Revista EIA, ISSN 1794-1237 Número 9, p. 31-42. Julio 2008
Escuela de Ingeniería de Antioquia, Medellín (Colombia)

UN MÉTODO DE INGENIERÍA INVERSA DE CÓDIGO JAVA HACIA DIAGRAMAS DE SECUENCIAS DE UML 2.0

Carlos Mario Zapata*
Óscar Andrés Ochoa**
Camilo Vélez**

RESUMEN

La Ingeniería inversa de software aparece como un proceso que ayuda al aseguramiento de la calidad y documentación de aplicaciones con deficiencias en los modelos de análisis y diseño. Además, ayuda en la disminución de costos y tiempos de mantenimiento. En la actualidad existen herramientas CASE y algunas propuestas de investigación que realizan el proceso de ingeniería inversa a diagramas UML, en especial a los diagramas de clases y secuencias. Algunas se encuentran en fases experimentales; otras se enfocan mucho más en el diagrama de clases que en el de secuencias. Un tercer grupo obtiene algunos elementos del diagrama de secuencias, pero no posee muchos de los elementos que hacen parte de la especificación de UML 2.0. En este artículo se propone un método que automatiza la conversión de código JAVA en diagrama de secuencias de UML 2.0, por medio de la aplicación de reglas de transformación que convierten los elementos del código en elementos del diagrama. Se presenta también un ejemplo de aplicación del método con un prototipo que lo emplea, el UNC-Inversor.

PALABRAS CLAVE: Ingeniería inversa; diagrama de secuencias; UML; JAVA; reglas de transformación.

---

* Ingeniero Civil, Especialista en Gerencia de Sistemas Informáticos, Magíster en Ingeniería de Sistemas y Doctor en Ingeniería con énfasis en Sistemas. Profesor Asociado de la Escuela de Sistemas, Facultad de Minas, Universidad Nacional de Colombia, Sede Medellín. Integrante del Grupo de Investigación en Lenguajes Computacionales. cmzapata@unal.edu.co

** Estudiante de Ingeniería de Sistemas e Informática de la Universidad Nacional de Colombia. Integrante del Grupo de Investigación en Lenguajes Computacionales de la Escuela de Sistemas, Facultad de Minas, Universidad Nacional de Colombia. oaochoa@unalmed.edu.co; cvelezp@unalmed.edu.co

Artículo recibido 20-III-2008. Aprobado 8-VI-2008
Discusión abierta hasta diciembre de 2008

ABSTRACT

Software reverse engineering seems to be the process for helping software quality assurance and documentation in applications with low-quality analysis and design models. It also helps for decreasing maintenance cost and time. Currently, some CASE tools and research proposals assist analysts to develop reverse engineering process with UML diagrams as a result (especially class and sequence diagram). Some of them have reached experimental phases. Some others are focused more on class diagram and less on sequence diagram. A third group of CASE tools and proposals obtains sequence diagram, but the resulting diagram lacks some of the elements of the UML 2.0 specification. In this paper, we propose a method for automating the conversion of JAVA code into UML 2.0 sequence diagram, by means of the application of transformation rules for converting code elements to the diagram elements. We also present an example of the method application through a prototype named UNC-Inversor.

KEY WORDS: Reverse engineering; sequence diagram; UML; JAVA; transformation rules.

1. INTRODUCCIÓN

Inmersa en el área de conocimiento de la ingeniería de software, se encuentra la ingeniería inversa como punto de apoyo para los procesos de análisis y diseño propuestos en diferentes métodos de desarrollo (Tilley *et al.*, 1994). Muchos proyectos de software eluden la aplicación de métodos de desarrollo que faciliten el mantenimiento posterior de las aplicaciones resultantes. Este tipo de aplicaciones carece de la documentación necesaria para realizar su adecuado mantenimiento (Fitzgerald, Russo y O'Kane, 2003).

La ingeniería inversa permite subsanar esta deficiencia, al documentar los productos de software a partir del código ejecutable, con el fin de obtener los artefactos de análisis y diseño. La ingeniería inversa es además una herramienta útil que ayuda en el proceso de aseguramiento de la calidad del software, mediante la comparación de diagramas de fases de análisis y desarrollo (Briand, Labiche y Miao, 2003). La ingeniería inversa de software se apoya comúnmente en la generación del diagrama de clases, debido a la importancia de este diagrama en la fase de diseño y su facilidad de generación. Existen también otros diagramas de interés como el de comunicación y el de secuencias, que modelan las características dinámicas de un producto de software.

Actualmente, existen herramientas y propuestas de investigación que realizan el proceso de ingeniería inversa hacia estos diagramas, que aún presentan algunos problemas: la mayoría de las herramientas CASE que realizan este proceso todavía lo hacen de forma experimental e incompleta; las herramientas y propuestas de investigación se enfocan, por lo general, en el diagrama de clases, y finalmente aquellas que intentan obtener otros diagramas (como los de interacción) aún no obtienen los nuevos elementos que define el estándar de UML 2.0.

En este artículo se propone un proceso de conversión de código JAVA a diagrama de secuencias de UML 2.0, en el cual se define un conjunto de reglas de transformación y que se ejemplifica con un caso de estudio en el prototipo de una herramienta que automatiza dicho proceso, denominada UNC-Inversor.

El artículo se organiza así: en la sección 2 se presenta un marco teórico que recopila los conocimientos básicos necesarios para comprender los procesos de ingeniería inversa; en la sección 3 se discuten los antecedentes y trabajos existentes en la materia, prestando especial atención a la ingeniería inversa del diagrama de secuencias a partir de código ejecutable; en la sección 4 se presenta el método para obtener el diagrama de secuencias

de UML 2.0 a partir de código JAVA, detallando las reglas y el método empleado en la transformación; en la sección 5 se presenta un caso de estudio que emplea el prototipo de la herramienta UNC-Inversor, que automatiza el método propuesto; al final, en las secciones 6 y 7, se presentan las conclusiones y el trabajo futuro, respectivamente.

2. MARCO TEÓRICO

La ingeniería inversa es el proceso para analizar componentes y relaciones entre componentes, con el fin de construir descripciones de un sistema en un nivel superior de abstracción (Chikofsky y Cross, 1990). La ingeniería inversa de software permite extraer los detalles estructurales y de comportamiento implícitos en el código de un producto de software y expresarlos de forma estándar, generalmente en diagramas UML (Tonella y Potrich, 2005).

Existen dos tipos de ingeniería inversa. El primero se basa en el código estático, es decir, cuando no se utiliza información de tiempo de ejecución, y el segundo analiza el código en ejecución. Esta diferencia es importante, en tanto establece la dificultad de la inversión y sus alcances. Mientras que la inversión del código estático se utiliza, ante todo, cuando se generan diagramas que describen su estructura, como el diagrama de clases y el de paquetes, la inversión que analiza el código en ejecución se utiliza, primordialmente, cuando se requieren modelos que revelen las características dinámicas del sistema, como el diagrama de objetos (Fowler, 2004). El diagrama de secuencias modela el comportamiento dinámico de un software, lo cual sugiere que se debe invertir usando el código en ejecución. Sin embargo, por ser la secuencia de mensajes la que interesa analizar, es posible utilizar la información estática obtenida en tiempo de compilación para invertir sus características esenciales.

Entre los usos más difundidos de la ingeniería inversa se encuentran (Tonella y Potrich, 2005): la documentación de aplicaciones con deficientes fases de análisis y diseño, con el fin de facilitar el mantenimiento y la corrección de defectos que el software pueda presentar; la adquisición de conocimiento de un producto de software, revelando los detalles estructurales y de comportamiento que se encuentran dispersos u ocultos, ofreciendo mayor independencia entre el producto de software y sus constructores; finalmente, el apoyo al aseguramiento de la calidad del software, al facilitar la comparación de los diagramas de análisis y diseño contra los diagramas generados mediante el proceso de ingeniería inversa.

Asimismo, la ingeniería inversa de software presenta riesgos y problemas que pueden ocasionar que un equipo de trabajo desestime su uso. Entre ellos se encuentran los siguientes:

- Las dificultades asociadas con la complejidad de la inversión y con la comprensión de los diagramas, puesto que en muchos casos se incluye información irrelevante.
- La escasa madurez de los procesos de ingeniería inversa, que implica un bajo refinamiento en la transformación y, por ende, incrementa la susceptibilidad a errores.
- El desconocimiento de la aplicación de la ingeniería inversa en entornos productivos y la desconfianza por el uso de nuevas tecnologías.

El diagrama de secuencias UML 2.0 es un diagrama de interacción que describe cómo un grupo de objetos intercambian mensajes para realizar una operación. Existen muchas características por las cuales el diagrama de secuencias UML 2.0 se usa para modelar el comportamiento dinámico de un sistema. Entre estas, se destaca la facilidad de comprensión, es decir, que no se necesita una explicación rigurosa de la notación de dicho diagrama y la posibilidad de modelar secuencias correspondientes a códigos escritos en lenguajes orientados a objetos, ya que estos se basan en el intercambio de mensajes (Fowler, 2004).

En el diagrama de secuencias, se distribuyen los elementos en dos ejes, en los cuales se ubican los

participantes (eje horizontal u objetual) y los mensajes y fragmentos combinados (eje vertical o temporal). Los elementos del diagrama de secuencias son: participantes, mensajes y fragmentos combinados (Fowler, 2004).

3. ANTECEDENTES

Ahora se dispone de varias herramientas CASE que automatizan los procesos de ingeniería inversa. Algunas de las más representativas son: NetBeans (Myatt, 2007), Together versión 2006 de Borland que se apoya en el entorno Eclipse (Thurston y Kanouse, 2005), Visual Paradigm (Tsang, 2005) y Visual Studio (Levinson y Nelson, 2006). Exceptuando el Visual Studio, que sólo genera unas estructuras de datos desde código, las demás herramientas generan de forma exitosa el diagrama de clases desde código en diferentes lenguajes de programación, pero todas ellas presentan inconvenientes al obtener el diagrama de secuencias, puesto que no obtienen la gran mayoría de los elementos de dicho diagrama pertenecientes al estándar de UML 2.0. Además, NetBeans y Together 2006 para Eclipse, que sí generan algunos de los elementos de UML 2.0, presentan problemas en la identificación del fragmento combinado LOOP, a partir de sentencias DO y DO WHILE.

El creciente interés en esta clase de trabajos se nota también en el trabajo de El-Attar y Miller (2008) quienes, a partir de las descripciones textuales de los casos de uso, aplican ingeniería inversa a la obtención del diagrama de casos de uso, que es uno de los diagramas de comportamiento de UML, el grupo de diagramas del cual descienden los de interacción. Existen propuestas de investigación que estudian la ingeniería inversa del código orientado a objetos a nivel general, como Baxter y Mehlich (1997), que consideran la ingeniería inversa como un caso especial de la ingeniería de software convencional; Cain y McCrindle (1999), que generan una taxonomía de clases a partir de código en C++, y Snavely, Debray y Andrews (2005), que obtienen diagramas de flujo y expresiones en lógica de predicados a partir de código C.

Otras propuestas se encargan únicamente del diagrama de clases, como Guéhéneuc (2004) desde lenguaje C; Sutton y Maletic (2005), Philippow *et al.* (2005), Yeh *et al.* (2007) y Sutton y Maletic (2007) desde C++; Keschenau (2004), Wang y Yuan (2006) y Dong, Lad *y* Zhao (2007) a partir de código JAVA.

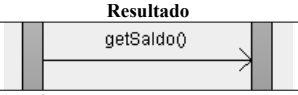
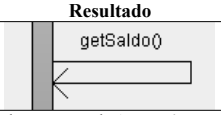
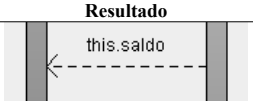
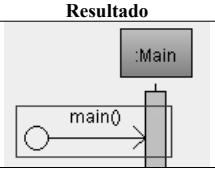
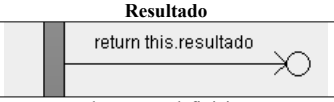
Un tercer grupo de propuestas se encarga de diagramas de interacción, como el de comunicación (colaboración en el estándar de UML 1.4) y el de secuencias. En este grupo se encuentran Kollman y Gogolla (2001), que definen los elementos básicos del diagrama de comunicación, como actores, objetos y mensajes; Briand, Labiche y Miao (2003) y López *et al.* (2006), que obtienen el diagrama de secuencias de UML a partir de código C++, pero que no obtienen los fragmentos combinados definidos en el estándar de UML 2.0; Tonella y Potrich (2005), que realizan una revisión de las técnicas disponibles para la ingeniería inversa de código orientado a objetos y dedican un capítulo especial a los diagramas de interacción, pero sin obtener los elementos nuevos definidos por el estándar de UML 2.0 (como los fragmentos combinados), y Merdes y Dorsch (2006), que obtienen los diagramas de secuencias de UML a partir de JAVA, pero sin incluir los fragmentos combinados. Propuestas como la de Rountev *et al.* (2005) y Rountev y Connell (2005) obtienen los diagramas de secuencias de UML 2.0 a partir de JAVA, incluyendo algunos fragmentos combinados, pero presentan problemas con el fragmento LOOP a partir de las sentencias DO y DO WHILE.

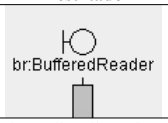
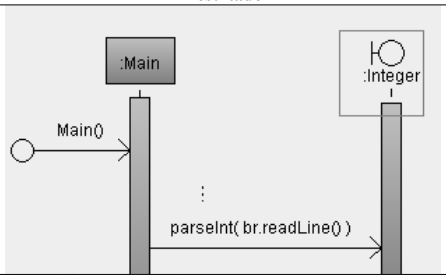
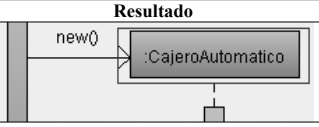
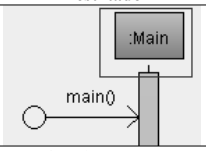
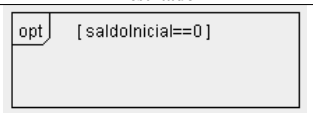
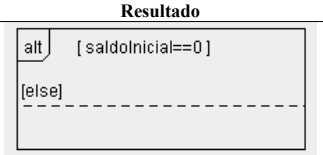
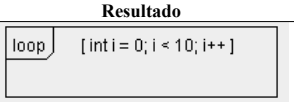
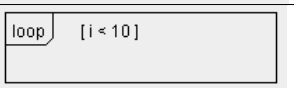
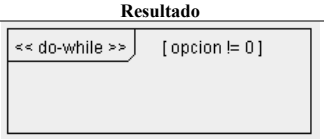
4. MÉTODO PARA LA INGENIERÍA INVERSA DEL DIAGRAMA DE SECUENCIAS DE UML 2.0 A PARTIR DE CÓDIGO JAVA

Como una forma de respuesta a los diferentes problemas enunciados en la sección anterior para la inversión del diagrama de secuencias, el grupo de Lenguajes Computacionales de la Universidad

Nacional de Colombia presenta, en este artículo, un método que permite obtener los diferentes elementos del diagrama de secuencias de UML 2.0, incluyendo los fragmentos combinados OPT, ALT y LOOP, obtenido este último a partir de sentencias DO y FOR. Para las sentencias DO WHILE, se optó por generar un fragmento combinado estereotipado <<DO-WHILE>>, puesto que su semántica es muy diferente de los fragmentos combinados LOOP convencionales. Las reglas de transformación se definen en JAVA, pero se podrían definir equivalencias a cualquier otro lenguaje orientado a objetos. En la tabla 1, se muestran algunas de las reglas de transformación (las demás se omitieron por razones de espacio). En "Condición" se incluye un ejemplo del fragmento que se busca en el código JAVA, en tanto que en "Resultado" se presenta la imagen del diagrama de secuencias resultante en cada caso. Se incluye para cada regla una descripción en lenguaje natural por efectos de claridad de la regla.

Tabla 1. Reglas de transformación de código JAVA al diagrama de secuencias de UML 2.0

No.	REGLAS DE TRANSFORMACIÓN	
1	**Mensaje Normal:** Llamado a un método con el patrón 'nombreObjeto.nombreMetodo()'.	
	Condición	**Resultado**
	miCajero.getSaldo();	
2	**Mensaje Nuevo:** Instancia de un objeto por medio del operador '*new*'.	
	Condición	**Resultado**
	CajeroAutomatico miCajero = new CajeroAutomatico(saldoInicial);	
3	**Mensaje Recurrente:** Llamado a un método de clase 'nombreMetodo()'. Se puede usar el operador '*this*' antecediendo el llamado del método.	
	Condición	**Resultado**
	this.getSaldo();	
4	**Mensaje Retorno:** Retorno de un método que se identifica por la palabra reservada '*return*'.	
	Condición	**Resultado**
	return this.saldo;	
5	**Mensaje Encontrado:** Método del cual empieza el proceso de inversión del código.	
	Condición	**Resultado**
	public class Main { public static void main(String args[]) throws IOException{ } }	
6	**Mensaje Perdido:** Caso especial del mensaje tipo Retorno, que se realiza en el ámbito del método del cual se empieza el proceso de inversión del código.	
	Condición	**Resultado**
	return this.resultado;	
7	**Participante Normal:** Representación de un objeto definido con una clase cuya definición se encuentra en la estructura del código invertido.	
	Condición	**Resultado**
	CajeroAutomatico miCajero = new CajeroAutomatico(saldoInicial);	

<table>
<tr><th>No.</th><th colspan="2">REGLAS DE TRANSFORMACIÓN</th></tr>
<tr><td rowspan="3">8</td><td colspan="2">Participante Límite: Representación de un objeto definido con una clase cuya definición no se encuentra en la estructura del código invertido.</td></tr>
<tr><td>Condición</td><td>Resultado</td></tr>
<tr><td>BufferedReader br;
br = new BufferedReader(isr);</td><td> 
</td></tr>
<tr><td rowspan="3">9</td><td colspan="2">Participante Estático: Representación de un objeto-clase al cual se le realiza llamado a métodos estáticos pertenecientes a éste.</td></tr>
<tr><td>Condición</td><td>Resultado</td></tr>
<tr><td>Integer.parseInt(br.readLine());</td><td> 
</td></tr>
<tr><td rowspan="3">10</td><td colspan="2">Participante Anónimo: Representación de un objeto instanciado como anónimo.</td></tr>
<tr><td>Condición</td><td>Resultado</td></tr>
<tr><td>new CajeroAutomatico(saldoInicial)</td><td> 
</td></tr>
<tr><td rowspan="3">11</td><td colspan="2">Participante “Inicial”: Representación de la clase que contiene el método del cual se inicia el proceso de inversión.</td></tr>
<tr><td>Condición</td><td>Resultado</td></tr>
<tr><td>public class Main {
public static void main(String args[]) throws IOException{
}
}</td><td> 
</td></tr>
<tr><td rowspan="3">12</td><td colspan="2">Fragmento OPT: Estructura de control alternativa ‘if’ que no tiene secuencia ‘else’.</td></tr>
<tr><td>Condición</td><td>Resultado</td></tr>
<tr><td>if (saldoInicial == 0)
System.out.println("Sin fondos");</td><td> 
</td></tr>
<tr><td rowspan="3">13</td><td colspan="2">Fragmento ALT: Estructura de control alternativa ‘if-else’.</td></tr>
<tr><td>Condición</td><td>Resultado</td></tr>
<tr><td>if (saldoInicial == 0)
System.out.println("Sin fondos");
else
System.out.println("Con fondos");</td><td> 
</td></tr>
<tr><td rowspan="3">14</td><td colspan="2">Fragmento LOOP: Estructuras de repetición o bucles ‘for’ y ‘while’.</td></tr>
<tr><td>Condición</td><td>Resultado</td></tr>
<tr><td>for(int i = 0; i < 10; i++) {
System.out.println("contador:"+i);
}</td><td> 
</td></tr>
<tr><td>while(i < 10) {
System.out.println("contador:"+i);
i++;
}</td><td> 
</td></tr>
<tr><td rowspan="3">15</td><td colspan="2">Fragmento <<<i>do-while</i>>>: Fragmento combinado estereotipado que representa la estructura de repetición ‘<i>do-while</i>’.</td></tr>
<tr><td>Condición</td><td>Resultado</td></tr>
<tr><td>do{
System.out.println(" 1. Consultar saldo");
System.out.println(" 0. Terminar");
} while(opcion != 0);</td><td> 
</td></tr>
</table>

El método enunciado se programó en el prototipo UNC-Inversor. La figura 1 ilustra los diferentes módulos que componen el UNC-Inversor y también ilustra el proceso de inversión (módulo de inversión).

El proceso de inversión comienza luego de hacer un análisis de código para identificar paquetes, clases, métodos y atributos y para ello es indispensable que se cree un proyecto que contenga los archivos fuente. Enseguida, se selecciona el método inicial de inversión y se continúa con la obtención de la representación computacional del diagrama de secuencias en lenguaje XML para, al final, presentarle al usuario el gráfico correspondiente a la inversión del método seleccionado como inicial. La inversión se puede hacer de dos formas diferentes: en la primera (forma no recurrente) se resuelven única y exclusivamente los mensajes pertenecientes al método elegido para invertir, mientras que en la segunda forma (recurrente) se ofrece una mayor flexibilidad y se resuelven los mensajes ocasionados por los propios mensajes contenidos en el método de inversión inicial.

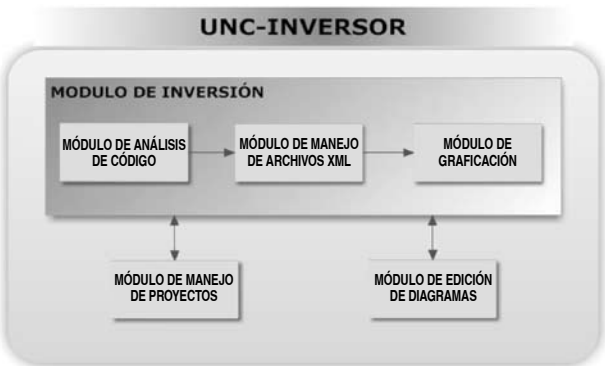


Figura 1. Módulos que componen UNC-Inversor

En esta última forma, la resolución de los mensajes permite condensar en un solo diagrama el comportamiento del sistema desde una perspectiva más global. Es de anotar que ninguna de las herramientas CASE comerciales resuelve los mensajes de forma recurrente. El UNC-Inversor detecta los métodos recurrentes que se encuentran dentro del código que se invierte, de manera que no se entra en un bucle infinito de recurrencia. El UNC-Inversor maneja este proceso con la ayuda de tres módulos: inversión, edición de diagramas y manejo de proyectos.

El módulo de Inversión es el núcleo de UNC-Inversor, que integra las diferentes funciones para asumir la complejidad que requiere el proceso de inversión. Este módulo se compone a su vez de tres módulos más pequeños: análisis de código, manejo de archivos XML y graficación. El módulo de análisis de código se encarga de recorrer los paquetes y clases de los diferentes archivos que forman un proyecto de programación en JAVA y obtiene estructuras de datos que representen el código por analizar en memoria, para formar estructuras para las clases incluyendo sus métodos, atributos, importaciones y clases interiores. Este módulo también se encarga de analizar el método inicial para crear la representación computacional del diagrama de secuencias. El módulo de manejo de archivos XML se encarga de generar la estructura para los archivos XML correspondientes a los diagramas de secuencias creados por el módulo de análisis de código; tiene la capacidad de guardarlos en disco y leerlos para su posterior análisis. El módulo de graficación se encarga de dibujar y situar los diferentes elementos de un diagrama de secuencias que se encuentra almacenado en un archivo XML.

El módulo de edición de diagramas le permite al UNC-Inversor editar el diagrama de secuencias, y lo convierte en una herramienta CASE convencional de modelado. Los diagramas invertidos también son susceptibles de modificación, pues permiten el manejo estético que el usuario requiera. Este módulo permite guardar los diagramas de secuencias generados en el proceso de inversión o editados por el usuario, en formato binario, como una imagen en formato jpg o como un archivo XML compatible con el usado por el módulo de graficación.

El módulo de manejo de proyectos absorbe la complejidad asociada con el manejo de proyectos y crea un entorno que facilita la inversión. De esta manera, permite manejar varios proyectos simultáneamente y crear las estructuras de archivos y carpetas necesarias para crear, abrir o editar un proyecto de inversión.

5. CASO DE ESTUDIO EN EL PROTOTIPO UNC INVERSOR

El siguiente es un ejemplo de conversión de código JAVA en el diagrama de secuencias de UML 2.0, a través de la herramienta UNC-Inversor.

El método que se analiza pertenece a la clase 'CajeroAutomatico', la cual hace parte de un código JAVA que emula el comportamiento de un cajero automático bancario. La figura 2 muestra el fragmento de código, que incluye el método analizado 'Depositar' y el método referenciado en el interior de éste 'sumarAlSaldo'. Por efectos de claridad del proceso, se omiten los demás métodos de la clase 'CajeroAutomatico'. Las reglas de transformación que se emplean para la inversión del fragmento de código de la figura 2 se muestran en la tabla 1. Usando la herramienta UNC-Inversor para realizar la inversión de este código, se obtiene el resultado que se presenta en la figura 3. En esta figura, se puede apreciar la secuencia de mensajes resultante del análisis del método 'Depositar'. En el ejemplo de inversión, se pueden observar las siguientes secuencias:

```
1  package paqueteCajero;
2  import java.io.BufferedReader;
3  import java.io.IOException;
4
5  public class CajeroAutomatico{
6      double saldo;
7      /* No se muestra el cuerpo de la clase CajeroAutomatico que incluye el
8       * constuctor y demás métodos necesarios para su implementación */
9      public void Depositar(BufferedReader br) throws IOException{
10         System.out.println("*          DEPOSITAR EN LA CUENTA              *");
11         System.out.println("*---------------------------------------------*");
12         System.out.print("Escriba la cantidad a depositar: " );
13         Double cantidadDouble = new Double( br.readLine() );
14         double cantidad = cantidadDouble.doubleValue();
15         if ( cantidad <= 0 ){
16             System.out.println( "Error!! Cantidad negativa." );
17             return;
18         } else {
19             sumarAlSaldo(cantidad);
20             System.out.println( "El nuevo saldo de la cuenta es: " + this.saldo );
21             return;
22         }
23     }
24
25     private void sumarAlSaldo(double d) {
26         this.saldo += d;
27     }
28 } // fin de la clase CajeroAutomatico
```

Figura 2. Fragmento de código perteneciente a la clase 'CajeroAutomatico'

- El participante 'CajeroAutomatico' representa la clase que contiene el método inicial de inversión. Se refiere a la regla 11 de la tabla 1 y se ve, en el código, en la línea 5.
- El mensaje 'Depositar()' es el método inicial de inversión, por lo cual se dibuja como un mensaje tipo *Encontrado*. Se asocia con la regla 5 de la tabla 1 y se ve en la figura 2 del código en la línea 9.
- Las secuencias asociadas a las líneas 10, 11, 12, 16 y 20 son mensajes 'System.out.println' y se refieren a interacciones con el usuario por medio la consola, por lo que se dibujan como mensajes recurrentes. Estos se asocian con la regla 3 de la tabla de reglas de transformación. De esta misma forma, se puede analizar el mensaje 'sumarAlSaldo()', que se encuentra en la línea 19 del código de la figura 2.
- En la línea 13 del código, se encuentra la secuencia de instanciación de un nuevo objeto, lo cual se representa por medio de un mensaje tipo *Nuevo* y la aparición de un participante tipo *Límite,* debido a que la clase *Double* no hace

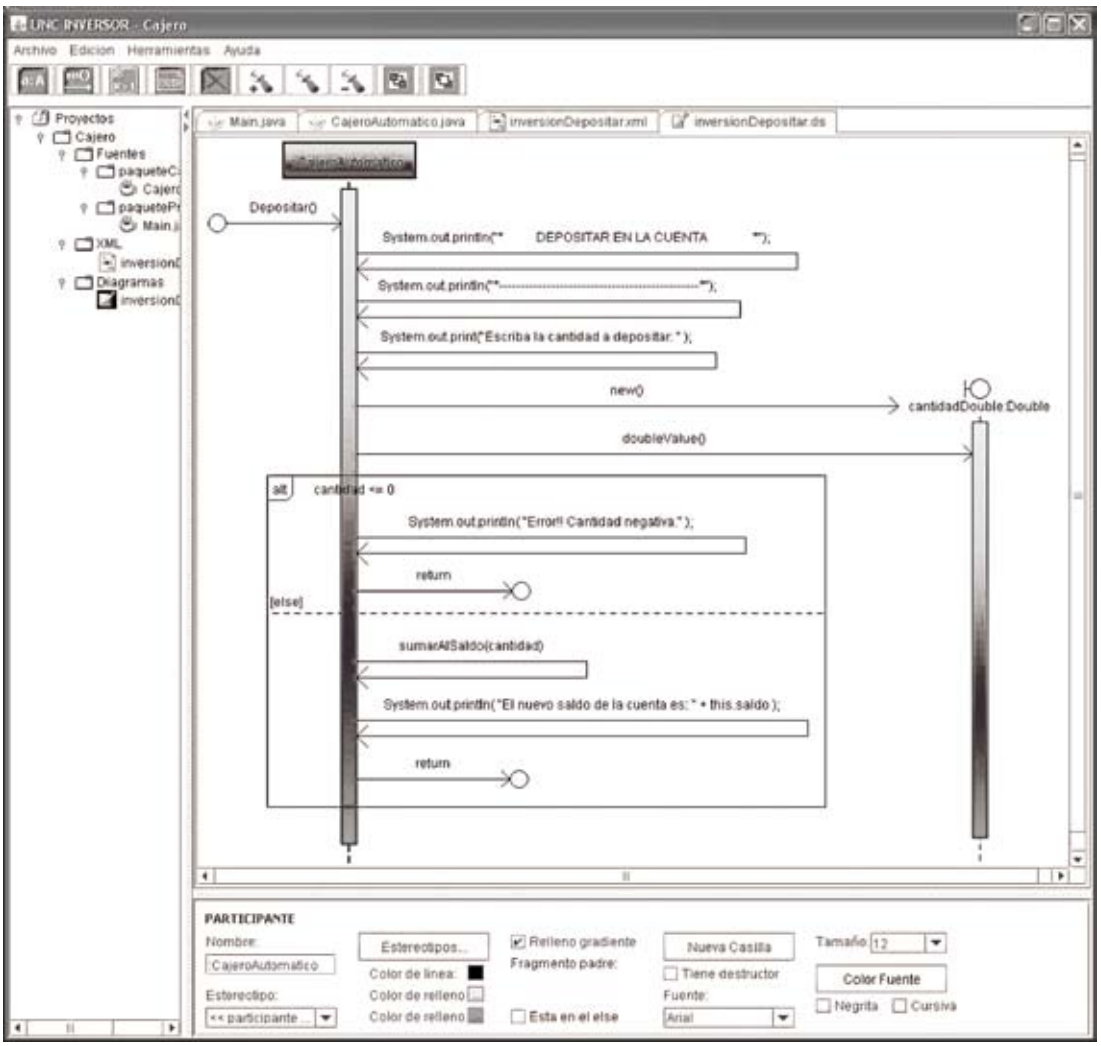


Figura 3. Resultado de la inversión del código de la figura 2 con UNC-Inversor

parte de la estructura del código invertido. Estas secuencias, se transforman aplicando las reglas 2 y 8 de la tabla de reglas de transformación.

- El mensaje 'doubleValue()' es de tipo *Normal,* se representa en la línea de código 14 y se invierte al aplicar la regla 1 de la tabla 1.
- La estructura de control alternativo *'if-else',* que abarca las líneas 15 a 23 del código, se transforma mediante la aplicación de la regla 13 de la tabla de reglas de transformación. En la figura 2, se aprecia su representación como el fragmento combinado ALT, el cual contiene todos los mensajes enviados dentro de la estructura de control.
- Los mensajes tipo *Retorno* se asocian con las líneas 17 y 21 del código invertido y representan el envío de un método *return* aplicando la regla 4 de la tabla 1.

6. CONCLUSIONES

La ingeniería inversa es un proceso multipropósito que contribuye a mejorar la documentación que se genera mediante la ingeniería de software. Dentro de la ingeniería de software, la ingeniería inversa hacia diagramas de secuencias permite la comprensión de los detalles dinámicos de un método por medio de la visualización del intercambio de mensajes en el tiempo. Este tipo de ingeniería inversa aún presenta problemas que motivan la investigación en este campo, pues, por lo general, los esfuerzos se encaminaron a la ingeniería inversa del diagrama de clases y los trabajos que se ocupan de los diagramas de interacción todavía presentan problemas, como la falta de identificación de los fragmentos combinados y los errores que se cometen al invertir sentencias DO y DO WHILE.

En este artículo se definió un método para obtener el diagrama de secuencias de UML 2.0 a partir de código JAVA, como una forma de superar algunas de las limitaciones encontradas en este proceso. Este método permite automatizar el proceso de ingeniería inversa a diagrama de secuencias de UML 2.0, mediante la aplicación de reglas de transformación que establecen una relación unívoca entre los elementos del diagrama y patrones de código, que se ejemplifica en JAVA, pero se podrían equiparar a cualquier lenguaje orientado a objetos.

La inversión del diagrama de secuencias, aunque es una tarea compleja que teóricamente se debe realizar con información de tiempo de ejecución y de tiempo de compilación, se puede simplificar para realizar la inversión tomando como base la información de tiempo de compilación, para obtener de esta manera la mayor cantidad de secuencias disponibles en el código.

7. TRABAJO FUTURO

Existen aspectos del código que el método definido en este artículo no contempla, como algunas operaciones que se corren en tiempo de ejecución, tales como polimorfismos y asociaciones tardías. Estos aspectos, al igual que la mayoría de los fragmentos combinados predefinidos en UML no los invierte este método, por no tener una representación lógica en el código. Asimismo, los esfuerzos subsecuentes a este trabajo se enfocarán en inversiones de más diagramas y desde más lenguajes, contemplando la mayor cantidad de detalles, con el objetivo de que el conjunto de documentos invertidos conformen entre todos una representación completa de un producto de software determinado.

Es importante, igualmente, la definición de un mecanismo de comparación que permita determinar la coherencia entre los diagramas de secuencia construidos en fases de análisis y diseño y los obtenidos a partir del proceso de inversión. Esta comparación permite establecer qué tan similar es lo construido con lo deseado. Además, si se supone que los diagramas de fase de análisis y diseño son correctos, se pueden realizar extrapolaciones para que la comparación determine si el producto de software cumple con las necesidades del interesado

o no. Este criterio de comparación también se debe definir para los otros diagramas que se obtengan mediante el proceso de inversión.

REFERENCIAS

Baxter, D. and Mehlich, M. (1997). Reverse engineering is reverse forward engineering. Proceedings of the 4th Working Conference on Reverse Engineering, Amsterdam (Países Bajos).

Briand, L. C.; Labiche, Y. and Miao, Y. (2003). Towards the reverse engineering of UML sequence diagrams. Proceedings of the 10th Working Conference on Reverse Engineering, Victoria (Canada), pp. 57-66.

Cain, J. W. and McCrindle, R. J. (1999). Software visualisation using C++ Lenses. Proceedings of the 7th International Workshop on Program Comprehension, Pittsburgh (Estados Unidos).

Chikofsky, E. and Cross, J. (1990). Reverse engineering and design recovery: A Taxonomy. IEEE Software, 7(1): 13-17.

Dong, J.; Lad, D. and Zhao, Y. (2007). DP-Miner: Design pattern discovery using Matrix. Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, Tucson (Estados Unidos). p. 371-380.

El-Attar, M. and Miller, J. (2008). Producing robust use case diagrams via reverse engineering of use case descriptions. Software and System Modeling 7:67-73.

Fitzgerald, B.; Russo, N. and O'Kane, T. (2003). Software development method tailoring at Motorola. Communications of the ACM, 46(4):64-70.

Fowler, M. 2004. UML distilled: A brief guide to the standard object modeling language. Addison Wesley, Boston.

Guéhéneuc, Y. (2004). A reverse engineering tool for precise class diagrams. Proceedings of the 2004 Conference on the Centre for Advanced Studies on Collaborative Research, Markham (Canadá), pp. 28-41.

Keschenau, M. (2004). Reverse engineering of UML specifications from Java programs. Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, Vancouver (Canadá), pp. 326-327.

Kollman, R. and Gogolla, M. (2001). Capturing dynamic program behaviour with UML collaboration diagrams. Proceedings of the 5th Conference on Software Maintenance and Reengineering, Lisboa (Portugal).

Levinson, J. and Nelson, D. 2006. Pro Visual Studio 2005 team system. Apress, Berkeley.

López, M.; Alfonzo, A.; Pérez, J.; González, J. and Montes, A. (2006). A metamodel to carry out reverse engineering of C++ Code into UML Sequence Diagrams. Proceedings of the Electronics, Robotics and Automotive Mechanics Conference, Cuernavaca (México).

Merdes, M. and Dorsch, D. (2006). Experiences with the development of a reverse engineering tool for UML sequence diagrams: A case study in modern Java development. Proceedings of the 4th International Symposium on Principles and Practice of Programming in Java, Mannheim (Alemania), pp. 125-134.

Myatt, A. 2007. Pro NetBeans IDE 5.5 Enterprise Edition. Apress, New York.

Philippow, I.; Streitferdt, D.; Riebisch, M. and Naumann, S. (2005). An approach for reverse engineering of design patterns. Software and System Modeling 4(1): 55-70.

Rountev, A. and Connell, B. (2005). Object naming analysis for reverse-engineered sequence diagrams. Proceedings of the 27th International Conference on Software Engineering, St. Louis (Estados Unidos), pp. 254-263.

Rountev, A.; Volgin, O. and Reddoch, M. (2005). Static control-flow analysis for reverse engineering of UML sequence diagrams. Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, Lisboa (Portugal), pp. 96-102.

Snavely, N.; Debray, S. and Andrews, G. (2005). Unpredication, unscheduling, unspeculation: reverse engineering Itanium executables. IEEE Transactions on Software Engineering, 31(2): 99-115.

Sutton, A. and Maletic, J. I. (2005). Mappings for accurately reverse engineering UML class models from C++. Proceedings of the 12th Working Conference on Reverse Engineering, Pittsburgh (Estados Unidos), pp. 175-184.

Sutton, A. and Maletic, J. I. (2007). Recovering UML class models from C++: a detailed explanation. information and software technology, 49: 212-219.

Thurston, M. and Kanouse, G. 2005. Eclipse distilled: a programmer's first look at Eclipse. Addison Wesley Professional, Indianapolis.

Tilley, S.; Wong, K.; Storey, M. and Muller, H. (1994). Programmable reverse engineering. International Journal of Software Engineering and Knowledge Engineering, 4(4):501-520.

Tonella, P. and Potrich, A. 2005. Reverse engineering of object oriented code. Springer, New York.

Tsang, C. 2005. Object-oriented technology from diagram to code with Visual Paradigm for UML. McGraw-Hill, Maidenhead.

Wang, X. and Yuan, X. (2006). Towards and AST-based approach to reverse engineering. Proceedings of the Canadian Conference on Electrical and Computer Engineering, Ottawa (Canadá), pp. 422-425.

Yeh, D.; Sun, P.; Chu, W.; Lin, C. and Yang, H. (2007). An empirical study of a reverse engineering method for the aggregation relationship based on operation propagation. Empirical Software Engineering 12:575-592.