarrollo de interfaces gráficas, simplificando la tarea del programador, y abstrayéndolo de cuestiones repetitivas. Muchos *frameworks* abarcan más ampliamente el desarrollo de las interfaces gráficas, incluyendo cuestiones del flujo navegacional, manejo de eventos y lógica de la aplicación, tal es el caso del patrón MVC (Model-View-Controller).

Para el contexto web, existen diversos *frameworks* que encapsulan y simplifican las tareas de diseño y programación de interfaces web.

En los siguientes capítulos, se presentará una comparación entre los distintos niveles que componen la arquitectura de la implementación de las interfaces de usuario, en cada contexto de aplicación: aplicaciones de consola, aplicaciones de escritorio, aplicaciones web, aplicaciones para dispositivos móviles y otros tipos de aplicaciones.

4.3. Interfaces de usuario en aplicaciones de consola

En el inicio, el ingreso de la información se realizaba a través de tarjetas perforadas; luego se debía consultar el resultado de su procesamiento en extensos listados que eran impresos. Con el correr del tiempo, las interfaces de usuario se fueron orientando al uso de texto proveniente de un teclado. Este texto usualmente consistía de un conjunto de comandos que el usuario debía memorizar y a los que la computadora respondía. Estas interfaces de aplicaciones de consola sólo eran capaces de desplegar códigos ASCII en pantalla. En contraste con los programas gráficos que tratan al monitor como un arreglo de píxeles, las interfaces de aplicaciones de consola trataban al área de despliegue como un arreglo de bloques, donde cada uno, sólo puede contener un carácter.

Interfaces de líneas de comandos

Las primeras interfaces utilizadas por las aplicaciones de consola fueron las interfaces de líneas de comandos (CLI, Command Line Interface). Estas interfaces sólo son capaces de mostrar texto. No utilizan opciones, ni menúes. Como dispositivo de entrada se utiliza únicamente el teclado, no se utiliza el *mouse*. La interacción se realiza mediante comandos que ingresa el usuario, y a continuación la tecla *Enter*. Un intérprete parsea el comando y lo ejecuta. Luego el resultado de la ejecución del comando se muestra en pantalla como texto. En la figura 8 se ve un ejemplo de este tipo de interfaces.

En la actualidad, las CLI son parte fundamental de los *shells* o emuladores de terminal. Aparecen en todos los escritorios (Gnome, KDE, Windows) como un método para ejecutar aplicaciones rápidamente; como interfaz de lenguajes interpretados tales como Java, Python, Ruby o Perl; en aplicaciones cliente-servidor, en bases de datos (PostgreSQL, MySQL, Oracle), en clientes FTP, etc. Las CLI son también un elemento fundamental en aplicaciones de ingeniería como Matlab y Autocad.

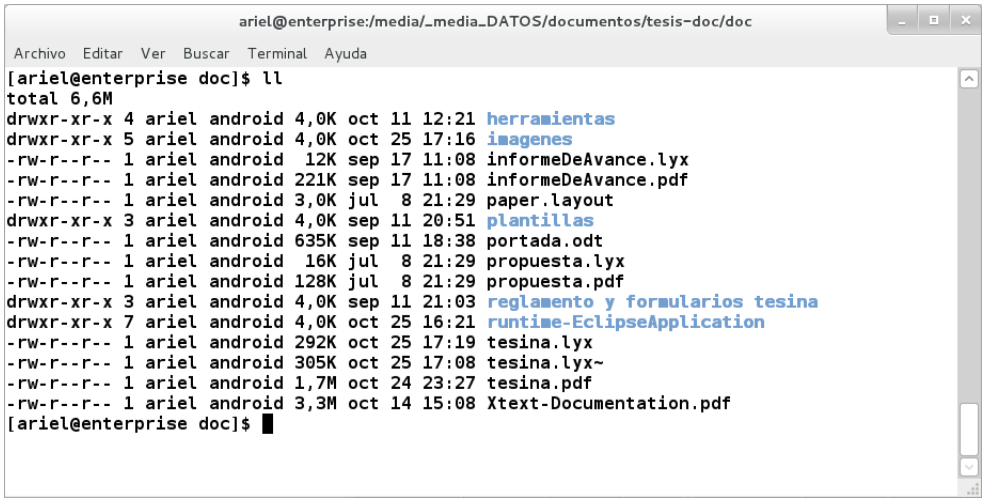

Figura 8: Interfaz de línea de comando de Bash

Interfaces de texto

Un paso intermedio entre la interfaz de línea de comando y línea las interfaces gráficas, fue el uso de interfaces basadas en menúes y ventanas en modo texto, donde se podía interactuar por medio de

un *mouse* en lugar de escribir el comando correspondiente en el teclado. Esto demandaba un esfuerzo considerable de programación, y por lo general no había una manera estandarizada de programar este tipo de interfaces. En este punto, aparecieron muchas librerías para interfaces de usuario, prometiendo disminuir los tiempos de desarrollo, pero con el advenimiento de las interfaces gráficas integradas en el sistema operativo, mucho de este trabajo fue efímero. En la figura 9 se muestra la aplicación Midnight Commander la cual utiliza interfaces de texto.

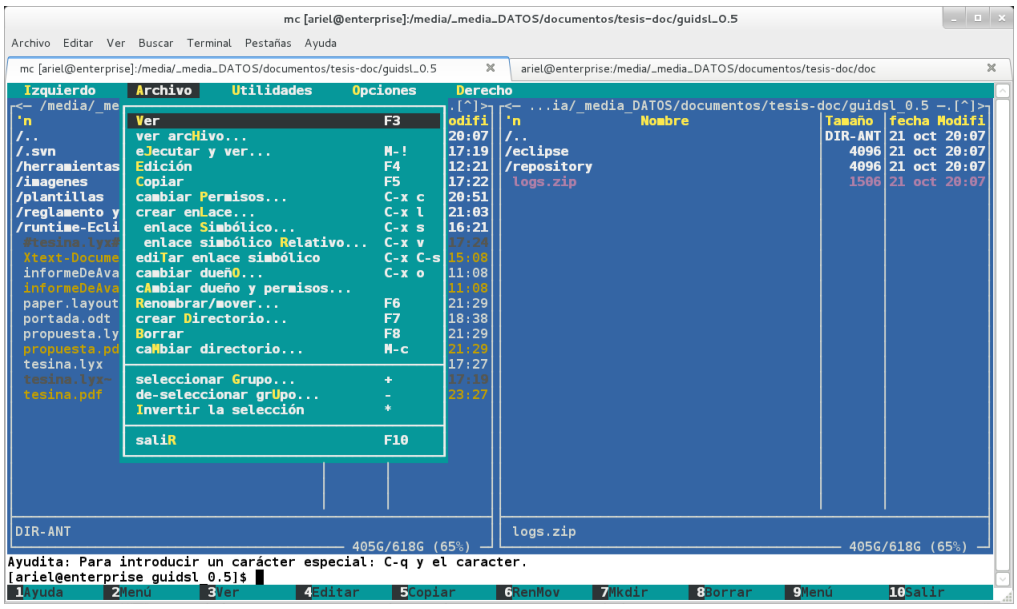

Figura 9: Interfaz del programa Midnight Commander

Los lenguajes de programación más populares que utilizan este tipo de interfaces son Clipper y Pascal.

4.4. Interfaces gráficas en aplicaciones de escritorio

Son aplicaciones creadas para ejecutarse en una computadora de escritorio, sobre un sistema operativo de interfaz visual.

Ventajas

- Robustez.
- Mayor rendimiento y velocidad, ya que el procesamiento de la información es de manera local.
- Seguridad, dado que la información se administra localmente o dentro de la *intranet* donde se utiliza la aplicación.

Desventajas

- Altamente acopladas al sistema operativo donde se ejecutan. Una aplicación de escritorio desarrollada para funcionar en Linux, no funciona en Windows, debe ser desarrollada nuevamente (salvo que se desarrolle en algún lenguaje multiplataforma).
- En general se requiere instalar librerías y componentes auxiliares que dan soporte a la aplicación. Estas instalaciones son necesarias en cada puesto de trabajo donde se necesite utilizar la aplicación, incrementando el costo de mantenimiento y actualización.
- No es posible accederlas de manera distribuida: sólo es posible accederlas dentro de la *intranet* donde se encuentran.

Implementación de interfaces a bajo nivel

X Window System

X Window System [XWINDOW] es un software desarrollado para proveer de una interfaz gráfica a los sistemas Unix. Este protocolo permite la interacción gráfica en red entre un usuario y una o más

computadoras haciendo transparente la red para éste. X es el encargado de mostrar la información gráfica de forma totalmente independiente del sistema operativo. El sistema de ventanas X distribuye el procesamiento de aplicaciones especificando enlaces cliente-servidor. El hecho que exista un estándar definido para X permite que se desarrollen servidores X para distintos sistemas operativos y plataformas, lo que hace que el código sea muy portable. La comunicación entre el cliente X y el servidor se realiza por medio de un protocolo conocido como Xprotocol.

Los aspectos de decoración y manejo de ventanas no están definidos en esta biblioteca. X es primariamente una definición de primitivas de protocolo y gráficas, no contiene especificaciones de diseño de interfaz de usuario, como estilos de botón, menú, barra de título para las ventanas, para ello necesita un gestor de ventanas, por ejemplo, Metacity en Gnome, KWin en KDE, Xfwm en Xfce, etc. X permite al usuario instalar uno o más gestores de ventanas.

Servidor X.org

X.Org Server [XORG] es una implementación de código abierto del sistema X Window System, el cual es el sistema estándar utilizado por la mayoría de distribuciones de Linux. Ha sido adoptado por Archlinux, Debian, Ubuntu, Gentoo, Fedora, Slackware, openSUSE, Mandriva, Cygwin/X y otras; también por el sistema operativo FreeBSD. Está escrito en C y se distribuye bajo licencia MIT.

GNU/Linux

- GNOME: fue desarrollado con el objetivo de dotar de un entorno gráfico de escritorio y una plataforma de desarrollo de aplicaciones totalmente libres en sistemas operativos GNU/Linux. El escritorio GNOME es similar en su infraestructura, al de Windows.
- KDE: el objetivo del proyecto fue desarrollar una interfaz gráfica que opere sobre sistemas operativos Unix, especialmente GNU/Linux y que posibilite un método de interacción amigable con la computadora similar a los que ofrecen Windows o Mac OS en otras plataformas.

Windows

Las interfaces gráficas de Windows son administradas por los siguientes componentes que forman parte de la API de Windows:

- Graphical Device Interface (GDI): esta librería permite mostrar gráficos y texto en pantalla o en la impresora utilizando un conjunto de objetos gráficos genéricos. Estos objetos son contenedores, mapas de bits, líneas rectas, líneas curvas y la capacidad de colorear figuras. A diferencia de DOS, en Windows no es necesario conocer detalles de hardware de los dispositivos que se manipulan. GDI encapsula estos detalles y permite interactuar con todas las tarjetas graficadoras e impresoras de la misma manera, comunicándose con el manejador del dispositivo a través de un conjunto de funciones de la API GDI. Con la aparición de Windows XP y Windows 2003, surgió una nueva versión de la API GDI, llamada GDI+, la cual optimizó funciones, incorporó nuevas características, como el manejo de gráficos vectoriales 2D, manejo de tipografías y también permitió manipular diversos formatos de imágenes. Las funciones provistas por la API GDI, están encapsuladas en la librería Gdi32.dll [GDI, GDI2].
- WindowsUSER: esta librería permite "dibujar" ventanas (tamaño, posicionamiento, título, controles de ventana, ventanas modales), menúes, mostrar mensajes y el escritorio (fondos, puntero del ratón) de Windows y es la responsable del "*look and feel*" de este sistema operativo. Invoca funciones de la librería GDI para "dibujar" los componentes necesarios. Algunas aplicaciones utilizan funciones de WindowsUser para crear una ventana, y luego invocan directamente funciones GDI para "dibujar" a más bajo nivel los objetos que contendrá la ventana. Las funciones provistas por la API WindowsUSER, están encapsuladas en la librería user32.dll.

Mac

La arquitectura del sistema operativo Mac OS X está basada en tecnología Unix. Mac OS X introduce un nuevo entorno gráfico denominado Aqua, el cual introduce la interfaz de Macintosh con transparencias, formas redondeadas en los acabados de las ventanas, iconos con formas sinuosas y degradadas, que trasciende el aspecto tosco y plano de los acabados con efecto tridimensional predominantes en la generación anterior de interfaces gráficas.

Como soporte se tiene un microkernel muy potente (Darwin), sobre el que se monta la capa del sistema de gráficos que integra 3 tecnologías:

- Quartz: es un servidor de ventanas y un motor de despliegue gráfico bidimensional que define las formas en términos de sus componentes vectoriales, y que usa como lenguaje gráfico de representación interna el estándar PDF (Portable Document Format).
- QuickTime: es una tecnología multimedia que permite la manipulación de imágenes, sonidos, texto, música, animación y realidad virtual. Esta tecnología, en combinación con las interfaces de programación Carbon, Cocoa (Java/Objective C), Java y el soporte de UNIX, permiten el desarrollo de aplicaciones con interfaz gráfica de usuario en diversas plataformas. Todo esto queda plasmado en el término definido por Apple como experiencia del usuario el cuál abarca la apariencia visual, comportamiento interactivo y capacidades asistivas del software.
- OpenGL (Open Graphic Library): es un estándar definido por Architectural Review Board (ARB) que establece una API multiplatarforma y multilenguaje para contenido gráfico 3D. Define un conjunto de funciones, con su interfaz y el comportamiento esperado de cada una de ellas. Las funciones de OpenGL, no definen como manipular imágenes, animaciones, objetos 3D, teclado ni *mouse*. Sólo establecen la interfaz, a bajo nivel, de cómo se comunicará el sistema operativo con el GPU (Graphics Processing Unit). Luego, una librería externa, como por ejemplo GLUT, será la responsable de implementar el estándar OpenGL y realizar estas funciones de más alto nivel. Diversos fabricantes de hardware implementan el estándar, los cuales luego de superar un test de conformidad, pueden exponer su certificación OpenGL. Existen implementaciones de OpenGL para GNU/Linux, Mac, Windows, varias plataformas Unix y PlayStation 3. Existen también varias implementaciones en software que permiten ejecutar aplicaciones que dependen de OpenGL sin soporte de aceleración hardware. Sus principales aplicaciones incluyen realidad virtual, representaciones científicas, videojuegos y simuladores de vuelo [OPENGL].

4.4.1. Librerías gráficas

Las aplicaciones de escritorio utilizan librerías de controles para graficar las interfaces gráficas. Algunas de estas librerías se encuentran acopladas al sistema operativo utilizando rutinas de más bajo nivel del mismo para dibujar los controles y formularios. En otros casos, utilizan librerías independientes del sistema operativo las cuales "dibujan" la interfaz utilizando la API de dibujo de éste.

A continuación se describen brevemente las librerías de controles utilizadas por los lenguajes más populares .

AWT

AWT (Abstract Window Toolkit) es la primer biblioteca de clases Java desarrollada para la creación de interfaces gráficas de usuario. Su arquitectura se basa en componentes y contenedores. Los controles son *renderizados* utilizando primitivas nativas del sistema operativo donde se ejecuta la aplicación.

Características principales

- Los contenedores contienen componentes, que son los controles básicos.
- Los componentes no se posicionan de manera absoluta, sino que se disponen dentro del contenedor de acuerdo a un *layout*.

Ventajas

- Apariencia semejante cualquiera sea su plataforma de ejecución.
- Mayor rendimiento en la *renderización* de los controles.

Desventajas

- Provee sólo un conjunto básico de componentes, comunes a todas las plataformas.
- Carece de un formato de recursos.
- Altamente acoplado al entorno de ejecución.
- No se puede separar el código de lo que es propiamente interfaz.
- No posee diseñador gráfico de interfaces.
- Una interfaz gráfica diseñada para una plataforma, puede no visualizarse correctamente en otra diferente.

SWT

SWT (Standard Widget Toolkit) es una librería de clases Java para la creación de interfaces gráficas de usuario, diseñada para ser portable y eficiente. Utiliza las librería del sistema operativo donde se implementa. La portabilidad que ofrece SWT está en realidad dada por la utilización de JNI (Java Native Interface) las cuales acceden a las APIs particulares de cada sistema operativo, por lo cual, su implementación permite abstraerse del hecho que cada JNI está en realidad implementado para cada plataforma.

Ventajas

- Alta performance.
- Gran integración con la plataforma nativa.

Desventajas

- Mayor costo para extender controles o crear nuevos.
- Exposición a los *bugs* específicos de cada plataforma.
- Al no formar parte del *release* de Java, no es portable en todas las plataformas que soporta Java.
- Costo de adaptación para cada nueva plataforma donde deba portarse.

Swing

Swing es una librería gráfica para Java que extiende AWT la cual mejora la apariencia y extiende la funcionalidad de los componentes. Utiliza la arquitectura MVC, donde el modelo se conforma de objetos *listeners* y la vista de controles UI. Los componentes Swing están escritos 100 % en código Java, sin utilizar librerías del sistema operativo. Cada componente es dibujado utilizando Java2D el cual invoca rutinas de bajo nivel del sistema operativo. Swing permite emular la apariencia de los componentes nativos manteniendo las ventajas de la independencia de la plataforma [SWING]. Existen las siguientes posibilidades para establecer la apariencia del *toolkit* de Swing para una aplicación:

- Determinado por el manejador Swing UI: si no se especifica ninguna apariencia, el manejador Swing UI utiliza la apariencia predeterminada, la cual es determinada por el proveedor JRE. Para el proveedor JRE de Sun se utiliza la apariencia de Java, llamada Metal, la cual funciona en todas las plataformas.
- Utilizar la apariencia nativa de la plataforma: si la aplicación se ejecuta sobre Windows, se utiliza el apariencia Windows; si se ejecuta sobre Mac, se utiliza la apariencia Aqua; si se ejecuta sobre plataformas Unix/Linux, se utiliza la apariencia GTK+ o CDE/Motif, de acuerdo a la configuración de escritorio del usuario.
- Especificar una apariencia específica, de las 4 apariencias disponibles (Windows, Java (Metal), GTK+, CDE/Motif).
- Crear una apariencia personalizada, utilizando el paquete Synth.
- Utilizar una apariencia de un proveedor externo.

Ventajas

- Fácilmente extensible: dada su arquitectura particionada, es fácil proveer otras implementaciones sin hacer cambios sustanciales en el modelo.
- Portable.
- Personalizable: fácil de personalizar la apariencia y comportamientos de los controles. Se pueden proveer implementaciones de apariencia y aplicar cambios uniformes de estilo, sin realizar cambios en la aplicación.
- Los componentes Swing soportan más características.
- Componentes escritos en código Java en su totalidad.
- Componentes livianos.

Desventajas

- Menor performance, ya que cada componente es "dibujado".
- Controles con un *"look and feel"* similar a un control nativo del sistema operativo puede no comportarse de la misma manera.

GTK+

Gimp Toolkit (GTK) es una librería para desarrollar interfaces gráficas multiplataforma, usada principalmente en entornos gráficos como GNOME, XFCE, LXDE. También es posible utilizarla en Windows, Mac OS y dispositivos móviles. Inicialmente fue creada para el software de edición de imágenes GIMP. GTK+ ofrece *bindings*, llamados *wrappers*, para diversos lenguajes como Java, C, Python, Perl, PHP, Ruby o Javascript y posee una amplia variedad de controles. GTK+ está escrita en C, es software libre y se distribuye bajo licencia GNU LGPL 3.0 [GTK].

GTK+ utiliza a su vez las siguientes librerías:

- ATK: bibliotecas para ofrecer accesibilidad, por ejemplo, a personas con alguna discapacidad.
- Pango: biblioteca para el diseño y *renderizado* de texto internacional.
- Cairo: biblioteca de *renderización* avanzada de controles de aplicación.

Ventajas

- Simple y fácil de usar, tanto para desarrolladores como usuarios.
- Gran comunidad activa.
- Es bien diseñado, flexible y extensible.
- Sofisticado *framework* de internacionalización.
- Es software libre.
- Es portable.

Desventajas

- La funcionalidad extra (XML, bases de datos, programación de *sockets*, etc.) no se encuentra incorporada, se deben utilizar librerías externas.

Qt

Qt es una biblioteca multiplataforma para desarrollar interfaces gráficas de usuario. Qt utiliza el lenguaje de programación C++ de forma nativa, pero puede ser utilizado en varios otros lenguajes de programación a través de *bindings*. El API de la biblioteca cuenta con métodos para acceder a bases de datos mediante SQL (Standard Query Language), así como uso de XML, gestión de hilos, soporte de red, una API multiplataforma unificada para la manipulación de archivos además de estructuras de datos tradicionales. Qt se distribuye bajo los términos de GNU Lesser General Public License (y otras), es software libre y de código abierto [QT, QT1, QT2].

El entorno de escritorio KDE para sistemas operativos Linux utiliza Qt como librería gráfica.

Ventajas

- Es compilado, por lo cual es más rápido.
- Multiplataforma.
- Buena herramienta para el diseño de interfaces.
- Código abierto.
- Utiliza librerías nativas para dibujar los controles.

Desventaja

- Requiere de un pre-proceso no estándard.

TK

TK [TK] es una biblioteca multiplataforma para desarrollar interfaces gráficas de usuario diseñada para ser utilizada en lenguajes de alto nivel como Perl, Python, Ruby, TCL y varios otros. Originalmente fue desarrollada como una extensión para el lenguaje TCL. Ha sido portada para correr en la mayoría de las variantes de Linux, Apple Macintosh, Unix y Windows. Desde la versión 8, TK ofrece *"look and feel"* nativo para la mayoría de sus controles.

TCL/TK es de código abierto, distribuido con licencia BSD.

wxWidget

wxWidgets [WXWIDGETS] es un librería de controles que permite crear aplicaciones para Linux, Unix, Windows, OSX, y también en plataformas para dispositivos móviles con iPhone o Windows Mobile. Utiliza las librerias GTK+ y está escrita en C++. Ofrece *bindings* a distintos lenguajes como Python, Perl y Ruby. wxWidgets permite un *"look and feel"* nativo de la plataforma, ya que utiliza las librerías del sistema operativo. Es libre y de código abierto.

Ventajas

- Ofrece gran variedad de controles y utilidades.
- Buen manejo de eventos. Permite definir eventos personalizados y *binding* con el código manejador del evento tanto estáticamente como dinámicamente.
- Ofrece utilidades para la depuración.
- Soporte para ejecución multihilos.
- Funcionalidad para acceder a bases de datos.
- Ofrece un conjunto de clases para soporte TCP/IP, FTP y otros protocolos.

Desventajas

- Exposición a los *bugs* específicos de la plataforma.
- Puede no comportarse igual en todas las plataformas.
- Debe compilarse de manera independiente (no provee binarios).

En el *apéndice B* se presenta un cuadro comparativo de las librerías descritas.

4.4.2. Lenguajes de programación

Java

Para implementar las interfaces gráficas en Java se puede utilizar alguna de las librerías gráficas que existen para este lenguaje: AWT, Swing o SWT (explicadas en detalle en la subsección 4.4.1).

.NET

Para las aplicaciones Windows (aplicaciones de escritorio) en CSharp, las interfaces se implementan a través de clases parciales que heredan de la clase Form. Estas clases tienen dos vistas posibles: la vista del diseñador, donde se observan de manera gráfica los controles y su ubicación dentro del formulario; y la vista del código, donde se encuentra el código manejador de todos los eventos del formulario, y demás rutinas auxiliares.

En .NET es posible crear controles personalizados de manera muy simple. Dependiendo de la manera en que están implementados, se pueden encontrar los siguientes tipos de controles de usuario:

- Controles de usuario: son los más simples. Éstos heredan de la clase *System.Windows.Forms.UserControl* y siguen un modelo de composición. En general estos controles, combinan dos o más controles estándares para formar una unidad lógica que los integra y les agrega funcionalidad.

- Controles heredados: estos controles son clases que heredan del control estándard que más se ajusta a la funcionalidad que se desea proveer. Luego agrega o redefine diseño o comportamiento del control que extiende. Son en general más potentes y flexibles.
- Controles "dibujados": estos controles son diseñados desde cero utilizando rutinas de dibujo GDI+. En general heredan de una clase base, como puede ser *System.Windows.Forms.Control*. Son más costosos de diseñar e implementar, pero proveen una interfaz más personalizada.
- *Extender providers*: no son necesariamente controles. Estos componentes agregan características a otros controles de un formulario. Proveen una potente manera de implementar interfaces de usuario extensibles.

A los controles personalizados se los puede incluir y compilar junto al proyecto que los utiliza, o compilarlos en un *assembly* separado para poder ser utilizado también por otros proyectos.

Delphi

En Delphi las interfaces gráficas se implementan a través de formularios, los cuales utilizan dos archivos para guardar su información: un archivo con extensión *.DFM* que contiene las propiedades de todos los controles que posee el formulario; y un archivo con extensión *.PAS* que contiene todo el código asociado a los eventos de los controles del formulario.

Es posible implementar controles personalizados muy fácilmente extendiendo la clase *TComponent*. También existen numerosas librerías de controles que ofrecen todo tipo de funcionalidad.

Visual Basic

En Visual Basic las interfaces se implementan de manera sencilla utilizando un único archivo para cada formulario. El archivo del formulario contiene dos partes: un control *VB.Form* donde constan los controles del formulario con sus propiedades, y otra sección donde consta el código asociado a los eventos y rutinas. Cada tipo de control define un conjunto de propiedades y eventos. El *bind* entre los eventos y el código que lo maneja se realiza por el nombre de la rutina, el cual se conforma de la siguiente manera: *nombreControl_nombreEvento*. Los controles se agregan al formulario de manera gráfica, arrastrando los mismos desde la barra de herramientas. No existe jerarquía de controles. Es posible crear controles personalizados, los cuales son compilados como controles OCX (OLE Control eXtension).

4.5. Interfaces gráficas en aplicaciones Web

Las aplicaciones web residen en un servidor y son accedidas por los usuarios a través de una conexión de red utilizando un software cliente, llamado navegador. El navegador se conecta con el servidor donde reside la aplicación, utilizando el protocolo HTTP. El navegador realiza peticiones, y el servidor responde enviando como respuesta contenido que puede ser interpretado y graficado por el navegador. Estos contenidos podrán ser páginas HTML (Hypertext Markup Language), comandos Javascript que permiten agregar comportamiento a la página HTML, archivos CSS (Cascading Style Sheets) que determinan el formato visual de la página y sus componentes, y contenido multimedia (archivos Flash, audio, videos, imágenes).

Ventajas

- Independiente del sistema operativo donde se ejecute.
- Portabilidad: se pueden ejecutar en cualquier computadora a través de un navegador, sin necesidad de instalar ningún software o componente adicional.
- Facilidad de actualización: cuando se lanza una nueva versión de la aplicación, todos los clientes disponen de ella de manera inmediata y transparente, sin necesidad de distribuir actualizaciones.
- Uniformidad: al ser accedidas todas las aplicaciones a través de un navegador, hace que el usuario se encuentre familiarizado con su uso.
- Bajo consumo de recursos de hardware en los clientes.

Desventajas

- Las aplicaciones web son, en general, más costosas de desarrollar. Aunque en la actualidad existen numerosos *frameworks* que encapsulan y ocultan la complejidad de la infraestructura de una aplicación web, es necesario mayor conocimiento de la arquitectura y posee mayores costos de desarrollo.
- Mayor tiempo de respuesta: aunque en la actualidad existen numerosas técnicas y herramientas para optimizar la respuesta de las aplicaciones web, como Ajax, compresión de datos, etc., aún no son equiparables con las aplicaciones de escritorio.

4.5.1. Implementación de interfaces a bajo nivel

Un motor de *renderizado* es un software que interpreta contenido marcado (HTML, XML, archivos de estilos) y realiza una representación visual en la pantalla. Formalmente, el motor de *renderizado*, define las normas de posicionamiento y ubica el contenido dentro de la página o contenedor. Generalmente el motor de *renderizado* se emplea en aplicaciones que muestran contenido web, como navegadores o clientes de e-mail.

Motores de renderizado más populares

Gecko

Gecko es un motor de *renderizado* de código abierto, utilizado principalmente por navegadores web. Está escrito en C++ y aunque originalmente fue desarrollado por Netscape, en la actualidad es desarrollado por Mozilla Coorporation y Mozilla Fundation. Es utilizado por múltiples aplicaciones en diversas plataformas, destacándose el navegador Firefox. Soporta varios estándares abiertos de internet, como HTML4, CSS 1, 2 y 3, W3C DOM, XML y Javascript. Gecko es también usado para la creación de interfaces de ciertas aplicaciones, dibujando *scrollbars*, *toolbars* y *menus* [GECKO].

WebKit

WebKit nació como una bifurcación de KHTML, un motor de *renderizado* HTML y del motor Javascript de KDE (KJS). La API de WebKit está desarrollada en Objective-C y permite interactuar con un servidor web para recuperar y *renderizar* páginas web, descargar archivos y administrar *plugins*. Webkit incluye dos *frameworks* de más bajo nivel: WebCore, un analizador sintáctico y motor de *renderizado* de HTML basado en KHTML, y JavaScriptCore, un intérprete de Javascript basado en KJS. WebKit soporta los estándares HTML, XML, DOM, XSLT, CSS, Javascript y SGV. Es de código abierto, y entre las aplicaciones más populares que lo utilizan, se destacan los navegadores Google Chrome y Safari [WEBKIT].

Trident

Trident es el motor de *renderizado* usado por Microsoft Internet Explorer. Fue diseñado como un componente de software que permitía a los programadores añadir la funcionalidad de navegación web a sus propias aplicaciones fácilmente. Presenta una interfaz COM para acceder y editar páginas web en cualquiera de los entornos que soporten COM, como C++ y .NET. Por ejemplo, un control de navegación web puede ser añadido a un programa en C++ y Trident puede usarse para acceder a la página mostrada en el navegador y acceder a valores de elementos. También se pueden capturar eventos de control del navegador web. Para poder habilitar la funcionalidad de Trident, es necesario conectar el archivo mshtml.dll al proyecto software [TRIDENT].

Comparación de los motores de renderizado más populares

	Gecko	WebKit	Trident
Lenguaje	C++	C++	C++
Desarrollado por	Mozilla Fundation / Netscape	Apple, Nokia, KDE, QT Software, Google y otros	Microsoft
Plataforma	Multiplataforma	Multiplataforma	Windows
Quien lo utiliza	Mozilla Firefox, Thunderbird, SeaMonkey, Camino, Pencil	Google Chrome, Safari, Konkeror, BlackBerry, Android	Internet Explorer, explorador de windows, RealPlayer, Outlook, reproductor Windows Media
Licencia	MPL, GPL, LGPL	GNU, LGPL, BSD	MS-EULA

4.5.2. Lenguajes de programación

Conforme fueron creciendo las aplicaciones web y sus distintos usos, se fueron complicando las páginas y las acciones que se querían realizar a través de ellas. Al poco tiempo quedó en evidencia que HTML no era suficiente para realizar todas las acciones que se pueden llegar a necesitar en una página web, ya que sólo sirve para presentar el texto en un página, definir estilos y navegabilidad. Al evolucionar los sitios web, una de las primeras necesidades fue que las páginas respondiesen a algunas acciones del usuario, para desarrollar pequeñas funcionalidades más allá de los propios enlaces.

El primer intento para cubrir las necesidades que estaban surgiendo fue Java, que había creado una manera de incrustar programas en páginas web. A través de la tecnología de los *applets*, se podían crear pequeños programas que se ejecutaban en el navegador dentro de las propias páginas web, pero que tenían posibilidades similares a los programas de propósito general. La programación de *applets* fue un gran avance ya que se había hecho posible la programación dentro de las páginas web.

HTML

HTML es un lenguaje utilizado para construir páginas web, que describe la estructura y el contenido de la página en forma de texto. Un documento HTML posee texto y etiquetas, llamadas *tags*, las cuales son anidadas conformando una estructura jerárquica. Estos *tags* representan distintos tipos de elementos y sus atributos. Permiten la utilización de hojas de estilo (archivos con extensión .CSS), archivos de *scripting*, como archivos Javascript, así como también embeber código Javascript dentro del código HTML. Un documento HTML es interpretado por el navegador, el cual a través de su motor de *renderizado*, grafica los componentes del documento.

Javascript

La incorporación de Javascript a HTML en el año 1995 significó el primer gran aporte a la programación del lado del cliente. Javascript es un lenguaje de *scripting*, interpretado, no tipado, orientado a objetos. Aunque es posible utilizarlo en el servidor, su uso se centra casi exclusivamente del lado del cliente. Comparado con los *applets* Java, que era la única posibilidad hasta el momento de incorporar código en las páginas HTML, Javascript es mucho más fácil de utilizar. Además, para programar Javascript no es necesario un *kit* de desarrollo, ni compilar los *scripts*, ni realizarlos en ficheros externos al código HTML, como ocurría con los *applets*.

Cascading Style Sheets

CSS es un lenguaje usado para describir la apariencia y el formato de documentos escritos en algún lenguaje de marcado, como XML, SVG y XUL. Su uso más común es para especificar el estilo de páginas HTML. CSS permite separar el código del documento de los detalles de su apariencia, lo cual aporta distintas ventajas:

- Control centralizado de la presentación de un sitio web completo con lo cual se agiliza de forma considerable la actualización del mismo.
- Los navegadores permiten a los usuarios especificar su propia hoja de estilo local, que será aplicada a un sitio web, con lo que aumenta considerablemente la accesibilidad. Por ejemplo, personas con deficiencias visuales pueden configurar su propia hoja de estilo para aumentar el tamaño del texto o remarcar más los enlaces.

- Una página puede disponer de diferentes hojas de estilo según el dispositivo que la muestre o, incluso, a elección del usuario. Por ejemplo, para ser impresa, para ser mostrada en un dispositivo móvil o para ser "leída" por un sintetizador de voz.
- El documento HTML en sí mismo es más claro de entender y se consigue reducir considerablemente su tamaño (siempre y cuando no se utilice el estilo embebido).

4.5.3. Frameworks para aplicaciones web

El diseño de aplicaciones web ha evolucionado a grandes pasos, hoy en día se dispone de numerosos *frameworks* para aplicaciones web haciendo uso de las bondades de la web 2.0. Éstos apoyan el desarrollo de sitios web dinámicos, aplicaciones web y servicios web. Intentan aliviar el exceso de carga asociado con actividades comunes usadas en desarrollos web facilitando la reutilización de código, por ejemplo, proporcionan bibliotecas para acceder a bases de datos, gestión de sesiones, seguridad, sistemas de manejo de *templates*, manejo de AJAX, *logging*.

Todos los *frameworks* y librerías de aplicaciones web intentan abarcar las mismas necesidades:

- Optimizar y maximizar el código que se ejecuta en el cliente a través del navegador.
- Minimizar procesamiento en el servidor.
- Minimizar la cantidad de información que se transfiere en cada petición.
- Optimizar los tiempos de respuesta.
- Simplificar el proceso de implementación, ofreciendo bibliotecas para acceder a bases de datos, gestión de sesiones, seguridad, sistemas de manejo de *templates*, manejo de AJAX, etc.

Ejemplos de frameworks web populares son:

- Java: Sping, Apache Wicket, JavaServer Faces, Apache Tapestry.
- .NET: ASP.NET MVC, MonoRail, DotNetNuke.
- Python: CherryPy, Django, TurboGears, Zope.
- PHP: CakePHP, Symfony, CodeIgniter.
- Javascript: Mootools, jQuery, PrototypeJS, DOJO, GWT (Google Web Toolkit).

4.5.4. Test Acid

Los tests Acid estan diseñados para evaluar en qué grado los navegadores respetan los estándares web. En la actualidad existen tres tests, los cuales fueron incluyendo nuevos aspectos a medida que la web fue evolucionando.

Acid Test 1

Esta basado exclusivamente en testear archivos de hoja de estilo (CSS), propiedades y valores válidos de los distintos elementos y resolución de conflictos entre distintas hojas de estilo. Para aprobar el test, el navegador debe *renderizar* una página HTML de manera que sea igual a una imagen de referencia.

Acid Test 2

Está basado en el test Acid 1, pero incorpora validaciones HTML, CSS 2.0, imágenes PNG y URIs. Al igual que el test Acid 1, éste es aprobado si el navegador logra *renderizar* una página HTML de manera idéntica a una imagen de referencia, la cual es una cara sonriente. Este test fue motivado especialmente para el navegador Internet Explorer de Microsoft, el cual hasta ese momento mostraba las páginas HTML de manera distinta al resto de los navegadores. El primer navegador en superar esta prueba fue Safari. Internet Explorer superó la prueba a partir de la versión 8.

Acid Test 3

El test Acid 3 está escrito en Javascript y se enfoca en testear las principales características de las aplicaciones de la web 2.0, como así también DOM y Javascript, incluyendo también los aspectos testeados por Acid 2. Para superar la prueba el navegador debe *renderizar* una página HTML, obteniendo así una puntuación de 1 a 100, la cual evalúa el grado de conformidad con el test. Además, la página debe ser cargada por el navegador en menos de 33 milisegundos.

4.6. Interfaces gráficas en aplicaciones para dispositivos móviles

Las interfaces para dispositivos móviles han crecido enormemente durante los últimos años. El avance de la tecnología sobre dispositivos inalámbricos y el desarrollo de tecnologías como GSM, GPRS y *bluetooth*, ofrece a los usuarios mayor flexibilidad y posibilidades.

Implementación de interfaces a bajo nivel para dispositivos móviles

Los sistemas operativos móviles son bastantes más simples y están orientados a la conectividad inalámbrica, los formatos multimedia para móviles y las diferentes maneras de introducir información en ellos. Requieren características de multitarea, personalización, manejo de pantalla táctil y tienda de aplicaciones.

Una de las grandes diferencias en la experiencia con la interfaz reside en el tipo de pantalla: capacitiva o resistiva. Mientras que las segundas han sobrevivido el paso del tiempo, las capacitivas son la nueva alternativa, en especial porque no requieren de objetos externos para funcionar.

Interfaces naturales de usuario

La interfaz natural de usuario (NUI) es aquella en las que se interactúa con una aplicación sin utilizar sistemas de mando o dispositivos de entrada como sería un *mouse*, teclado, lápiz óptico, *touchpad*, *joystick,* etc. En su lugar, se hace uso de movimientos gestuales tales como las manos o el cuerpo, en el caso de pantallas capacitivas multitáctiles la operación es por medio de la yemas de los dedos en uno o varios contactos. También se están desarrollando sistemas operativos controlados por medio de voz humana y control cercano a la pantalla pero sin tocarla [NUI].

Android

Android es un sistema operativo basado en Linux para dispositivos móviles, desarrollado por Open Handser Alliance y Google inc. El núcleo de Android está escrito en C y C++, y las librarías de la interfaz de usuario escritas en Java, siguiendo el estándar OpenGL. Posee un administrador de interfaz gráfica (*surface manager*), un *framework* OpenCore, una base de datos relacional SQLite, una API gráfica OpenGL ES 2.0 3D, un motor de *renderizado* WebKit, un motor gráfico SGL, SSL y una biblioteca estándar de C Bionic.

La plataforma es adaptable a pantallas más grandes, VGA, biblioteca de gráficos 2D, biblioteca de gráficos 3D basada en las especificaciones de la OpenGL ES 2.0.

El navegador web incluido en Android está basado en el motor de *renderizado* WebKit, emparejado con el motor JavaScript V8 de Google Chrome. El navegador obtiene una puntuación de 93/100 en el test Acid3.

Android soporta *tethering*, el cual permite al teléfono ser usado como un punto de acceso alámbrico o inalámbrico.

Blackberry OS

Blackberry OS [BLACKBERRY] es un sistema operativo propietario desarrollado para los dispositivos móviles Blackberry por la empresa Research In Motion. Está escrito en C++, y está fuertemente orientado a ofrecer al usuario una experiencia multimedia ágil, un buen manejo de correo electrónico, y una amplia variedad de aplicaciones. Para esto posee manejo multitarea y soporta diversos dispositivos de entrada como pantalla táctil, *trackball*, *trackwheel* y *trackpad*.

Es posible crear aplicaciones utilizando MIDP (Mobile Information Device Profile) 2.0 APIs , Scalable Vector Graphics APIs o BlackBerry UI APIs. Esta última API es desarrollada para la construcción de interfaces de usuario sobre la plataforma Blackberry, y ofrece un conjunto de clases y funciones especialmente diseñados para el manejo de correo electrónico, navegador web y funciones específicas del teléfono[BLACKBERRYUI].

iOS

La plataforma móvil iOS es una de las más avanzada y constantentemente se redefine para mejorar la interacción del usuario. Para desarrollar aplicaciones se utiliza el SDK de iOS junto con el IDE Xcode y el *framework* Cocoa. Permite definir todo tipo de objetos para vistas y controles, además existen muchas herramientas externas para el diseño de la interfaz [IOS].

Symbian

Symbiam es un sistema operativo propietario (desde 2010, antes era software libre) desarrollado por la compañía Symbian Ltd; y que luego fue adquirido por Nokia. Originalmente Symbian no proveía ninguna interfaz gráfica; la misma debía ser desarrollada de manera independiente al sistema operativo, ofreciendo así mayor flexibilidad. Las interfaces gráficas para Symbian son MOAP, Series 60, Series 80, Series 90 y UIQ. La versión Symbian^3 incluye el *framework* Qt, el cual es el recomendado para el desarrollo de nuevas aplicaciones. La nueva versión Symbian^4 proyecta incorporar un nuevo *framework* gráfico basado en Qt, llamado UIEMO (UI Extensions for Mobile).

Symbian fue el primer sistema operativo para dispositivos móviles en utilizar un navegador propio basado en el motor de *renderizado* WebKit.

Windows Phone

Desarrollado por Microsoft y diseñado para su uso en teléfonos inteligentes (s*martphones*) y otros dispositivos móviles. Se basa en el núcleo del sistema operativo Windows CE y cuenta con un conjunto de aplicaciones básicas utilizando las API de Microsoft Windows. Está diseñado para ser similar estéticamente a las versiones de escritorio de Windows. Además, existe una gran oferta de software de terceros disponible para Windows Phone, la cual se puede adquirir a través de *Windows Marketplace for Mobile*.

Para desarrollar se cuenta con el *Windows Phone SDK*, que agrega al IDE del Visual Studio una barra de herramientas que incluye controles de usuario, un diseñador de *skin*, *templates* para crear proyectos específicos para móviles, y un emulador, en el cual se puede desplegar y realizar depuración y testeo de las aplicaciones [WINPHONE].

Meego

Meego es un sistema operativo de código abierto, diseñado para diversos tipos de dispositivos móviles, como *tablets*, *smartphones*, dispositivos para vehículos y televisores. Meego surge como la fusión de los sistemas operativos Maemo de Nokia y Moblin de Intel. El *core* de MeeGo es una distribución de Linux, basada en Debian de Maemo y Fedora de Moblin [MEEGO].

Los diversos tipos de dispositivos móviles utilizan el mismo *core* de Meego, pero con distintas interfaces de usuarios, las cuales son llamadas *"User Experiences"*. Para cada tipo de dispositivo, la interfaz gráfica es implementada de manera diferente, la mayoría utiliza Qt y el *framework* MeeGo Touch para los dispositivos táctiles. Para el desarrollo de aplicaciones, además de Qt, también es posible utilizar GTK.

4.7. Otros tipos de interfaces

En un futuro la forma de interacción de los humanos con las computadoras cambiará de forma radical. Al aumentar el poder de cómputo, formas de interacción planteadas como futuristas se están haciendo realidad. Algunos ejemplos de esto son:

Interfaz de usuario basada en el habla (SUI - Speech User Interface)

Es aquella que soporta un diálogo interactivo entre el usuario y una aplicación de software, ofreciendo una forma de interacción más natural para el usuario. Un escenario propicio para este tipo de interfaces es el de dispositivos móviles, en especial para plataformas como Auto-PC [AUTOPC] donde se requiere que el usuario tenga desocupadas las manos para conducir.

Interfaces neuronales

Los dispositivos de interfaz neuronal (Neural Interface Devices) permiten a los usuarios aprovechar las señales eléctricas generadas por sus cuerpos para controlar una computadora o dispositivos eléctricos conectados a ella. Las señales pueden adquirirse de forma directa o indirecta. Los métodos directos implican implantes quirúrgicos en el cuerpo del usuario. El implante puede localizarse en el cerebro o en algún miembro amputado, dependiendo de la aplicación. Este tipo de tecnología aún está en desarrollo. Los métodos indirectos de obtención de señales no requieren cirugía. Estas señales neuronales indirectas pueden obtenerse de los músculos, movimiento de los ojos o de las ondas cerebrales.

Después de que las señales eléctricas se obtienen del cuerpo se amplifican y se traducen en su equivalente digital para que el software pueda interpretar las señales. Dicho software permite a los usuarios controlar un cursor, jugar videojuegos especialmente diseñados o controlar otros dispositivos conectados a la computadora [NEI].

4.8. Conclusión

Las interfaces de usuario cumplen un rol fundamental en la usabilidad de las aplicaciones. Las mismas han evolucionado enormemente en los últimos años, y en la actualidad se cuenta con interfaces sofisticadas, que intentan reducir las distancias tecnológicas entre el usuario y las computadoras, haciendo uso natural de las mismas a través de las manos, movimientos o la voz. También se han desarrollado numerosos *frameworks* que simplifican y mejoran la implementación de las interfaces gráficas, especialmente para plataformas web.

Por otro lado se analizó la implementación de las interfaces en las distintas plataformas (en aplicaciones de escritorio, en aplicaciones web, en aplicaciones para dispositivos móviles), la cual se compone de una arquitectura en capas, donde las capas inferiores manejan cuestiones de bajo nivel, asociadas al hardware, y a medida que se asciende, cada capa abstrae aspectos de implementación. Se analizaron cuáles aspectos de las interfaces de usuario son comunes a todas las plataformas, determinando controles, propiedades y eventos que tienen similar utilidad en todas ellas. Estas cuestiones analizadas, fueron un gran aporte a la herramienta que se presentará en la parte III.

Parte III
GuiDSL

Una herramienta MDA enfocada a interfaces gráficas

En la parte III se presenta la herramienta GuiDSL: sus características y beneficios principales, se describe su lenguaje, desde el punto de vista sintáctico y semántico, y se especifica cada uno de los elementos del lenguaje. Luego se explica como GuiDSL utiliza las transformaciones para obtener el código fuente de la aplicación. Se muestra cómo utilizar la herramienta con un ejemplo sencillo, abarcando todos los pasos del proceso: desde la creación del modelo de la aplicación, hasta ejecutar el código generado. Se presentan los detalles de implementación más importantes y se explican las herramientas utilizadas, cómo se implementan los chequeos del lenguaje y la arquitectura de las transformaciones. Finalmente se muestra cómo es posible extender la herramienta, tanto el lenguaje como las transformaciones, y se explica conceptualmente cómo hacerlo.

5. Presentando GuiDSL

5.1. Introducción

Como se analizó en el capítulo 3, las herramientas MDA existentes, poseen aún algunas limitaciones importantes:

- las aplicaciones generadas requieren ser adaptadas o configuradas para su ejecución,
- generar para un único lenguaje o plataforma,
- la especificación del PIM no abarca todos los aspectos necesarios para definir de manera completa la aplicación.

Estas cuestiones de índole técnica, hacen que aún la utilización de herramientas MDA no se encuentre instalada masivamente en el proceso de desarrollo de software. Por otro lado, tal como se analizó en el capítulo 2, las interfaces gráficas son costosas de implementar y cumplen un rol clave en el éxito de una aplicación.

Como consecuencia se desarrolló la herramienta GuiDSL, la cual ofrece una solución a estos inconvenientes ya que facilita y mejora el proceso de implementación de interfaces gráficas.

Con una visión global de la herramienta, se puede describir a GuiDSL de la siguiente manera:

El lenguaje

Se creó un DSL, el cual ofrece elementos básicos, pero suficientes, para definir cualquier tipo de aplicación de manera completa. Posee reglas que representan sentencias básicas, presentes en la mayoría de los lenguajes de programación, las cuales permiten especificar el manejo de eventos, la lógica del negocio y cualquier método o rutina que se requiera. Esto aporta gran flexibilidad, ya que permite definir de manera completa la aplicación, lo cual se traduce luego, en código 100 % listo para ejecutar. Posee reglas que permiten especificar las interfaces gráficas, definiendo controles, con sus propiedades, eventos y apariencia. El lenguaje posee distintos tipos de chequeos sintácticos y semánticos, los cuales verifican que los PIMs sean válidos. De ser necesario, el lenguaje podrá ser extendido fácilmente, incorporando reglas nuevas, o reglas que optimicen, personalicen o amplíen las existentes.

La herramienta

La herramienta se integra al IDE Eclipse como un *plugin*. Permite definir de manera textual la aplicación, a través de un editor que posee distintas utilidades como resaltador de sintaxis, autocompletar y chequeos sintácticos y semánticos.

Las transformaciones

Incluye transformaciones para algunos lenguajes populares como CSharp, Java o PHP; e incluye también el uso de librerías estándar como Nhibernate, Castle Monorail o Castle ActiveRecord. Estas transformaciones se implementan a través de clases que respetan determinada estructura. Es así, que respetando dicha estructura, es posible fácilmente extender estas transformaciones o crear nuevas, que se adapten a necesidades específicas.

Combinando distintas transformaciones, se crean los PSM para una plataforma, lenguaje y tecnología específica. GuiDSL incluye un PSM completo a modo de ejemplo para plataforma web, utilizando el *framework* Castle (MVC + NHibernate).

La generación

Una vez definida la aplicación en nuestro lenguaje, y determinado qué PSM utilizar, se realiza la generación del código fuente. Ésto es posible realizarlo de manera integrada al IDE, basta con seleccionar el PSM a utilizar.

5.2. Principales características

Reusabilidad, portabilidad

Siguiendo los principios básicos de MDA, reusabilidad y portabilidad son las principales características de la herramienta. La aplicación se define en GuiDSL una única vez, luego podrá ser transformada en múltiples lenguajes sin necesidad de redefinir o adaptar ningún aspecto original. Esto reduce enormemente

los costos de migración a otras tecnologías o plataformas. Todo lo que se especifica en el nivel de PIM es completamente portable y sólo depende de las herramientas de transformación disponibles. Se escribe una vez, se transforma a múltiples tecnologías o plataformas.

Definición personalizada de UI

GuiDSL permite especificar de manera detallada el diseño de las interfaces de usuario, estilos y manejo de eventos. Ésta es una importante ventaja sobre la gran mayoría de los generadores de código, ya que éstos sólo generan interfaces sencillas para realizar operaciones tipo CRUD sobre las entidades del modelo.

Posibilidad de inclusión de código

GuiDSL permite incluir código especificado con sentencias propias, las cuales son un conjunto básico de sentencias con las que cualquier programador se encuentra familiarizado. El código permite especificar el manejo de eventos, las reglas del negocio y cualquier rutina que sea necesaria para la aplicación. Esta característica permite definir el modelo de manera completa, sin necesidad de alterar manualmente el código generado para incorporar las rutinas necesarias, como ocurre con la mayoría de los generadores de código.

Extensibilidad

Los *templates* utilizados para las transformaciones a código ejecutable, son clases independientes (las cuales serán descritas en el capítulo 7). Es posible extender estas clases para incorporar o modificar aspectos particulares de la transformación, incorporar el uso de una librería, o definir una transformación para un nuevo lenguaje o plataforma.

Modularización de templates

El esquema de generación de código utiliza una arquitectura de *templates* modularizada, donde es posible intercambiar una capa o aspecto en particular de la transformación. Este caso puede darse por un cambio de arquitectura o cambio de librerías. Por ejemplo, si cuento con una transformación que realiza el acceso a datos utilizando NHibernate, y luego deseo utilizar Active Records; o también ante la necesidad de extender u optimizar alguna transformación utilizada; por ejemplo, si deseo incorporar una librería propia que extiende NHibernate para loguear modificaciones sobre la base de datos. Ante estas circunstancias es posible modificar únicamente las clases que realizan esa parte de la transformación, sin necesidad de alterar el resto de las transformaciones.

Mantenimiento

Cualquier requerimiento de cambio o nueva característica, podrá ser incorporada al modelo y luego verla reflejada en código ejecutable fácilmente. Cuando la aplicación se encuentre productiva en múltiples plataformas o tecnologías, con sólo modificar el metamodelo se puede regenerar cada una de sus instancias, sin necesidad de sumergirse en las cuestiones técnicas de cada una de ellas.

Legibilidad

GuiDSL posee un conjunto de primitivas de sintaxis sencilla, que aportan legibilidad al modelo y que permiten al programador abstraerse de los detalles técnicos de las distintas implementaciones. Un conjunto de primitivas de GuiDSL pueden ser transformada en numerosas líneas de código o diversos archivos, dependiendo de la transformación aplicada. Por ejemplo, un vista definida en GuiDSL en unas pocas lineas de código, y que luego es transformada en una aplicación web que utiliza jQuery, el código resultante será mucho más extenso y estará compuesto por distintos tipos de archivos, como hojas de estilo, archivos Javascript y archivos HTML.

Facilidad de documentación

La documentación fue siempre un punto débil en el proceso del desarrollo de software. Una solución a este problema, a nivel de código, es la facilidad de generar documentación directamente desde el código fuente, asegurándose que esté siempre actualizada. Esta solución, sin embargo, soluciona únicamente el problema de la documentación a nivel código fuente.

Dada la complejidad de las aplicaciones que se construyen, la documentación en un nivel más alto de abstracción resulta obligatoria. Con GuiDSL el desarrollador puede enfocarse sólo en el modelo, el cual a su vez tiene un nivel más abstracto que el del código. El modelo es utilizado para generar el código

ejecutable, por lo tanto, será una exacta representación del código. Así el modelo definido en GuiDSL formará parte de la documentación, necesaria en cualquier aplicación.

La gran ventaja es que el modelo no se deja de lado luego de ser escrito. Cuando surja alguna necesidad de cambio en la aplicación, éstos se aplicarán sobre el modelo para luego volver generar el código, el cual incluirá dichos cambio. Esto nos garantiza integridad entre el modelo y el código ejecutable generado a partir del mismo, aún en etapas de mantenimiento.

Por otro lado, también es posible crear un PSM que permita documentar la aplicación a partir del PIM.

Productividad

Todas las características descritas anteriormente hacen que la productividad se vea directamente beneficiada:

- Reusabilidad del modelo.
- Posibilidad de personalización de los *templates*.
- Facilidad de mantenimiento.
- No es necesario sumergirse en cuestiones técnicas de la herramienta o de la tecnología cuando se dispone del correspondiente PSM.
- Abstracción del problema, el programador se centra en el modelo manejando una única herramienta, y no en detalles técnicos de implementación para un lenguaje particular.

En los capítulos venideros, se describirá GuiDSL en detalle: cómo es el lenguaje, cómo se realizan las transformaciones y cómo utilizarlo, con un ejemplo completo.

5.3. GuiDSL y la arquitectura de las interfaces de usuario

La arquitectura de las interfaces de usuario se compone de capas que interactuan ofreciendo servicios a la capa superior. A bajo nivel encontramos los sistemas operativos que se comunican con los dispositivos físicos a través de sus controladores. Luego se encuentran las distintas librerías gráficas las cuales son las responsables de la apariencia y comportamiento de los controles. Luego, los lenguajes de programación pueden hacer uso de estas librerías gráficas externas a través de *bindings*, o utilizar librerías de controles propias. Finalmente, a más alto nivel, encontramos los distintos frameworks, que encapsulan comportamiento y resuelven distintos aspectos de programación, no sólo a nivel de las interfaces gráficas. Muchos de estos conceptos fueron presentados en el capítulo 4 donde se analizó en detalle la implementación de estas capas de acuerdo a las plataformas. Estos conceptos fueron los que sirvieron de base para el diseño de GuiDSL, tanto para el lenguaje como para la arquitectura de las transformaciones.

En la figura 10 se puede observar la arquitectura en capas de las interfaces de usuario y cómo se relacionan entre ellas de acuerdo al contexto de aplicación. Como se puede observar, GuiDSL abstrae en una única herramienta todos estos niveles, permitiendo generar código través de transformaciones que combinan distintas librerías, lenguajes o *frameworks* para cada nivel de la arquitectura. Por ejemplo, se podría utilizar una transformación que genere código para un dispositivo móvil a través de una aplicación Java, que utilice la librería gráfica Qt, y se ejecute sobre el sistema operativo Symbian. Luego, utilizando el mismo modelo de entrada, se podría generar la aplicación para una plataforma web, utilizando el *framework* Castle MVC, el lenguaje de programación CSharp y ejecutarla sobre un sistema operativo Windows a través de un navegador.

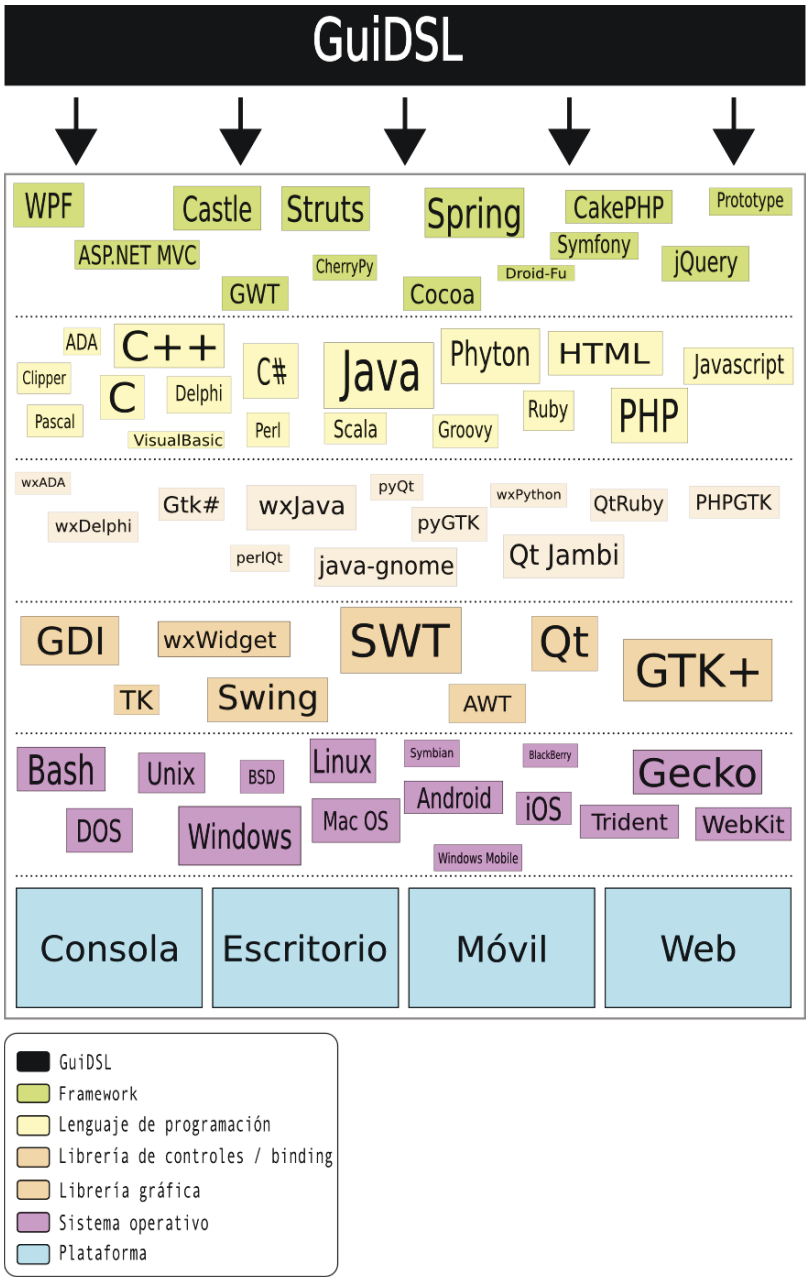

Figura 10: GuiDSL y su interacción con las distintas arquitecturas de las interfaces de usuario

5.4. Correspondencia con la arquitectura de la OMG

GuiDSL es una herramienta que implementa la arquitectura MDA definida por la OMG. Tal como se analizó en el capítulo 3, MDA establece 4 niveles los cuales se encuentran presentes en nuestra solución:

M3. Meta-metamodelo

GuiDSL está implementado utilizando el *framework* Xtext, el cual a su vez se basa en el conjunto de herramientas EMF. EBNF es el lenguaje de meta-metamodelado de base.

M2. Metamodelo

Para el metamodelo GuiDSL define un DSL, el cual permite definir una aplicación de manera completa.

M1. Modelo

La aplicación se define haciendo uso de los elementos que ofrece el DSL, en archivos con una extensión particular.

M0. Aplicación

El modelo de la aplicación es transformado utilizando un PSM específico. Así se obtiene código ejecutable para la plataforma y el lenguaje de programación correspondiente al PSM utilizado.

5.5. Conclusión

Apoyándose en los conceptos básicos de la iniciativa MDA, y teniendo en cuenta el costo de implementar las interfaces gráficas y la importancia que éstas tienen, la presente tesina se basa en la definición de una herramienta enfocada en la creación de interfaces gráficas. Así, a través de transformaciones sobre un único modelo de entrada, se puede obtener la generación completa de una aplicación en distintos lenguajes de programación y plataformas, permitiendo generar la apariencia, comportamiento y navegabilidad de las interfaces gráficas del proyecto.

6. El lenguaje

6.1. Introducción

En este capítulo se describirá en detalle la sintaxis y la semántica el metalenguaje de GuiDSL, y se analizarán características y propiedades de cada uno de sus elementos.

6.2. Sintaxis

6.2.1. Elección de la sintaxis

Al comenzar a diseñar GuiDSL se evaluaron diversas alternativas sobre cómo especificar la sintaxis del lenguaje.

En primer lugar, se evaluó utilizar una sintaxis secuencial, donde los elementos se definían uno debajo de otro, y cada uno de ellos, poseía una referencia a su elemento contenedor. En el siguiente ejemplo se muestra la sintaxis evaluada.

Bloque de código 1 Ejemplo de sintaxis secuencial

```
1  Form fEditar = new Form();
2  fEditar.id = "frmEditar";
3  fEditar.isInitial = true;
4
5  Form fAlta = new Form();
6  fAlta.id = "frmAlta";
7  fAlta.isInitial = false;
8
9  Button bGrabar = new Button();
10 bGrabar.id = "btnGrabar";
11 bGrabar.refView = fEditar;
12
13 Button bCancelar = new Button();
14 bCancelar.id = "btnCancelar";
15 bCancelar.refView = fAlta;
```

Esta opción se descartó debido a las siguientes razones:

- Poco legible, ya que los elementos no están necesariamente ordenados por su contenedor.
- Se debe recordar y repetir el identificador de cada elemento contenedor.
- Escritura costosa, ya que se debe referenciar el objeto contenedor para cada elemento lo cual implica más código a escribir.

Luego se decidió utilizar elementos anidados para resolver estos inconvenientes. Así se evaluó una sintaxis tipo XML (bloque de código 2), la cual resuelve los dos primeros inconvenientes, ya que al ser de estructura anidada, los elementos se encuentran agrupados por su contenedor y no es necesario referenciarlo. Sin embargo, esta notación sigue siendo de escritura costosa ya que cada elemento requiere de un *tag* particular de apertura y otro de cierre.

Bloque de código 2 Ejemplo de sintaxis anidada (XML)

```
<form>
    <id>fEditar</id>
    <properties>
        <property name=isInitial>
            true
        </property>
    </properties>
    <controls>
        <button>
            <id>btnGrabar</id>
        </button>
    </controls>
</form>

<form>
    <id>fAlta</id>
    <properties>
        <property name=isInitial>
            false
        </property>
    </properties>
    <controls>
        <button>
            <id>btnCancelar</id>
        </button>
    </controls>
</form>
```

Finalmente se decidió utilizar otra sintaxis, tipo JSON, donde cada elemento también posee delimitadores de apertura y cierre, pero que utiliza el mismo para todos los tipos de elementos. En este caso se utilizó el carácter "{" para apertura y el carácter "}" para el cierre (bloque de código 3).

Bloque de código 3 Ejemplo de sintaxis anidada (simil JSON)

```
form fEditar {
    isInitial   true
    controls [
        button btnEditar { ... }
    ]
}

form fAlta {
    isInitial   false
    controls [
        button btnCancelar { ... }
    ]
}
```

Cada elemento tendrá un tipo, un identificador, el delimitador de apertura "{" y luego de manera anidada podrá contener propiedades simples, listas u otros elementos. Finalmente tendrá el delimitador de cierre "}", tal como se puede apreciar en el bloque de código 4.

Bloque de código 4 Ejemplo sintaxis GuiDSL

```
tipoElementoA identificadorElemA {
    simpleProperty1 "string value"
    simpleProperty2 true
    simpleProperty3 34

    listPropertyB [
        tipoElementoC identificadorElemC { ... }
        ...
    ]

    composePropertyD identificadorElemD {
        property4 ...
        ...
    }
    ...
}
```

6.2.2. Características

GuiDSL está basado en una gramática libre de contexto del tipo EBNF. La sintaxis es simple y clara, puede ser leída fácilmente y utiliza una mínima cantidad de código para cada elemento [GJC].

Indentación, espacios en blanco y manejo de lineas

GuiDSL ignora los espacios en blanco, tabuladores y retornos de carro, ya que no forman parte de ninguna sentencia del lenguaje. Las sentencias son *parseadas* de manera continua, sin tener en cuenta ninguno de estos caracteres. Por este motivo, tampoco es necesaria la indentación de código (aunque es sumamente útil para la legibilidad del mismo).

Palabras clave

GuiDSL posee un conjunto acotado de palabras claves las cuales no pueden ser utilizadas como identificadores de ningún tipo de elemento del modelo, ni en expresiones.

Literales

Son valores constantes. Podrán ser:

- Números enteros.
- Cadenas de caracteres.
- Valores lógicos.

Operadores

GuiDSL posee operadores lógicos, de comparación y aritméticos.

Comentarios

GuiDSL posee comentarios de línea y de bloque.

6.3. Semántica

GuiDSL realiza distintos chequeos semánticos de manera estática. Luego de chequear la sintaxis, el compilador ejecuta chequeos de tipos, propiedades, acciones, alcance de las variables, operadores, etc. Estos chequeos podrán ser de tipo advertencia, los cuales no impiden la compilación; o de tipo error, que deben ser corregidos para la compilación.

Cuando alguno de estos chequeos no se cumple, el compilador advierte de la situación.

Chequeo de tipos

GuiDSL es fuertemente tipado. Chequea la correspondencia entre los tipos de las expresiones y los tipos esperados de acuerdo a cada sentencia En la figura 11 se observa cómo GuiDSL controla en tiempo de diseño la correspondencia entre los tipos de las expresiones y de las variables.

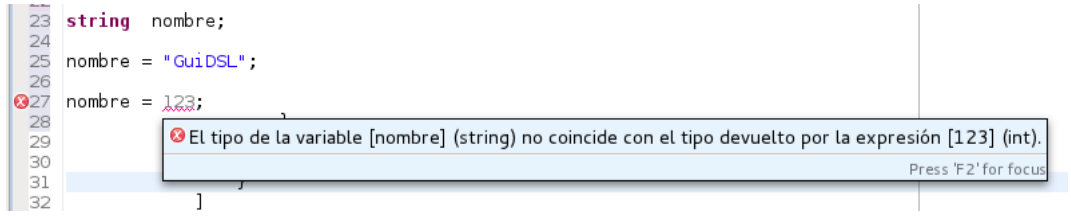

Figura 11: Chequeo de tipos en GuiDSL

Chequeos de propiedades y acciones

GuiDSL chequea que las propiedades y acciones de cada elemento sean válidas para el tipo de elemento.

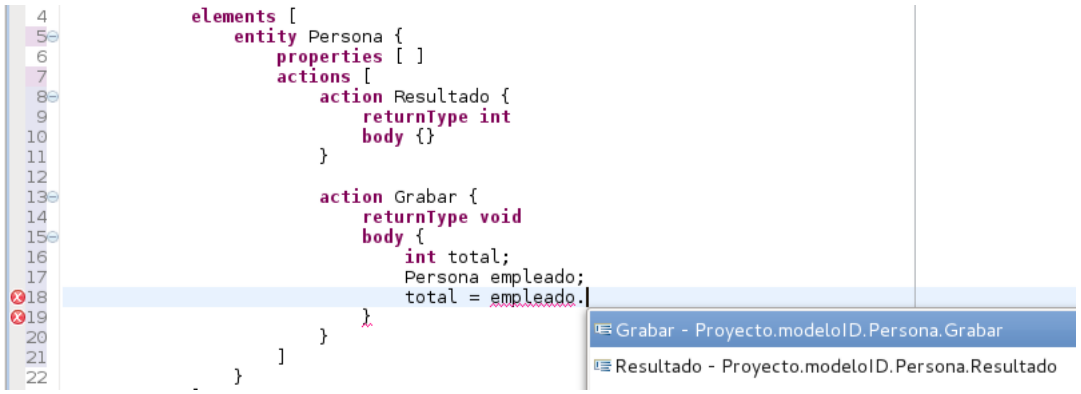

Figura 12: Chequeos de propiedades y acciones

Alcance de las variables

Las variables, ya sean locales o parámetros, deben estar definidas explícitamente. El alcance es dentro del módulo que la declara y a partir de su definición.

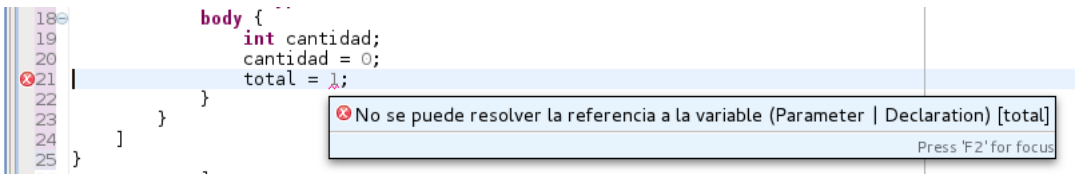

Figura 13: Alcance de las variables

Otros chequeos

GuiDSL realiza también los siguientes chequeos semánticos:

- Unicidad de identificadores: los identificadores de elementos no podrán repetirse dentro del proyecto.
- Referencias entre elementos: el compilador chequeará que los identificadores referenciados puedan ser resueltos, y que el tipo coincida con el esperado.

- Chequeo sobre el tipo *void: void* es un valor especial, el cual sólo es posible utilizarlo como tipo de retorno. GuiDSL controla que *void* sólo forme parte se sentencias de tipo *returnType* (no es posible utilizarlo como tipo para la sentencias declaración, parámetros o propiedades).
- Chequeo de expresiones bien formadas: para las expresiones que se compongan de más de un término o factor, y contengan operadores, GuiDSL controla que los operadores de correspondan con los factores sobre los que operan.
- Chequeo sobre el factor *null:* se controla que el factor *null* no sea asignado a variables booleanas.
- Proyecto inicial: controla que el proyecto tenga un grupo inicial, y sólo uno.
- Vista inicial: controla que cada grupo del proyecto tenga una vista inicial y sólo una.
- Correspondencia de argumentos y parámetros: controla la correspondencia en cantidad y tipo entre los parámetros definidos en una acción, y los argumentos definidos en cada invocación a la acción.
- Chequeo de nombre de entidades: chequea que los nombres de las entidades comiencen con mayúscula. Este chequeo es de tipo advertencia.

Existen también chequeos semánticos particulares para cada tipo de elemento del DSL, los cuales serán descritos en detalle en el capítulo 6.4.

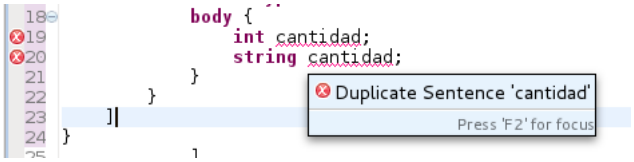

Figura 14: Otros chequeos, unicidad de identificadores

Vista Markers

GuiDSL posee una vista que muestra el total de errores y advertencias del proyecto abierto.

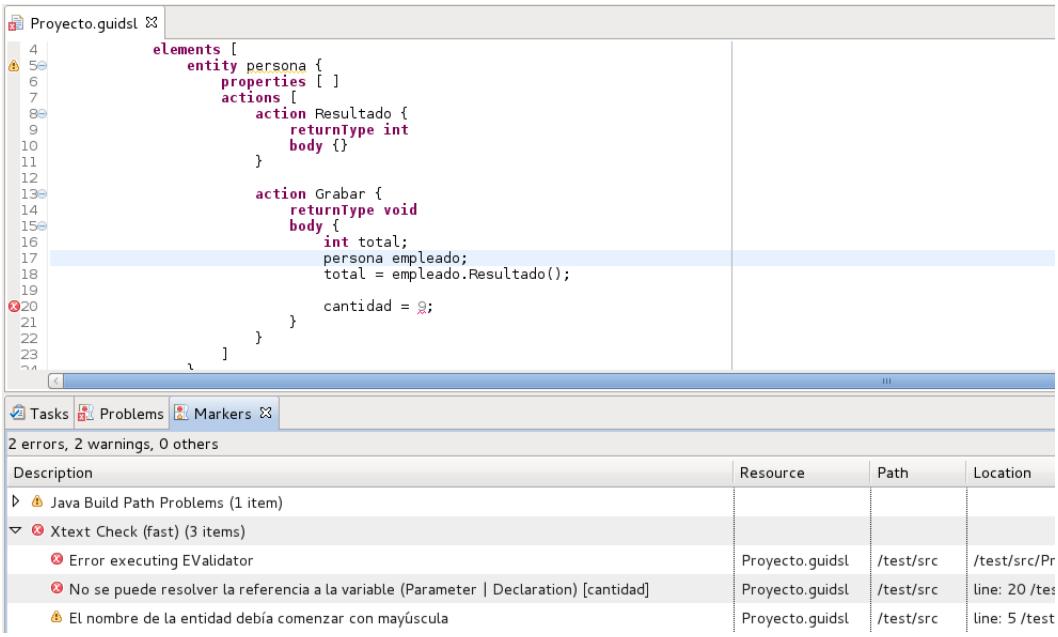

Figura 15: Vista Markers, agrupa errores y advertencias sintácticas y semánticas

6.4. Especificación del lenguaje

El lenguaje de GuiDSL posee reglas que permiten definir una aplicación de manera completa: las entidades del modelo, acciones y sus interfaces gráficas con su apariencia y el manejo de todos los eventos. Para esto, GuiDSL utiliza un conjunto mínimo de reglas que abarcan todos estos aspectos. Las reglas que permiten definir las interfaces gráficas fueron determinadas como consecuencia del análisis de la arquitectura de éstas presentado en el capítulo 4. Se determinó cuáles son los controles presentes en todas las plataformas, cuáles son sus propiedades y cómo es el manejo de sus eventos. Las reglas del lenguaje de acciones también se establecieron analizando los lenguajes de programación más populares para las distintas plataformas. Así, se determinó un conjunto mínimo de sentencias que permiten realizar todo lo necesario en las acciones.

Los componentes de GuiDSL se clasifican en 4 grupos principales: sistema de tipos, modelo, interfaces gráficas y lenguaje de acciones, los cuales están contenidos dentro del proyecto (figura 16). Para cada uno de los componentes se describirá su gramática, elementos, chequeos semánticos y se mostrará un ejemplo de su uso.

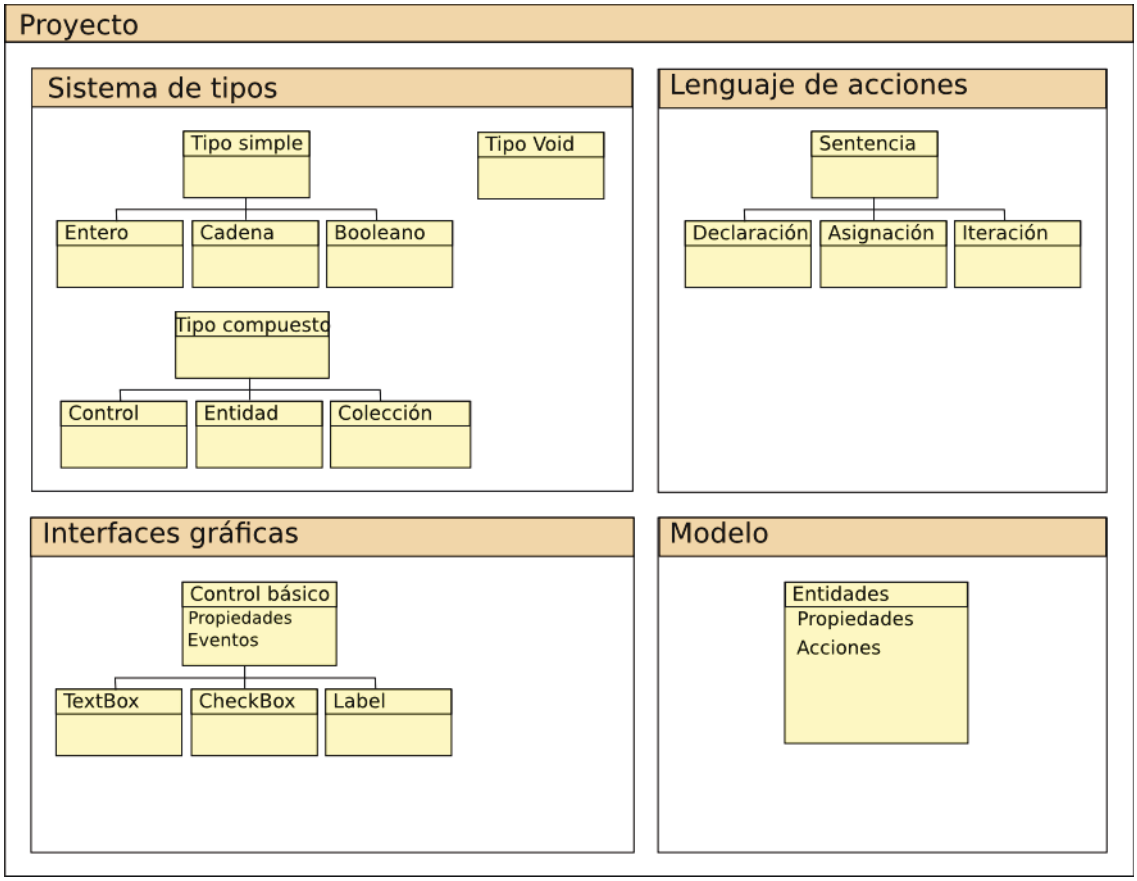

Figura 16: Estructura del lenguaje GuiDSL

6.4.1. Proyecto

Regla Project

Una aplicación en GuiDSL comienza con la definición de un proyecto, el cual será el objeto contenedor de todos los elementos de la aplicación. Existirá un único proyecto para la aplicación.

Gramática

```
Project:
    'project' name=ID '{'
        ('modelAssociated' model=Model)?
        ('groups' '[' (groups+=Group)+ ']')?
        ('actions' '[' (actions+=Action)+ ']')?
        ('styles' '[' (styles+=StyleDeclaration)+ ']')?
'}';
```

Elementos

Elemento	Requerido	Descripción
name	Si	Es el identificador del proyecto.
modelAssociated	No	Es el modelo asociado al proyecto. El modelo contendrá las entidades que conforman el dominio de la aplicación.
groups	No	Lista de uno o más elementos de tipo *Group*. Los grupos permiten agrupar conceptualmente las vistas y las acciones, permitiendo ordenar y modularizar la aplicación.
actions	No	Lista de uno o más elementos de tipo *Action*. Estas acciones serán globales, es decir, podrán ser accedidas desde cualquier punto de la aplicación. Este elemento será útil cuando se deseen modelar aplicaciones sin interfaces gráficas, de tipo librería.
styles	No	Lista de uno o más elementos de tipo *StyleDeclaration*. Estos elementos permiten definir estilos que podrán ser utilizados en las vistas.

Comentarios

Un proyecto no necesariamente debe definir vistas. Es posible modelar aplicaciones de tipo librería, las cuales podrían ser importadas y utilizadas por otras aplicaciones.

Ejemplo

Bloque de código 5 Ejemplo de uso de la regla *Project*

```
project ProyectoDeEjemplo {
    modelAssociated model ModeloA {
        ...
    }

    groups [
        group GrupoFacturacion {
            views [
                    view AltaFactura {
                                    ...
                    }
            ]
        }
        ...
    ]

    styles [
        style linkVisitado {
            properties [
                foreColor violet
            ]
        }
    ]
    ...
}
```

Regla Group

La regla Group permite agrupar lógicamente vistas y acciones, permitiendo modularizar la aplicación.

Gramática

```
Group:
    'group' name=ID '{'
        ('isInitial' isInitial=BoolLiteral)?
        ('actions' '[' (actions+=Action)* ']')?
        ('views' '[' (views+=View)* ']')?
    '}';
```

Elementos

Elemento	Requerido	Descripción
name	Si	Es el identificador del grupo.
isInitial	No	Indica si es o no el grupo inicial del proyecto.
actions	No	Lista de uno o más elementos de tipo *Action*.
views	No	Lista de uno o más elementos de tipo *View*.

Ejemplo

Bloque de código 6 Ejemplo de uso de la regla *Group*

```
project ProyectoDeEjemplo {
    ...
    groups [
        group GrupoFacturacion {
            views [
                    view AltaFactura {
                                    ...
                    }
            ]
        }
        ...
    ]
    ...
    groups [
        group GrupoControlDeStock {
            views [
                    view ConsultarStock {
                                    ...
                    }
            ]
        }
        ...
    ]
    ...
}
```

6.4.2. Sistema de tipos

GuiDSL ofrece un conjunto básico y acotado de tipos, necesarios para poder representar modelos mínimos. Estos tipos podrán ser extendidos fácilmente de acuerdo a las necesidades del modelo a representar. Por ejemplo, del tipo *entero*, es posible derivar los tipos *entero corto* o *entero largo*. También es posible crear nuevos tipos de datos estándares, como el tipo *fecha y hora*, o definir tipos personalizados, explotando el uso de GuiDSL. Por ejemplo, se podría definir el tipo de datos "*nombre propio*", el cual deriva del tipo *cadena* y agrega restricciones: debe comenzar con mayúscula y debe contener solo un conjunto determinado de caracteres.

El sistema de tipos contempla tipos de datos simples y compuestos.

Gramática

```
Type:
    SimpleType | CompositeType | VoidType;

SimpleType:
    IntType | BoolType | StringType;

CompositeType:
    ControlType | EntityType | CollectionType;
```

Tipos de datos simples

Los tipos de datos simples son:

Entero

Está representado por la regla *IntType*. Admite valores enteros mayores o iguales a 0 y hasta 2^{31}-1. Su valores primitivos están definidos a través de *IntLiteral*.

Boolean

Está representado por la regla *BoolType*. Su valores primitivos están definidos a través de *BoolLiteral* ('true' o 'false').

String

Está representado por la regla *StringType*. Su valores primitivos están definidos a través de *StringLiteral*. Los valores deben escribirse entre comillas simples o dobles. Es posible utilizar el caracter de escape "\" para incluir caracteres especiales. En el ejemplo se observa como es posible asignar el valor *Juan "Cacho" Perez* a una variable de tipo *string*.

Bloque de código 7 Ejemplo de utilización de tipos simple

```
int cantidad;
cantidad = 0;
cantidad = cantidad;

bool flag;
flag = false;
flag = true;

string nombre;
nombre = "Juan";
nombre = 'Perez';
nombre = "Juan \"Cacho\" Perez \n";
```

Tipos de datos compuestos

Los tipos de datos compuestos son:

Controles

Están representados por la regla *Control*Type. Los controles forman parte de las vistas. Definen propiedades, eventos y cuestiones de apariencia. Los controles, junto con las vistas, componen la interfaz gráfica.

Entidades

Están representados por la regla *Entity*Type. Las entidades conforman el dominio de la aplicación, poseen acciones y propiedades.

Colecciones

Están representados por la regla *CollectionType*. Permite representar listas homogéneas de cualquier tipo.

```
CollectionType:
    type=Collection;

Collection:
    Array | List;

Array:
    type='Array' '<' content=Type '>';

List:
    type='List' '<' content=Type '>';
```

La regla *CollectionType* ofrece dos tipos para modelar las colecciones:

- *Array*: es una colección indexada de tamaño fijo. Permite acceder los elementos por el índice.
- *List*: es una colección de tamaño variable.

GuiDSL ofrece operaciones básicas para hacer uso de las colecciones (agregar, quitar y acceder elementos).

Tipo Void

La regla *VoidType* es utilizada para representar la ausencia de valor. *VoidType* no puede ser utilizado como los demás tipos de datos, sólo se utiliza para indicar que una acción no retorna ningún valor.

6.4.3. Lenguaje de acciones

GuiDSL ofrece un conjunto básico de sentencias, similares a las presentes en la mayoría de los lenguajes de programación. Posee sentencias que permiten manipular las entidades del modelo, acceder y manipular a los controles de las vistas, manejo de variables de tipo simple o compuesto y manejo de acciones. Las sentencias son agrupadas en bloques, las cuales son utilizadas dentro de las acciones.

Action

Las acciones están presentes en las entidades, en los grupos y en el proyecto. Una acción define un bloque de código, puede tener o no una lista de parámetros. Desde el código de las acciones definidas dentro un grupo, es posible acceder a los controles de las vistas del grupo al cual pertenece.

Gramática

```
Action:
    'action' name=ID '{'
        (
            ('visibility' visibility=Visibility)?
            & ('isStatic' isStatic=BoolLiteral)?
            & ('isInitial' isInitial=BoolLiteral)?
            & ('description' description=STRING)?
        )
        ('parameters' '[' (parameters+=Parameter)* ']')?
        'returnType' returnType=Type
        'body' body=Block
    '}';

Visibility:
    Public | Private | Protected;

Public:
    visibility='public';

Private:
    visibility='private';

Protected:
    visibility='protected';

Parameter:
    'parameter' name=ID '{'
        'type' type=Type
    '}';
```

Elementos

<table>
<tr><th>Elemento</th><th>Requerido</th><th>Descripción</th></tr>
<tr><td>name</td><td>Si</td><td>Es el identificador de la acción.</td></tr>
<tr><td>visibility</td><td>No</td><td>Indica la visibilidad de la acción, determinando su accesibilidad. Cuando una acción no posee definida su visibilidad, será el PSM quien determine la visibilidad por defecto.
Las acciones definidas dentro del proyecto o dentro de un grupo sólo podrán tener visibilidad pública.
Las acciones definidas dentro de una entidad podrán tener tener los siguientes tipos de visibilidad:
<table>
<tr><td>public</td><td>Es accesible desde todas las acciones</td></tr>
<tr><td>protected</td><td>Es accesible dentro de la entidad y de sus subclases</td></tr>
<tr><td>private</td><td>Sólo es accesible dentro de la entidad</td></tr>
</table></td></tr>
<tr><td>isStatic</td><td>No</td><td>Indica si la acción es estática o no. Se aplica sólo a acciones definidas dentro de una entidad. Cuando la acción sea estática no es necesario instanciar la clase para invocar a la acción.</td></tr>
<tr><td>isInitial</td><td>No</td><td>Indica si la acción es la inicial del grupo. Se aplica sólo a las acciones definidas dentro de un grupo.</td></tr>
<tr><td>description</td><td>No</td><td>Texto que describe la acción.</td></tr>
<tr><td>parameters</td><td>No</td><td>Lista de cero o más elementos de tipo Parameter.</td></tr>
<tr><td>returnType</td><td>Si</td><td>Indica el tipo de datos que retorna la acción.
Cuando no retorne ningún valor, returnType será de tipo VoidType.</td></tr>
<tr><td>body</td><td>Si</td><td>De tipo Block. El bloque podrá ser vacío o no. Contiene las sentencias de la acción.</td></tr>
</table>

Chequeos semánticos

Chequea que si el tipo de *returnType* es distinto de *VoidType*, la acción posea una sentencia *return*, y que el tipo retornado coincida con el declarado en *returnType*.

Ejemplo

Bloque de código 8 Ejemplo de uso de la regla *Action*

```
1  action grabarPersona{
2      parameters[
3          parameter nombre{type string}
4          parameter documento{type int}
5      ]
6      visibility public
7      returnType void
8      body {
9              Persona p;
10             p = new Persona();
11             p.Nombre = nombre;
12             p.Documento = documento;
13             p.Grabar();
14             redirectToAction(listarPersonas);
15     }
16 }
```

En el bloque de código 8, la acción *grabarPersona* recibe dos parámetros: nombre de tipo cadena y documento de tipo entero. Se observa que la acción es pública (línea 6) y no retorna ningún valor (línea 7). En el cuerpo de la acción, se declara una variable p de tipo Persona (línea 9), la cual luego es instanciada

a través de la sentencia *new* (línea 10). En las líneas 11 y 12, se le asignan valor a las propiedades nombre y documento, con los valores recibidos por parámetro. En la línea 13 se invoca al método *Save* de la clase Persona, el cual grabará a la persona con los datos de la variable *p*. Finalmente, en la línea 14 se ejecuta la sentencia *redirectToAction*, la cual redireccionará a la vista *listarPersonas*.

Como se puede observar en este ejemplo, con esta acción compuesta de un conjunto mínimo de sentencias sencillas, se puede especificar un método simple para persistir una entidad del modelo. A continuación se describirán en detalle cada una de la sentencias utilizadas en el ejemplo.

Comentarios

Las acciones aportan gran flexibilidad ya que permiten definir de manera completa, el comportamiento, el manejo de eventos y las reglas de negocio de la aplicación.

Block

La regla *Block* permite agrupar sentencias. Existen bloques vacíos, definidos a través de la regla *EmptyBlock*, y bloques con sentencias, definidos a través de la regla *NotEmptyBlock*.

Gramática

```
1  Block:
2      NotEmptyBlock | EmptyBlock;
3
4  NotEmptyBlock:
5      '{' (sentences+=Sentence)+ '}';
6
7  EmptyBlock:
8      '{' '}';
9
10 Sentence:
11     Declaration | Assigment | Selection | Iteration | Return |
12     Increment | Decrement | Navigation | MVCMethods | Invoke;
```

Elementos

Elemento	Requerido	Descripción
sentences	No	Lista de una o más sentencias.

Sentencia Declaration

La regla *Declaration* sirve para declarar variables. El alcance de la variable será dentro de la acción que la declara.

Gramática

```
1  Declaration:
2      type=Type name=ID ';';
```

Elementos

Elemento	Requerido	Descripción
type	Si	Indica el tipo de la variable.
name	Si	Es el identificador de la variable.

Chequeos semánticos

Se chequeará que el identificador de la variable no esté utilizado aún.

Ejemplo

Bloque de código 9 Ejemplo de declaraciones

```
body {
    Persona p;
    List<Persona> l;
    string nombre;
}
```

Sentencia Assigment

La regla *Assigment* sirve para asignar valor a una variable o a una propiedad de una entidad o control.

Gramática

```
Assigment:
    variable=RefVariable ('.' property=[Property])? '=' expression=Expression ';';
```

Elementos

Elemento	Requerido	Descripción
variable	Si	Referencia a una variable.
property	No	Referencia a una propiedad cuando variable sea de tipo entidad o control.
expression	Si	Expresión a asignar.

Chequeos semánticos

Se chequeará que el tipo de la expresión coincida con el tipo de la variable o propiedad a asignar.

Ejemplo

Bloque de código 10 Ejemplo de cómo asignar una variable

```
body {
    Persona p;
    p = new Persona();
    ...
    string nombre;
    nombre = "Juan Perez";
    p.Nombre = nombre;
    ...
}
```

Sentencia Selection

La regla *Selection* es una sentencia condicional, es decir, se utiliza para determinar el flujo de ejecución, de acuerdo a una condición booleana.

Gramática

```
Selection:
    'if' '(' expression=Expression ')' then=Block ('else' else=Block)?;
```

Elementos

Elemento	Requerido	Descripción
expression	Si	Es la expresión a evaluar. Debe ser una expresión booleana.
then	Si	Bloque de sentencias que se ejecutará si la expresión de evalúa como verdadera.
else	No	Bloque de sentencias que se ejecutará si la expresión de evalúa como falsa.

Chequeos semánticos

Se chequeará *expression* sea booleana.

Ejemplo

Bloque de código 11 Ejemplo de uso de la regla *Selection*

```
body {
    int a;
    int b;
    a = 1;
    b = 2;
    if (a > b) {
        return("Uno es mayor a dos...");
    } else {
        return("Uno no es mayor a dos...");
    }
}
```

Sentencia Iteration

La regla *Iteration* permite ejecutar iterativamente un bloque de sentencias, mientras se evalúe como verdadera la expresión booleana.

Gramática

```
Iteration:
    For | While;

While:
    'while' '(' expression=Expression ')' body=Block;

For:
    'for' '(' init=Declaration expression=Expression ';' update=Expression ')'
        body=Block;
```

Elementos

Elemento	Requerido	Descripción
init	Si	Es la declaración de la variable que se utiliza para iterar.
expression	Si	Es la expresión que se evalúa en cada iteración. Debe ser una expresión booleana.
update	Si	Es la expresión que actualiza la variable de iteración.
body	Si	Bloque de código que se ejecuta en cada iteración.

Chequeos semánticos

Se chequeará *expression* sea booleana.

Ejemplo

Bloque de código 12 Ejemplo de uso de las reglas *While* y *For*

```
body {
    int x;
    x = 0;
    while (x < 10) {
        a = a + x;
        x++;
    }
    ...
    for (int i; i < 10; i++) {
        a = a + lista.get(i);
    }
}
```

Sentence Increment

La regla *Increment* permite incrementar en uno, una variable o propiedad entera.

Gramática

```
Increment:
    variable=RefVariable ('.' property=[Property])? '++';
```

Elementos

Elemento	Requerido	Descripción
variable	Si	Referencia a la variable a incrementar, o a la entidad de la propiedad a incrementar.
property	Si	Es la propiedad a incrementar.

Chequeos semánticos

Se chequeará que la variable o propiedad sean de tipo entero.

Ejemplo

Bloque de código 13 Ejemplo de uso de la regla *Increment*

```
body {
    int x;
    x++;
    ...
    for (int i; i < 10; i++) {
        a = a + lista.get(i);
    }
}
```

Sentence Decrement

La regla *Decrement* permite decrementar en uno, una variable o propiedad entera.

Gramática

```
Decrement:
    variable=RefVariable ('.' property=[Property])? '--';
```

Elementos

Elemento	Requerido	Descripción
variable	Si	Referencia a la variable a decrementar, o a la entidad de la propiedad a decrementar.
property	Si	Es la propiedad a decrementar.

Chequeos semánticos

Se chequeará que la variable o propiedad sean de tipo entero.

Ejemplo

Bloque de código 14 Ejemplo de uso de la regla *Decrement*

```
body {
    int x;
    x--;
    ...
    }
}
```

Sentencia Return

La regla *Return* permite retornar un valor.

Gramática

```
Return:
    'return' (expression=Expression)? ';';
```

Elementos

Elemento	Requerido	Descripción
expression	Si	Es la expresión a retornar.

Chequeos semánticos

Se chequeará que el tipo *expression* coincida con el tipo de retorno de la acción que contiene la sentencia.

Ejemplo

Bloque de código 15 Ejemplo de uso de la regla *return*

```
action factorial{
    parameters[
        parameter numero{type int}
    ]
    visibility public
    returnType int
    body {
        if (numero == 1) {
            return 1;
        } else {
            return (numero * factorial(numero-1));
        }
    }
}
```

Sentencia Navigation

La regla *Navigation* permite derivar el flujo navegacional hacia una vista, a través de la regla *Render*, o hacia una acción en el controlador, a través de la regla *RedirectToAction*.

Gramática

```
Navigation:
    Render | RedirectToAction;

Render:
    'render' '(' view=[View] ')' ';';

RedirectToAction:
    'redirectToAction' '(' action=ID ')' ';';
```

Elementos Render

Elemento	Requerido	Descripción
view	Si	Es el identificador de la vista a *renderizar*.

Elementos RedirectToAction

Elemento	Requerido	Descripción
action	Si	Es el identificador de la acción a ejecutar. Esta acción deberá estar definida dentro de un grupo.

Ejemplo

Bloque de código 16 Ejemplo de uso de las reglas *render y redirectToAction*

```
action grabarPersona{
    parameters[
        parameter nombre{type string}
        parameter documento{type int}
    ]
    visibility public
    returnType void
    body {
            Persona p;
            p = new Persona();
            p.Nombre = nombre;
            p.Documento = documento;
            p.Grabar();
            redirectToAction(listarPersonas);
    }
}
...
action listarPersonas{
    visibility public
    returnType void
    body {
            ...
            render(vListarPersonas);
    }
}
```

Sentencia MVCMethods

La regla *MVCMethods* define la regla *SetValue*. Ésta es utilizada para pasar valores entre las vistas y los controladores.

Gramática

```
MVCMethods:
    SetValue;

SetValue:
    'setValue' '(' refVariable=RefVariable ',' value=Expression ')' ';';
```

Elementos SetValue

Elemento	Requerido	Descripción
refVariable	Si	Es un identificador, el cual no se corresponde con una variable local ni con un parámetro. La expresión asignada a este identificador podrá ser utilizada en la vista a través de la regla *GetValue*.
value	Si	Es la expresión a asignar al identificador.

Chequeos semánticos

El identificador *refVariable* es un identificador establecido libremente, es decir, el compilador no chequeará que sea una variable definida o un parámetro.

Ejemplo

Bloque de código 17 Ejemplo de uso de la regla *SetValue*

```
action altaPersona {
    returnType void
    body {
        Persona persona;
        persona = new Persona();
        setValue(VarPersona, persona);
        render(vAltaPersona);
    }
}
```

Sentencia Invoke

Las acciones son invocadas desde otras acciones o desde los eventos de los controles. La acción invocada a través de una sentencia *invoke*, puede ser una acción de proyecto, de grupo, de entidad o de colección.

Gramática

```
Invoke:
    InvokeGenericAction ';';

InvokeGenericAction:
    (variable=RefVariable '.')? action=GenericAction '(' (argumentList=ArgumentList)? ')';

GenericAction:
    RefAction | ListAction | ArrayAction;

RefAction:
    action=[Action];

ArrayAction:
    AddElementInPosition | Clone;

ListAction:
    AddElement | RemoveElement | Revert;

ControlAction:
    OnClick | OnLostFocus | OnGotFocus;

ArgumentList:
    arguments+=Expression (',' arguments+=Expression)*;
```

Elementos

Elemento	Requerido	Descripción
variable	No	La variable deberá ser una referencia a un grupo, a una entidad o a una colección.
action	Si	Es el identificador de la acción que se invoca. Cuando no se especifique variable, la acción podrá ser una acción definida dentro del grupo donde se invoca, o una acción definida dentro de la entidad donde se invoca; o una acción global de proyecto.
argumentList	No	Lista de cero o más argumentos. Un argumento es una expresión.

Chequeos semánticos

Se chequeará que el tipo y la cantidad de argumentos de la sentencia *invoke*, coincida con los parámetros que define la acción invocada.

Ejemplo

Bloque de código 18 Ejemplo de uso de la regla *Invoke*

```
...
action grabarPersona{
    parameters[
        parameter nombre{type string}
        parameter documento{type int}
    ]
    visibility public
    returnType void
    body {
            Persona p;
            p = new Persona();
            p.Nombre = nombre;
            p.Documento = documento;
            p.Grabar();
            redirectToAction(listarPersonas);
    }
}
...
body{
    grabarPersona();
    ...
    Persona p;
    p = new Persona();
    p.Grabar();
    ...
    List<Persona> l;
    l = new List<Persona>();
    l.add(p);
    ...
}
```

Sentencia Expression

La regla *Expression* define una expresión, la cual podrá ser compuesta por términos y operadores, o simplemente un término. Esta regla es *parseada* recursivamente en forma de árbol, donde los nodos son términos, operadores y expresiones, y las hojas son los factores.

Gramática

```
Expression:
    left=Term (operator=Operator right=Expression)?;

Term:
    left=Factor (operator=Operator right=Factor)?;

Factor:
    GetValue | IntLiteral | StringLiteral | BoolLiteral | RefVariable |
    ParenthesesExpression | InvokeGenericAction |
    InvokeGenericProperty | Constructor | Null | Increment | Decrement;
```

En la figura 17 se observa el árbol de *parseo* de la regla *Expression*, la cual se define recursivamente como una composición de términos, operadores y factores.

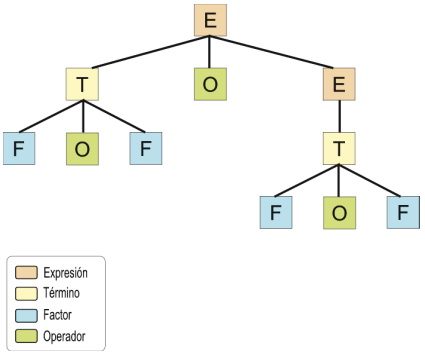

Figura 17: Árbol de *parseo* de la regla *Expression*

Elementos Expression

Elemento	Requerido	Descripción
left	Si	Es un término.
operator	No	Es el operador a aplicar entre los elementos *left* y *right*.
right	No	Es una expresión.

Elementos Term

Elemento	Requerido	Descripción
left	Si	Es un factor.
operator	No	Es el operador a aplicar entre los elementos *left* y *right*.
right	No	Es un factor.

Ejemplo

Bloque de código 19 Ejemplo de uso de la regla *expression*

```
...
int x;
int y;
int z;
z = x + y + 1;
z = x / (8 - x);
...
bool ok;
bool flag;
flag = ok && (x < y);
...
string nombre;
string apellido;
nombre = "Juan" + " " + apellido;
...
```

Sentencia GetValue

La regla *GetValue* permite acceder al valor asociado previamente a un identificador a través de la regla *SetValue*. Esta regla permite pasar valores entre las vistas y los controladores.

Gramática

```
GetValue:
    '%' refVariable=QualifiedName;
```

Elementos

Elemento	Requerido	Descripción
refVariable	Si	Es un identificador, el cual no se corresponde con una variable local ni con un parámetro. Este identificador podría haber sido asignado a través de la regla *SetValue*.

Ejemplo

Bloque de código 20 Ejemplo de uso de la regla *GetValue*

```
action altaPersona {
    returnType void
    body {
        Persona persona;
        persona = new Persona();
        setValue(VarPersona, persona);
        render(vAltaPersona);
    }
}
...
action grabarPersona{
    visibility public
    returnType void
    body {
            Persona p;
            p = %VarPersona;
            p.Save();
            redirectToAction(listarPersonas);
    }
}
```

RefVariable

La regla *RefVariable* representa una referencia a una variable. El compilador chequeará la validez semántica del *id*, el cual deberá ser el identificador de una variable declarada localmente o recibida como parámetro.

Gramática

```
RefVariable:
    ref=ID;
```

RefVariableType

La regla *RefVariableType* representa una referencia a una variable declarada o recibida como parámetro.

Gramática

```
RefVariableType:
    RefParameter | RefDeclaration;

RefParameter:
    value=[Parameter];

RefDeclaration:
    value=[Declaration];
```

Regla ParenthesesExpression

La regla *ParenthesesExpression* permite establecer un orden de precedencia en la evaluación de expresiones compuestas.

Gramática

```
ParenthesesExpression:
    '(' value=Expression ')';
```

Elementos

Elemento	Requerido	Descripción
value	Si	Es una expresión de cualquier tipo.

Ejemplo

Bloque de código 21 Ejemplo de uso de la regla *ParenthesesExpression*

```
...
int x;
int z;
z = x / (8 - (x*3));
...
bool ok;
bool flag;
flag = ok && (x < y);
```

Regla Constructor

La regla *Constructor* permite instanciar una variable de tipo compuesto.

Gramática

```
Constructor:
    'new' type=CompositeType '(' (argumentList=ArgumentList)? ')';
```

Elementos

Elemento	Requerido	Descripción
type	Si	Es un *CompositeType*.
argumentList	No	Lista de argumentos del constructor.

Comentarios

Todos los tipos compuestos definen un constructor sin parámetros. Algunos constructores podrán definir parámetros los cuales se utilizan para, además de instanciar la variable, inicializarla con valores.

Ejemplo

Bloque de código 22 Ejemplo de uso de la regla *Constructor*

```
...
entity Persona{
    void Persona(int documento){
        ...
    }
    ...
}
...
Persona pA;
Persona pB;
pA = new Persona();
pB = new Persona(123456);
...
...
Array<Partido> partidos;
partidos = new Array<Partido>(30);
...
```

Regla Operator

La regla *Operator* define un conjunto de operadores aritméticos, lógicos y de comparación.

Gramática

```
Operator:
    ArithOperator | LogicOperator | ComparisonOperator;

ArithOperator:
    TermOperator | FactorOperator;

TermOperator:
    PlusOperator | MinusOperator;

FactorOperator:
    MultOperator | DivOperator;

LogicOperator:
    AndOperator | OrOperator;

ComparisonOperator:
    EqualOperator | DifferentOperator | GreaterOperator | LessOperator;

PlusOperator:
    {PlusOperator} '+';

MinusOperator:
    {MinusOperator} '-';

MultOperator:
    {MultOperator} '*';

DivOperator:
    {DivOperator} '/';

AndOperator:
    {AndOperator} '&&';

OrOperator:
    {OrOperator} '||';

EqualOperator:
    {EqualOperator} '==';

DifferentOperator:
    {DifferentOperator} '!=';

GreaterOperator:
    {GreaterOperator} '>';

LessOperator:
    {LessOperator} '<';
```

Operadores aritméticos

Regla	Descripción
PlusOperator	Operador binario. Aplica sobre expresiones enteras o booleanas. Para expresiones enteras realiza la suma de las expresiones. Para expresiones de cadena, realiza la concatenación de las expresiones.
MinusOperator	Operador binario. Aplica sobre expresiones enteras. Realiza la resta de las expresiones.
MultOperator	Operador binario. Aplica sobre expresiones enteras. Realiza la multiplicación de las expresiones.
DivOperator	Operador binario. Aplica sobre expresiones enteras. Realiza la división de las expresiones.

Operadores lógicos

Regla	Descripción
AndOperator	Operador binario. Aplica sobre expresiones booleanas. Realiza el *and* lógico de las expresiones.
OrOperator	Operador binario. Aplica sobre expresiones enteras. Realiza el *or* lógico de las expresiones.

Operadores de comparación

Regla	Descripción
EqualOperator	Operador binario. Aplica sobre expresiones de cualquier tipo. Retorna verdadero si ambas expresiones son iguales. Retorna falso en caso contrario.
DifferentOperator	Operador binario. Aplica sobre expresiones de cualquier tipo. Retorna verdadero si ambas expresiones son distintas. Retorna falso en caso contrario.
GreaterOperator	Operador binario. Aplica sobre expresiones enteras o de cadena. Retorna verdadero si la expresión izquierda es mayor que la expresión derecha. Retorna falso en caso contrario.
LessOperator	Operador binario. Aplica sobre expresiones enteras o de cadena. Retorna verdadero si la expresión izquierda es menor que la expresión derecha. Retorna falso en caso contrario.

Ejemplo

Bloque de código 23 Ejemplo de uso de los distintos operadores

```
...
int a;
int b;
...
a = a + b;
b = b * ;
...
...
bool ok;
bool flag;
...
ok = ok || flag;
flag = ok && (a < b);
...
...
string nombre;
string apellido;
nombre = nombre + " " + apellido;
...
```

6.4.4. Modelo

Regla Model

La regla *Model* permite definir el conjunto de entidades que forman el dominio de la aplicación.

Gramática

```
Model:
    'model' name=ID '{'
        ('elements' '[' (elements+=Entity)* ']')?
    '}';
```

Elementos

Elemento	Requerido	Descripción
name	Si	Es el identificador del modelo.
elements	No	Lista de uno o más elementos de tipo *Entity*.

Entity

La regla Entity permite modelar las entidades del modelo de la aplicación. A través de las propiedades es posible establecer las relaciones entre las entidades. GuiDSL permite herencia simple entre las entidades del modelo, a través de la cláusula *extends*.

Gramática

```
Entity:
    'entity' name=ID('extends' extends=[Entity])?
    '{'
        ('properties' '[' (properties+=Property)* ']')?
        ('actions' '[' (actions+=Action)* ']')?
    '}';
```

Elementos

Elemento	Requerido	Descripción
name	Si	Es el identificador de la entidad.
extends	No	Referencia a otra entidad de la cual hereda sus métodos y propiedades.
properties	No	Lista de cero o más elementos de tipo *Property*.
actions	No	Lista de cero o más elementos de tipo *Action*.

Regla Property

La regla Entity permite modelar las entidades del modelo de la aplicación. A través de las propiedades es posible establecer las relaciones entre las entidades. GuiDSL permite herencia simple entre las entidades del modelo, a través de la cláusula *extends*.

Gramática

```
Property:
    'property' name=ID '{'
        ('isPrimaryKey' isPrimaryKey=BoolLiteral)?
        'type' type=Type
    '}';
```

Elementos

Elemento	Requerido	Descripción
name	Si	Es el identificador de la propiedad.
isPrimaryKey	No	Indica si la propiedad es la clave primaria de la entidad. Podrá indicarse sólo una propiedad por entidad como clave primaria.
type	Si	Indica el tipo de la propiedad.

Comentarios

Cada entidad deberá tener una y sólo una propiedad indicada como clave primaria.

Ejemplos de las reglas que componen el modelo

Bloque de código 24 Ejemplo de uso de la regla *Model, Entity y Property*

```
model modeloPersonas {
    elements [
        entity Persona
        {
            properties [
                        property Id
                        {
                            isPrimaryKey true
                            type int
                        }
                        property Nombre {type string}
                        property Domicilio {type Domicilio}
                        ]
            actions [
                        action TraerPorId
                        {
                            isStatic    true
                            returnType  Equipo
                            body {}
                        }
                    ]
        }
        entity Domicilio
        {
            properties [
                        property Id
                        {
                            isPrimaryKey true
                            type int
                        }
                        property Calle {type string}
                        property Numero {type int}
                        property Localidad {type Localidad}
                        ]
        }
        ...
    ]
}
```

Bloque de código 25 Ejemplo de uso de la cláusula *extends*

```
entity Socio extends Persona
{
    properties [
                property NroSocio{type int}
                ]
}
```

6.4.5. Interfaces gráficas

GuiDSL ofrece un conjunto mínimo de controles, necesarios para modelar interfaces de usuario con operaciones básicas. Al igual que con los tipos de datos simples, también es posible extender GuiDSL, y definir controles específicos, de acuerdo a las necesidades del modelo y de interacción de la aplicación.

Por ejemplo, es posible definir un control lista chequeable, o más específico, un control que extienda el control imagen que permita mostrar una imagen y seleccionar zonas aleatorias sobra la misma.

Regla View

La regla *View* permite definir las vistas de la aplicación, con sus controles, apariencia y comportamiento.

Gramática

```
View:
    'view' name=ID '{'
        ('isInitial' isInitial=BoolLiteral)?
        ('mainContainer' mainContainer=Container)?
    '}';

Container:
    'container' name=ID '{'
        'layout' layout=Layout
    '}';

Layout:
    FlowLayout | GridLayout | BorderLayout;

FlowLayout:
    'FlowLayout' name=ID '{'
        ('controls' '[' (controls+=Control)* ']')?
    '}';

GridLayout:
    'GridLayout' name=ID '{'
        ('properties' '[' properties+=ControlProperty* ']')?
        'rows' '[' (rows+=Row)* ']'
    '}';

BorderLayout:
    'BorderLayout' name=ID '{'
        ('properties' '[' properties+=ControlProperty* ']')?
        'controls' '[' (controls+=Control)* ']'
    '}';

Row:
    'row' name=ID '{
        ('properties' '[' properties+=ControlProperty* ']')?
        'cells' '[' (cells+=Cell)+ ']'
    '}';

Cell:
    'cell' name=ID '{'
        ('properties' '['
            properties+=ControlProperty*
        ']')?
        'controls' '[' (controls+=Control)* ']'
    '}';
```

Elementos de View

Elemento	Requerido	Descripción
name	Si	Es el identificador de la vista.
isInitial	No	Indica si es o no la vista inicial del grupo.
mainContainer	No	Elemento de tipo *Container*, el cual contendrá los controles de la vista.

Elementos de Container

Elemento	Requerido	Descripción
name	Si	Es el identificador del container.
layout	Si	Elemento de tipo *Layout*, el cual define la disposición de los controles de la vista.

Elementos de FlowLayout

Elemento	Requerido	Descripción
name	Si	Es el identificador del *layout*.
controls	Si	Lista de cero o más controles.

Elementos de GridLayout

Elemento	Requerido	Descripción
name	Si	Es el identificador del *layout*.
properties	No	Lista de cero o más elementos de tipo *ControlProperty*. Estas propiedades definen la apariencia del *layout*.
rows	Si	Lista de cero o más elementos de tipo *Row*.

Elementos de Row

Elemento	Requerido	Descripción
name	Si	Es el identificador del *layout*.
properties	No	Lista de cero o más elementos de tipo *ControlProperty*. Permiten darle a cada fila una apariencia personalizada.
cells	Si	Lista de cero o más elementos de tipo *Cell*, las cuales contendrán lo controles de la vista.

Elementos de Cell

Elemento	Requerido	Descripción
name	Si	Es el identificador del *layout*.
properties	No	Lista de cero o más elementos de tipo *ControlProperty*. Permiten darle a cada fila una apariencia personalizada.
controls	Si	Lista de cero o más elementos de tipo *Control*.

Ejemplo

Bloque de código 26 Ejemplo de uso de la *regla View* y sus componentes

```
view index {
    isInitial true
    mainContainer container layContainer {
        layout GridLayout gridLay1 {
                rows [
                        row row1 {
                            cells [
                                cell cell0 {
                                    controls [
                                        ...
                                        ...
                                    ]
                                }
                            ]
                        }
                        row row2 {
                            cells [
                                cell cel20 {
                                    controls [
                                        ...
                                        ...
                                    ]
                                }
                            ]
                        }
                ]
        }
    }
}
```

Controles

La regla *Control* define un conjunto de controles gráficos que permiten realizar operaciones básicas en las vistas. Para cada control es posible definir su apariencia y el manejo de sus eventos.

Gramática

```
Control:
    type=ControlBase name=ID '{'
        ('properties' '[' properties+=ControlProperty* ']')?
        ('events' '[' actions+=ControlAction* ']')?
    '}';

ControlBase:
    ComboBox | CheckBox | TextBox | Label | Button |
    Container | ListBox | Menu | Option | Image | Options;

ComboBox:
    control='comboBox';

CheckBox:
    control='checkBox';

TextBox:
    control='textBox';

Label:
    control='label';

Button:
    control='button';

ListBox:
    control='listBox';

Image:
    control='image';

Option:
    control='option';

Options:
    control='options' '[' options +=Control ']';

Menu:
    control='menu';
```

```
ControlProperty:
    Text | FieldValue | FieldText | SelectedValue | SelectedText |
    Checked | DataSource | Password | Path | Ident | MaxLength |
    ForeColor | BackgroundColor | FontFamily | FontSize |
    FontStyle | FontWeight | HorizontalAlignment | VerticalAlignment |
    Visible | Enabled | Size | Style;

ControlAction:
    OnClick | OnLostFocus | OnGotFocus;

OnClick:
    action='onClick' delegateAction=RefAction '(' (argumentList=ArgumentList)? ')';

OnGotFocus:
    action='onGotFocus' delegateAction=RefAction '(' (argumentList=ArgumentList)? ')';

OnLostFocus:
    action='onLostFocus' delegateAction=RefAction '(' (argumentList=ArgumentList)? ')';
```

Elementos

Elemento	Requerido	Descripción
type	Si	Elemento de tipo *ControlBase*. Indica el tipo de control.
properties	No	Lista de una o más elementos de tipo *ControlProperty* que definen la apariencia del control.
actions	No	Lista de unoo más elementos de tipo *ControlAction* que definen el manejo de los eventos del control.

Tipos de ControlBase

ControlBase	Descripción	Propiedades	Eventos
ComboBox	Lista deplegable que permite seleccionar un único valor.	Text FieldValue FieldText SelectedValue SelectedText DataSource MaxLength ForeColor BackgroundColor FontFamily FontSize FontStyle FontWeight HorizontalAlignment VerticalAlignment Visible Enabled Size Style	OnClick OnLostFocus OnGotFocus

ControlBase	Descripción	Propiedades	Eventos
CheckBox	Permite ser chequeado o deschequeado.	Text Checked DataSource ForeColor BackgroundColor FontFamily FontSize FontStyle FontWeight HorizontalAlignment VerticalAlignment Visible Enabled Size Style	OnClick OnLostFocus OnGotFocus

ControlBase	Descripción	Propiedades	Eventos
TextBox	Caja que permite ingresar texto.	Text SelectedText DataSource Password MaxLength ForeColor BackgroundColor FontFamily FontSize FontStyle FontWeight HorizontalAlignment VerticalAlignment Visible Enabled Size Style	OnClick OnLostFocus OnGotFocus

ControlBase	Descripción	Propiedades	Eventos
Label	Muestra un texto	Text DataSource MaxLength ForeColor BackgroundColor FontFamily FontSize FontStyle FontWeight HorizontalAlignment VerticalAlignment Visible Enabled Size Style	OnClick

ControlBase	Descripción	Propiedades	Eventos
Button	Botón que permite invocar una acción al *clickearlo*.	Text MaxLength ForeColor BackgroundColor FontFamily FontSize FontStyle FontWeight HorizontalAlignment VerticalAlignment Visible Enabled Size Style	OnClick OnLostFocus OnGotFocus

ControlBase	Descripción	Propiedades	Eventos
ListBox	Lista que permite seleccionar un valor.	Text FieldValue FieldText SelectedValue SelectedText DataSource MaxLength ForeColor BackgroundColor FontFamily FontSize FontStyle FontWeight HorizontalAlignment VerticalAlignment Visible Enabled Size Style	OnClick OnLostFocus OnGotFocus

ControlBase	Descripción	Propiedades	Eventos
Image	Muestra una imagen	Text Path BackgroundColor HorizontalAlignment VerticalAlignment Visible Size Style	OnClick

ControlBase	Descripción	Propiedades	Eventos
Option	Elemento del menú	Text DataSource Path Ident ForeColor BackgroundColor FontFamily FontSize FontStyle FontWeight HorizontalAlignment VerticalAlignment Visible Enabled Size Style	OnClick OnLostFocus OnGotFocus

ControlBase	Descripción	Propiedades	Eventos
Options	Elemento del menú que posee sobopciones.	Text DataSource Path Ident ForeColor BackgroundColor FontFamily FontSize FontStyle FontWeight HorizontalAlignment VerticalAlignment Visible Enabled Size Style	OnClick OnLostFocus OnGotFocus

ControlBase	Descripción	Propiedades	Eventos
Menu	Menu que posee opciones y subopciones.	Text DataSource Path Ident ForeColor BackgroundColor FontFamily FontSize FontStyle FontWeight HorizontalAlignment VerticalAlignment Visible Enabled Size Style	OnClick OnLostFocus OnGotFocus

Tipos de ControlAction

ControlAction	Descripción
OnLostFocus	Evento que se dispara cuando el control pierde el foco.
OnGotFocus	Evento que se dispara cuando el control obtiene el foco.
OnClick	Evento que se dispara al *clickear* el control.

Estilos

Los estilos sirven para definir la apariencia de los controles de las vistas. Cada estilo define una o más propiedades, las cuales se aplicarán al control que posea el estilo.

Gramática

```
Style:
    property='style' value=[StyleDeclaration];

StyleDeclaration:
    'style' name=ID     '{'
    ('properties' '[' (properties+=StyleProperty)+ ']')?
    '}';

StyleProperty:
    MaxLength | ForeColor | BackgroundColor |
    FontFamily | FontSize | FontStyle |
    FontWeight | HorizontalAlignment | VerticalAlignment |
    Visible | Enabled | Size;

enum Color:
    WHITE='white' | BLACK='black' | RED='red' | BLUE='blue' |
    YELLOW='yellow' | GREEN='green' | ORANGE='orange' | VIOLET='violet';

enum Font:
    ARIAL='arial' | TIMESNEWROMAN='timesNewRoman' |
    VERDANA='verdana' | COURIER='courier';

enum FontSizeType:
    SMALL='small' | MEDIUM='medium' | LARGE='large' | EXTRALARGE='extraLarge';

enum FontStyleType:
    NORMAL='normal' | ITALIC='italic' | OBLIQUE='oblique';

enum FontWeightType:
    NORMAL='normal' | LIGHT='light' | BOLD='bold';

enum HAlignment:
    CENTER='center' | LEFT='left' | RIGHT='right';

enum VAlignment:
    CENTER='center' | TOP='top' | BOTTOM='bottom';
```

Elementos de StyleDeclaration

Elemento	Requerido	Descripción
name	Si	Es el identificador del estilo.
properties	No	Lista de cero o más elementos de tipo *StyleProperty*.

Tipos de propiedades de controles y estilos

Regla	Gramática	Descripción
MaxLength	*maxLength [IntLiteral]*	Indica el tamaño máximo del texto.
ForeColor	*foreColor [Color]*	Indica el color del texto.
BackgroundColor	*backgroundColor [Color]*	Indica el color de fondo del control.
FontFamily	*fontFamily [Font]*	Indica el tipo de letra del control.
FontSize	*fontSize [FontSizeType]*	Indica el tamaño de la letra del control.
FontStyle	*fontStyle [FontStyleType]*	Indica el estilo de la letra del control (normal, itálica u obliqua).
FontWeight	*fontWeight [FontWeightType]*	Indica si la letra del control está negrita o no.
HorizontalAlignment	*horizontalAlignment [HAlignment]*	Indica la alineación horizontal del texto dentro del control (a izquierda, a derecha o centrada).
VerticalAlignment	*verticalAlignment [VAlignment]*	Indica la alineación vertical del texto dentro del control (arriba, centrada o abajo).
Visible	*visible [BoolLiteral]*	Indica si el control está visible o no.
Enabled	*enabled [BoolLiteral]*	Indica si el control está habilitado o no.
Size	*size [IntLiteral]*	Indica el tamaño del control.
Text	*text [StringLiteral]*	Texto del control.
FieldValue	*fieldValue [StringLiteral]*	Indica la propiedad que se utilizará para enlazar el identificador de cada elemento de la lista.
FieldText	*field*Text *[StringLiteral]*	Indica la propiedad que se utilizará para enlazar el texto de cada elemento de la lista.
SelectedValue	selectedValue *[ID]*	Indica el identificar del elemento de la lista a seleccionar.
SelectedText	*selectedText [StringLiteral]*	Indica el texto del elemento de la lista a seleccionar.
Checked	*checked [BoolLiteral]*	Indica si chequear o no el control.
DataSource	*datasource [ID] [QualifiedName] [QualifiedName] [QualifiedName]*	Indica el origen de datos del control.
Password	*password [BoolLiteral]*	Indica si la modalidad del texto es de tipo contraseña.
Path	*path [QualifiedName]*	Indica el *path* de la imagen.

Ejemplo

Bloque de código 27 Ejemplo de controles con sus propiedades

```
...
controls [
    label lblDocumento {
        properties [
            text       "Documento:"
            style     StyleVerde
        ]
    }
    textBox txtDocumento {
        properties [
            size          10
            maxLength   10
            enabled   false
            datasource   persona documento
        ]
    }
]
...
styles [
        style StyleVerde {
            properties [ backgroundColor green ]
        }
]
```

6.4.6. Otras reglas

Sentencia QualifiedName

La regla *QualifiedName* define un identificador el cual podrá contener el caracter ".".

Gramática

```
QualifiedName:
    ID ('.' ID)*;

terminal ID :
    '^'?('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
```

6.5. Conclusión

GuiDSL posee un DSL textual, de sintaxis clara y sencilla. Ofrece un conjunto mínimo de reglas que permiten definir de manera completa una aplicación junto con su interfaz gráfica. Además posee reglas que representan sentencias básicas, presentes en la mayoría de los lenguajes de programación, las cuales permiten especificar el manejo de eventos, la lógica del negocio de la aplicación, y cualquier método o rutina que se requiera. Esto aporta gran flexibilidad a la hora de definir la aplicación.

El lenguaje posee distintos tipos de chequeos sintácticos y semánticos, los cuales verifican que los PIMs sean válidos.

El lenguaje podrá ser extendido fácilmente, incorporando reglas nuevas, o reglas que optimicen, personalicen o amplíen las existentes, tal como se verá en la sección 9.3.1.

7. Transformaciones

7.1. Introducción

Tal como ha sido presentado en el capítulo 5, GuiDSL es una herramienta MDA que utiliza transformaciones entre modelos para obtener el resultado final, que es la aplicación implementada en alguna plataforma y lenguaje de programación específico, lista para ser utilizada.

Para generar el código de la aplicación GuiDSL utiliza PSMs, los cuales generan la aplicación para un lenguaje de programación, plataforma y tecnología específica. Los PSMs se componen de un conjunto de archivos, donde cada uno genera una parte particular del proyecto.

Cuando la tecnología del PSM no soporte alguna característica del PIM, es decir, cuando el modelo utilice alguna regla del lenguaje que no existe para el PSM utilizado, el programador del PSM podrá tomar alguna de las siguientes decisiones:

- Transformar la regla como una combinación de sentencias que posean igual significado semántico. En el ejemplo se observa un modelo que utiliza la regla *Increment* (bloque de código 28), la cual no es soportada en algunos lenguajes de programación. En ese caso, el PSM reemplazará esta regla por una combinación sentencias que realizan la misma acción: incrementar en 1 la variable i (bloque de código 29).

Bloque de código 28 Utilización de la sentencia *Increment*

```
1 ...
2 int i;
3 i = 0;
4 i++;
5 ...
```

Bloque de código 29 Reemplazo de la sentencia *Increment*

```
1 ...
2 int i;
3 i = 0;
4 i = i + 1;
5 ...
```

- Transformar la regla que no es soportada con el uso de una librería tipo TAD (Tipo Abstracto de Datos) que provea y encapsule su funcionalidad. Por ejemplo, si el PIM utiliza listas, y el lenguaje de programación a utilizar es Pascal, el PSM deberá proveer un TAD para el manejo de listas, y luego utilizarlo en la transformación de las reglas no soportadas.
- Excepcionalmente, en caso que la regla no pueda ser soportada de ninguna manera en la tecnología del PSM, la misma será ignorada. En este caso, eventualmente se podrían obtener proyectos con errores sintácticos o con comportamiento distinto al esperado.

7.2. Estructura de los PSM

Para realizar las transformaciones GuiDSL utiliza PSMs. Un PSM se compone de distintos tipos de archivos:

- Archivos propios de la transformación a realizar: para cada PSM existen archivos que realizan transformaciones teniendo en cuenta el lenguaje de programación, la librería de mapeo y persistencia, y la tecnología de las interfaces gráficas. También se incluyen archivos *template*, los cuales realizan transformaciones de otros aspectos del proyecto, los cuales son más personalizados y acoplados a un PSM en particular (por ejemplo las transformaciones sobre las vistas del proyecto).
- Archivos que dan soporte a las transformaciones (*helpers*): ofrecen distintas funciones, que permiten acceder y recorrer el modelo de entrada. Son reusados por distintas transformaciones ya que son independientes de éstas.

- Archivos fijos auxiliares: estos archivos son copiados al proyecto generado, sin aplicar ninguna transformación sobre los mismos. Podrán ser librerías externas, archivos de configuración o archivos de recursos (hojas de estilo, archivos Javascript, imágenes, etc).

Dentro de la estructura de los PSM, y respecto de los archivos propios de cada transformación, GuiDSL organiza las transformaciones en tres grupos principales:

- Transformaciones a nivel lenguaje de programación
- Transformaciones a nivel de mapeo y persistencia de las entidades del modelo
- Transformaciones a nivel navegacional, para las interfaces gráficas

GuiDSL incorpora las siguientes transformaciones respecto de los lenguajes de programación:

- Java
- CSharp
- PHP

A nivel de modelo, respecto del mapeo y la persistencia de las entidades, GuiDSL incorpora las siguientes transformaciones:

- Hibernate
- Castle ActiveRecords

Para el esquema de navegación de la interfaces gráficas, GuiDSL incorpora la transformación Castle MonoRail.

Los PSM que utiliza GuiDSL no son cerrados: pueden extenderse o crearse nuevos muy fácilmente (esto se analizará en detalle en los próximos capítulos). Así es posible incorporar transformaciones para otros lenguajes de programación, como podrían ser Python o Ruby, o para otros *frameworks* de mapeo y persistencia, como podría ser NHibernate; basta con implementar la interfaz correspondiente.

Por otro lado, al estar modularizadas las transformaciones, es posible reutilizar parte de ellas. Por ejemplo, si se desea incorporar una transformación que utilice Java, Spring y Struts, respecto de Java, no es necesario desarrollar las transformaciones para las sentencias, basta con reutilizar la que incorpora GuiDSL. La figura 18 muestra la estructura de los PSMs y cómo es posible desarrollar nuevos reutilizando transformaciones existentes.

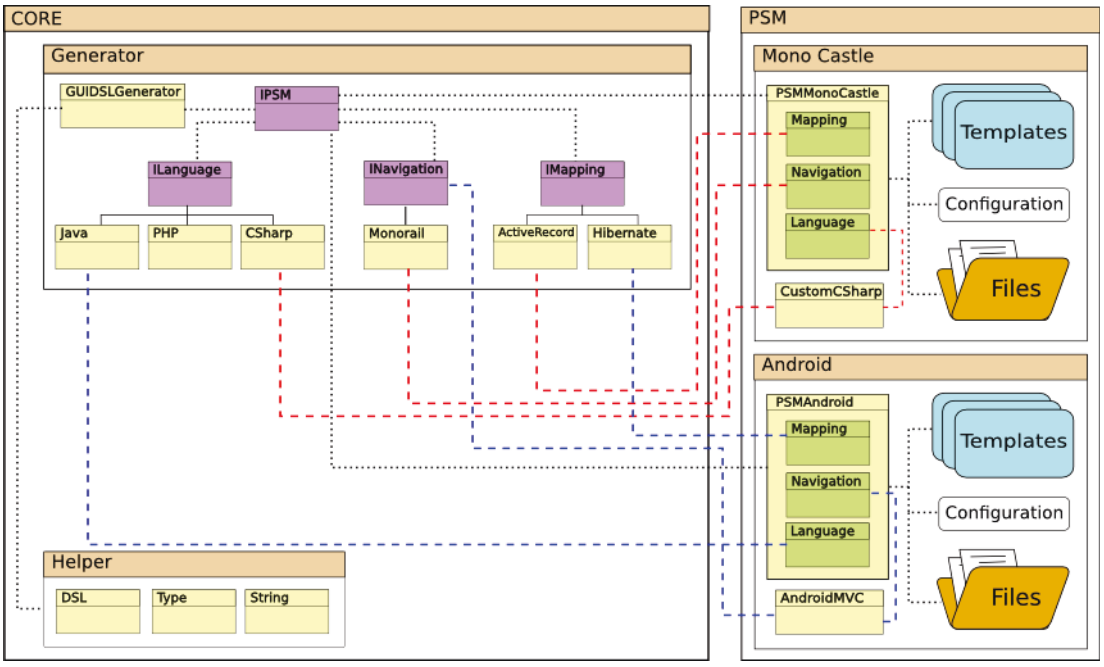

Figura 18: GuiDSL y los PSMs

La figura 18 muestra la arquitectura de las transformaciones en GuiDSL. En el *core* de la herramienta se incluyen transformaciones a nivel lenguaje de programación para Java, PHP y CSharp; a nivel de mapeo, transformaciones para Active Records e Hibernate; y a nivel navegacional, una transformación para el framework MVC Monorail. Se observan también dos PSM: uno para Mono Castle y otro para Android. El PSM para Mono Castle posee su propia implementación de las transformaciones a nivel lenguaje de programación, utilizando CustomCSharp. En cambio en el PSM para Android, observamos que a nivel lenguaje de programación utiliza la transformación para Java que viene incluida en la herramienta.

7.3. Conclusión

GuiDSL hace uso de transformaciones para obtener la aplicación implementada sobre una plataforma tecnológica específica. Estas transformaciones están modularizadas y son fácilmente extensibles. Cada PSM utiliza un conjunto de archivos, algunos propios de una transformación específica y otros auxiliares, que son utilizados para dar soporte a las transformaciones. Así es posible desarrollar nuevos PSMs, reutilizando parte de otro existente.

8. Utilizando GuiDSL

8.1. Introducción

En el presente capítulo se mostrará cómo utilizar GuiDSL, desde el diseño y la escritura del PIM, hasta obtener la aplicación implementada en una plataforma y lenguaje de programación específico. Para eso, se mostrará paso a paso utilizando un ejemplo concreto y sencillo.

8.2. Modelando la aplicación

Para comenzar a escribir el modelo de la aplicación, se deben realizar los siguientes pasos:

Ejecutar la herramienta

Al ejecutar la herramienta se abrirá el IDE que permite crear y transformar proyectos.

Crear el proyecto

Crear un nuevo proyecto de tipo "GuiDSL Project". Este tipo de proyecto, incorpora un archivo GuiDSL, el cual posee un pequeño modelo a modo de plantilla (figura 19). Se debe asignar un nombre al proyecto, en el ejemplo se asigna el nombre "ProyectoPrueba" (figura 20).

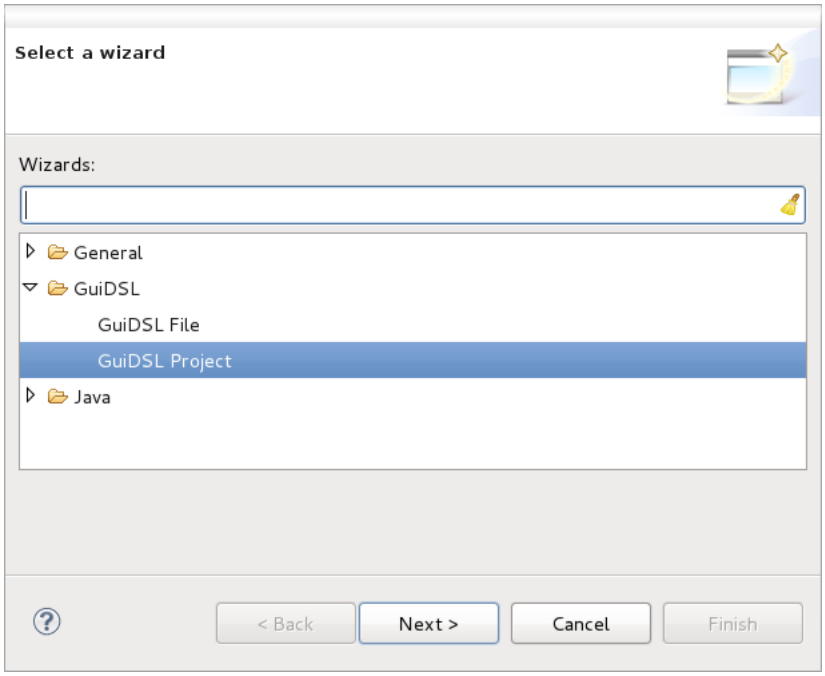

Figura 19: Selección del tipo de proyecto

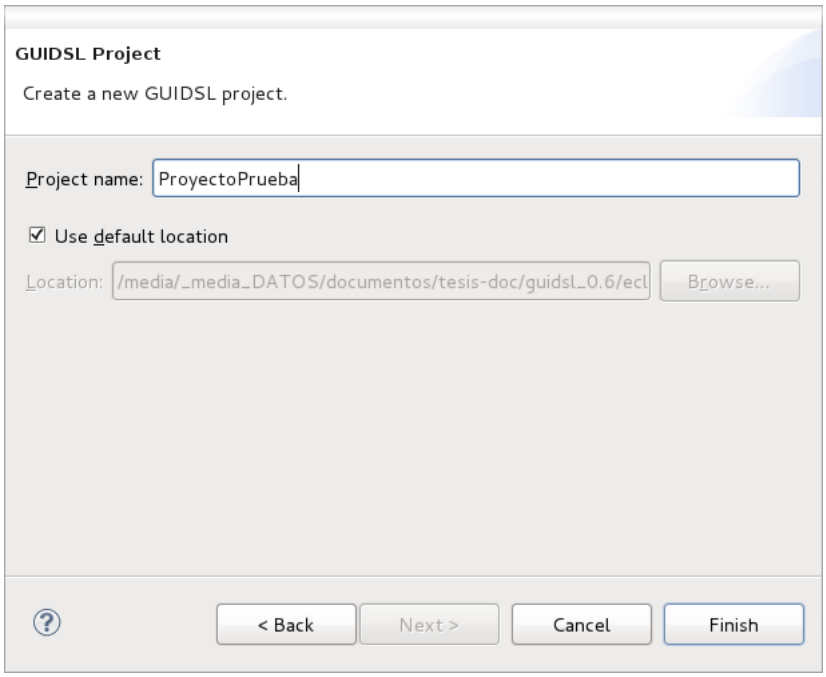

Figura 20: Nombre del proyecto GuiDSL

En la figura 21 se observa como es la estructura del proyecto creado.

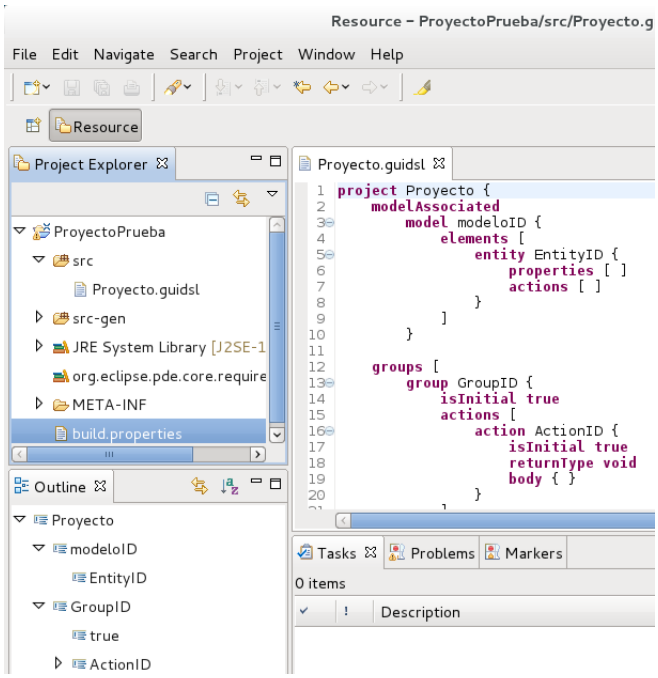

Figura 21: Estructura del proyecto GuiDSL generado

Escribir el modelo

Se debe crear un nuevo archivo de tipo "GuiDSL File" (figura 22). Asignarle un nombre, como contenedor seleccionar el directorio *src* y guardarlo (figura 23).

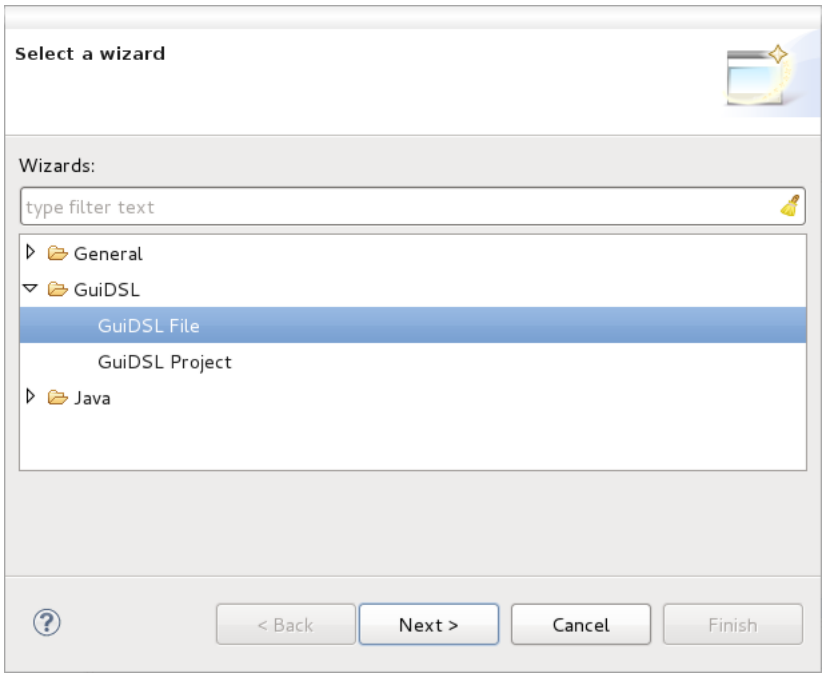

Figura 22: Selección de tipo de archivo GuiDSL

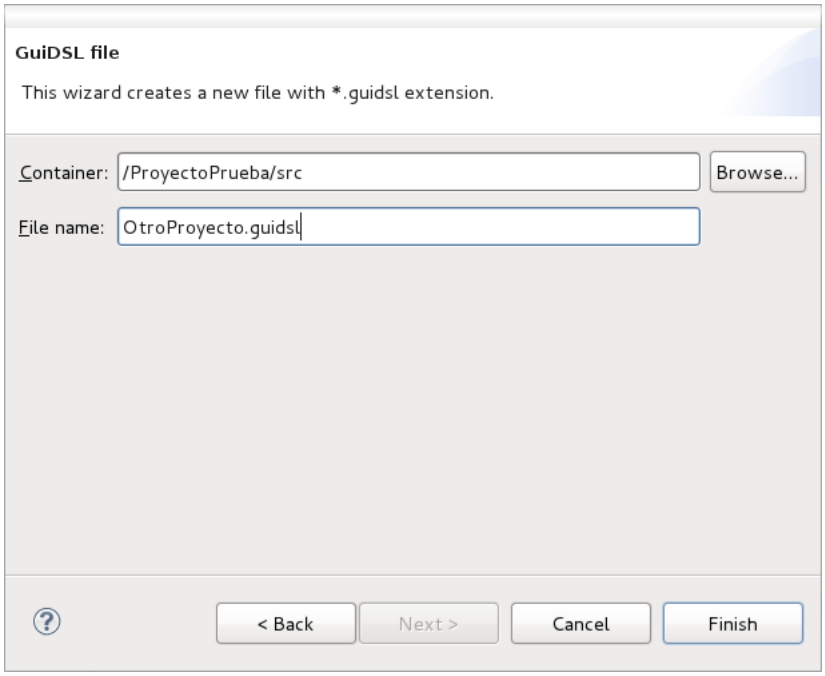

Figura 23: Especificación del contenedor y nombre de archivo

En la figura 24 se observa el archivo creado, el cual posee una plantilla a modo de ejemplo.

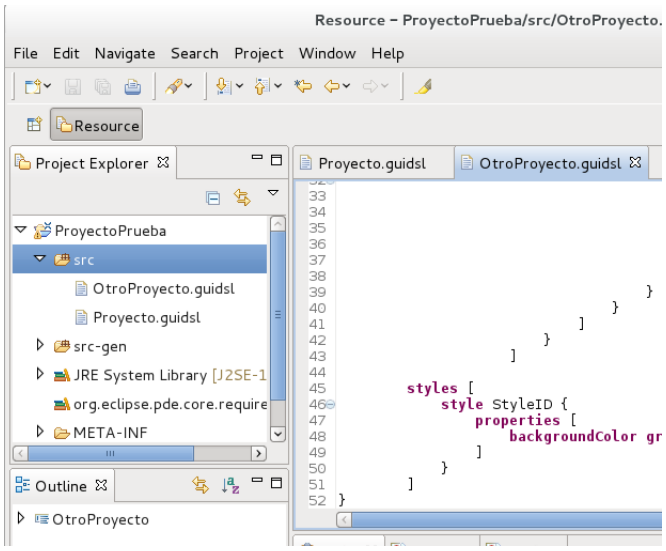

Figura 24: Vista del nuevo archivo en el IDE

En el archivo creado, se comienza a escribir el modelo de la aplicación. El editor posee características que ayudan a la escritura del modelo, como asistente de contexto y resaltador de sintaxis (en la sección 8.2.1 se detallan las principales características del editor).

En el ejemplo se modelará una aplicación pequeña, para dar de alta personas. En el bloque de código que se muestra en el *apéndice C* se muestra el código del modelo.

8.2.1. Editor

El editor textual de GuiDSL ofrece diferentes características, que ayudan al usuario en la escritura del modelo. A continuación se describirá brevemente cada una de ellas.

Proveedor de etiquetas (Label Provider)

Para cada elemento del DSL, se muestra una descripción que ayuda al usuario a entender la función del elemento y cómo se utiliza.

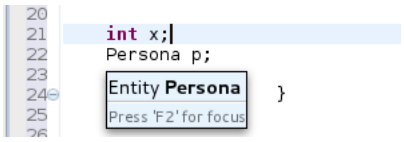

Figura 25: Proveedor de etiquetas

Asistente de contenido (Content Assist)

Esta funcionalidad permite escribir código más rápido y de manera más eficiente. Presionando *Ctrl + barra espaciadora*, el editor ofrece los posibles elementos a escribir, de acuerdo a la sintaxis y al contexto del modelo. Seleccionando uno de los elementos de la lista y presionando *enter*, se inserta en el modelo el elemento seleccionado.

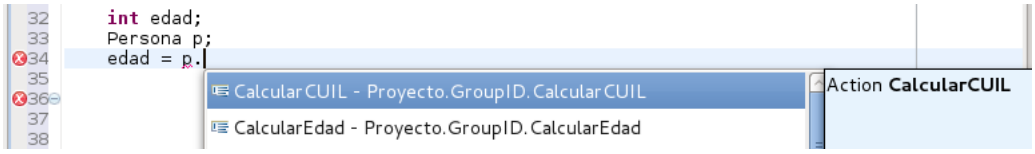

Figura 26: Asistente de contenido

Solución rápida (Quick Fixes)

Cuando el editor detecta algún error sintáctico o semántico en el modelo, remarca el mismo con una linea ondulada roja o amarilla, de acuerdo a si es advertencia o error, y aparece un icono en el margen izquierdo de la linea. Haciendo *click* sobre el icono, o presionando *Ctrl + 1* sobre la linea, el editor ofrecerá posibles soluciones. El *quick fix* también es útil para escribir código más rápido. Por ejemplo, si deseo utilizar una clase que aún no definí, puedo referenciarla por su nombre, y luego utilizando el *quick fix*, el editor me autocompletará el *template* que crea dicha clase, con lo cual, se ahorra escritura de código.

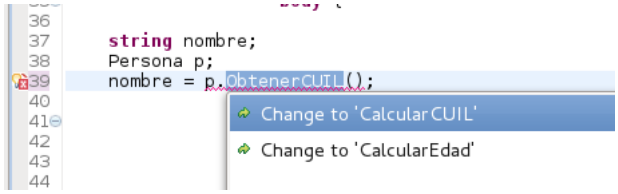

Figura 27: Solución rápida

Propuesta de *templates* (Template Proposals)

Es posible definir *templates* para cada elemento del lenguaje. Estos *templates*, son un pequeño bloque de código que contiene el "esqueleto" de la sintaxis del elemento. Así, presionando *Ctrl + barra espaciadora,* se abrirá un menú el cual en la parte inferior permite seleccionar el *template* deseado. Haciendo *click* sobre el *template*, el editor insertará el bloque de código asociado al elemento, agilizando enormemente la escritura del código.

En la figura 28se observa el editor de templates, en el cual es posible manipular *templates*. Se observa un *template* para la sentencia *for*, cuyo código se muestra en el *preview*.

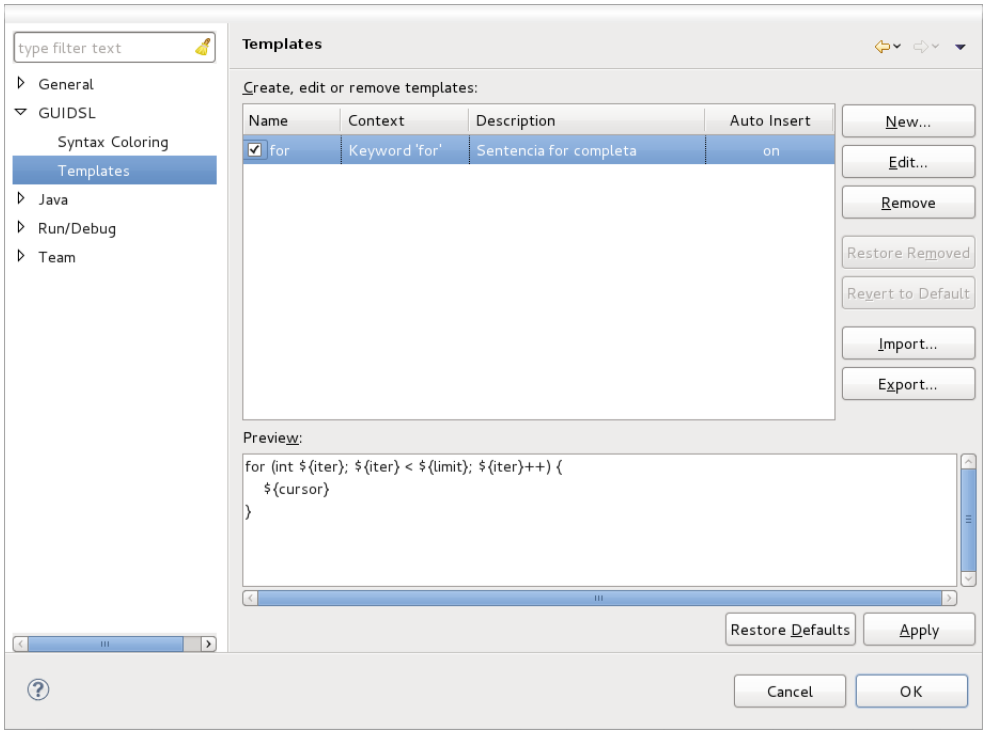

Figura 28: Vista de creación y edición de *templates*

En la figura 29 se observa como el editor ofrece el *template* para la sentencia for.

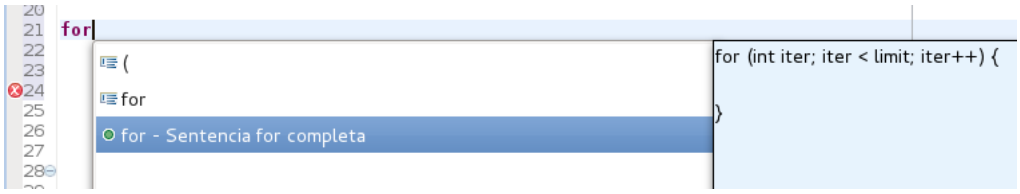

Figura 29: Utilización del *template*

En la figura 30 se observa como el editor completa el código de la sentencia con el *template* seleccionado.

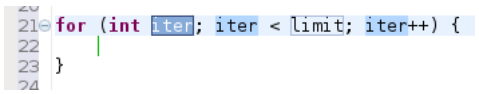

Figura 30: *Template* generado

Vista de outline (Outline View)

La vista de *outline* permite visualizar y navegar los elementos del modelo de manera práctica. Los elementos de muestran jerárquicamente, pero también es posible ordenarlos alfabéticamente. Haciendo doble *click* sobre un elemento en la vista de *outline*, el editor se posiciona en la definición del elemento seleccionado.

El editor posee también un *quick outline*. Presionando *Ctrl + o*, se muestra un pop up que contiene la vista de *outline*, y permite en la parte superior realizar búsqueda de elementos. A medida que se ingresan los caracteres de búsqueda, el *quick outline* se va posicionando en los elementos que coinciden con la búsqueda ingresada. Luego posicionándose sobre un elemento y presionando *enter* sobre el mismo, el editor se posiciona en la definición del elemento.

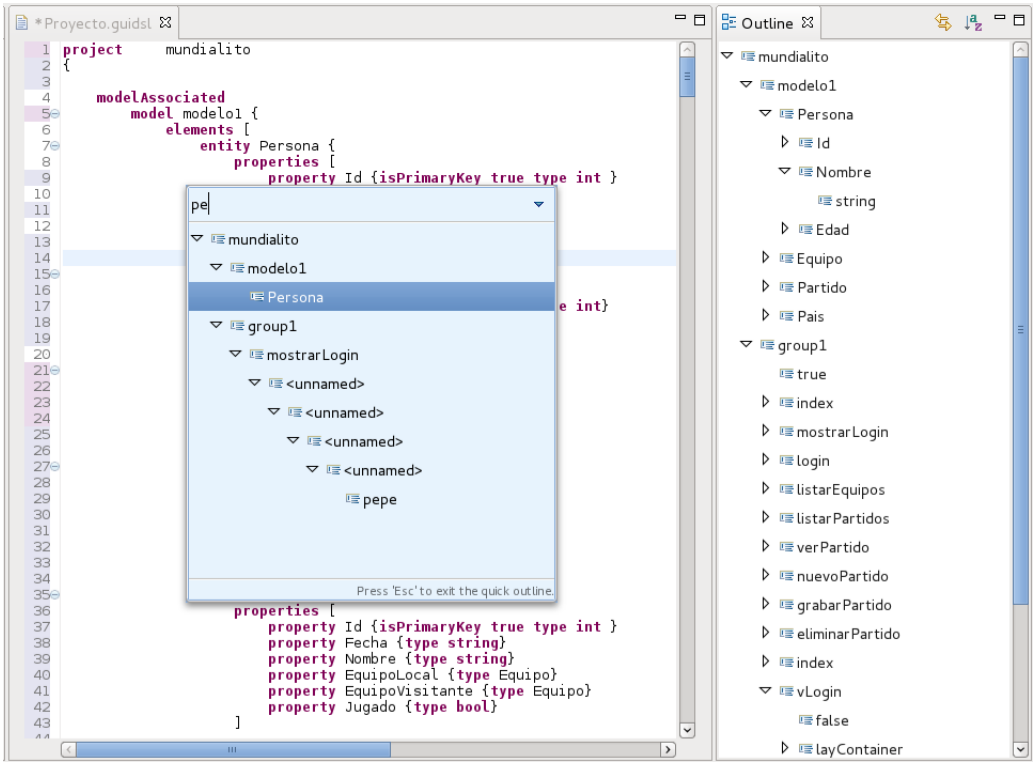

Figura 31: Vista de outline

Vínculos dinámicos (Hyperlinking)

Presionando *F3* o *Ctrl + click* con el mouse sobre una referencia a un elemento del modelo, el editor se posicionará sobre la definición de dicho elemento.

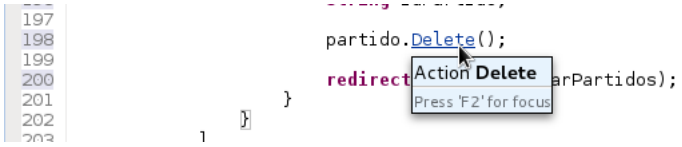

Figura 32: Vínculos dinámicos

Resaltador de sintaxis (Syntax Coloring)

El editor de GuiDSL posee resaltador de sintaxis, el cual ayuda enormemente a la legibilidad del modelo. Se pueden configurar colores y fuentes para mostrar cada elemento del DSL. Esta "decoración" no afecta a la semántica de los elementos, pero ayuda a detectar errores sintácticos más fácilmente.

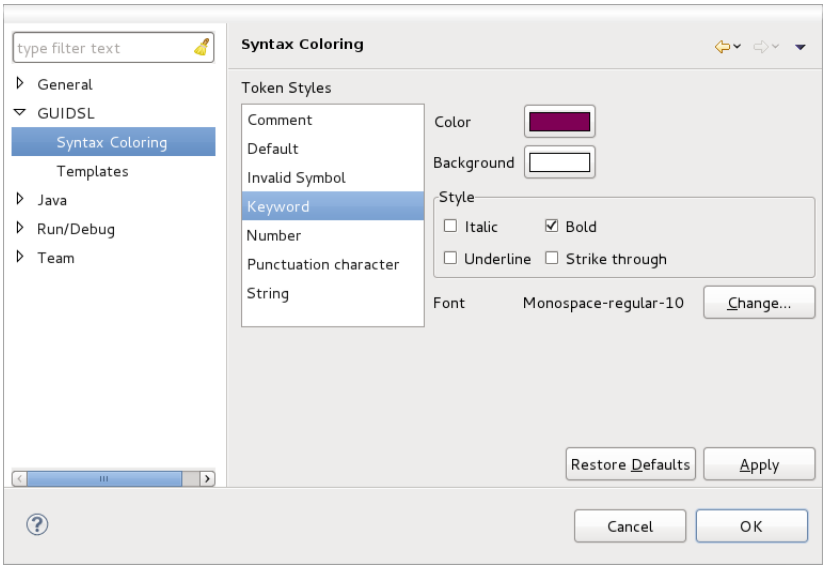

Figura 33: Vista de la configuración del resaltado de sintaxis

Refactorización de identificadores

Al modificar el identificador de algún elemento del modelo, el editor detecta el cambio y actualiza automáticamente todas las referencias al elemento renombrado.

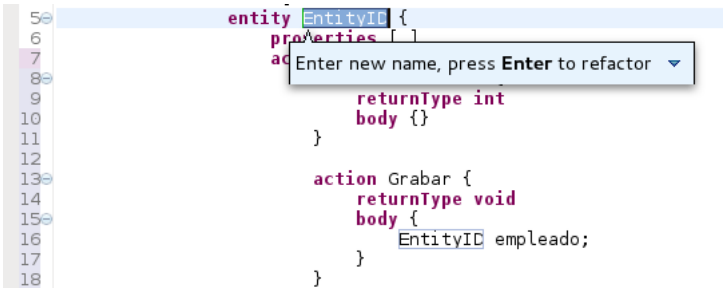

Figura 34: Refactorización de identificadores

8.3. Transformando la aplicación

Una vez que se tiene el modelo escrito en el lenguaje de GuiDSL, se está en condiciones de transformarlo en código ejecutable. Sobre el archivo del modelo, hacer *click* derecho, y en el menú contextual aparecerá

una opción para cada PSM que dispone la herramienta. Hacer *click* sobre el PSM deseado y GuiDSL transformará el modelo en código ejecutable. GuiDSL creará una estructura de directorios dentro del proyecto llamada *generated-code/[Nombre PSM]/[Nombre Proyecto]*. Allí se guardará el código generado, junto todos los archivos necesarios para que la aplicación pueda ser ejecutada. Estos archivos auxiliares podrán ser librerías externas, archivos de configuración o archivos de recursos (hojas de estilo, archivos Javascript, imágenes, etc).

Como se observa en la figura 35, en el ejemplo seleccionamos el PSM *mono_castle*, el cual transformará la aplicación a código CSharp, utilizando el *framework* MVC Castle Project.

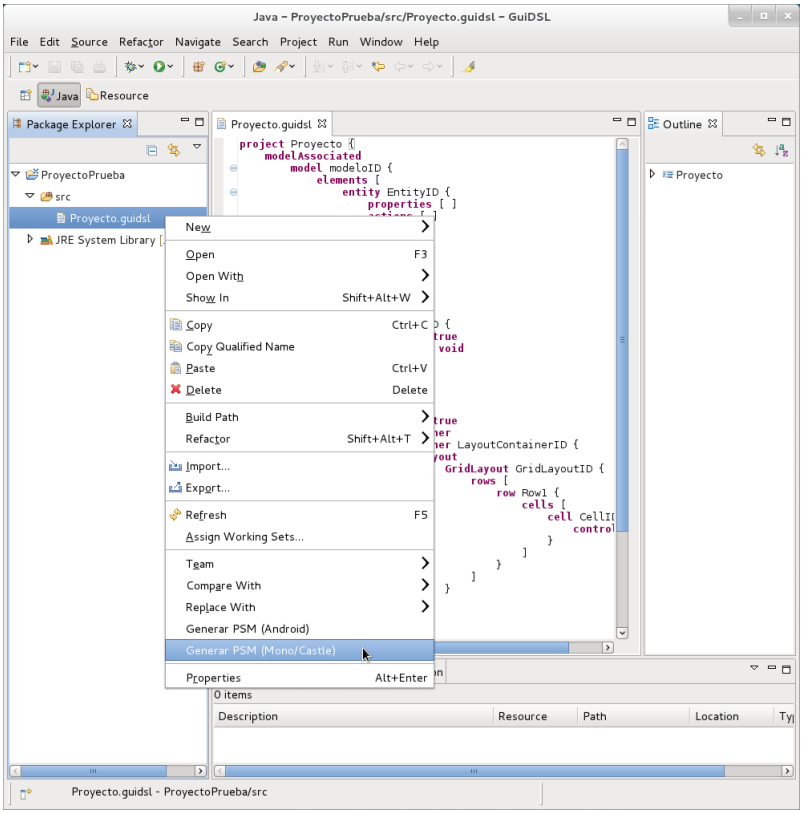

Figura 35: Transformando la aplicación

Como se observa en la figura 36, luego de la transformación, GuiDSL creó la estructura de directorios *generated-code|mono_castle\ProyectoPersonas* la cual contiene el proyecto generado, junto con todos sus archivos auxiliares.

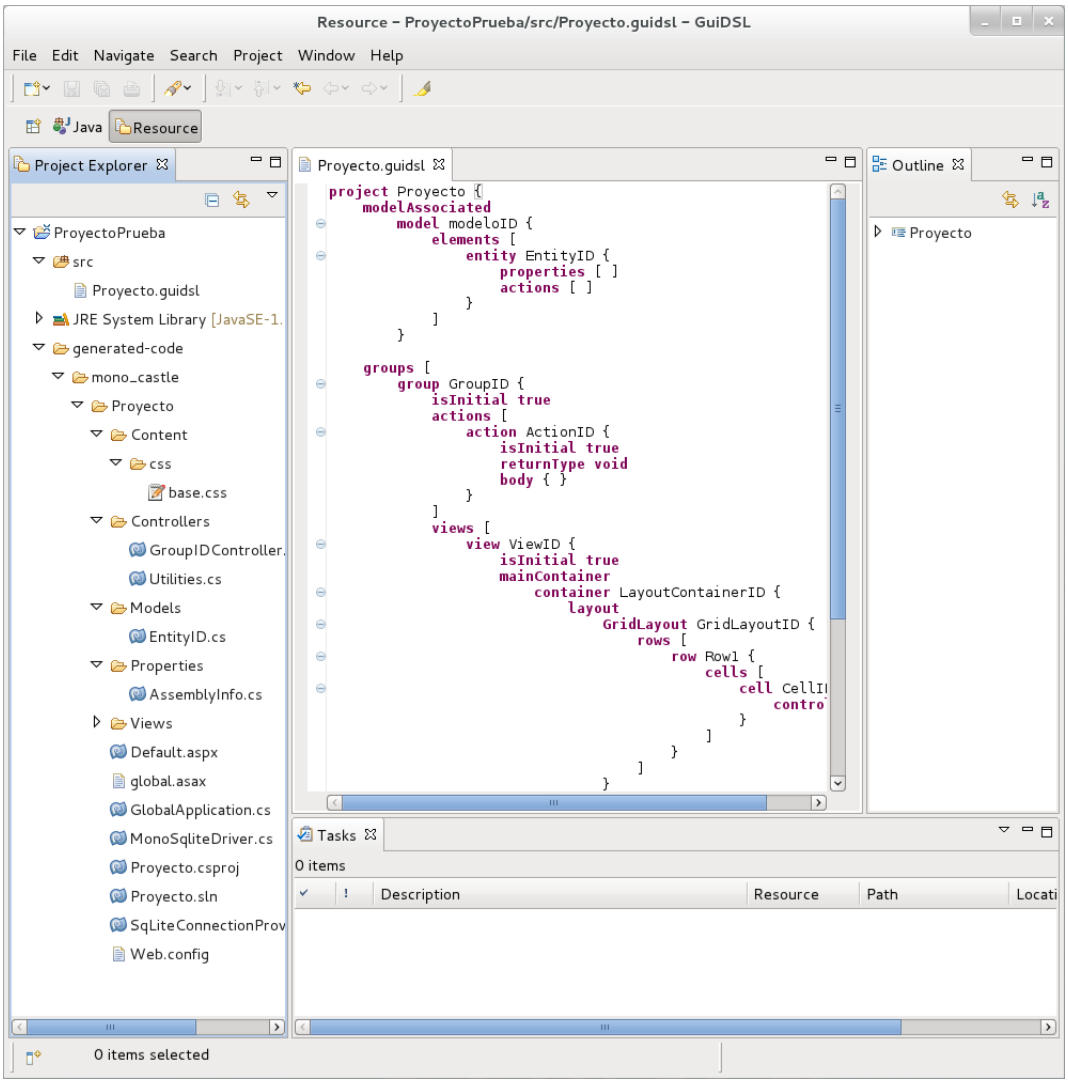

Figura 36: Código generado luego de la transformación

8.4. Utilizando la aplicación generada

Luego de realizada la transformación al PSM deseado, se puede abrir el código fuente y ejecutar la aplicación. Para eso, se debe abrir el IDE correspondiente al lenguaje de programación del PSM. Para este ejemplo, se utiliza el IDE MonoDevelop, ya que el lenguaje de programación del PSM es CSharp. Para eso se debe abrir el archivo de la solución, ProyectoPersona.sln.

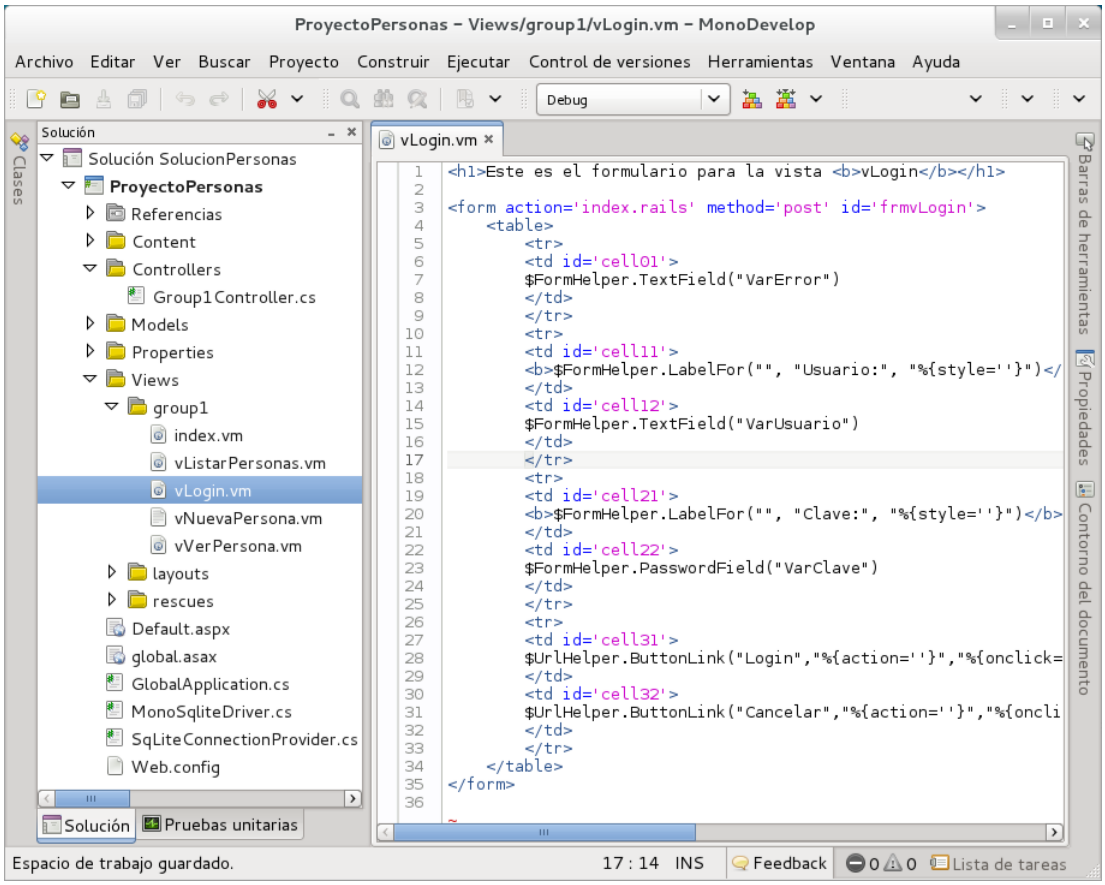

Figura 37: Abriendo la aplicación generada

Como se observa en la figura 37, el proyecto contiene, a excepción de la base de datos, todos los archivos auxiliares necesarios para ser ejecutado sin necesidad de agregarlos o configurarlos manualmente. Cuando la aplicación utilice algún tipo de base de datos, se asumirá que la misma ya existe. De no ser así, el usuario deberá crearla en el servidor o *path* correspondiente, y luego ajustar la referencia en el archivo donde se configura la conexión a la base de datos.

Así, ya se puede ejecutar la aplicación haciendo *click* en la opción *Run* en el menú de MonoDevelop, o presionando la tecla F5.

Cómo se puede observar en la figura 38 ya se cuenta con la aplicación ejecutándose en un navegador, de acuerdo al modelo definido en el capítulo 8.2 y sin necesidad de alterar en absolutamente nada el código generado.

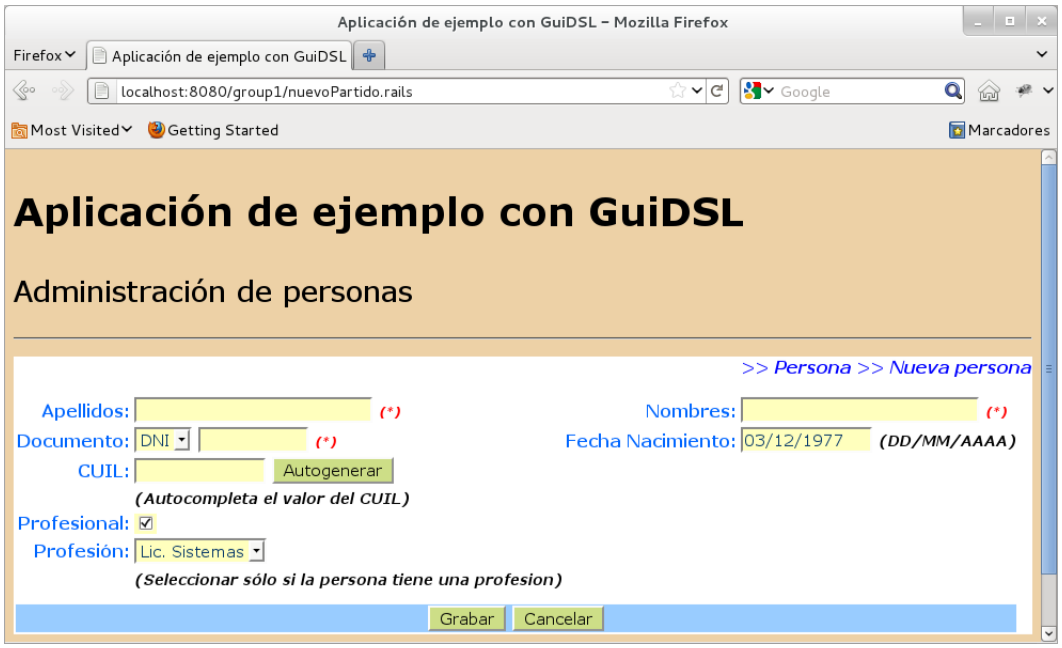

Figura 38: Ejecutando la aplicación generada

8.5. Conclusión

Como se analizó a través del presente capítulo, aprender a utilizar GuiDSL es muy sencillo. Su lenguaje posee sintaxis sencilla e intuitiva. Por otro lado, cuenta con un editor con múltiples características que guían al usuario y simplifican su tarea, por lo cual, la escritura del modelo se convierte en un proceso sencillo y amigable. Como también se presentó en el ejemplo, el lenguaje de GuiDSL permite definir rutinas de código, las cuales podrán manejar eventos o definir la lógica de la aplicación. Esto aporta gran flexibilidad, ya que permite definir de manera completa la aplicación en el modelo de entrada, lo cual se traduce luego, en código 100 % listo para ejecutar.

La transformación del modelo se realiza muy fácilmente, de manera integrada al IDE. No es necesario ejecutar comandos o configurar archivos por fuera de la herramienta. Basta con seleccionar el PSM deseado en el menú contextual, y GuiDSL generará todo el código necesario para utilizar la aplicación. Así, se pudo observar en el ejemplo, cómo es posible abrir el código generado y ejecutar directamente la aplicación, sin necesidad de ser alterado o ajustado por el usuario.

9. Arquitectura de la implementación

9.1. Herramienta utilizada

GuiDSL se desarrolló utilizando Xtext [Xtext]. El mismo es un *framework* de desarrollo de lenguajes basado en EMF, permite crear desde un DSL simple hasta un verdadero lenguaje de propósito general en poco tiempo, utilizando un ambiente de desarrollo sofisticado basado en Eclipse, el cual proporciona características de edición muy completas.

Permite configurar o cambiar, de manera muy sencilla, el comportamiento por defecto de las características más comunes (resaltador de sintaxis, asistente de código, *scoping*, etc.) con implementaciones propias.

Los componentes del compilador del lenguaje creado, son independientes de Eclipse y se pueden utilizar en cualquier ambiente Java. Estos componentes incluyen un analizador, un árbol sintáctico abstracto, un serializador y formateador de código, un *framework* de alcance y vinculación, chequeos sintácticos y semánticos y un generador de código [AHO].

9.2. Estructura

Como se ve en la figura 39 la herramienta está estructurada en distintos módulos: *helper, validation, exceptions, Ide* y *generator*.

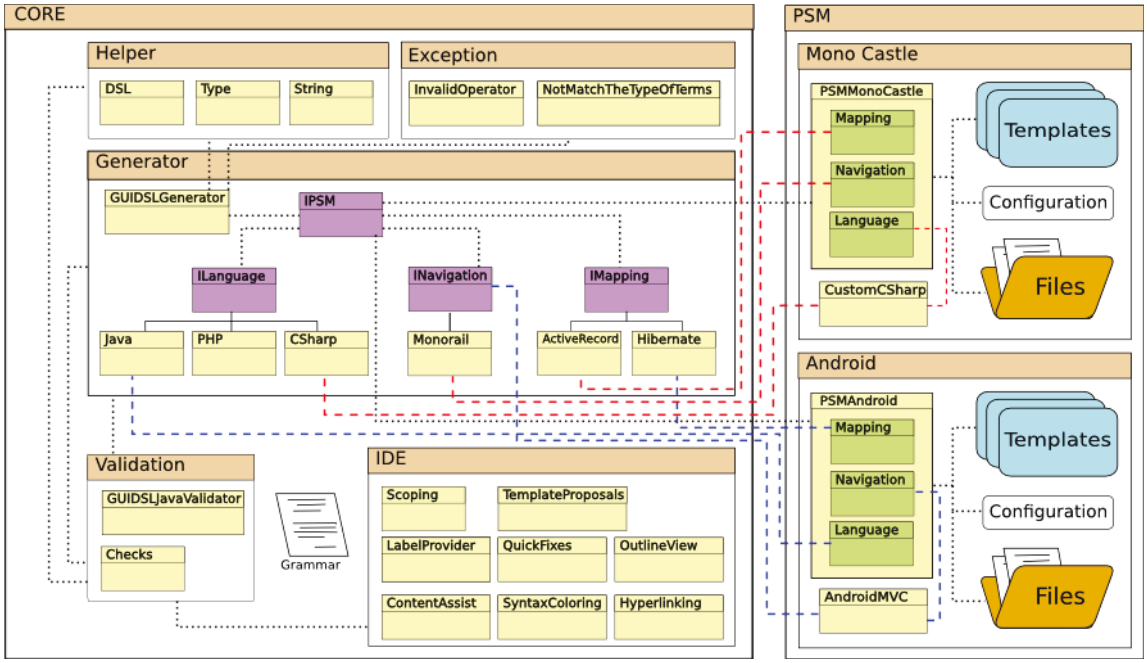

Figura 39: Estructura completa de GuiDSL

Gramática (Grammar)

Define la gramática de GuiDSL con una sintaxis tipo EBNF. Es el archivo *GUIDSL.xtext*. Al ejecutar el *workflow GenerateGUIDSL.mwe2*, se genera el modelo Ecore, junto con todas las clases que representan el meta-modelo.

Validaciones (Validation)

Este paquete está compuesto por la clase *GUIDSLJavaValidator*, contiene los métodos que realizan las validaciones semánticas. Cada vez que se modifica un archivo *.guidsl* se invoca a cada uno estos métodos. Llevan la anotación *@Check*.

Las validaciones pueden ser de tipo error o advertencia.

Excepciones (Exceptions)

Este paquete contiene excepciones personalizadas de GuiDSL. Se refieren a errores de validación del DSL.

Generador (Generator)

La generación de la aplicación se inicia en la clase *GUIDSLGenerator*, la cual implementa la interfaz *IGenerator* de Xtext.

Esta clase en la encargada de invocar a cada uno de los módulo que componen el PSM, los cuales se clasifican en estructura de base, modelo, vistas y controladores (bloque de código 30).

Cada módulo generará una partes de la aplicación final, el cuál recibirá como parámetro variables para acceder a métodos del PSM, el lenguaje y mapeo, además de las variables *resource* y *fsa*, las cuales permiten el acceso al modelo instanciado y realizar operaciones sobre el sistema de archivos respectivamente.

Bloque de código 30 Código de la clase *GUIDSLGenerator*

```
class GUIDSLGenerator implements IGenerator {
    @Inject IPSM psm
    @Inject ILanguage language
    @Inject IMapping mapping

    override void doGenerate(Resource resource, IFileSystemAccess fsa) {
        psm.generateBaseStructure(resource, fsa);
        psm.generateModel(resource, fsa, language, mapping);
        psm.generateView(resource, fsa, language, mapping);
        psm.generateController(resource, fsa, language, mapping);
    }
}
```

Clases de ayuda (Helper)

Este paquete cuenta con clases que ayudan al programador encapsulando funciones o métodos de uso repetitivo.

Son invocados desde el generador o desde las validaciones.

- **HelperDSL**: métodos que realizan operaciones sobre el modelo instanciado, generalmente recorriendo los nodos y haciendo distintas validaciones, por ejemplo, determinar si la cantidad de parámetros coincide con la cantidad de argumentos.
- **HelperType**: esta clase contiene métodos de validaciones entre tipos de datos y operadores.
- **HelperString**: contiene utilidades para manipular *strings*, por ejemplo, pasar la primera letra a mayúscula.
- **GUIDSLExtension**: similar a HelperDSL, pero más enfocado en cuestiones del DSL, por ejemplo, obtener el grupo inicial del proyecto.

Entorno de desarrollo (IDE)

Conjunto de clases que extienden clases Xtext y permiten personalizar el comportamiento de las características que provee el IDE de Eclipse, por ejemplo, agregar íconos a los nodos de la vista *outline* o resaltar una expresión según cumpla una regla semántica.

PSM

Estos paquetes contienen la especificación de cada uno de los PSM. Su estructura se explicó en detalle en la sección 7.2.

9.3. Cómo extender la herramienta

9.3.1. Extender el lenguaje

En caso de ser necesario extender el lenguaje de GuiDSL para incorporar nuevas reglas u optimizar, personalizar o extender reglas existentes, es posible realizarlo siguiendo los siguientes pasos:

1. Incorporar o modificar las reglas correspondiente en el archivo de la gramática *GUIDSL.xtext*.

2. Ejecutar el *workflow GenerateGUIDSL.mwe2*, el cual generará el modelo Ecore, junto con todas las clases que representan el meta-modelo.

3. De acuerdo al tipo de regla agregada o modificada, se debe agregar en la interfaz correspondiente (*ILanguage*, *IMapping* o *INavigation*) los métodos necesarios que realicen la transformación de la regla.

4. Modificar las clases que implementan la interfaz afectada en el punto anterior, para que implementen los nuevos métodos.

5. En caso de la nueva regla requiera métodos de soporte para su transformación, se deben modificar los *helpers*.

6. Incorporar, de ser necesario, en la clase *GUIDSLJavaValidator* los chequeos semánticos correspondientes.

7. Incorporar, de ser necesario, los métodos que determinan el alcance (*scoping*) de la nueva regla.

8. Adaptar los *templates* y extensiones de cada PSM para que generen la nueva regla.

9.3.2. Crear nuevos PSM

Cuando sea necesario incorporar a la herramienta nuevos PSMs, se podrá realizar fácilmente siguiendo los siguientes pasos:

1. Para comenzar a abstraer las características del lenguaje de programación y la tecnología del PSM necesario, se aconseja programar una aplicación funcional con dicha tecnología, de la manera tradicional. Esta aplicación deberá abarcar todas las características del PSM (utilizar todas las sentencias, todos los tipos de controles gráficos, manejar todos los tipos de eventos, etc). Luego analizar el código fuente de la aplicación, abstrayendo características fijas de la arquitectura del proyecto, de sus archivos y de sus dependencias. El esfuerzo invertido en esta tarea es alto, pero luego se ve amortizado con cada generación que utilizará el PSM creado.

2. Copiar y pegar todos los archivos que componen el proyecto programado, y determinar cuál código debería cambiar en función del modelo de entrada, es decir, cuáles características no son fijas de la tecnología y dependen directamente del PIM. Ubicado este código, generar la transformación específica, utilizando funciones, clases externas, *helpers*, las clases del lenguaje, etc.

3. **Posibilidad de reutilizar parte de otro PSM:** respecto de las clases internas del PSM, que implementan las interfaces *ILanguage*, *IMapping* o *INavigation*, es posible reutilizarla alguna de ellas cuando el nuevo PSM comparta alguna característica con otro PSM existente. Por ejemplo, si el nuevo PSM utiliza un nuevo *framework* MVC para Java, las transformaciones sobre las sentencias para el lenguaje de programación Java, no es necesario implementarlas, ya que posible reutilizar esta parte, la cual ya ha sido desarrollada y es compartida por varios PSMs. En cambio, si las transformaciones existentes no se adaptan a las necesidades particulares del programador del PSM, éste debería extender la implementación existente o crear una propia. En algunos casos el programador deberá crear estas clases, por ejemplo, cuando surja un nuevo lenguaje de programación o un nuevo *framework*.

4. Crear un paquete para el nuevo PSM que contenga todos los archivos necesarios: clases que implementan las interfaces, templates, archivos de configuración, librerías y recursos externos, etc.

9.4. Conclusión

GuiDSL está desarrollado utilizando Xtext. Éste es un framework de desarrollo de lenguajes, basado en las herramientas de EMF e integrado con Eclipse. Posee un editor potente, y herramientas que dan muy buen soporte a la escritura del lenguaje, de sus validaciones y de su compilador.

La herramienta está diseñada de manera modular, donde cada módulo tiene una función bien definida. Así la arquitectura de GuiDSL es muy simple de entender y aprender.

Cuando se desee extender la herramienta, el lenguaje o los PSMs, será muy fácil de realizar, ya que estos cambios no implican grandes modificaciones en sus archivos. Se explicó paso a paso como hacerlo, y se destacó la importancia y utilidad de poder reutilizar parte de otros PSMs existentes.

Parte IV
Conclusiones

En la parte IV se realiza un resumen de los contenidos más importantes vistos a lo largo de la tesina, se presentan los inconvenientes con los trabajos relacionados, y conclusiones y aportes de la herramienta desarrollada. Finalmente se exponen posibles líneas de trabajo sobre la investigación y la herramienta, para quienes estén interesados en contribuir al área de estudio de la tesina.

10. Conclusiones

Las interfaces gráficas de usuario constituyen hoy en día uno de los factores más relevantes a la hora del diseño y el desarrollo de una aplicación para garantizar la aceptación de los usuarios. Un buen diseño de interfaz requiere tiempo, esfuerzo y conocimiento acerca de buenas prácticas de diseño centrado en el usuario, accesibilidad y facilidad de uso, cuestiones en las cuales muchas veces los programadores de sistemas no se encuentran familiarizados o ni siquiera se sienten interesados.

Debido a la diversidad de plataformas en las cuales operan los sistemas, como consola de texto, interfaz gráfica de escritorio, interfaz web y, las más recientes, interfaces para dispositivos móviles, surge constantemente la necesidad de adaptación o migración de plataformas o lenguajes para una misma aplicación. Esto conlleva a invertir nuevamente los costos de diseñar la nueva interfaz de usuario.

Se destacó lo costoso que es diseñar interfaces que satisfagan correctamente las necesidades del usuario y también los costos de investigación y aprendizaje de cada tecnología a utilizar.

El paradigma MDE soluciona la mayoría de estos inconvenientes y ofrece una solución integral, basada en transformaciones de modelos, que permite generar la aplicación en múltiples plataformas o tecnologías a partir de un modelo de entrada. Además permite a los programadores abstraerse de las cuestiones técnicas de cada implementación, ya que si se dispone del PSM deseado, éste es el responsable de generar el código para dicha tecnología. Esto simplifica significativamente el proceso de desarrollo de software, permitiendo reducir tiempos y costos. Así los programadores pueden enfocarse a desarrollar modelos que se ajusten mejor a las necesidades del negocio de la aplicación, y a diseñar interfaces gráficas más sofisticadas, que ayuden a una mejor experiencia de uso por parte del usuario.

Como se ha analizado, existen numerosos trabajos relacionados a este tema, y debido a sus evidentes beneficios, el paradigma MDE ha evolucionado enormemente, y se han desarrollado numerosas herramientas, las cuales se encuentran en proceso de expansión y maduración. Sin embargo, estas herramientas poseen limitaciones de índole técnica, las cuales hacen que aún, la utilización de las mismas no se encuentre instalada masivamente en el proceso de desarrollo de software. Estas limitaciones, son a veces determinantes a la hora de decidir incorporarlas al desarrollo de una aplicación:

- En general no son multiplataforma o multilenguaje, es decir, generan código para una única plataforma o lenguaje.
- No permiten definir comportamiento en el PIM, o si lo hacen, es de manera muy limitada.
- No generan código 100 % listo para ejecutar. El programador debe modificar o completar el código generado, lo cual, para las herramientas que no permiten definir regiones protegidas de código, esto deberá repetirse en cada iteración de la generación. Esto "empaña" el concepto básico de MDE sobre los modelos, y la bidireccionalidad modelo - código. Por otro lado ante la necesidad cambiar de implementación, se deberá replicar y mantener manualmente este código adicional.
- Respecto de las interfaces gráficas, sólo generan interfaces tipo CRUD para las entidades del modelo. No es posible definir eventos o apariencia de manera personalizada.

Así surge la herramienta GuiDSL, que permite minimizar estos inconvenientes y ayuda a agilizar y mejorar el proceso de implementación de las interfaces gráficas. A través de transformaciones sobre un único modelo de entrada, se puede obtener la aplicación en distintos lenguajes de programación y plataformas, permitiendo incorporar la lógica del negocio; y para sus interfaces gráficas, su apariencia, comportamiento y navegabilidad.

Para la definición del PIM de la aplicación, GuiDSL posee un DSL textual, de sintaxis clara y sencilla. Este lenguaje posee reglas que permiten definir una aplicación de manera completa: las entidades del modelo, acciones, sus interfaces gráficas con su apariencia y comportamiento. Para esto, GuiDSL utiliza un conjunto mínimo de reglas que abarcan todos estos aspectos. Las reglas que permiten definir las interfaces gráficas fueron establecidas como consecuencia del análisis de la arquitectura de éstas presentado en el capítulo 4.

Se determinaron cuáles son los controles presentes en todas las plataformas, cuáles son sus propiedades y cómo es el manejo de sus eventos. Las reglas del lenguaje de acciones también se establecieron analizando los lenguajes de programación más populares para las distintas plataformas. Así, se determinó un conjunto mínimo de sentencias que permiten realizar todo lo necesario en las acciones, como manejo de eventos, lógica del negocio, y cualquier método o rutina que se requiera. Esto aporta gran flexibilidad, ya que permite definir de manera completa la aplicación, lo cual se traduce luego, en código 100 % listo para ejecutar. El lenguaje posee distintos tipos de chequeos sintácticos y semánticos, los cuales verifican que los PIMs sean válidos. De ser necesario, el lenguaje podrá ser extendido fácilmente, incorporando reglas nuevas, o reglas que optimicen, personalicen o amplíen las existentes.

GuiDSL hace uso de transformaciones para obtener la aplicación implementada sobre una plataforma tecnológica específica. Estas transformaciones están modularizadas y son fácilmente extensibles. Cada PSM utiliza un conjunto de archivos, algunos propios de una transformación específica y otros auxiliares, que son utilizados para dar soporte a las transformaciones. Así es posible desarrollar nuevos PSMs, reutilizando parte de otro existente.

Aprender a utilizar GuiDSL es muy sencillo. El IDE posee un editor con diversas funcionalidades, las cuales hacen que la escritura del PIM sea un proceso simple y amigable. La transformación se realiza muy fácilmente, de manera integrada al IDE. No es necesario ejecutar comandos o configurar archivos por fuera de la herramienta. Basta con seleccionar el PSM deseado en el menú contextual, y GuiDSL genera la estructura completa del proyecto, conteniendo todos los archivos auxiliares necesarios para ser ejecutado, sin necesidad de agregarlos o configurarlos manualmente. Es posible ejecutar el proyecto en el IDE correspondiente directamente, sin necesidad de ser alterado o ajustado por el usuario.

GuiDSL posee todas las características y funcionalidades necesarias para ser utilizado sin inconvenientes, se destaca su capacidad de ser extendido fácilmente, tanto su lenguaje como sus transformaciones, lo cual le permite acompañar los avances tecnológicos que surgen constantemente, o adaptarse ante nuevas necesidades.

Por otro lado es importante destacar que GuiDSL está construido sobre el framework de modelado de Eclipse (EMF), el cual, además de ser software libre, se encuentra en constante crecimiento, aportando un importante respaldo y soporte tecnológico.

GuiDSL propone una solución basada en MDE que reduce la mayoría de los inconvenientes más importantes que enfrenta el proceso de desarrollo de software y, en especial, la implementación de las interfaces gráficas de las aplicaciones. El esfuerzo de desarrollo se concentra en la escritura del modelo de la aplicación, sin detalles tecnológicos de la implementación. Luego las transformaciones serán las encargadas de generar el proyecto final, por lo cual los costos tecnológicos se ven considerablemente reducidos.

11. Trabajos futuros

En este capítulo se presentan posibles lineas de trabajo para extender y mejorar GuiDSL.

Editor gráfico

El lenguaje de GuiDSL es textual. Posee un editor muy completo, con funcionalidad que ayuda y mejora el proceso de escritura del modelo. Sería útil contar con un editor gráfico que permita agregar y relacionar ciertos elementos del modelo visualmente. Para el diseño de las interfaces gráficas contar un editor visual es primordial, ya que es costoso definir textualmente los estilos y la posición de cada control de las vistas. Contar con un editor gráfico que ofrezca esta posibilidad simplificaría significativamente la definición de las vistas, además de permitirle al programador ir evaluando visualmente el diseño de las mismas. Este editor también podría utilizarse para la definición del modelo de entidades de la aplicación.

Extensiones del lenguaje

El lenguaje de GuiDSL abarca un conjunto mínimo de reglas, pero suficientes, que permiten definir una aplicación. Como se ha expuesto a lo largo de la tesina, este lenguaje es fácilmente extensible, incorporando nuevas reglas y chequeos. Sería útil extender el lenguaje incorporando reglas, tanto en el lenguaje de acciones, como reglas relacionadas a la definición de las vistas:

- Reglas en el lenguaje de acciones: se podrían incorporar nuevas sentencias, como por ejemplo *foreach* o *switch*. También sería útil contar con reglas para el manejo de librerías (importar librerías) y para el manejo de archivos (crear y editar archivos).
- Reglas para nuevos tipos de datos: GuiDSL posee un conjunto reducido de tipos. Los mismos podrían extenderse para ofrecer mayor flexibilidad, o se podrán crear nuevos tipos:
  - Extender tipos existentes: GuiDSL incluye el tipo entero, el cual se podría extender incorporando reglas para el tipo número flotante o entero corto.
  - Crear tipos de datos de uso frecuente, como por ejemplo el tipo fecha y hora.
  - Incorporar el tipo enumerativo.
  - Incorporar tipos de datos complejos, que se ajusten a algún dominio específico (por ejemplo para una aplicación que realice cálculos matemáticos se podría necesitar el tipo número complejo).
  - Incorporar tipos de datos definidos por el usuario.
- Reglas para la definición de las interfaces gráficas: sería útil incorporar reglas que definan otros tipos de controles presentes en algunos lenguajes modernos de programación como por ejemplo grillas o listas seleccionables. También se podrían definir controles más sofisticados que encapsulen la funcionalidad de dos o más controles, permitiendo definir a través de la regla todo su comportamiento (por ejemplo un *checkBox* y una lista desplegable, la cual se habilita dependiendo si el *checkBox* está chequeado o no).

Definición del modelo en varios archivos

En GuiDSL el modelo de la aplicación se escribe en un único archivo (con extensión ".GuiDSL"). Sería deseable tener la posibilidad de definir el proyecto en más de un archivo, lo cual aportaría las siguientes ventajas:

- Permitiría ordenar, clasificar y simplificar la escritura del modelo. Se podría organizar la definición de la aplicación en distintos archivos, por ejemplo, un archivo para cada grupo, o vista, otro archivo que defina el modelo de entidades, etc. Esto aportaría además mejor legibilidad al modelo.
- Permitiría reusar partes del modelo definidas previamente en algún otro archivo. Por ejemplo, si se requiere escribir el modelo de una aplicación que administre los recursos humanos de una empresa, y se dispone con anterioridad del modelo de una aplicación que manipula personas, se podrían reutilizar las vistas y el modelo de entidades de ésta última.

Chequeos semánticos

GuiDSL posee distintos chequeos semánticos, los cuales garantizan la correctitud de modelo de entrada. Sería de utilidad incorporar nuevos chequeos que ayuden a la escritura y correctitud del modelo (por ejemplo un chequeo semántico que valide que cada variable esté inicializada antes de su uso).

Módulos de deploy

Luego de generar la aplicación, para poder utilizar el código obtenido, se requieran tal vez distintas tareas de *deploy* (instalar servicios, registrar controles, copiar archivos, etc) las cuales deben realizarse manualmente luego de la generación. Sería muy útil que el PSM incluya la automatización de estas tareas de *deploy* sobre el código generado, ya que las mismas dependen directamente de la tecnología aplicada por el PSM.

Nomenclatura

API Application programming interface

ASMX Active Server Method File

ATL Atlas Transformation Language

BSD Berkeley Software Distribution

CARAT Java 2 Platform, Enterprise Edition

CLI Command Line Interface

CRUD Create Read Update Delete

DAO Data Access Component

EBNF Extended Backus–Naur Form

EJB Enterprise Java Beans

EMF Eclipse Modeling Framework

GDI Graphics Device Interface

GNU GNU is Not Unix

GWT Google Web Toolkit

IDE Integrated development environment

J2EE Java 2 Platform, Enterprise Edition

JMI Java Metadata Interface

JNI Java Native Interface

JSF Java Server Face

JSON JavaScript Object Notation

LGPL Lesser General Public License

MDA Model-Driven Architecture

MDE Model-driven engineering

MIDP Mobile Information Device Profile

MIT Massachusetts Institute of Technology

MOF Meta-Object Facility

MPL Mozilla Public License

OCL Object Constraint Language

OCX OLE Control eXtension

PIM Platform-Independent Model

PSM Platform Specific Model

QVT Query View Transformation

RIA Rich Internet Aplication

SQL Structured Query Language

TAD Tipo Abstracto de Datos

UI User Interface

UML Unified Modeling Language

XMI XML Metadata Interchange

XML Extensible Markup Language

Referencias

[WEBRATIO] Web Models. WebRatio. Online, accedido en Agosto de 2011.
URL http://www.webratio.com

[UJN] Jacob Nielsen. *Desingning Web Usability*. New Riders, 1999.

[USA] Jakob Nielsen. Usability 101: Definition and Fundamentals - What, Why, How. Online, accedido en Junio de 2011.
URL http://www.useit.com/alertbox/20030825.html

[MDE] Douglas C. Schmidt. Model Driven Engineering. *IEEE Computer Society*, 2006.

[MDA] MDA Guide, Junio 2003.
URL http://www.omg.org/

[PONS] Gabriela Pérez Claudia Pons, Roxana Giandini. *Desarrollo de Software Dirigido por Modelos: Conceptos teóricos y su aplicación práctica*. Editorial de la Universidad Nacional de La Plata, 2010.

[EMF] Eclipse. Eclipse Modeling Framework. Online, accedido Septiembre de 2011.
URL http://www.eclipse.org/modeling/emf

[MSDSL] Short K. Cook S. Kent S. Greenfield, J. *Software Factories*. Wiley, 2004. ISBN 0-471-20284-3.

[OZGUR] Turhan Özgür. *Comparison of Microsoft DSL Tools and Eclipse Modeling Frameworks for Domain-Specific Modeling In the context of the Model-Driven Development*. Master's thesis, School of Engineering - Blekinge Institute of Technology, 2007.

[ANDROMDA] AndroMDA 123. Online, accedido en Julio de 2011.
URL http://www.andromda.org

[ARCSTYLER] Interactive Object Software. ArcStyler. 2005.

[OPTIMALJ] Corporation Compuware. Using OptimalJ 3.1.
URL http://www.uio.no/studier/emner/matnat/ifi/INF5120/v04/verktoy/OptimalJTutorials.pdf

[ACCELEO] Obeo. Acceleo. Online, accedido en Julio de 2011.
URL http://www.acceleo.org

[WEBML] WebML. WebML. Online, accedido en Agosto de 2011.
URL http://www.webml.org

[XWINDOW] X Window System wikipedia. Online, accedido en Mayo de 2011.
URL http://es.wikipedia.org/wiki/X_Window_System

[XORG] X.Org. Online, accedido en Mayo de 2011.
URL http://www.x.org/wiki/

[GDI] GDI. Online, accedido en Mayo de 2011.
URL http://msdn.microsoft.com/en-us/library/aa925824.aspx

[GDI2] GDI wikipedia. Online, accedido en Mayo de 2011.
URL http://en.wikipedia.org/wiki/Graphics_Device_Interface

[OPENGL] Open GL. Online, accedido en Mayo de 2011.
URL http://www.opengl.org

[SWING] Oracle. Swing. Online, accedido en Mayo de 2011.
URL http://download.oracle.com/javase/tutorial/ui

[GTK] GTK. Online, accedido en Mayo de 2011.
URL http://www.gtk.org

[QT] Qt wikipedia. Online, accedido en Mayo de 2011.
URL http://en.wikipedia.org/wiki/Qt_%28framework%29

[QT1] Qt. Online, accedido en Mayo de 2011.
URL http://qt.gitorious.org

[QT2] Qt Documentation. Online, accedido en Mayo de 2011.
URL http://doc.qt.nokia.com

[TK] TK. Online, accedido en Mayo de 2011.
URL http://www.tcl.tk

[WXWIDGETS] wxWidget. Online, accedido en Agosto de 2011.
URL http://www.wxwidgets.org

[GECKO] Gecko. Online, accedido en Mayo de 2011.
URL https://developer.mozilla.org/en/Gecko_FAQ

[WEBKIT] WebKit. Online, accedido en Mayo de 2011.
URL http://www.webkit.org

[TRIDENT] Trident. Online, accedido en Mayo de 2011.
URL http://msdn.microsoft.com/en-us/library/aa741312%28v=vs.85%29.aspx

[NUI] Daniel Priego. Interfaz natural de usuario. *Expression Lab*, 2009.
URL http://expressionlab.net/tag/interfaz-natural-de-usuario/

[BLACKBERRY] BlackBerry. Online, accedido en Mayo de 2011.
URL http://m.blackberry.com/ar/es/blackberry-os-6.html

[BLACKBERRYUI] Blackberry UI. Online, accedido en Mayo de 2011.
URL http://www.blackberry.com/developers/docs/5.0.0api/UI-summary.html

[IOS] iOS. Online, accedido en Marzo de 2011.
URL http://developer.apple.com/technologies/ios

[WINPHONE] Microsoft. Windows Phone SDK Tools. Online, accedido en Marzo de 2011.
URL http://msdn.microsoft.com/en-us/library/ff402523%28v=vs.92%29.aspx

[MEEGO] Meego. Online, accedido en Mayo de 2011.
URL https://meego.com

[AUTOPC] Using Automotive Speech Architecture. Online, accedido en Mayo de 2011.
URL http://msdn.microsoft.com/en-us/library/ms834174.aspx

[NEI] ATRC – Neural interface devices. Online, accedido en Mayo de 2011.
URL http://www.utoronto.ca/atrc/reference/tech/neuralinterface.html

[GJC] Ghezzi Carlo Jazayeri Mehdi. *Programming languaje concepts.* Wiley, 3 edition edition, 1998.

[Xtext] Xtext. Online, accedido en Enero de 2011.
URL http://xtext.org

[AHO] Ravi Sethi Jeffrey Ullman Alfred V. Aho, Mónica S. Lam. *Compiladores, principios, técnicas y herramientas.* Addison-Wesley Iberoamericana, 2007.

[ACID] Acid Tests. Online, accedido en Mayo de 2011.
URL http://www.acidtests.org

[ANDROID] Android. Online, accedido en Mayo de 2011.
URL http://www.android.com

[CLJ] Luis Joyanes Albeiro Cuesta, Marcelo López. Comparativo de herramientas MDA. 2009.

[DOJO] Foundation Dojo. Dojo Toolkit. Online, accedido en Noviembre de 2011.
URL http://dojotoolkit.org

[JQUERY] jQuery. Online, accedido en Noviembre de 2011.
URL http://jquery.com

[MOOTOOLS] Mootools. Online, accedido en Noviembre de 2011.
URL http://mootools.net

[PROTOTYPEJS] PrototypeJS. Online, accedido en Noviembre de 2011.
URL http://www.prototypejs.org

[UJN2] Jakob Nielsen. *Eyetracking Web Usability*. New Riders Press, 2009.

Apéndice

	AndroMDA	ArcStyler	OptimalJ	Acceleo
Lenguaje de modelado	UML (MagicDraw, Poseidon, ArgoUML entre otras)	UML (MagicDraw)	UML (MagicDraw)	UML
Calidad del código generado	Bien documentado y legible	Bien documentado y legible	Bien documentado pero poco legible ya que utiliza varios patrones en el código	Bien documentado y legible
Transformaciones	La aplicación permite tanto la creación de nuevos PSMs (permitiendo usar hasta la propia herramienta para generarlos ya que son archivos .JAR) así como también la extensión de los existentes	Permite la creación de nuevos PSMs y también la extensión de los existentes	La edición OptimalJ Professional no permite la modificación de las transformaciones pero la edición OptimalJ Architecture brinda control absoluto sobre ellas	Permite la creación de nuevos PSMs y también la extensión de los existentes de acuerdo a su licencia
Generación UI (User Interface)	Utiliza JSP o Struts para Java	Dispone de un buen mecanismo para el diseño de las interfaces gráficas a través del modelo Accessor	Modelo de presentación poco expresivo que no permite especificar el flujo de control de la navegación web, imprescindible para construir aplicaciones “a medida”. Modificar la navegación generada por defecto es bastante complejo y requiere cambios en el código fuente generado o bien definir patrones propios	Provee varios PSM, algunos para Java que generan para Struts o Spring, otros para PHP que utilizan Smarty
Regiones protegidas de código	Si	Si	Si	Si
Varias implementaciones	J2EE. Puede ampliarse a cualquier otra plataforma	Java2, EJB, Servicios Web, Corba, .NET y gracias a CARAT permite definir nuevos PSMs	Genera tres tipos distintos de PSMs a partir del mismo PIM, pero todas van dirigidos a la misma plataforma J2EE	Existen distintos PSMs para Python, PHP, Java y CSharp

Librería	Plataforma	Lenguaje	Utiliza componentes nativos?	Diseñador	Integración	Limitaciones	Ventajas
AWT	Multiplataforma	Java	Si	No posee	Con Python	Gran acoplamiento. Sólo componentes comunes en todos los SO	Rendimiento
SWT	Multiplataforma	Java	Si, a través de JNI	Si	Java, Python, Ruby, Perl, C	Costo de extensión	Rendimiento
Swing	Multiplataforma	Java	No	Si	Java, Python, Ruby, Perl, C	Menor rendimiento	Fácil de extender, mejora en los componentes
Qt	Multiplataforma	C++	No	Si	Con Ada, Pascal, Perl, PHP, Ruby, Python y Java entre otros	Genera archivos de gran tamaño	Flexible, portable
Tk	Multiplataforma	TCL	Si	Si	Algunos lenguajes utilizan TCL para integrarse con TK: Perl, Python y Ruby	Costo de aprendizaje, es interpretado	Multiplataforma. Fácil de extender
wxWidgets	Multiplataforma	C++	Si	Si (wxGlade)	Provee *binding* con Python, Perl, Java, Lua, Lisp, Erlang, Eiffel, C, Ruby Javascript	No es libre para uso comercial	Gran comunidad, Posee foros, wiki. Rendimiento. Multiplataforma
GTK+	Multiplataforma	C	No	Si	Ruby, Python, PHP, Java, etc.	Escrita en C, se dificulta su programación	Multiplataforma. Gran comunidad.

Apéndice C. Ejemplo de una aplicación pequeña

En el ejemplo, se modeló la clase Persona, con las propiedades *Nombre*, *Documento* y *Id*, esta última será la clave primaria de la persona.

Se creó el grupo *grupoPersonas*, el cual tendrá las vistas y acciones que manipulen personas. Dentro del grupo se crearon dos vistas: *index*, será la vista inicial del grupo; y la vista *vAltaPersona* la cual se utilizará para dar de alta nuevas personas. También se crearon tres acciones dentro del grupo: *index*, la cual será la acción inicial del grupo, la acción *altaPersona* la cual se ejecuta al presionar el botón "Alta persona" que se encuentra en la vista *index* y la acción grabarPersona, la cual se ejecuta al presionar el botón "Grabar persona" que se encuentra en la vista *vAltaPersona*.

La vista *vAltaPersona* posee dos cajas de texto, las cuales se utilizarán para ingresar el nombre y el documento de la nueva persona. A través del botón "Guardar persona", se invocará a la acción "Guardar" de la clase persona, para dar de alta la nueva entidad.

```
project proyectoPersonas {
    modelAssociated model modeloPersonas {
        elements[
            entity Persona{
                properties [
                    property Id {
                        isPrimaryKey true
                        type int
                    }
                    property Nombre {type string}
                    property Documento {type int}
                ]
                actions [
                    action BuscarPorNombre {
                        returnType List<Persona>
                        body{ ... }
                    }
                    action Grabar {
                        returnType void
                        body { ... }
                    }
                ]
            }
        ]
    }

    groups[
        group grupoPersonas{
            isInitial true
            actions [
                action index {
                    isInitial   true
                    returnType  void
                    body { ... }
                }
                action altaPersona {
                    returnType void
                    body {
                        Persona persona;
                        persona = new Persona();
                        setValue(VarPersona, persona);
                        render(vAltaPersona);
                    }
                }
            ]
```

```
            views [
                view index {
                    isInitial true
                    mainContainer container layContainer {
                        layout GridLayout gridLay1 {
                            rows [
                                row row1 {
                                    cells [
                                        cell cell0 {
                                            controls[
                                                button btnAltaPersona{
                                                    properties[ text "Alta persona" ]
                                                    events [ onClick altaPersona() ]
                                                }
                                            ]
                                        }
                                    ]
                                }
                            ]
                        }
                    }
                }

                view vAltaPersona {
                    isInitial false
                    mainContainer container layContainer {
                        layout GridLayout gridLay1 {
                            rows [
                                row row1 {
                                    cells [
                                        cell cell11 {
                                            controls[
                                                label lblNombre{
                                                    properties [
                                                        text    "Nombre y apellido:"
                                                        size    80
                                                    ]
                                                }
                                            ]
                                        }
                                        cell cell12 {
                                            controls[
                                                textBox txtNombre
                                                {
                                                    properties [
                                                        size        80
                                                        maxLength   80
                                                        enabled     true
                                                        datasource  persona nombre
                                                    ]
                                                }
                                            ]
                                        }
                                    ]
                                }
```

```
                                row row2 {
                                    cells [
                                        cell cell21 {
                                            controls [
                                                label lblDocumento  {
                                                    properties [ text "Documento:" ]
                                                }
                                            ]
                                        }
                                        cell cell22 {
                                            controls [
                                                textBox txtDocumento {
                                                    properties [
                                                        size        10
                                                        enabled     true
                                                        datasource  persona documento
                                                    ]
                                                }
                                             ]
                                         }
                                    ]
                                }
                                row row3 {
                                    cells [
                                        cell cell31 {
                                            controls [
                                                button btnGrabarPersona{
                                                    properties[ text "Grabar persona" ]
                                                    events [ onClick grabarPersona() ]
                                                }
                                            ]
                                        }
                                    ]
                                }
                            ]
                        }
                    }
                }
            }
        }
    ]
}
```