UNIVERSIDAD NACIONAL DEL CENTRO DE LA PROVINCIA DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS

DESARROLLO DIRIGIDO POR MODELOS EN UN PROCESO DE REINGENIERÍA: MIGRACIÓN DE APLICACIÓN DE ESCRITORIO A PLATAFORMA MÓVIL

Améndola Federico

Directora
Lic. Favre Liliana

Comisión de Investigaciones Científicas de la Provincia de Buenos Aires

Beca de Estudio 2012

RESUMEN

El presente informe propone una forma de integrar técnicas de ingeniería inversa tradicionales, como el análisis estático y el análisis dinámico, con las ideas detrás de la Arquitectura Dirigida por Modelos (*MDA, Model Driven Architecture*) con el objetivo de constituir un proceso de reingeniería para aplicaciones existentes.
Primero se presentan los conceptos más importantes que respaldan y definen esta integración, haciendo principal énfasis en la interoperabilidad entre plataformas. Además, se describen y comparan algunas de las herramientas que hacen posible la automatización de partes importantes de este proceso.
Por último, se presenta un caso de estudio perteneciente a un contexto particular, que permitirá ver la aplicación de los conceptos y las técnicas presentadas.

CONTENIDO

LISTA DE FIGURAS

LISTA DE CÓDIGOS

INTRODUCCIÓN

La necesidad de mantenimiento, reuso y reingeniería de sistemas de software existentes se ha incrementado dramáticamente en los últimos años. Requerimientos que cambian o la necesidad de migrar un sistema para adaptarse a las nuevas tecnologías han planteado nuevos objetivos dentro de la ingeniería de software.
Reutilizar, adaptar o modificar un sistema existente puede constituir un proceso complejo y costoso debido, principalmente, al prolongado trabajo que requiere la comprensión del sistema en sí que se está analizando. Aplicaciones cada vez más extensas y complejas, con una documentación quizás poco adecuada (o aún inexistente), plantean las principales dificultades de este proceso de comprensión.
Contar con métodos de ingeniería de software y herramientas que faciliten el entendimiento de un programa se ha establecido como una necesidad fundamental para asegurar el éxito del nuevo software [Sys00]. Debido a esto es que surge la **ingeniería inversa** como una disciplina que apunta a soportar la comprensión de un programa, explotando el código fuente como principal recurso de información acerca de la organización y el comportamiento de un programa, y extrayendo un conjunto de vistas potencialmente útiles que se presentan en forma de diagramas [TP05].
Para recolectar esta información es que se dispone de distintas técnicas comprendidas en dos tipos principales de análisis: el **análisis estructural o estático** y el **análisis de comportamiento o dinámico**. A partir de ambos análisis se construirán los artefactos de software necesarios para describir al sistema que se está analizando. En este punto es donde se debe considerar la dependencia que tienen los artefactos obtenidos con las tecnologías utilizadas para implementar el sistema bajo análisis. Esta dependencia no debería impactar en los artefactos que se construirán para describir el nuevo sistema a implementar. Para evitar estas situaciones es que se propone la integración de las técnicas de ingeniería inversa con el estándar de **Arquitectura Dirigida por Modelos** [OMG13c] propuesto por el Grupo de Gestión de Objetos (*OMG, Object Management Group*).
MDA es una realización específica del Desarrollo Dirigido por Modelos (*MDD, Model Driven Development*) que apunta a la interoperabilidad entre plataformas y a la independencia de las tecnologías, proponiendo que todos los artefactos involucrados en un proceso de desarrollo se representen a partir de un lenguaje común de metamodelado llamado MOF (*Meta-Object Facility*) [OMG13d]. MOF define una forma común de capturar todas las construcciones de modelado e intercambio que son usadas, y es la esencia de MDA al permitir que diferentes tipos de artefactos de software sean usados juntos en un mismo proyecto. A estos modelos, obtenidos del sistema bajo análisis, se les puede aplicar una de las operaciones más importantes dentro del enfoque de MDD: **la transformación entre modelos**. Estas transformaciones permiten

obtener modelos equivalentes para representar el nuevo sistema que se desea implementar. Hay distintas formas de llevar a cabo las transformaciones, por ejemplo, utilizando un lenguaje de programación de propósito general. En este proyecto se utilizará, en cambio, un lenguaje de transformación entre modelos especializado denominado Lenguaje de Transformación ATLAS (*ATL, ATLAS Transformation Language*) [Fou13a].

El proceso de reingeniería que se pretende describir se puede presentar mediante la siguiente figura:

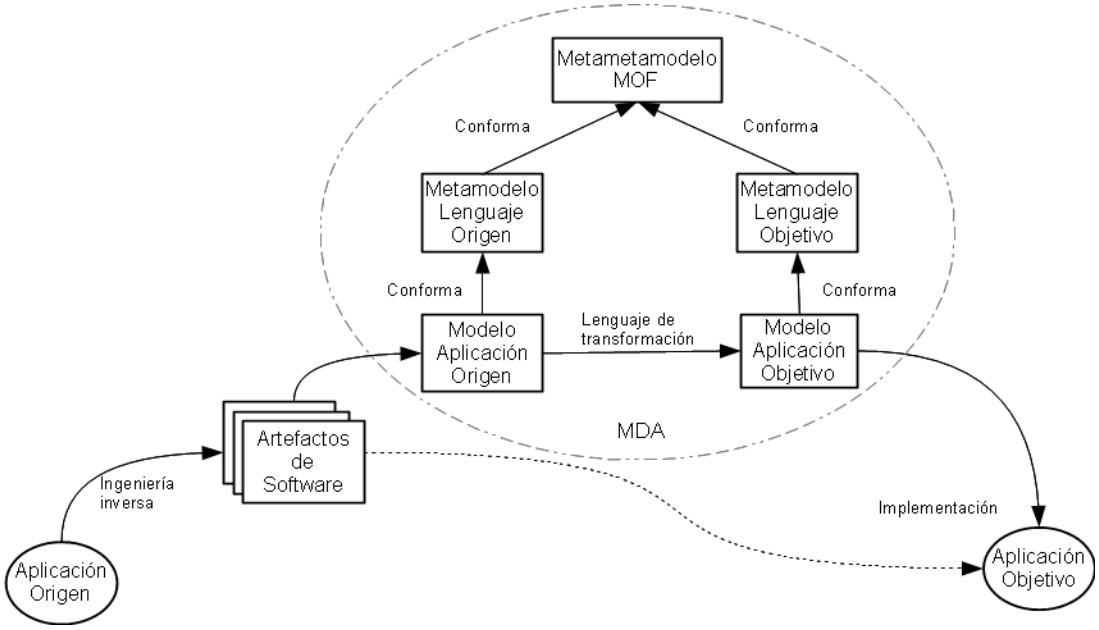

Figura 1: **Proceso de reingeniería.**

Para asegurar el éxito de las tareas que involucran la ingeniería inversa y el desarrollo dirigido por modelos, en cada una de las etapas descritas en la figura anterior, es necesario contar con herramientas que asistan y automaticen partes del proceso. Estas herramientas se denominan **Ingeniería de Software asistida por computadoras** (*CASE, Computer Aided Software Engineering*) y cada una presenta características y soporte distintos para las técnicas involucradas en el proceso de reingeniería.

El presente informe se divide en dos partes principales. En la primera parte se describen los conceptos teóricos involucrados en el proceso de reingeniería propuesto y se realiza una comparación de las herramientas CASE más importantes; mientras que en la segunda parte se presenta un caso de estudio que ejemplifica dicho proceso.

CONCEPTOS TEÓRICOS

Los conceptos teóricos que se presentan a continuación se centran principalmente en la ingeniería inversa, presentando una descripción general y un detalle en la evolución de las técnicas que la componen.
En una segunda parte, se describen las características más importantes de la Arquitectura Dirigida por Modelos, junto con los conceptos que hacen posible su vinculación con la ingeniería inversa en el proceso de desarrollo propuesto.

INGENIERÍA INVERSA

La primera etapa en un proceso de reingeniería es comprender el sistema bajo análisis y recuperar el diseño del mismo; una tarea compleja debido, principalmente, a la falta de una documentación adecuada que describa el sistema en sí. Ésta situación, sumado quizás a la imposibilidad de consultar a los desarrolladores del sistema lleva a que, generalmente, la única fuente de documentación confiable es el código fuente.
La ingeniería inversa surge como un enfoque para ayudar en la comprensión de un sistema, intentando minimizar las dificultades mencionadas anteriormente. Estas dificultades también impactan negativamente en el proceso de ingeniería inversa, con lo cual es fundamental contar con herramientas que asistan y automaticen dicho proceso, como se verá en las secciones siguientes.

En busca de una definición formal se han planteado distintos enfoques. Una definición más práctica define a la ingeniería inversa como el proceso de analizar artefactos de software disponibles como los requerimientos, el diseño, la arquitectura, el código fuente o binario, con el objetivo de extraer información y proveer vistas de alto nivel del sistema bajo estudio [FMP11].

Los objetivos claves que debe perseguir la ingeniería inversa, asistida por las mejoras tecnológicas, se pueden resumir en:

1. **Lidiar con la complejidad**: Se necesitan métodos para tratar con la complejidad de los sistemas. La integración de métodos y herramientas de ingeniería inversa con ambientes CASE provee una forma de extraer información relevante.
2. **Generar vistas alternativas**: Las representaciones gráficas han sido muy bien aceptadas para ayudar a comprender un sistema. Las herramientas de ingeniería inversa pueden generar vistas adicionales a partir de distintas perspectivas (diagramas de flujo de datos, flujo de control, gráficos de estructuras, diagramas de entidad-relación, entre otros).
3. **Recuperar información perdida**: La evolución continua de grandes sistemas de software frecuentemente lleva a modificaciones no reflejadas en la documentación de diseño. Sin ser un substituto para preservar el historial de diseño, la ingeniería inversa puede ayudar a recuperar la información sobre un sistema existente.
4. **Detectar efectos secundarios**: Las continuas modificaciones a un sistema pueden llevar a ramificaciones no intencionales o efectos secundarios que atentan contra la performance de un sistema. La ingeniería inversa puede ayudar a detectar estos problemas y a proveer una perspectiva distinta a la conseguida con la ingeniería directa.
5. **Sintetizar abstracciones de mayor nivel**: La ingeniería inversa requiere métodos y técnicas para crear vistas alternativas que impulsen niveles de abstracción mayores. La tecnología de sistemas expertos juega un rol fundamental en este objetivo.
6. **Facilitar el reuso**: La ingeniería inversa puede ayudar a detectar componentes de software, en los sistemas presentes, candidatos a ser reutilizados en un futuro.

Tomando en cuenta estos objetivos se puede plantear una definición más general: la ingeniería inversa incluye cualquier método pensado para recuperar conocimiento acerca de un sistema de software existente, con el objetivo de asistir en la ejecución de una tarea de ingeniería de software. Recuperar conocimiento significa que, en general, múltiples vistas y perspectivas de un sistema de software dado son posibles y potencialmente se pueden completar entre sí [TTBS07].
Todas las definiciones coinciden en que la ingeniería inversa es un proceso de examen no un proceso de cambio y, por lo tanto, no involucra cambiar el software que se está examinando; a diferencia de la reingeniería, definida como el examen y alteración de un sistema para reconstituirlo en una nueva forma y su subsecuente implementación.

Una arquitectura básica para describir las herramientas de ingeniería inversa se puede ver en la siguiente figura, propuesta por Chikofsky y Cross [CC90]:

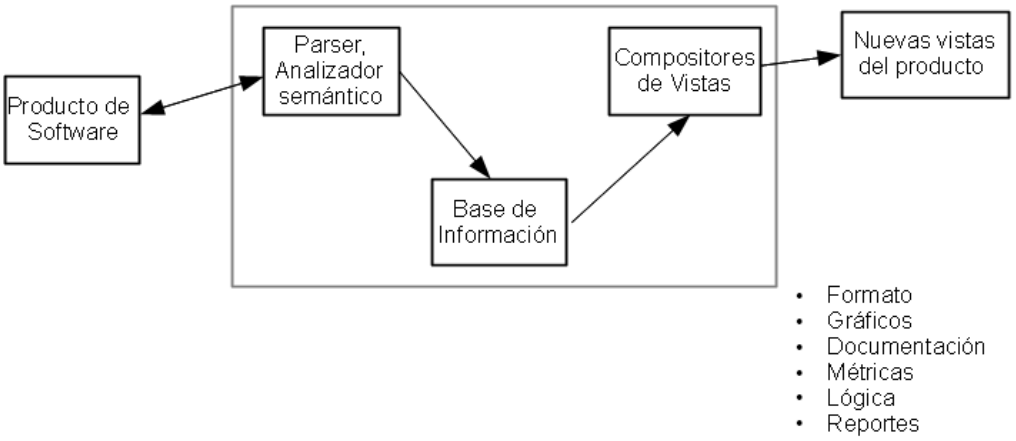

Figura 2: **Arquitectura de las herramientas de ingeniería inversa.**

El producto de software, bajo estudio, es analizado y los resultados son almacenados en una base de información. Cuando se diseña una base, es deseable hacerlo usando un esquema común compartido por diferentes herramientas. Los esfuerzos en esta dirección produjeron esquemas como el lenguaje de intercambio de grafos (*GXL, Graph eXchanged Language*) [HSSW13], el formato estándar de Rigi (*RSF, Rigi Standard Format*) [Mul13], el metamodelo FAMIX [tec13] utilizado por Moose o el metamodelo de descubrimiento de conocimiento (*KDM, Knowledge Discovery Metamodel*) [OMG13b], recientemente propuesto por la OMG.
Los datos de la base de información luego son utilizados por los compositores de vistas para producir vistas alternativas del producto de software. De acuerdo a las técnicas de ingeniería inversa utilizadas se pueden distinguir algunos artefactos: Documentación de usuario, requerimientos, modelos de diseño y código fuente, a partir del análisis estático; y trazas de ejecución y registros de utilización de memoria, si se utiliza el análisis dinámico; por mencionar dos de las principales técnicas.
El rol de la ingeniería inversa, en relación con estos artefactos, es:

- Reconstruir artefactos no disponibles o que no reflejan el sistema bajo estudio.
- Construir otras vistas de alto nivel para proveer información necesaria para entender el sistema.
- Proveer la información necesaria para realizar tareas de cambios en el software en un escenario de reingeniería dirigida por modelos.

Como se mencionó, muchos de estos artefactos pueden no estar disponibles y, en gran parte de los casos, solamente se contará con el código fuente o aún peor, los binarios de un sistema como documentación del mismo. Para recuperar la información necesaria que permita construir los artefactos deseados y, por ende, entender el sistema de software bajo estudio, la ingeniería inversa ha evolucionado incorporando distintos tipos de técnicas para aplicar durante el análisis.

Evolución de las técnicas de ingeniería inversa

Inicialmente la ingeniería inversa estaba centrada en identificar y modelar la estructura de un programa examinando su código fuente. Esto se debía a la naturaleza de los sistemas bajo investigación y a los objetivos establecidos para las actividades de ingeniería inversa. Por un lado, los sistemas de software que tradicionalmente habían sido objeto de estudio, en su mayoría, estaban escritos en lenguajes procedurales y se ejecutaban en una sola computadora. Por otro lado, los objetivos de mantenimiento, en donde se aplicaba la ingeniería inversa, habían sido la adaptación de un sistema legacy para extender su funcionalidad o para realizar una migración a una plataforma más moderna [SS02].
En este contexto, las primeras técnicas de ingeniería inversa estaban basadas, básicamente, en el análisis del programa y en el concepto de una interpretación abstracta, la cual representa los cálculos de un programa utilizando las descripciones de los valores (o valores abstractos) en lugar de los valores realmente calculados. La interpretación abstracta permite obtener información acerca del comportamiento de un programa en tiempo de ejecución sin tener que ejecutar los programas con todas las variantes de datos de entrada existentes.
Además de la interpretación abstracta, se desarrollaron técnicas basadas en la teoría de compiladores, donde se partía del código fuente y la gramática del lenguaje de programación fuente, y mediante una serie de etapas se conseguían distintos tipos de información, de acuerdo al análisis realizado. A este análisis se lo denominó **análisis estático** [Fav10].

Tanto la naturaleza de los sistemas bajo investigación como los objetivos de mantenimiento han evolucionado. Una gran parte de los sistemas legacy actuales están desarrollados con lenguajes orientados a objetivos y a menudo sus componentes están distribuidos en arquitecturas multicapas. Además, uno de los objetivos de mantenimiento que más ha crecido es la migración de sistemas legacy a ambientes distribuidos para interoperar con otros sistemas. Este nuevo objetivo tiene gran impacto en los tipos de modelos que se espera que resulten de la ingeniería inversa. En vez de identificar las interdependencias entre las estructuras de datos originales y los procedimientos de un sistema, los resultados esperados son especificaciones de los subsistemas funcionales describiendo comportamientos de alto nivel que puedan ser integrados para interoperar en un sistema [SS02].
Este nuevo contexto, donde la demanda de procesos de reingeniería para sistemas orientados a objetos fue en aumento, planteó la necesidad de desarrollar nuevos enfoques para, por ejemplo, identificar objetos en código legacy y traducir este código a lenguajes orientados a objetos.
Las técnicas basadas en la teoría de compiladores fueron adaptadas para representar la propagación de datos entre objetos durante la ejecución y así poder seguir el ciclo de vida de los objetos desde su creación.

Los programas orientados a objetos son esencialmente dinámicos y presentan problemas particulares relacionados con el polimorfismo y el binding dinámico, las clases abstractas y los lenguajes tipados dinámicamente. Por ejemplo, algunos lenguajes orientados a objetos introducen conceptos como la reflexión y la posibilidad de cargar en memoria clases dinámicamente, aunque estos mecanismos son muy útiles afectan negativamente las técnicas de ingeniería inversa.
Características tales como la redefinición y resolución dinámica de métodos y la asociación dinámica entre clases requieren capturar estados del sistema durante el tiempo de vida de los objetos, para esto se necesita un nuevo tipo de análisis, que se denominó **análisis dinámico**.
A partir de la programación orientada a objetos, el centro del análisis de software pasó del análisis estático al dinámico, más precisamente el análisis estático fue complementado por el dinámico.

Análisis estático y análisis dinámico

Para entender los sistemas de software existentes se necesita la información estática y la información dinámica. La información estática describe la estructura del software como fue escrita en código fuente, mientras que la información dinámica describe el comportamiento en tiempo de ejecución. En el marco de la ingeniería reversa, esta información se utiliza para producir modelos de diseño del software bajo análisis. Este enfoque también es útil cuando se construye software a partir de información de diseño de alto nivel, por ejemplo, durante la ingeniería directa. Los modelos estáticos extraídos pueden ser usados para asegurar que las guías de arquitectura son seguidas y para obtener una visión global del estado actual del software. Los modelos dinámicos se pueden usar para tareas de depuración, encontrar código muerto y entender el comportamiento actual del software [Sysoo].
Con el objetivo de recuperar estas dos clases de información, acerca de los artefactos de software y sus relaciones, es que se utilizan diversas técnicas dentro del análisis estático y del análisis dinámico.

Análisis estático

El análisis estático examina el código fuente e infiere todos los posibles comportamientos que pueden surgir durante la ejecución del programa. Este tipo de análisis, generalmente, opera construyendo un modelo abstracto del estado del programa, que pierde algo de información, pero es compacto y fácil de manipular; luego determina cómo el programa reacciona a ese estado. Debido a que existen muchas ejecuciones posibles, el análisis debe considerar múltiples diferentes estados.
Se puede definir al análisis estático como un análisis *conservador* y *confiable*. La confiabilidad garantiza que los resultados serán una descripción precisa del comportamiento del programa, sin importar los datos de entrada o el ambiente en

donde se ejecute. La característica de ser un análisis conservador se debe a que puede detectar propiedades más débiles que las reales; se garantiza que las propiedades débiles serán verdaderas, preservando la confiabilidad, pero pueden no ser lo suficientemente fuertes para ser útiles. Por ejemplo, dada una función `f`, un análisis conservador puede reportar que `f` devuelve un valor no negativo, lo cual es más débil (pero fácil de establecer) que detectar que `f` devuelve el valor absoluto de su argumento. Para mantener la confiabilidad, el análisis debe producir un resultado verdadero sin importar el valor de los componentes del estado abstraído, esto quizás lleve a resultados menos precisos y más aproximados y conservadores [Ern03].
Las optimizaciones del compilador y el análisis de flujo de datos son ejemplos de técnicas de análisis estático. La información estática, recuperada mediante el análisis, incluye artefactos de software y sus relaciones: clases, interfaces, métodos, variables, invocación entre métodos, relaciones entre clases, chequeo de sintaxis, chequeo de tipos y análisis de flujo de datos y de control.

Análisis dinámico

El análisis dinámico opera ejecutando el programa y observado las ejecuciones resultantes. Es un análisis *preciso* debido a que no se necesita construir una aproximación o una abstracción: el análisis puede examinar el comportamiento exacto y actual de un programa, en tiempo de ejecución. Hay poca incertidumbre en qué camino del flujo de control fue tomado, qué valores computados, cuánta memoria consumida, cuánto tardó el programa en ejecutar u otras métricas de interés. El análisis dinámico puede ser tan rápido como la ejecución del programa en sí, a diferencia del análisis estático donde, generalmente, obtener resultados exactos implica un gran número de cómputos y largas esperas.
La desventaja del análisis dinámico es que sus resultados no se pueden generalizar a ejecuciones futuras, es decir, no hay garantía que el caso de prueba (esto es, el conjunto de datos de entrada) para el cual se ejecutó el programa sea característico de todas las posibles ejecuciones del mismo.
Mientras que el desafío principal al construir un análisis estático es la elección de una buena función de abstracción, el desafío principal al realizar un buen análisis dinámico es seleccionar un conjunto de caso de pruebas representativos. Una buena selección de este conjunto puede revelar propiedades del programa o de su contexto de ejecución; pero si el conjunto no es representativo, el análisis dinámico indicará propiedades del caso de prueba en sí, haciendo difícil reconocer si una propiedad particular es un artefacto del caso elegido o una propiedad verdadera del programa.
Las técnicas de testing, debug y profiling componen generalmente un análisis dinámico estándar.
La información dinámica también incluye artefactos de software: información secuencial de traza de eventos, información sobre el comportamiento actual, administración de memoria y cobertura de código. Extraer información dinámica es especialmente importante cuando se examina software orientado a objetos. Esto es por

la naturaleza dinámica de los programas orientados a objetos: creación de objetos, eliminación de objetos/garbage collector y el binding dinámico hacen muy difícil, y muchas veces imposible, entender el comportamiento de un programa sólo examinando su código fuente.

Sinergia de los análisis

De acuerdo a las características descritas anteriormente se puede concluir que ambos análisis son complementarios. El análisis estático es conservador y confiable: los resultados pueden ser más débiles de lo deseado, pero está garantizado que se pueden generalizar a futuras ejecuciones. El análisis dinámico es eficiente y preciso: no requiere análisis costosos, pero si requiere la selección de los casos de prueba, y proporciona resultados altamente detallados respecto a estos casos de prueba. Estos aspectos permiten aplicar ambos análisis a un mismo problema produciendo resultados que son útiles en diferentes contextos. La observación clave es que los dos análisis pueden considerar directamente sólo un subconjunto de las ejecuciones del programa. La generalización a partir de esas ejecuciones es la fuente de la pérdida de confianza en el análisis dinámico y de la imprecisión en el análisis estático.
El conjunto de ejecuciones consideradas por el análisis dinámico son exactamente aquellas que aparecen en el caso de prueba o que fueron observadas durante la ejecución. Este conjunto es fácil de enumerar y puede caracterizar un ambiente particular; sin embargo, el conjunto puede ser difícil de formalizar en notación matemática [Ern03].

Nuevos desafíos

Cuando surgió el Lenguaje Unificado de Modelado (*UML, Unified Modeling Language*) [OMG13h], también apareció un nuevo problema: cómo extraer vistas de alto nivel de un sistema para representarlas mediante diferentes clases de diagramas UML. En este sentido, los diagramas que podían ser obtenidos mediante las técnicas de ingeniería inversa eran parciales. Un nuevo desafío fue cómo identificar diferentes relaciones, por ejemplo, dependencia, asociación, agregación y composición. Además, si bien existen muchos avances para extraer diagramas UML a partir del código fuente (diagramas de clase, de estado y de secuencia, por mencionar algunos), todavía es necesario realizar mejoras. Por ejemplo, es un problema muy común la necesidad de extraer diagramas dinámicos y la integración de especificaciones en el Lenguaje de Restricciones de Objetos (*OCL, Object Constraint Language*) [OMG13f]. Por otra parte, existen herramientas para asistir en las diferentes dimensiones de la ingeniería inversa (particionar, refactorizar, diseñar con patrones y realizar casos de prueba) pero, en general, no están integradas entre sí debido a que el software evoluciona constantemente.

Actualmente, tanto la industria de la ingeniería de software como los sistemas han evolucionado para tratar con nuevas tecnologías de plataformas, técnicas de diseño y procesos. Un nuevo framework técnico para integración de información e interoperabilidad entre herramientas, como el Desarrollo Dirigido por Modelos (*MDD, Model Driven Development*), ha creado la necesidad de desarrollar nuevas herramientas de análisis y técnicas específicas. MDD se refiere a un conjunto de enfoques de desarrollo que se basan en el uso de modelos de software como entidades de primera clase. El enfoque más conocido es el estándar de la OMG, la Arquitectura Dirigida por Modelos (*MDA, Model Driven Architecture*) [OMG13c], el cual es una realización del Desarrollo Dirigido por Modelos.
MDA es una evolución de los estándares de la OMG [OMG13e] para soportar el desarrollo centrado en los modelos, con el objetivo de incrementar el grado de automatización de procesos tales como traducción de código fuente, ingeniería inversa, ingeniería directa y la reingeniería de datos. El alcance de MDA no está restringido a sistemas de software y muchas clases de dominios, desde ingeniería de sistemas, negocios y manufacturación pueden beneficiarse de los conceptos detrás de MDA [Fav10].

ARQUITECTURA DIRIGIDA POR MODELOS (MDA)

Las ideas principales detrás de la arquitectura dirigida por modelos son:

- separar la especificación de la funcionalidad de un sistema de su implementación en una plataforma específica,
- administrar la evolución de un software desde modelos abstractos a implementaciones (incrementando el grado de automatización), y
- obtener interoperabilidad con múltiples plataformas, lenguajes de programación y lenguajes formales [FMP11].

Estas ideas, al compararlas con los objetivos que persigue la ingeniería inversa, permiten utilizar la Arquitectura dirigida por Modelos como contexto de aplicación de las distintas técnicas de la ingeniería inversa.
Como ya se mencionó, MDA incrementa el rol de los modelos en el proceso de desarrollo. Estos modelos se utilizan para diseñar, modificar y mantener los sistemas de software, en distintos niveles de abstracción.
Se pueden distinguir los siguientes modelos principalmente:

- Modelo del dominio (*CIM, Computation Independent Model*): un modelo que describe un sistema desde un punto de vista independiente del sistema de software a construir. Se centra en el ambiente y en los requerimientos del sistema.
- Modelo independiente de la plataforma (*PIM, Platform Independent Model*): un modelo con un alto nivel de abstracción, independiente de cualquier tecnología de implementación. Ejemplos de diagramas útiles para representar estos modelos son los diagramas de casos de uso, de actividades y de estados.

- Modelo específico de la plataforma (*PSM, Platform Specific Model*): un modelo que describe el sistema en términos de las construcciones de implementación disponibles para una plataforma específica. Para este tipo de modelo, son útiles los diagramas de clases, de objetos, de interacción y de paquetes.
- Modelo específico de la implementación (*ISM, Implementation Specific Model*): una descripción del sistema en código fuente [Bro04].

MDA es implementada como una secuencia de transformaciones entre modelos, a continuación se distinguen las principales transformaciones.
Un **refinamiento** es el proceso de construcción de una especificación más detallada que conforma a otra más abstracta. Por otro lado, un **anti-refinamiento** es el proceso de extraer de una especificación más detallada (o código) otra especificación más abstracta que es conformada por la especificación más detallada. **Refactorizar** significa cambiar un modelo sin modificar su comportamiento pero mejorando algún atributo de calidad no funcional, como por ejemplo, simplicidad, flexibilidad, facilidad de comprensión y performance.
Una de las características esenciales de MDA es que todos los artefactos involucrados en un proceso de desarrollo se representan a partir de un lenguaje de metamodelado común, el lenguaje MOF (Meta-Object Facility) [OMG13d]. MOF define una forma común de capturar todas las construcciones de modelado e intercambio que son usadas, y es la esencia de MDA al permitir que diferentes tipos de artefactos de software sean usados juntos en un mismo proyecto. Además del estándar MOF, para definir metamodelos, MDA está asociado con otros dos estándares de la OMG para expresar transformaciones: el Lenguaje Unificado de Modelado UML [OMG13h] y el metamodelo de Consultas, Vistas y Transformaciones (*QVT, Query View Transformation Metamodel*) [OMG13g].
MOF utiliza un framework de modelamiento de objetos que es, esencialmente, un subconjunto del núcleo de UML. Los cuatro conceptos de modelamiento principales son las clases, las asociaciones, los tipos de datos y los paquetes.

La difusión inicial de MDA enfatizaba su relación con UML como lenguaje de modelado. Sin embargo, hay usuarios de UML que no usan MDA y usuarios de MDA que utilizan otros lenguajes de modelado específicos (*DSL, Domain Specific Language*). El modelamiento específico del dominio (*DSM, Domain Specific Modeling*) es una forma de diseñar y desarrollar sistemas en un nivel de abstracción más alto, enfocándose de cerca en el dominio del problema, y usualmente aplicado junto a la programación generativa. Utilizando lenguajes DSM muy expresivos se puede mejorar la calidad del código y aumentar la productividad debido a la generación automática de código [ALC06].
En [DF08] se plantea una comparación interesante entre los conceptos de MDA y DSM. En dicha comparación se describen las características más importantes y los puntos débiles de cada enfoque. Además, se menciona que actualmente MDA corre con ventaja por sobre DSM debido, principalmente, a la existencia de un gran número de herramientas que asisten a los usuarios en distintas etapas del proceso de desarrollo.

En este mismo sentido, se puede asegurar que el éxito de MDA depende de la existencia de herramientas CASE que ayuden en la automatización de procesos de reingeniería, generando código fuente a partir de modelos (ingeniería directa) y modelos desde código fuente (ingeniería inversa).
Por ejemplo, la computación integrada al modelo (*MIC, Model-Integrated Computing*) es una técnica que convierte modelos específicos del dominio en código ejecutable. MIC provee un framework para la producción de software utilizando ambientes de metamodelado e intérpretes de modelos. También soporta la creación flexible de ambientes de modelamiento y ayuda a seguir los cambios en los modelos.
En la próxima sección se realizará una comparación de las herramientas CASE más utilizadas dentro de este contexto.

Transformación entre modelos

La transformación entre modelos es una de las operaciones fundamentales dentro del enfoque de MDD. Esto se debe a que permiten, por un lado, conseguir una representación de todo un sistema al combinar sus diferentes modelos; por otro lado, organizar los modelos del sistema en distintos niveles de abstracción; y, por último, derivar el código fuente del sistema mediante sucesivas transformaciones entre modelos, desde la representación más abstracta, pasando por la capa intermedia de modelos. Para esto, las transformaciones construyen nuevos modelos a partir de un modelo de entrada. El modelo resultante puede ser una representación en una plataforma distinta o describir, en un nuevo nivel de abstracción, las características del modelo original.
Con el objetivo de reducir el esfuerzo y evitar la generación de errores, a la hora de realizar la transformación entre modelos, es que se ha intentado automatizar el proceso de transformación en la medida de lo posible. Esta automatización ha llevado a definir distintas formas de llevar a cabo las transformaciones, por ejemplo, mediante lenguajes de propósito general, representaciones intermedias o lenguajes específicos de transformación [SK03].
Se supone que utilizando un lenguaje de propósito general los desarrolladores no necesitarán demasiado entrenamiento para escribir las transformaciones. Sin embargo, las interfaces de programación de estos lenguajes generalmente restringen el tipo de transformación que se puede realizar. Además, debido a que el lenguaje es de propósito general, no se contará con el suficiente nivel de abstracción para especificar las transformaciones, haciendo que estas transformaciones sean difíciles de escribir, mantener y comprender. Ejemplos de herramientas para esta forma de realizar las transformaciones son Rational Rose, que provee una versión de Visual Basic, y Rational XDE, que propone una interfaz de programación que puede ser usada desde Java, Visual Basic o C#.
El enfoque de representaciones intermedias plantea exportar los modelos en un formato estándar, por ejemplo XML, para que una herramienta externa pueda realizar la

conversión entre modelos. Esta alternativa puede requerir experiencia y un esfuerzo considerable para definir un modelo simple de transformaciones. Además, las transformaciones generalmente son costosas de realizar por lo que no pueden realizarse de forma interactiva con el usuario. Uno de estos formatos intermedios es el estándar de Intercambio de Metadata XML (*XMI, XML Metadata Interchange*) [OMG13i] propuesto por la OMG. XMI fue pensado como un estándar para intercambiar información acerca de metadata (es decir, información relacionada con un conjunto de datos, su consistencia y organización). Este estándar integra los estándares XML, UML y MOF.

La última alternativa propone utilizar un lenguaje específico de transformación que permita especificar cómo generar un modelo objetivo, que conforma un metamodelo objetivo, a partir de un modelo origen que conforma un metamodelo origen. Esta alternativa, en comparación con las dos alternativas ya mencionadas, ofrece el mayor potencial ya que el lenguaje puede ser especializado para este propósito. Dentro de estos lenguajes se pueden mencionar QVT y ATL.

Como su nombre lo indica, el estándar QVT (*Query, View, Transformation*) propuesto por la OMG, abarca las transformaciones, las vistas y las consultas. Este estándar define tres lenguajes de transformación de modelos (QVT-Operacional, QVT-Relacional y QVT-Núcleo) que conforman los metamodelos MOF 2.0. Una transformación en cualquiera de estos lenguajes puede considerarse como un modelo que conforma a uno de los metamodelos especificados en el estándar. El estándar QVT integra el estándar OCL 2.0 y lo extiende con características de lenguaje imperativo.

El lenguaje ATL está inspirado en los requerimientos del estándar QVT y construido sobre el formalismo del estándar OCL. Este lenguaje será utilizado en el caso de estudio presentado en las secciones siguientes, razón por la cual, a continuación, se realiza una descripción resumida de sus características principales.

El lenguaje de transformación ATLAS (ATL)

ATL es un lenguaje híbrido que permite especificar transformaciones entre modelos de forma declarativa e imperativa. Este lenguaje es descrito mediante una sintaxis abstracta (un metamodelo MOF), y una sintaxis textual concreta.

El estilo declarativo para la especificación de transformaciones tiene una serie de ventajas. Por lo general, se basa en la especificación de relaciones entre patrones de fuente y destino y, por lo tanto, tiende a estar más cercano a la manera en que los desarrolladores intuitivamente perciben una transformación. Este estilo hace hincapié en codificar las relaciones y ocultar los detalles vinculados con la selección de los elementos de origen, la ejecución y el orden de las reglas, la trazabilidad, etc. Gracias a esto se pueden esconder, detrás de una sintaxis simple, complejos algoritmos de transformación. Sin embargo, a veces es difícil proporcionar una solución declarativa completa para un determinado problema de transformación. En ese caso, los

desarrolladores pueden recurrir a las características imperativas del lenguaje [JABK08].

Una transformación en ATL se puede especificar mediante los módulos del lenguaje. Un módulo ATL está compuesto de los siguientes elementos:

- una sección de encabezado, que define los nombres del módulo de transformación y las variables de los metamodelos de origen y destino.
- una sección opcional de importación, que permite utilizar bibliotecas ATL existentes.
- un conjunto de helpers, que pueden ser utilizados para definir variables y funciones.
- un conjunto de reglas, que definen cómo los elementos del modelo de origen son asociados y visitados para crear e inicializar los elementos de los modelos de destino. Los modelos de origen y destino conforman un metamodelo de origen y de destino, respectivamente.

El término ***Helper*** proviene de la especificación OCL que define dos tipos de *helpers*: uno para atributos y el otro para operaciones. Un *helper* sólo puede especificarse para un tipo OCL o un tipo del metamodelo de origen, ya que los modelos de destino no son accesibles.
Los *helpers* para operaciones se definen en el contexto de un elemento del modelo o de un módulo. Mientras que los *helpers* para atributos son usados para asociar valores sólo lectura a elementos del modelo de origen. Estos *helpers* pueden ser considerados como una forma de editar los modelos de origen antes de la ejecución de la transformación.

La **regla de transformación** es la construcción básica para expresar la lógica de la transformación. Las reglas ATL pueden ser especificadas de forma declarativa o imperativa.
Las reglas declarativas se llaman reglas *matched*. Están compuestas de un patrón de origen y un patrón de destino. El patrón de origen especifica un conjunto de tipos de origen (a partir de los tipos OCL y los metamodelos de origen) y una expresión OCL booleana (denominada *guard*). El patrón de destino está compuesto de un conjunto de elementos. Cada elemento especifica un tipo de destino (a partir del metamodelo de destino) y un conjunto de *bindings*. Un *binding* se vincula con una característica de un tipo de destino (un atributo, una referencia o un extremo de asociación) y especifica una expresión de inicialización para el valor de esa característica.
Las reglas *matched* son ejecutadas cuando un elemento del metamodelo de origen coincide con el patrón de origen. En ese caso, los elementos de destino de los tipos especificados son creados en el modelo de destino y sus características inicializadas utilizando los *bindings*.
Algunas veces puede ser demasiado difícil establecer una solución puramente declarativa para algoritmos de transformación complejos. Para estos casos, ATL provee un estilo imperativo basado en dos construcciones principales:

- las reglas *called*, las cuales son básicamente procedimientos que se invocan a partir de su nombre.
- los bloques de acción, que se definen como secuencias de sentencias imperativas que pueden utilizarse en los patrones de destino de las reglas *called* o *matches*. Las sentencias imperativas disponibles son las construcciones clásicas para especificar flujo de control como condiciones, ciclos, asignaciones, por mencionar algunas.

El algoritmo de ejecución de las transformaciones ATL comienzo al llamar a una regla *called* opcional definida como punto de entrada. Esta regla puede invocar otras reglas *called*. Luego el algoritmo ejecuta las reglas *matched* estándar (las cuales pueden contener bloques de acción).

Además del lenguaje, ATL define un conjunto de herramientas construidas sobre la plataforma Eclipse. Las Herramientas de Desarrollo de ATLAS (*ADT, ATLAS Development Tools*) están compuestas de un motor de transformación y de un ambiente de desarrollo integrado (editor, compilador y depurador). El motor de transformación es responsable de compilar y ejecutar las transformaciones. Estas transformaciones son compiladas a un código de máquina virtual específica, orientada a la transformación y manipulación de modelos.

HERRAMIENTAS CASE

Contar con herramientas que asistan durante el proceso de reingeniería propuesto es fundamental para asegurar su exitosa aplicación. Las tareas que involucran los procesos de ingeniería (directa e inversa) y las etapas de modelamiento y traducción entre modelos de MDA, requieren de herramientas que automaticen y simplifiquen su ejecución.
En los últimos años, un gran número de herramientas en el contexto de ingeniería inversa y recuperación de diseño han sido desarrolladas para uso industrial e investigación académica. Estas herramientas proveen y combinan funcionalidades como:

- permitir un entendimiento del software mediante la construcción de modelos de alto nivel de la estructura y/o el comportamiento del software.
- analizar las propiedades del software mediante métricas.
- brindar facilidades para ingeniería directa e inversa.

Si bien la mayoría de las herramientas CASE dan soporte para la ingeniería inversa, existen dos dificultades principales. Por un lado, la mayoría de estas herramientas pueden obtener diagramas de clase, pero hay una falta de soporte considerable para extraer otro tipo de diagramas. Por otro lado, sólo usan características de notación básicas, con una representación de código directa, y producen diagramas de gran tamaño. Los procesos de ingeniería inversa se logran agregando anotaciones en el código generado, las cuales funcionan como vínculos entre los elementos del modelo y el lenguaje.
En [ALC06] se presentan las siguientes tablas que resumen los aspectos más importantes de las principales herramientas que pueden asistir durante un proceso de reingeniería completo.

Como se puede ver la comparación se establece para dos tipos de herramientas: las herramientas que dan soporte para ingeniería inversa y las herramientas CASE.

HERRAMIENTAS PARA INGENIERÍA INVERSA

La entrada de muchas herramientas de ingeniería inversa es un archivo o un Árbol de Sintaxis Abstracta (*AST, Abstract Syntax Tree*), creado por otras herramientas como *parsers* y extractores de hechos. Estas herramientas no se encargan de extraer el código fuente sino que se centran en hacer una abstracción y visualización de las entradas provistas.
Columbus [FrBTG02] es un framework que contiene un analizador de código C/C++, que puede extraer diagramas UML, AST y grafos de llamados. También provee

Herramienta	Datos de entrada	Técnica	Construcción de diagramas	Reconocimiento de patrones
Columbus	C++	Matching entre grafos	Clases UML	+
CPP2XMI	CPPML, XMI	Análisis dinámico	Secuencia, Actividad y Clases UML	-
Rigi	C++	Análisis de relación de dependencia	Grafos jerárquicos	-
SHriMP	Rigi RSF	Visualización	Vista jerárquica de grafos	-
Bookshelf	Rigi RSF	Llamado de funciones, variables, análisis cruzado de archivos	Vista de panorama del software	-
GUPRO	GXL	Consultas de grafos	-	-
CrocoPat	Rigi RSF	Búsqueda en grafos	Visualización de grafos en 2D y 3D	+
FUJABA	Código Java	Análisis dinámico, lógica difusa	Diagrama de historia + Clases UML	+
SPOOL	CDIF basado en UML	Consultas en base de datos	Clases UML, HTML	+
PINOT	Java	Análisis de flujo de datos, entre clases, considera el uso de clases utilitarias de Java	-	+
PtideJ	Código binario Java	Análisis (solver) de restricciones	+	+

Herramienta CASE	Lenguajes	Transformación	Modelamiento
Borland Together	Java, C++, VB, C#	Código fuente – Clases UML, Modelo de base de datos – sincronización base de datos física, código fuente a diagrama de secuencia, Clases UML a cualquier modelo con QVT	Clases UML, base de datos
Rational XDE	Varios, incluyendo Java, Delphi, VB, C#	Código fuente – Clases UML, sincronización base de datos física, código fuente a diagrama de secuencia	Clases UML, base de datos
Eclipse GMT	Independiente del lenguaje	Transformaciones basada en modelos	Modelos, metamodelos
MetaEdit+	Independiente del lenguaje	Modelos generales a código fuente, API para que el usuario escriba programas de ingeniería inversa	Cualquier modelo específico del dominio
FUJABA	Java	Diagramas de historia + Clases UML – código fuente ejecutable de reingeniería	Diagrama de historia, Clases UML

Figura 3: **Tabla comparativa de herramientas de ingeniería inversa y herramientas CASE.**

identificación de patrones de diseño y una herramienta de auditoría de código.
CPP2XMI [KPBM06] permite extraer diagramas UML de clases, de secuencia y de actividad en formato XMI a partir de código C++. La herramienta procesa la salida en formato Lenguaje de marcaje de C++ (*CPPML, C++ Markup Language*) de *Columbus* y la presenta mediante vistas en 2D o 3D.
Rigi [Mul13] es una herramienta interactiva de visualización de estructuras de software, que analiza las dependencias entre artefactos de software a partir del código fuente. Las relaciones de procedimientos, invocaciones, accesos a datos y variables, entre otros flujos de control y de datos, son almacenadas en un formato estándar (RSF) y luego visualizadas en diagramas jerárquicos de grafos.
SHriMP (Simple Hierarchical Multi Perspective) [Gro13] es una aplicación y una técnica de visualización para explorar estructuras jerárquicas como si fuera código de software.
Bookshelf [Hol13] es un conjunto de herramientas que apunta a la visualización y navegación de grandes sistemas de software.
GUPRO [rEKRW02] es un entorno de trabajo integrado basado principalmente en el modelamiento de grafos y en los algoritmos. El código fuente se almacena en un grafo de repositorio interno, lo cual permite elegir la representación de código o la técnica de análisis.

Otra rama de investigación de la ingeniería inversa es el reconocimiento de patrones de diseño [GHJV95] a partir del código fuente. Detectar la presencia de patrones en una arquitectura hace más fácil la comprensión de las cuestiones de diseño de un sistema de software. Los patrones de diseño pueden ser identificados mediante distintos enfoques, por ejemplo, a partir de relaciones entre clases en invocaciones a métodos, análisis de flujo de datos, lógica difusa, matching en grafos y semántica formal.
La herramienta *CrocoPat* [BL03] procesa archivos con el formato estándar Rigi (RSF) que contienen al grafo de un sistema y busca, utilizando su propio lenguaje imperativo, los patrones predefinidos entre las relaciones de herencia de clases y las invocaciones a

métodos. *Columbus* utiliza algoritmos de matching entre grafos. También hay otros métodos disponibles, *Ptidej* [AACGJ01], que utiliza análisis (resolución) de restricciones o *SPOOL* [KSRP99], que utiliza consultas en base de datos. *PINOT* [SO06] es una herramienta de inferencia y recuperación de patrones que reclasifica los patrones definidos por [GHJV95] e implementa un análisis estático liviano entre clases y de flujo de datos.

Además, de las herramientas vistas en la tabla comparativa se pueden mencionar: *Dali* [KC99] es un ambiente de trabajo para la extracción de arquitectura, la manipulación y las pruebas de conformidad. Esta herramienta integra varias herramientas de análisis y almacena la información obtenida (modelada mediante un grafo Rifi) en un repositorio. Este grafo contendrá información estática e información del comportamiento del sistema de software objetivo, extraída utilizando profilers y herramientas de pruebas de cobertura.
Imagix4D [Cor13] provee soporte para ingeniería inversa y documentación de sistemas C/C++. El código fuente del sistema bajo análisis puede ser analizado y presentado mediante distintas vistas, en cualquier nivel de abstracción.
DESIRE [Big89] es un sistema de recuperación de diseño, basado en modelos, que puede ser utilizado para reconocer conceptos y ayudar en la comprensión de un programa. Provee utilidades de asistencia inteligente para identificar instancias de conceptos definidos por el usuario, identificar conceptos que se correspondan con algún concepto del modelo de dominio y proponer la asignación de un concepto para un grupo de interés dado.
Ovation [PHKV93] utiliza vistas de patrones de ejecución para visualizar y explorar una ejecución de un programa en diferentes niveles de abstracción.
Sefika [SSC96] introduce un enfoque de visualización orientada a la arquitectura que puede ser utilizada para ver el comportamiento de un sistema objetivo en diferentes niveles de granularidad. Esta herramienta utiliza una técnica llamada instrumentación con conocimiento de la arquitectura, que permite al usuario obtener información del sistema objetivo en el nivel de abstracción deseado (subsistema, framework, patrones, clases, objetos y métodos).
SCED (scenario editor) [Sys00] fue diseñado para mejorar el soporte automático para el modelado dinámico en la construcción de software orientado a objetos. Cuando existe una cantidad suficiente de escenarios son transformados en un diagrama de estados para los objetos participantes. Si bien fue creado para ingeniería directa, se puede usar para ingeniería reversa.

HERRAMIENTAS CASE

Con respecto a las herramientas CASE, las más populares como *Rational XDE* [IBM13], *Borland Togheter* [Bor13], *Eclipse GMT* [Fou13b] y *Fujaba* [NNWZ00] soportan reingeniería. La tecnología *LiveSource* [Bor13] automáticamente sincroniza modelos y código, por lo tanto el modelo se basa directamente en el código fuente en sí; el diagrama de clase UML será, por lo tanto, una vista de la implementación. Cuando el modelo cambia, el código fuente es actualizado automáticamente, resultando en un diseño siempre actualizado.

MetaEdit+ [Met13] es un ambiente de modelado y metamodelado integrado para lenguajes específicos del dominio (DSL). Los desarrolladores pueden definir un lenguaje de modelado específico del dominio con roles, restricciones y especificar el mapeo de estos elementos a fragmentos de código en un generador específico del dominio. La ejecución del modelo puede ser simulada, MetaEdit+ muestra que elementos del modelo están siendo ejecutados y los vincula con el código fuente correspondiente.

Los sistemas *Fujaba* introducen modelamiento Dirigido por Historias (*Story-Driven*) como un nuevo método para el desarrollo de software, el cual se basa en un lenguaje de programación visual de alto nivel llamado diagramas de historias. *Fujaba* es un ambiente de reingeniería que crea diagramas de clase y diagramas de historia a partir del grafo de sintaxis abstracto de un código Java. Los diagramas de historias pueden visualizar los aspectos dinámicos de un sistema como el flujo de control, adaptar clases UML, actividades y diagramas de colaboración, y pueden ser traducidos a Java. El generador *Fujaba* traduce diagramas de historia en el cuerpo de métodos de las clases descritas en los diagramas de clase.

Eclipse es un framework de código abierto, el cual puede convertirse en un ambiente de desarrollo integrado completo, mediante la utilización de plugins externos. Varias herramientas de transformación entre modelos son implementadas como plugins de *Eclipse*. Por ejemplo, el plugin UML2 que provee una implementación del metamodelo UML de la OMG. Si bien esta implementación no incluye herramientas para el modelado en sí, existen otros plugins, como UML2Tools que provee un editor para manipular modelos UML. También *Borland Together* está disponible como un plugin. Al brindar esta posibilidad de extensión, *Eclipse* ha permitido que se creen diversos proyectos específicos para un dominio determinado. Esta quizás sea una de las razones más fuertes de su elección por parte de una masiva comunidad y del porqué de su utilización, en el presente trabajo, como entorno principal.

A continuación podemos listar algunos de estos proyectos:

- Las Tecnologías de Modelado Generativo (*GMT, Eclipse Generative Modeling Technologies*), son un conjunto de herramientas pertenecientes al área de desarrollo dirigido por modelos. La transformación entre modelos se mantiene como la operación esencial, pero otras facilidades de modelado, como la composición de modelos, fueron propuestas para expandir el alcance del proyecto GMT. Por ejemplo, el plugin para el lenguaje de transformación ATLAS provee un conjunto de herramientas de transformación para el proyecto GMT.
- El Framework de Modelado (*EMF, Eclipse Modeling Framework*) fue creado para facilitar el modelado de sistemas y la generación automática de código Java. EMF empezó como una implementación de MOF resultando en Ecore, el metamodelado EMF comparable con EMOF. EMF comenzó a partir de la experiencia de la comunidad de usuarios de Eclipse y fue evolucionando para implementar una variedad de herramientas, altamente relacionadas con MDE. Por ejemplo, herramientas comerciales como IBM Rational Software Architect, Spark System Enterprise Architect o Together están integrados con Eclipse-EMF.
- El proyecto de Transformación Modelo a Modelo (*MMT, Model-to-Model Transformation*), es un sub-proyecto del Proyecto de Modelado de Eclipse (*EMP, Eclipse Modeling Project*) que provee un framework para lenguajes de transformación entre modelos. Las transformaciones son ejecutadas mediante motores de transformación que son añadidos en la infraestructura de modelado de Eclipse. Los motores de transformación principales desarrollados en el alcance del proyecto son ATL [Fou13a] y QVT [OMG13g]. Actualmente, el componente de QVT declarativo está en fase de "incubación" y provee sólo funcionalidades de edición para soportar el lenguaje QVT.

Además del mencionado Eclipse MMT, pocas herramientas CASE basadas en MDA soportan cualquiera de los lenguajes QVT. Por ejemplo, *IBM Rational Software Architect* soporta transformación modelo a modelo y modelo a texto pero no MOF y QVT. *Spark System Enterprise Architect* está basado en MDA y UML 2.1 y por lo tanto es compatible con MOF. Otras herramientas soportan parcialmente QVT, por ejemplo, *Together* permite definir y modificar transformaciones modelo a modelo (M2M) y modelo a texto (M2T) que cumplen con el QVT Operacional. *Medini QVT* [it13] soporta parcialmente MOF e implementa QVT. Está integrado con Eclipse y permite la ejecución de transformaciones expresadas en el lenguaje QVT Relacional.

También es importante mencionar, por fuera de la tabla comparativa presentada, la herramienta *MoDisco* [Fou13e] perteneciente a la fundación Eclipse. *MoDisco* provee un framework extensible para desarrollar herramientas dentro del contexto del desarrollo dirigido por modelos con el objetivo de brindar soporte para los casos de uso en la modernización de sistemas de software existentes. *MoDisco* combina los proyectos Eclipse mencionados anteriormente para esta finalidad: utiliza EMF para describir y manipular los modelos, M2M para implementar la transformación entre modelos, M2T

para implementar la generación de texto y Eclipse JDT para crear modelos a partir de código fuente Java.
El proyecto de código abierto *Eclipse-MDT MoDisco* está considerado por la Fuerza de Tareas de Modernización Dirigida por Arquitecturas de la OMG (*ADMTF, The Architecture-Driven Modernization Task Force*) como el proveedor de referencia para la implementación de varios de sus estándares. Es un framework basado en modelos reutilizable y extensible que facilita la construcción de aplicaciones para ingeniería inversa. Actualmente, el enfoque *MoDisco* sólo soporta la ingeniería inversa de diagramas de clase.
La Fuerza de Tareas de Modernización Dirigida por Arquitecturas (ADMTF) [OMG13a] fue formada para crear especificaciones y promover consenso en la industria sobre la modernización de las aplicaciones existentes, sin importar la plataforma sobre la que corren, el lenguaje en el que fueron escrito, o el tiempo que se encuentran en producción. La Modernización Dirigida por Arquitectura (ADM) es el proceso de entender y evolucionar los valores de los sistemas de software existentes con el fin de mejorar, modificar, lograr la interoperabilidad, refactorizar, reestructurar, reutilizar, migrar e integrar aplicaciones de software.

Actualmente no hay herramientas que asistan en un proceso de reingeniería completo como el propuesto en este trabajo. Sin embargo, varias herramientas están disponibles para utilizarse durante su aplicación. En la sección siguiente se presenta el caso de estudio propuesto. Para cada etapa de aplicación del proceso de desarrollo se presentará la herramienta utilizada y se detallarán las ventajas/desventajas encontradas en su utilización.

CASO DE ESTUDIO

Para ejemplificar la utilidad de los conceptos involucrados en el proceso de reingeniería propuesto, y ver la asistencia que brindan distintas herramientas durante este proceso; se propone, como caso de estudio, la migración de una aplicación existente desarrollada para correr en una computadora de escritorio, a una nueva plataforma de desarrollo móvil.

INTRODUCCIÓN

El sistema de software pertenece a la categoría de sistemas *CRM (Customer Relationship Management)*, es decir, un software que permite administrar la relación con los clientes. Este tipo de sistemas apoyan y guían la relación con los clientes creando un perfil dinámico de los mismos, mediante el registro de datos importantes como: datos personales, servicios y productos contratados registrando importe, frecuencia y lugar de compra, canales de contacto que suele utilizar, acciones comerciales ya realizadas y su respuesta en cada una de ellas, entre otros [Cav04].
Además, un sistema CRM hace accesible toda esta información a los usuarios autorizados, permitiéndoles disponer de ella en el momento adecuado (es decir, cuando es viable interactuar con el cliente con el fin de modificar su conducta).
Con el avance de la tecnología, estos sistemas han pasado de ser simples aplicaciones de escritorio cliente-servidor, a grandes aplicaciones web (como puede ser el caso de Salesforce.com), hasta convertirse en una de las aplicaciones móviles más solicitadas hoy en día. La aparición de los dispositivos móviles ha permitido mejorar y agilizar la atención de la relación con el cliente, además de simplificar notablemente el intercambio de información entre un cliente y un consultor. Mediante un dispositivo móvil un consultor puede realizar una captura de las solicitudes del cliente, obtener un resumen de las últimas interacciones con el mismo, entre otras actividades.
Es por esto que se propone una plataforma de desarrollo móvil como tecnología objetivo del proceso de reingeniería, más precisamente, la plataforma de desarrollo móvil Android [Goo13b].
Android, además de ser un sistema operativo basado en el kernel de Linux funcionando sobre arquitecturas ARM, es una plataforma de desarrollo de código abierto que soporta una implementación de un subconjunto del lenguaje JAVA proveyendo un framework completo de desarrollo de aplicaciones.

Desarrollar una aplicación para un dispositivo móvil implica adoptar y entender las características de estos dispositivos. Si bien se cuenta con características avanzadas como bases de datos integradas, soporte multimedia y mecanismo de comunicación y geolocalización; también se presentan importantes restricciones en cuanto al tamaño de la pantalla disponible, la utilización de memoria primaria y las bibliotecas de desarrollo disponibles.

DESCRIPCIÓN DE LA APLICACIÓN ORIGEN

La aplicación que se utilizará como caso de prueba se denomina **SellWin** [App13].
SellWin es un CRM simple orientado a las ventas desarrollado hace un par de años que centra la gestión de datos alrededor de lo que denomina oportunidad de venta (*Opportunity*). Permite administrar los datos de los clientes, los usuarios del sistema y de distintas entidades relacionadas con las oportunidades de venta como, por ejemplo, actividades, pronósticos, cotizaciones y ordenes. El análisis realizado en el presente caso de estudio priorizará estas entidades para ejemplificar el proceso de migración de la aplicación.

Desde el punto de vista técnico se puede mencionar que SellWin es una aplicación código abierto implementada completamente mediante el lenguaje de programación Java, con lo cual puede ejecutarse en cualquier plataforma Java (J2SE). Utiliza la interfaz de programación *Swing* para la interfaz con el usuario y *JDBC* para la conexión con bases de datos, con lo cual puede utilizarse en cualquier plataforma y para conectarse a cualquier base de datos utilizando el lenguaje *SQL* (particular de la base de datos elegida).
La arquitectura elegida sigue los lineamientos de una arquitectura cliente-servidor, donde la interfaz con el usuario (el cliente) es una aplicación aislada que corre en una única computadora. El servidor está implementado para responder a las solicitudes de datos aislando a los clientes de la forma de comunicación y base de datos particular utilizada.
La arquitectura simple de esta aplicación sigue un diseño orientado a componentes, separados en distintos módulos, de acuerdo a la funcionalidad a proveer:

- Dominio: los componentes del dominio representan la información acerca de las entidades que representan la lógica del negocio. Cada entidad está definida por un objeto separado que encapsula los atributos correspondientes.
- Base de datos: estos componentes se utilizan para mapear cada entidad del dominio. Cada componente contiene las rutinas de acceso a base de datos relacionadas con el objeto de dominio que representan. Todos los accesos están escritos utilizando la interfaz de programación JDBC.

- Servidor: los componentes del lado del servidor son objetos que manejan las solicitudes cliente-servidor en la forma de llamadas remotas desde la interfaz del usuario. Generalmente, las invocaciones desde el cliente solicitan información a obtener desde la base de datos.
- Interfaz con el usuario: estos componentes implementan una aplicación independiente que corre en la computadora del usuario y presenta la interfaz de acceso a la aplicación.

La elección de SellWin para ejemplificar el proceso de reingeniería descrito es debido a sus características simples y a la posibilidad de contar con un sistema CRM de código abierto. Además, no cuenta con una documentación adecuada para poder comprender el diseño del mismo, lo cual permite analizar las fuerzas y debilidades de la aplicación de las técnicas de ingeniería inversa para su comprensión.
A continuación se detallan los pasos realizados para la traducción de la aplicación elegida a la plataforma Android.

APLICACIÓN DEL PROCESO DE REINGENIERÍA PROPUESTO

La primera etapa del proceso se centra en recuperar los artefactos de software necesarios para comprender el diseño y las decisiones de implementación de la aplicación elegida. El objetivo es detectar las clases que conforman la aplicación y los objetos que participan en las distintas funcionalidades brindadas.
Como ya se mencionó, al no contar con una documentación adecuada, el código fuente es el único repositorio de información para recuperar el diseño del sistema. Debido a esto, esta primera etapa consta de la aplicación de distintas técnicas de ingeniería inversa para generar los diagramas necesarios.

Análisis estático: Diagrama de clases

El primer paso fue recuperar el diagrama de clases para detectar las relaciones existentes entre los distintos componentes que conforman los módulos principales. Este es el primer momento donde es necesario contar con una herramienta que asista y automatice el proceso de generación del diagrama de clases a partir del código fuente. Esta funcionalidad es una de las más comunes que brindan las herramientas CASE; sin embargo, todavía es necesario considerar situaciones particulares que no se pueden resolver por completo de forma automática. Una de estas situaciones es la utilización de colecciones genéricas.
La aplicación SellWin fue desarrollada previamente a la incorporación de colecciones genéricas en Java [Ora13]; por lo tanto, para trabajar con una lista, por ejemplo, que contenga las actividades o las órdenes de una oportunidad se utiliza un fragmento de código como el siguiente:

```
public class Opportunity {
    private ArrayList activities;
    private ArrayList orders;
    ...
    public final ArrayList getOrders() { return orders; }
    public final void addOrder(Order o) { orders.add(o); }
    public final ArrayList getActivities() { return activities; }
    public final void addActivity(Activity a) { activities.add(a); }
    ...
}
```

Código 1: **Ejemplo de utilización de colecciones genéricas.**

A la hora de recuperar las relaciones entre clases, este tipo de código no permite que las herramientas detecten que existe una relación de asociación de actividades y órdenes con oportunidad, sólo detectan una dependencia, como se puede ver en la figura 4a.
Por lo tanto, es necesario un pre-procesamiento del código para incorporar los tipos de datos y adaptar el código existente a las nuevas versiones de J2SE. Cabe mencionar que entornos de desarrollo avanzados, como *Eclipse*, detectan esta situación de colecciones sin especificación de tipos, por lo cual genera una advertencia y propone una alternativa de resolución automática.
Una vez determinado los tipos de datos de las colecciones, al generar nuevamente el diagrama de clases se puede observar la detección de las relaciones de asociación entre las entidades (figura 4b).

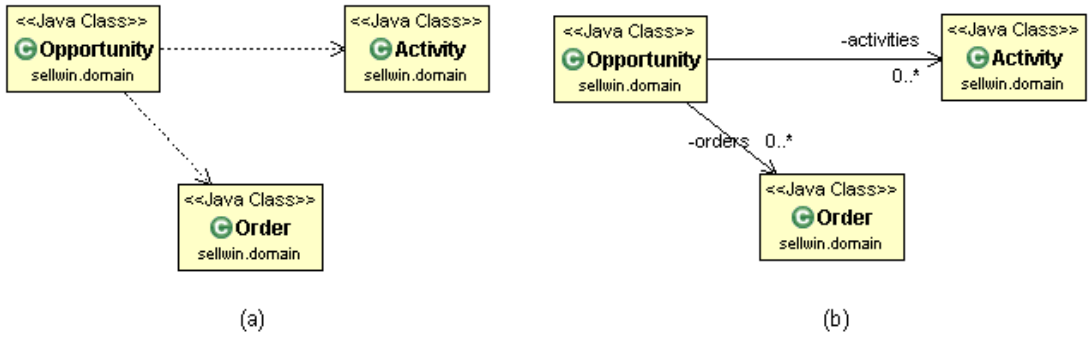

Figura 4: **Fragmento del diagrama de clases recuperado. (a) Sin tipos de datos. (b) Con tipos de datos.**

Todos los diagramas de clases que se presentan en el informe fueron recuperados utilizando la herramienta *ObjectAid UML Explorer* [LLC13a] que se integra con el entorno de desarrollo *Eclipse*. *ObjectAid* permite generar, de forma simple, diagramas de clases a partir del código fuente sin necesidad de pasar primero por una representación intermedia (situación común en otras herramientas). Es una herramienta gratuita para trabajar con diagramas de clases pero que restringe el acceso a los diagramas de secuencia mediante una licencia comercial.

Luego de recuperar el diagrama de clases de toda la aplicación se puede visualizar la separación entre módulos, descripta anteriormente, y cómo se relacionan cada uno de los componentes para implementar las funcionalidades mencionadas.
A modo de ejemplo se presenta el diagrama de clases de la administración de clientes (Figura 6). Desde la perspectiva de la interfaz del usuario, la pantalla de administración de clientes presenta una lista con los clientes existentes. Al seleccionar cada uno de ellos se puede ver, en distintas secciones, datos particulares de las ventas, información de los productos adquiridos y también información de contacto del cliente particular (Figura 5).

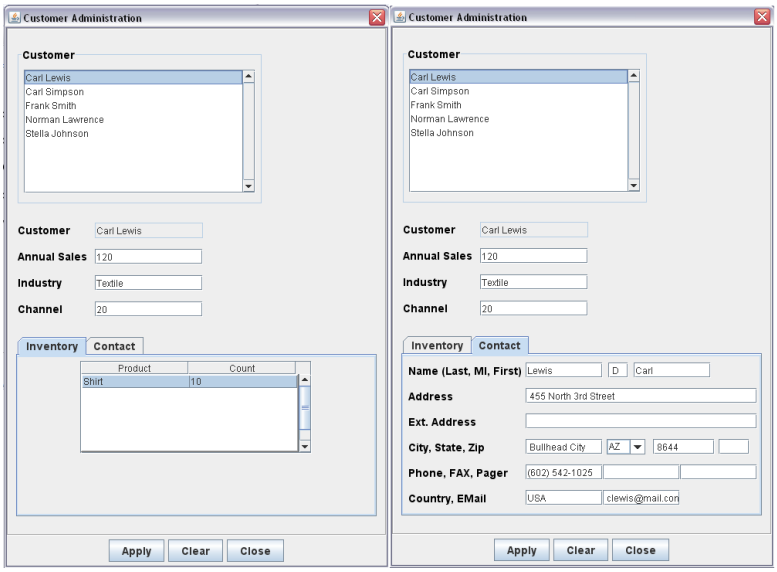

Figura 5: **Pantalla de administración de clientes. (a) Vista de productos adquiridos. (b) Vista de datos de contacto.**

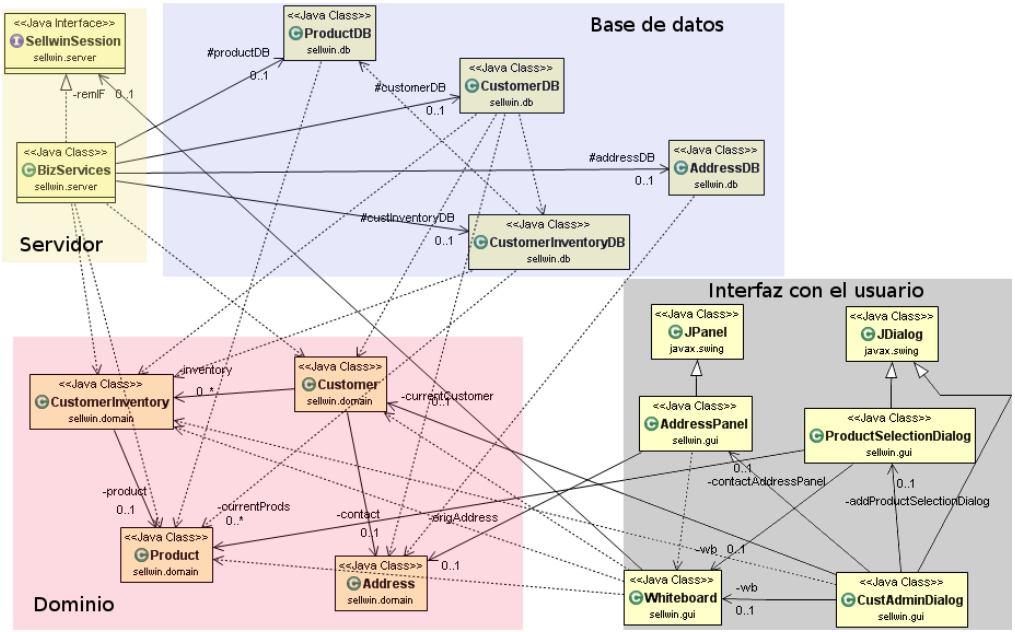

Figura 6: **Diagrama de clases de una parte de la aplicación.**

Sin entrar en detalles de cada clase concreta y su interfaz el objetivo de este diagrama, como ya se mencionó, es visualizar las relaciones entre los distintos módulos. Como se puede ver, el módulo de interfaz del usuario no tiene relación con la base de datos; el acceso a los datos está provisto por el módulo de servidor, con el cual mantiene una asociación directa a través de una interfaz definida. Por otro lado, la interfaz de usuario es la única que tiene asociaciones directas con el dominio, ya que tanto el servidor como la base de datos sólo registran dependencias de acuerdo a métodos de la interfaz de cada clase.

Análisis dinámico: Trazas de ejecución y estado de la memoria

Mediante el análisis estático se detectaron las clases que componen la aplicación y las relaciones existentes entre sí. Para detectar cómo interactúan en la resolución de las funcionalidades brindadas, es que se aplica el análisis dinámico.
En este caso se recupera información aplicando dos técnicas: trazas de ejecución y detección del estado de la memoria.
Para obtener las trazas de ejecución de la aplicación se utilizó la Plataforma de Herramientas de Prueba y Performance del entorno Eclipse (*Eclipse TPTP, Eclipse Test & Performance Tools Platform Project*) [Fou13d]. Esta plataforma permite ejecutar una instancia de la aplicación y registrar las invocaciones realizadas por cada objeto creado, para brindar la funcionalidad bajo análisis.
Por ejemplo, al inicializar la pantalla que permite administrar los contactos del sistema (Figura 5) se obtiene, de forma simplificada, la siguiente traza de ejecución:

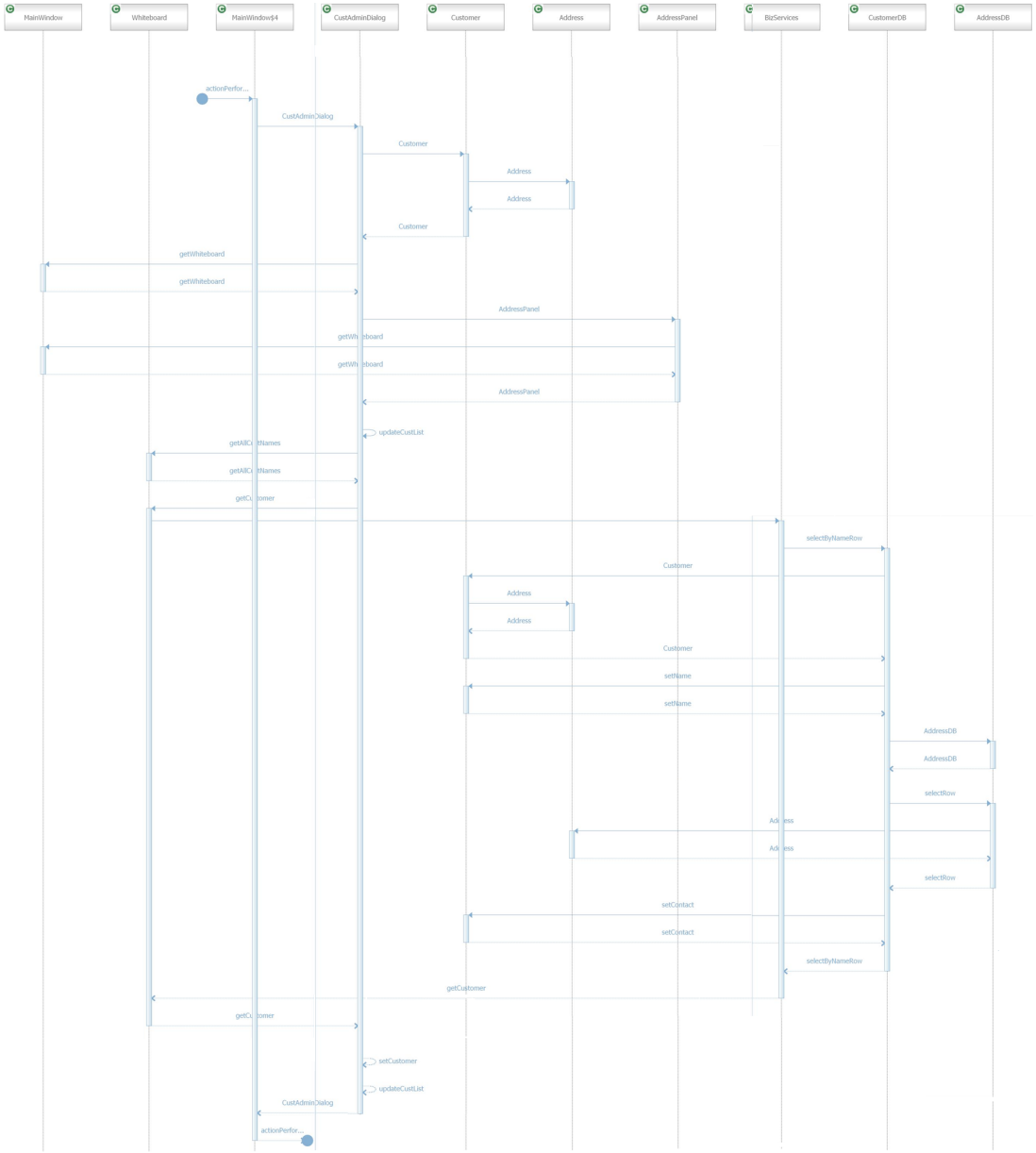

Figura 7: **Diagrama de traza de ejecución de la pantalla de administración de clientes.**

Si bien el resultado no es un diagrama de secuencia clásico (debido a que no detecta sentencias de control, por ejemplo) es una buena aproximación para detectar los métodos que intervienen en cada funcionalidad concreta y el orden de invocación de los mismos. También permite ver cómo los componentes de la interfaz del usuario interactúan con los componentes de dominio directamente pero no así con los componentes de base de datos. Para acceder a los datos provistos por estos componentes se utiliza la clase `BizServices` que representa al componente del servidor.

La última técnica de análisis dinámico que se utilizó es la evaluación del estado de la memoria. Mediante este análisis lo que se pretende recuperar son los valores reales de cada uno de los atributos de los objetos creados durante la ejecución de la aplicación.

Esta información, no sólo es muy importante para poder implementar correctamente la aplicación en la plataforma de destino, sino también será útil para la etapa de modelamiento, como se verá en la sección siguiente.
Para detectar el estado de la memoria se utilizó una herramienta comercial que permite su utilización gratuita para evaluación por un tiempo limitado llamada *Yourkit Java Profiler* [LLC13b]. Esta herramienta permite ejecutar la aplicación y capturar la información de los objetos que se van creando en memoria.
A continuación se presenta una captura de pantalla que refleja parte de la información obtenida por la herramienta para el caso de la administración de clientes.

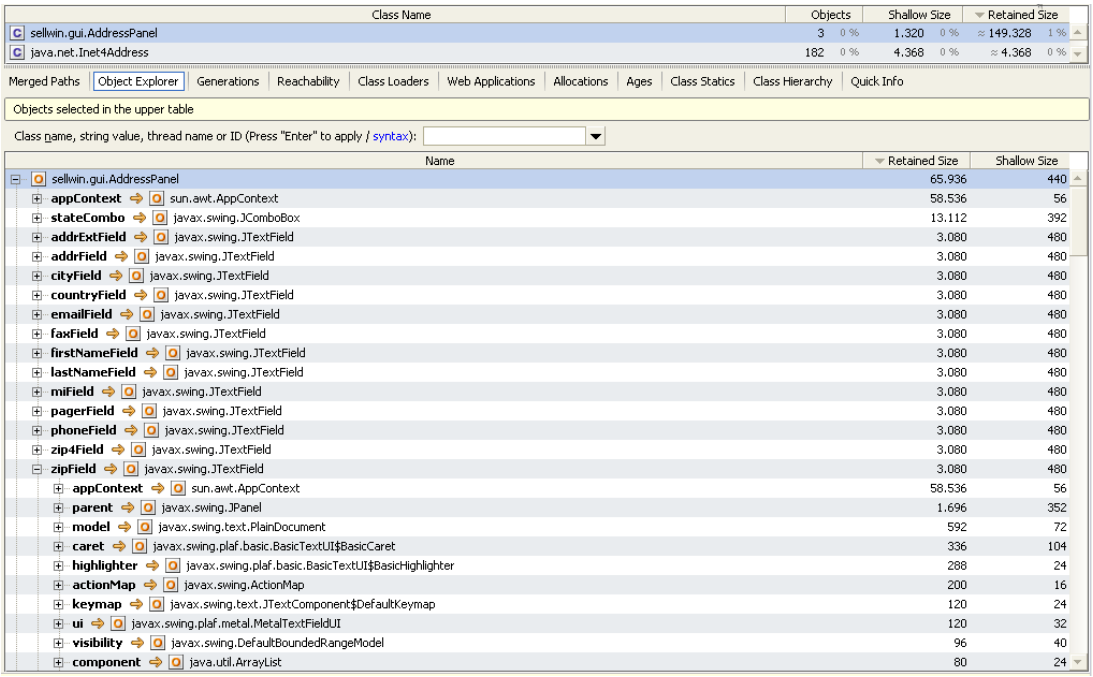

Figura 8: **Información del estado de la memoria.**

Como resultado de las técnicas de análisis aplicadas se han conseguido recuperar distintos artefactos que permiten reconstruir el diseño de la aplicación bajo estudio. A partir de este diseño se podrá implementar la misma aplicación en la plataforma de destino, realizando las modificaciones necesarias de acuerdo a las limitaciones de la misma (espacio de memoria, tamaño de pantalla, limitaciones de uso, entre otras).
El paso siguiente integra estos artefactos con las ideas detrás del desarrollo dirigido por modelos para conseguir una representación independiente de las plataformas involucradas hasta el momento.

MDA: REGLAS DE TRADUCCIÓN A LA PLATAFORMA DESTINO

La arquitectura dirigida por modelos apunta a la interoperabilidad entre plataformas y a la independencia de las tecnologías, proponiendo que todos los artefactos involucrados en un proceso de desarrollo se representen a partir del lenguaje de metamodelado MOF.
El objetivo de integrar estos artefactos, representados en MOF, con las técnicas de análisis estático y dinámico es generar modelos independientes de las tecnologías particulares; además de evaluar su consistencia mediante el modelamiento y la aplicación de traducciones entre modelos. Para esto último, se utilizó un lenguaje de transformación entre modelos especializado denominado ATL (ATLAS Transformation Language) [Fou13a]. ATL es un lenguaje híbrido que permite especificar transformaciones entre modelos de forma declarativa e imperativa. Es descrito mediante una sintaxis abstracta (un metamodelo MOF), y una sintaxis textual concreta. Un modelo de transformación en ATL es expresado como un conjunto de reglas de transformación.
La herramienta elegida para esta etapa del proceso de traducción es el Proyecto de Modelado del entorno de desarrollo Eclipse (*EMP, Eclipse Modeling Framework)* [Fou13c], el cual provee herramientas que permiten definir los metamodelos, definir las reglas de transformación ATL y ejecutar el proceso de traducción.

La plataforma de destino Android proporciona una versión del lenguaje Java, como lenguaje de programación principal para sus aplicaciones, diferente a la encontrada en los ambientes de ejecución estándar (JRE). Una de las diferencias más importantes se encuentra en la forma de construir interfaces gráficas, donde no se cuenta con frameworks como *Swing* o *AWT*, sino que la plataforma provee sus propias bibliotecas de componentes, denominados *widgets*.
Debido a esto es que los ejemplos de traducción entre modelos que se presentarán a continuación se centran en los componentes del módulo de interfaz del usuario, donde el diseño de estos componentes requiere cambios considerables. A su vez, los componentes del dominio permanecerán invariables mientras que los componentes de base de datos y servidor cambian principalmente en la implementación de los métodos de la interfaz y en las bibliotecas utilizadas para llevar a cabo sus funciones.

La arquitectura de traducción del presente caso de estudio se puede representar mediante la figura siguiente:

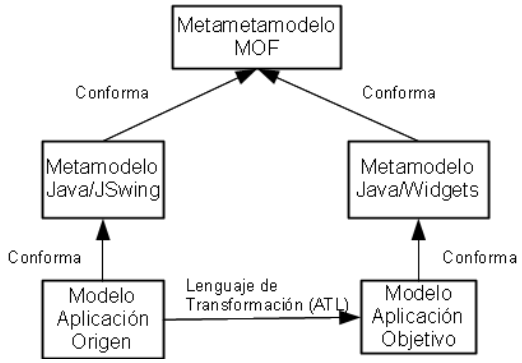

Figura 9: **Arquitectura del proceso de traducción.**

Siguiendo con el ejemplo de la pantalla de administración de clientes (Figura 5), a continuación se presenta el metamodelo **Java/JSwing** que define algunas de clases (y atributos) utilizadas para su construcción. Además, se muestra el modelo resumido de las clases de la pantalla en sí, que conforman al metamodelo anterior (Figura 10).
Por otro lado, también se presenta el metamodelo **Java/Android** de destino y el modelo concreto de la aplicación para implementar la pantalla de administración de clientes (Figura 11).

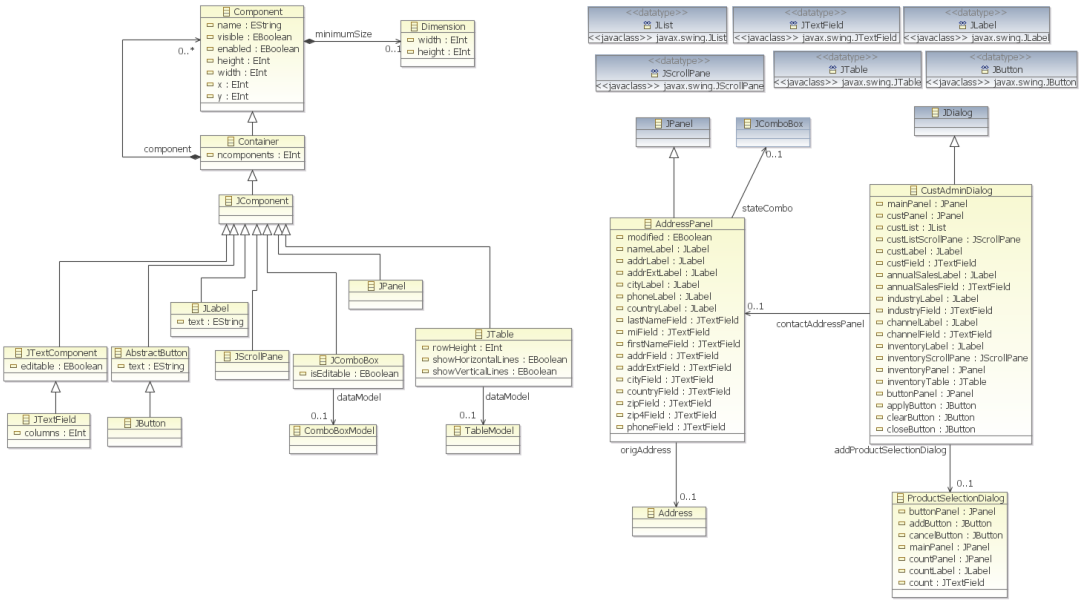

Figura 10: **(a) Metamodelo Java/JSwing. (b) Modelo de la aplicación origen.**

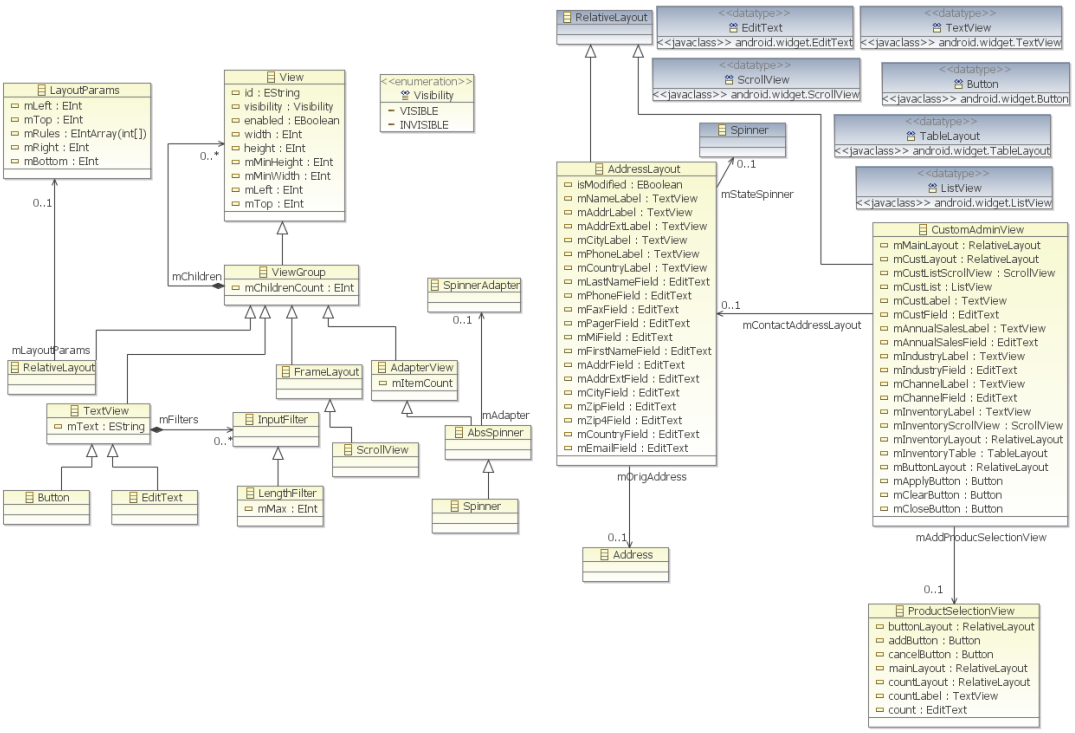

Figura 11: **(a) Metamodelo Java/Android. (b) Modelo de la aplicación destino.**

Analizando el metamodelo de destino se puede observar que la principal diferencia es el hecho de no contar con controles de interfaz que brinden la misma funcionalidad para todos los casos. En algunas situaciones, debido a las restricciones tecnológicas y características particulares de la plataforma de destino, se deberán crear funcionalidades equivalentes mediante la utilización de distintos *widgets*. Uno de estos casos puede ser la clase `JTable`, la cual implementa una tabla de datos, funcionalidad que no tiene su equivalente en Android y que deberá ser implementada a partir de la combinación de distintos controles.

En otros casos también se ven casos de restricciones, que se configuraban a partir de atributos de un control, pasan a ser asociaciones entre *widgets*. Por ejemplo, la situación de establecer un tamaño máximo para la cantidad de caracteres que se pueden ingresar en un control de edición, clase `JTextField` atributo `maxChars`, se representa en Android mediante la asociación de la clase `EditText` con un filtro de entrada de longitud, clase `LengthFilter`, y la configuración de su atributo `mMax`.

Estas consideraciones al pasar de la aplicación de origen a la aplicación de destino se harán presentes al momento de establecer las reglas de traducción mediante el lenguaje ATL.

Como ya se mencionó en la sección del lenguaje ATLAS, existen tres tipos de reglas que se corresponden con los dos paradigmas de programación provistos por dicho lenguaje, la programación declarativa y la programación imperativa. En este ejemplo se utilizarán principalmente reglas *Matched*. Estas reglas establecen el núcleo de una transformación declarativa ATL, debido a que permiten especificar e inicializar los elementos del

modelo de destino, generados a partir de los elementos del modelo de origen. Se denominan reglas *Matched* debido a que son aplicadas cuando el elemento coincide con el patrón definido por la regla. Debido a esto, es que tiene que existir una regla que defina un patrón de entrada para cada elemento del modelo de origen a transformar.
También se utilizarán los denominados helpers de ATL los cuales se definen como métodos que pueden ser llamados desde distintos puntos de una transformación permitiendo, de esta forma, la factorización de código.

A continuación se presentan las reglas de traducción para la clase `AddressPanel` y sus atributos que implementan el panel de contacto de la pantalla de administración de clientes (Figura 5b).

```
module JSwingToAndroid;
create OUT : JavaAndroid from IN : JavaSwing;

helper context JavaSwing!Component def: getVisibility():
    JavaAndroid!Visibility =
        if self.visible = true then
            #VISIBLE
        else
            #INVISIBLE
        endif;

helper context JavaSwing!Component def: getWidth(s: JavaSwing!Dimension):
    Integer =
        if s.oclIsUndefined() then
            0
        else
            s.width
        endif;

helper context JavaSwing!Component def: getHeight(s: JavaSwing!Dimension):
    Integer =
        if s.oclIsUndefined() then
            0
        else
            s.height
        endif;

rule ComponentToView {
    from
        jc: JavaSwing!Component
    to
        tv: JavaAndroid!View (
            visibility <- jc.getVisibility(),
            id <- jc.name,
            enabled <- jc.enabled,
```

```
            width <- jc.width,
            height <- jc.height,
            mLeft <- jc.x,
            mTop <- jc.y,
            mMinHeight <- jc.getHeight(jc.minimumSize),
            mMinWidth <- jc.getWidth(jc.minimumSize)
        )
}

rule ContainerToViewGroup extends ComponentToView {
    from
        jc: JavaSwing!Container
    to
        tv: JavaAndroid!ViewGroup (
            mChildren <- jc.component,
            mChildrenCount <- jc.ncomponents
            )
}

rule JComponentToViewGroup extends ContainerToViewGroup {
    from
        jc: JavaSwing!JComponent
    to
        tv: JavaAndroid!ViewGroup
}

rule JLabelToTextView extends JComponentToViewGroup {
    from
        jc: JavaSwing!JLabel
    to
        tv: JavaAndroid!TextView(
            mText <- jc.text
        )
}

rule JTextFieldWithColumnsToEditText extends JComponentToViewGroup {
    from
        jc: JavaSwing!JTextField(jc.columns > 0)
    to
        tv: JavaAndroid!EditText(
            enabled <- jc.editable,
            mFilters <- filters
        ),
        filters: JavaAndroid!LengthFilter (
                mMax <- jc.columns)
}
```

```
rule JTextFieldToEditText extends JComponentToViewGroup {
    from
        jc: JavaSwing!JTextField(jc.columns = 0)
    to
        tv: JavaAndroid!EditText (
            enabled <- jc.editable
        )
}

rule JButtonToButton extends JComponentToViewGroup {
    from
        jc: JavaSwing!JButton
    to
        tv: JavaAndroid!Button(
            mText <- jc.text
        )
}

rule JScrollPaneToScrollView extends JComponentToViewGroup {
    from
        jc: JavaSwing!JScrollPane
    to
        tv: JavaAndroid!ScrollView
}

rule JPanelToRelativeLayout extends JComponentToViewGroup {
    from
        jc: JavaSwing!JPanel
    to
        tv: JavaAndroid!RelativeLayout
}

rule JComboBoxToSpinner extends JComponentToViewGroup {
    from
        jc: JavaSwing!JComboBox
    to
        tv: JavaAndroid!Spinner
}

rule AddressPanelToAddressLayout extends JPanelToRelativeLayout {
    from
        jc: JavaSwing!AddressPanel
    to
        tv: JavaAndroid!AddressLayout (
            isModified <- jc.modified,
            mNameLabel <- jc.nameLabel,
            mAddrLabel <- jc.addrLabel,
            mAddrExtLabel <- jc.addrExtLabel,
```

```
                mCityLabel <- jc.cityLabel,
                mPhoneLabel <- jc.phoneLabel,
                mCountryLabel <- jc.countryLabel,
                mLastNameField <- jc.lastNameField,
                mFirstNameField <- jc.firstNameField,
                mMiField <- jc.miField,
                mAddrField <- jc.addrField,
                mAddrExtField <- jc.addrExtField,
                mCityField <- jc.cityField,
                mCountryField <-jc.countryField,
                mZipField <- jc.zipField,
                mZip4Field <- jc.zip4Field,
                mPhoneField <- jc.phoneField,
                mStateSpinner <- jc.stateCombo
        )
}
```

Código 2: **Reglas de traducción ATL.**

Debido a que los componentes principales del metamodelo de origen se relacionaban entre sí mediante una jerarquía, las reglas seguirán la misma estructura. Por lo tanto, la primer regla presentada establecerá la forma de transformar la metaclase padre de origen `Component` en la metaclase padre de destino `View`. La transformación se hace atributo a atributo de forma casi directa, salvo para los atributos en que es necesario invocar los *helpers* definidos anteriormente.
La siguiente regla especifica la transformación de `Container` a `ViewGroup`. Para esto se va a utilizar una característica agregada en el release ATL 2006, la herencia entre reglas. La regla creada para esta traducción (`ContainerToViewGroup`) extiende la mecánica de transformación de las clases padres de cada componente involucrado (regla `ComponentToView`)
La asociación de componentes (atributo `component`) en el contexto de `Container` se transforma directamente en la asociación definida en el contexto de `ViewGroup` (atributo `mChildren`). Esto es posible debido a la semántica de ejecución de las reglas *Matched* y al algoritmo de resolución de ATL. Este algoritmo determina que, luego de detectar que es necesaria una vinculación entre objetos de los distintos metamodelos, primero se intente resolver cada uno de los objetos antes de realizar la vinculación. En este caso puntual, esto resulta que los componentes encontrados en `components` primero son evaluados, para ver si existen reglas que definan en que elementos deben transformarse, y luego son asignados al atributo `mChildren`. Debido a esto es que se definen las reglas que se listaron anteriormente (`JButtonToButton`, `JScrollPaneToScrollView` y `AddressPanelToAddressLayout` por mencionar algunas), para transformar cada uno de los posibles elementos encontrados en components a su equivalente en el metamodelo Android.

Para evaluar estas reglas de traducción se utilizará la información de los atributos de los objetos recuperada a partir del análisis de estado de la memoria. Con esta información se construye un caso de entrada en formato xml que se presenta a continuación:

```
<AddressPanel name="addressPanel" visible="true" enabled="true" height="198"
    width="457" x="2" y="28" modified="false" ncomponents="9">
    <minimumSize height="198" width="457"/>
        <component xmi:type="JLabel" name="name" visible="true" enabled="true"
            height="19" width="147" x="5" y="7" text="Name (Last, MI, First)">
        </component>
        <component xmi:type="JLabel" name="addr" visible="true" enabled="true"
            height="19" width="147" x="5" y="40" text="Address">
        </component>
        <component xmi:type="JLabel" name="addrExt" visible="true" enabled="true"
            height="19" width="147" x="5" y="73" text="Ext. Address">
        </component>
        <component xmi:type="JLabel" name="city" visible="true" enabled="true"
            height="19" width="147" x="5" y="106" text="City, State, Zip">
        </component>
        <component xmi:type="JLabel" name="phone" visible="true" enabled="true"
            height="19" width="147" x="5" y="139" text="Phone, FAX, Pager">
        </component>
        <component xmi:type="JLabel" name="country" visible="true" enabled="true"
            height="19" width="147" x="5" y="172" text="Country, EMail">
        </component>
        <component xmi:type="JTextField" name="lastName" visible="true"
            enabled="true" height="20" width="100" x="0" y="0" editable="true">
        </component>
        <component xmi:type="JTextField" name="mi" visible="true" enabled="true"
            height="20" width="26" x="107" y="0" editable="true" columns="2">
        </component>
        <component xmi:type="JComboBox" name="state" visible="true"
            enabled="true" height="20" width="50" x="105" y="0" isEditable="false">
            <dataModel />
        </component>
</AddressPanel>
```

Código 3: **Caso de entrada XML para las reglas de traducción.**

Luego de ejecutar el código de la transformación sobre la máquina virtual de ATL se obtiene el equivalente en el metamodelo Android.
En este caso se puede ver como se generaron la misma cantidad de controles pero adaptando las características particulares del metamodelo. Por ejemplo, se ve que la configuración del tamaño mínimo de cada control deja de ser un elemento dentro del control (atributo `minimumSize`) para ser dos atributos propios de dicho control (`width` y `height`).

Otra diferencia importante se puede ver en la restricción de la cantidad de caracteres permitida por los controles de texto (atributo `columns`). En el caso resultante se ha creado una asociación con la clase `LenghtFilter` para cada control que imponía este tipo de restricciones.

```
<AddressLayout id="addressPanel" enabled="true" width="457" height="198"
    mMinHeight="198" mMinWidth="457" mLeft="2" mTop="28" mChildrenCount="9"
    mNameLabel="/2" mAddrLabel="/3" mAddrExtLabel="/4" mCityLabel="/5"
    mPhoneLabel="/6" mCountryLabel="/7" mLastNameField="/8" mMiField="/9"
    mFirstNameField="/10" mPhoneField="/17" mAddrField="/11" mAddrExtField="/12"
    mCityField="/13" mZipField="/15" mZip4Field="/16" mCountryField="/14"
    mStateSpinner="/1">
    <mChildren xsi:type="TextView" id="name" enabled="true" width="147" height="19"
        mLeft="5" mTop="7" mText="Name (Last, MI, First)"/>
    <mChildren xsi:type="TextView" id="addr" enabled="true" width="147" height="19"
    mLeft="5" mTop="40" mText="Address"/>
    <mChildren xsi:type="TextView" id="addrExt" enabled="true" width="147" height="19"
        mLeft="5" mTop="73" mText="Ext. Address"/>
    <mChildren xsi:type="TextView" id="city" enabled="true" width="147" height="19"
        mLeft="5" mTop="106" mText="City, State, Zip"/>
    <mChildren xsi:type="TextView" id="phone" enabled="true" width="147" height="19"
        mLeft="5" mTop="139" mText="Phone, FAX, Pager"/>
    <mChildren xsi:type="TextView" id="country" enabled="true" width="147" height="19"
        mLeft="5" mTop="172" mText="Country, EMail"/>
    <mChildren xsi:type="EditText" id="lastName" enabled="true" width="100" height="20"/>
    <mChildren xsi:type="EditText" id="mi" enabled="true" width="26" height="20"
        mLeft="107">
        <mFilters xsi:type="LengthFilter" mMax="2"/>
    </mChildren>
    <mChildren xsi:type="Spinner" id="state" enabled="true" width="50" height="20"
        mLeft="105"/>
</AddressLayout>
```

Código 4: **Caso resultante XML de la aplicación de las reglas de traducción.**

APLICACIÓN RESULTANTE EN ANDROID

La última etapa del proceso de reingeniería es realizar la implementación de la aplicación origen en la plataforma móvil Android. Los diagramas de clase y la información dinámica, recolectada mediante las distintas técnicas de ingeniería inversa, permiten establecer el diseño que debe seguir la aplicación a implementar; mientras que el modelamiento, a partir de la creación de un modelo de origen y la posterior traducción al modelo de destino, permiten validar los datos de las instancias concretas, que responden al diseño mencionado.

Las dificultades a la hora de realizar la implementación en la plataforma de destino se relacionan con las limitaciones y las características particulares de una plataforma móvil. El tamaño de la pantalla, las interrupciones generadas por eventos del teléfono (mensajes y llamadas, por ejemplo), el espacio de memoria de almacenamiento disponible y los métodos utilizados por los dispositivos para indicar entradas del usuario, son las principales dificultades que hay que superar al momento de crear una aplicación para una plataforma móvil.
En este sentido, la implementación de la interfaz de usuario en la nueva plataforma constituyó la etapa de migración más compleja. Si bien está disponible el lenguaje Java como lenguaje de desarrollo en Android, solamente contamos con una versión reducida que no contiene los frameworks básicos de controles de interfaz.
Para crear aplicaciones Android, en un ambiente de desarrollo clásico, se cuenta con el conjunto de Herramientas de Desarrollo de Software (*SDK, Software Development Kit*) [Goo13c], las cuales dan soporte completo para compilar y depurar durante el desarrollo.
En este proyecto, además, se utilizó un plugin que permite integrar estas herramientas con el entorno de desarrollo integrado de Eclipse, llamado ADT Plugin [Goo13a]. ADT extiende las facilidades de Eclipse para permitir crear fácilmente un nuevo proyecto Android, crear una aplicación de interfaz de usuario, depurar una aplicación utilizando el SDK y también exportar los archivos necesarios para distribuir la aplicación creada.
A continuación, para cerrar el ejemplo presentado anteriormente, se muestra la pantalla resultante del panel de datos de contacto de la administración de clientes.

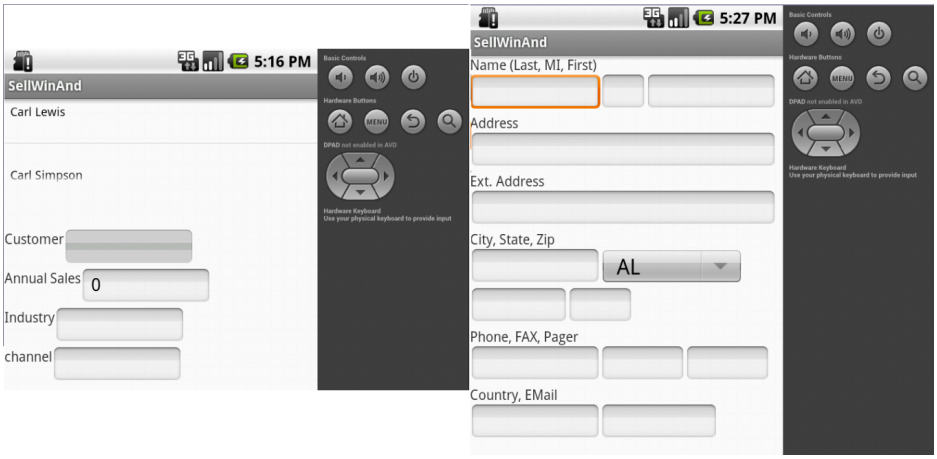

Figura 12: **Pantallas de administración de clientes de la aplicación resultante Android**

CONCLUSIONES Y TRABAJO FUTURO

El objetivo del presente trabajo fue proponer un proceso de reingeniería que integrara las técnicas de ingeniería inversa tradicionales con el desarrollo dirigido por modelos, para simplificar la traducción de aplicaciones de escritorio a plataformas móviles. Esta propuesta apunta a mejorar la productividad del desarrollo mediante la utilización de las siguientes estrategias:

- trabajar a un alto nivel de abstracción haciendo foco en el diseño en lugar de la implementación,
- encapsular modelos de plataformas móviles en una especificación de un metamodelo que será reutilizada por diferentes aplicaciones, y
- vincular diferentes herramientas CASE con diferentes actividades, de acuerdo al soporte que brindan.

El enfoque propuesto fue ejemplificado mediante un caso de estudio para modernizar un sistema CRM antiguo, realizando su traducción a una plataforma móvil. La idea es crear aplicaciones para plataformas móviles obteniendo, a partir de una aplicación de escritorio y mediante la ingeniería inversa, un modelo de alto nivel independiente de la plataforma; para luego transformar este modelo en código específico de la plataforma.
Proponer un proceso de desarrollo que considere modelos independientes de las plataformas es una práctica muy importante para prevenir una futura duplicación de esfuerzos, cuando se intenta proveer una aplicación a una nueva plataforma de destino.
En la creación de este proceso, se han detectado algunos inconvenientes que es necesario mencionar.
Cuando la única fuente de información de una aplicación es el código fuente, el éxito de la aplicación de la ingeniería inversa depende fuertemente de la existencia de herramientas que automaticen y asistan este trabajo. Esta es una de las complicaciones más importantes que surgen cuando se intenta migrar la lógica de un sistema a una nueva aplicación. De la misma forma, la poca documentación y maduración de las herramientas necesarias para el metamodelado y la transformación entre modelos, es en sí mismo, un inconveniente importante.
También se han destacado las principales complicaciones que involucra el desarrollo particular en la plataforma móvil de destino y que deben tenerse en cuenta en este tipo de procesos de migración.

Más allá de estas dificultades, se ha mostrado la aplicación aceptable del proceso de reingeniería propuesto mediante la integración de diferentes herramientas CASE y de asistencia en la ingeniería inversa. Además de mencionar la importancia de estas herramientas y como fueron aplicadas a los problemas concretos, se han remarcado sus fortalezas y debilidades en tareas de asistencia y automatización.

Por último, se puede establecer como trabajo a futuro la recuperación de otros tipos de artefactos, para los cuales no se cuenta hasta el momento con herramientas de asistencia, pero que su generación podría mejorar la creación de los modelos y la etapa final del proceso de reingeniería, es decir, la implementación de la aplicación en la plataforma de destino.
A su vez es interesante seguir explorando la aplicación del proceso de reingeniería propuesto en distintos contextos (aplicaciones de origen y plataformas de destino) con el objetivo de seguir sumando experiencias que permitan validar y adaptar su estructura para soportar la mayor cantidad de casos posibles.

BIBLIOGRAFÍA

[AACGJ01] Hervé Albin-Amiot, Pierre Cointe, Yann-Gaël Guéhéneuc, and Narendra Jussien. *Instantiating and Detecting Design Patterns: Putting Bits and Pieces Together*. 16th IEEE conference on Automated Software Engineering, 2001.

[ALC06] László Angyal, László Lengyel, and Hassan Charaf. *An Overview of the State-of-the-Art Reverse Engineering Techniques*. In Proceedings of the 7th International Symposium of Hungarian Researchers on Computational Intelligence. pag 507-516, 2006.

[App13] Open App. Sellwin, 2013.

[Big89] Ted Biggerstaff. *Design recovery for maintenance and reuse*. IEEE Software. pag 36-49, 1989.

[BL03] Dirk Beyer and Claus Lewerentz. *CrocoPat: Efficient pattern analysis in object-oriented programs*. In Proceedings of the 11th IEEE International Workshop on Program Comprehension. pag 294-295, IEEE Computer Society, 2003.

[Bor13] Borland. Borland together home page, 2013.

[Bro04] Alan Brown. *An introduction to model driven architecture*. IBM Developer Works, 2004.

[Cav04] Lluís Renart Cava. *CRM: Tres estrategias de éxito*. e-business Center PricewaterhouseCoopers & IESE, 2004.

[CC90] Elliot J. Chikofsky and James H. Cross. *Reverse engineering and design recovery: A taxonomy*. IEEE Software. pag. 13-17, 1990.

[Cor13] Imagix Corporation. Imagix - reverse engineering and source code analysis tools, 2013.

[DF08] Mark Dalgarno and Matthew Fowler. *UML vs. Domain-Specific Languages*. Software Development Magazine. Methods and Tools. pag 507-516, 2008.

[Ern03] Michael D. Ernst. *Static and Dynamic Analysis: Synergy and duality*. Proceedings of ICSE Workshop on Dynamic Analysis (WODA). pag 24-27, 2003.

[Fav10] Liliana Favre. *Model Driven Architecture for Reverse Engineering Technologies: Strategic Directions and System Evolution*. Engineering science reference, 1 edition usa, 2010.

[FMP11] Liliana Favre, Liliana Martínez, and Claudia Pereira. *Emerging Technologies for the Evolution and Maintenance of Software Models*. Software System Modernization: An MDA-Based Approach, 2011.

[Fou13a] Eclipse Foundation. Atl documentation, 2013.

[Fou13b] Eclipse Foundation. Eclipse gmt home page, 2013.

[Fou13c] Eclipse Foundation. Eclipse modeling framework, 2013.

[Fou13d] Eclipse Foundation. Eclipse test & performance tools platform project, 2013.

[Fou13e] Eclipse Foundation. Modisco, 2013.

[FrBTG02] Rudolf Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimothy. *Columbus – Reverse Engineering Tool and Schema for C++*. IEEE International Conference on Software Maintenance. pag 172-181, 2002.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Goo13a] Google. Adt tools, 2013.

[Goo13b] Google. Android platform, 2013.

[Goo13c] Google. Sdk tools, 2013.

[Gro13] Chisel Group. Shrimp home page, 2013.

[Hol13] Ric Holt. Bookshelf, 2013.

[HSSW13] Ric Holt, Andy Schürr, Susan Elliott Sim, and Andreas Winter. Graph exchange language, 2013.

[IBM13] IBM. Rational xde home page, 2013.

[it13] ikv++ technologies. Medini qvt, 2013.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. *ATL: A model transformation tool*. Science of Computer Programming 72. pag. 31-39, 2008.

[KC99] Rick Kazman and Jeromy Carrière. *Playing Detective: Reconstructing Software Architecture from Available Evidence*. Automated Software Engineering. pag 107-138, 1999.

[KPBM06] E. Korshunova, Marija Petkovic, Mark Van Den Brand, and Mohammad Mousavi. *CPP2XMI: Reverse Engineering of UML Class, Sequence, and Activity Diagrams from C++ Source Code*. Working Conference on Reverse Engineering, 2006.

[KSRP99] Rudolf Keller, Reinhard Schauer, Sébastien Robitaille, and Patrick Pagé. *Pattern-based Reverse- Engineering of Design Components*. In Proc. ICSE. pag. 226-235. ACM, 1999.

[LLC13a] ObjectAid LLC. Objectaid, 2013.

[LLC13b] YourKit LLC. Yourkit java profiler 12, 2013.

[Met13] MetaCase. Metaedit+ domain-specific modeling (dsm) environment, 2013.

[Mul13] Hausi Muller. Rigi group, 2013.

[NNWZ00] Ulrich Nickel, Jörg Niere, Jörg Wadsack, and Albert Zündorf. *Roundtrip Engineering with FUJABA*. Proceedings of 2nd Workshop on Software-Reengineering, 2000.

[OMG13a] OMG. Architecture-driven modernization task force, 2013.

[OMG13b] OMG. Knowledge discovery metamodel, 2013.

[OMG13c] OMG. Mda. the model-driven architecture, 2013.

[OMG13d] OMG. Meta object facility (mof) core specification version 2.4.1, 2013.

[OMG13e] OMG. The object management group consortium, 2013.

[OMG13f] OMG. Ocl. object constraint language, version 2.2, 2013.

[OMG13g] OMG. Qvt: Mof 2.0 query, view, transformation. version 1.1, 2013.

[OMG13h] OMG. Uml. unified modeling language: Infrastructure. version 2.4.1, 2013.

[OMG13i] OMG. Xml metadata interchange, 2013.

[Ora13] Oracle. Java programming language: Generics, 2013.

[PHKV93] Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides. *Visualizing the Behavior of Object-Oriented Systems*. In proc. of the 8th Annual Conference on Object-Oriented Programming systems, Languages and applications. pag 326-337, 1993.

[rEKRW02] ürgen Ebert, Bernt Kullbach, Volker Riediger, and Andreas Winter. *GUPRO – Generic Understanding of Programs An Overview*. Electronic Notes in Theoretical Computer Science 72 No. 2, 2002.

[SK03] Shane Sendall and Wojtek Kozaczynski. *Model Transformation: The Heart and Soul of Model-Driven Software Development*. IEEE Computer Society, 2003.

[SO06] Nija Shi and Ronald Olsson. *Reverse Engineering of Design Patterns from Java Source Code*. 21st IEEE International Conference on Automated Software Engineering. pag 123-134, 2006.

[SSo2] Eleni Stroulia and Tarja Systä. *Dynamic Analysis for reverse engineering and program understanding*. ACM SIGAPP Applied Computing Review. pag 8-17, 2002.

[SSC96] Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. *Architecture-Oriented Visualization*. In Proc. of the eleventh annual conference on Object-oriented programming systems, languages, and applications, 1996.

[Syso0] Tarja Systä. *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*. Ph.D Thesis. University of Tampere. Report A-2000-4, 2000.

[tec13] Moose technology. Famix, 2013.

[TPo5] Paolo Tonella and Alessandra Potrich. *Reverse Engineering of Object Oriented Code*. Monographs in Computer Science. Heidelberg: Springer-Verlag, 2005.

[TTBSo7] Paolo Tonella, Marco Torchiano, Bart Du Bois, and Tarja Systä. *Empirical studies in reverse engineering: state of the art and future trends*. Empir Software Eng. Springer Science + Business Media, 2007.