Analizador de Ensamblados Orientados a Objetos para el Mantenimiento de Software

Luis Felipe Fernández[1]; Javier Galindo Ruiz[2] y Raúl Alejandro Vargas Acosta[3]

[1]lfernand@uacj.mx 2. javiergalindor@hotmail.com 3. alejandrovargas123@hotmail.com

Resumen

Uno de los principales problemas que presentan el mantenimiento de software, se deriva de no contar con la documentación adecuada del sistema; cuando se hace necesario modificar código, con la intención de corregir o agregar funcionalidad, la posibilidad de que se induzcan efectos secundarios o colaterales es alta. Entendemos por efectos secundarios aquellos que se presentan después de haber realizado el mantenimiento (correctivo, perfectivo o adaptativo) y que generalmente se manifiestan en una afectación de la funcionalidad del sistema.

Cuando la documentación disponible no refleja claramente el código fuente, es difícil conocer cómo un cambio en éste, afecta otras operaciones de otras clases. Esto da como resultando que el sistema presente un comportamiento anormal en su funcionalidad. Para solventar este problema, se creó AEMS (Analizador de Ensamblados para el Mantenimiento de Software por sus siglas en español), el cual analiza el archivo ejecutable del sistema que está bajo mantenimiento y proporciona diagramas de dependencia interactivos. Esto ayuda a que el la persona que da el mantenimiento observe no solamente los métodos dependientes directamente de algún método, sino también aquellos que son afectados indirectamente a causa de un cambio en él. AEMS ofrece también diferentes métricas como lo son complejidad y el acoplamiento, entre otras. De esta manera, esta herramienta proporciona información que auxilia al encargado de dar mantenimiento a algún sistema. AEMS fue desarrollado para aplicaciones que se ejecutan en la plataforma .Net de Microsoft.

Introducción

El mantenimiento del software es sin duda una etapa inevitable en el ciclo de vida del software (fig. 1) y en la mayoría de los escenarios no es un trabajo trivial; uno de los problemas potenciales que conlleva esta etapa es la aparición de los efectos secundarios. El modificar la estructura de un sistema de alguna manera afecta tanto a los elementos transformados como a todos aquellos que se hallen relacionados a estos. Una manera de minimizar el problema de los efectos secundarios consiste en diseñar la estructura del sistema de tal forma que el acoplamiento entre sus elementos sea el menor posible.

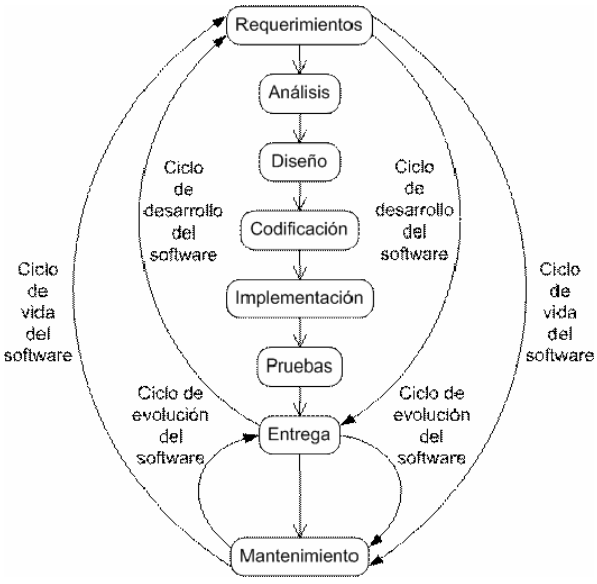

Fig. 1. Ciclo de vida del software (Bañuelos 2006)

Aunque existen distintos tipos de efectos secundarios, tales como los efectos secundarios en el código, en los datos o incluso en la documentación, en el presente trabajo hace énfasis en la prevención de los efectos secundaros en el código. Más específicamente, se buscó predecir qué funciones se verán afectadas después de modificar una en particular. Es decir, el problema consiste en encontrar qué funciones dependen de otra. La tentación inicial es pensar que ya existen herramientas que ayudan automáticamente a hacer esto y que se agrupan bajo el concepto de "Refactoring" o Reestructuración; por definición, este concepto se refiere a rehacer el código con el fin de lograr que sea más legible o mejorar el diseño de manera que sea más fácil darle mantenimiento [Refactoring]. La mayoría de estas herramientas soportan bien cambios sencillos, pero cuando el asunto no es trivial, hay que darles una manita.

Para dar solución a este problema se propuso la creación de una herramienta de software que le permitiera al desarrollador visualizar gráficamente las dependencias entre las funciones del sistema. A esta herramienta también se le incorporó la funcionalidad de mostrar distintas métricas del sistema, útiles para el análisis durante el mantenimiento de software.

Desarrollo

Antes de comenzar con el desarrollo de la solución, es necesario definir algunos puntos importantes como:

¿En qué lenguaje de programación deberá estar hecho el sistema para que pueda ser analizado?

¿Cuáles son los elementos del sistema que se analizarán?

¿Qué opciones de análisis se tienen?

¿Cómo se mostrará al usuario el resultado del análisis?

Es evidente que existe una gran variedad de lenguajes de programación, cada uno con alguna característica o paradigma que lo distingue. Pero para este caso el lenguaje debe ser orientado a objetos, ya que este paradigma permite definir claramente cada elemento que forma parte de un sistema y por lo tanto es posible realizar un análisis de dependencias entre sus elementos. Sin embargo, son muchos los lenguajes que pertenecen al paradigma orientado a objetos, entre ellos: C++, C#, Visual Basic .Net, J#. Inicialmente se pensó en que la herramienta permitiera el análisis de sistemas construidos en C#, pero como se verá más adelante, es posible el análisis de sistemas hechos en distintos lenguajes.

Una vez definido el paradigma sobre el cual se estará trabajando, es posible definir claramente los elementos que se analizarán. En este caso se deberán analizar las clases que componen a un sistema, y más específicamente, los métodos o funciones de dichas clases. Dentro de las funciones es necesario examinar cada instrucción y determinar cuáles instrucciones corresponden a llamadas de función. Cada llamada implica una dependencia de la función que hace la llamada hacia la función destino.

```
Class sum{
//La clase contiene más métodos.
  public  int GetSum()
{
    return x+y;
}
//La clase contiene mas campos.
  private int x;
  private int y;
};
```

```
Class operations{
//La clase contiene más métodos y
//campos.
  private void printResults()
{
    int res;
    sum obj = new sum();
    res=obj.GetSum();
    cout<<endl<<"La suma es: "
```

Figura 2. Ejemplo de dependencia de métodos. En este caso el método "printResults" depende del método "GetSum"

En cuanto al método para realizar el análisis, inicialmente se intentó desarrollar un *parser*, que permitiera crear un árbol que representara la estructura del sistema. Para ello fue necesario conseguir la gramática del lenguaje a analizar, que en este caso era C#. Pero fue desde esta parte donde empezaron a surgir problemas para su desarrollo. En primer lugar, la gramática de C# no estaba completa del todo. Aún cuando estaba basada en la especificación proporcionada por Microsoft, resultó difícil, y en ciertas partes imposible, convertir dicha especificación a la requerida por *GoldParser*, herramienta que se estaba utilizando para desarrollar el *parser*. Aún así se creó uno sencillo, que permitiera el análisis de programas simples. Aunque el *parser* había sido creado, su funcionamiento era bastante limitado, sobre todo debido a que el análisis requerido iba más allá de un simple análisis sintáctico. Era necesario analizar más a profundidad el significado de algunos elementos para poder obtener la información requerida.

Debido a estos problemas, fue necesario buscar otra alternativa para lograr un análisis más completo, sin la necesidad de crear prácticamente otro compilador para el lenguaje. Una gran ventaja que vino a dar solución a este problema fue que al ser C# un lenguaje creado para cumplir con las especificaciones del .Net, es posible aprovechar las características de los programas compilados para esta plataforma de Microsoft. La característica que facilita el análisis de la estructura del programa es la inclusión de los llamados "metadatos" en cada uno de los archivos compilados. Los metadatos proporcionan la información necesaria para identificar cada elemento del programa, facilitando enormemente el proceso de análisis.

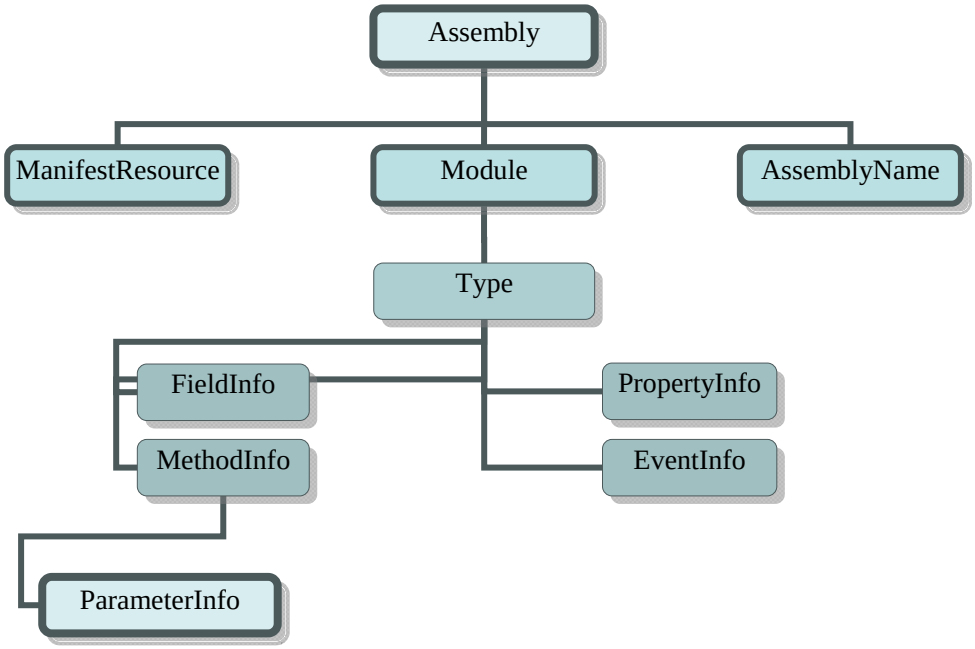

Figura 3. Jerarquía de los Metadatos desde el punto de vista del API *Reflection*

Pero incluso con la extracción de los metadatos fue necesario el uso de una librería auxiliar: *PERWAPI* (por sus siglas *Portable Executable Reader / Writer Application Programming Interface*) desarrollada por John Gough y Diane Corney en Queensland University of Technology. Esta librería permite leer y escribir archivos ejecutables creados para el .Net. Al leer un archivo mediante esta librería es posible determinar los tipos de instrucción que contiene cada elemento del programa, lo que resulta difícil con el uso del *API* del .Net solamente, ya que éste devuelve las instrucciones en código binario y por lo tanto habría que tener un amplio conocimiento del código intermedio de los ensamblados del .Net.

De esta manera, ahora se estará analizando el código compilado del programa y no el código fuente del mismo. Y ya que el código compilado para el .Net es en realidad un código intermedio generado por distintos compiladores, ahora no es necesario limitarse al uso del lenguaje C#, sino que también será posible el uso de otros lenguajes, como el Visual Basic .Net y el J#.

En cuanto a las métricas que proporciona la herramienta, se contemplaron sólo algunas de ellas. Estas fueron: complejidad, acoplamiento, número de instrucciones, métodos ponderados por clase y carencia de cohesión.

La complejidad calculada se refiere a la "complejidad ciclomática", propuesta por McCabe [Pianttini, 2001], la cual está definida como la cantidad de caminos lineales que contiene un programa. Para su cálculo fue necesario buscar las estructuras de control dentro de los métodos del programa y en base a ellas, determinar el número mínimo de caminos posibles.

El número de instrucciones se calcula de forma sencilla, al igual que los métodos ponderados por clase, los cuales se refieren simplemente al número total de métodos contenidos para cada clase.

Para la carencia de cohesión o *LCOM* [Pianttini, 2001] (por sus siglas en inglés *Lack of Cohesion in Methods*) se usaron distintas variantes de la misma:
LCOM1: define la carencia de cohesión en base al uso compartido de variables.
LCOM2 y LCOM3: al igual que LCOM1, se basa en el uso compartido de variables, con la diferencia de que estas variantes proporcionan una medida más confiable.
LCOM4: en esta variante la carencia de cohesión se calcula no sólo en base al uso compartido de variables, sino que también se incluye la relación "en cadena" de un método con otro.

Funcionalidad del sistema

Con el propósito de facilitarle al desarrollador la búsqueda de efectos secundarios debido al mantenimiento, se desarrolló un sistema que genera diagramas de dependencia para los métodos de las clases, además de que muestre distintas métricas de los mismos. La funcionalidad del sistema se muestra por medio de casos de uso, figura 4.

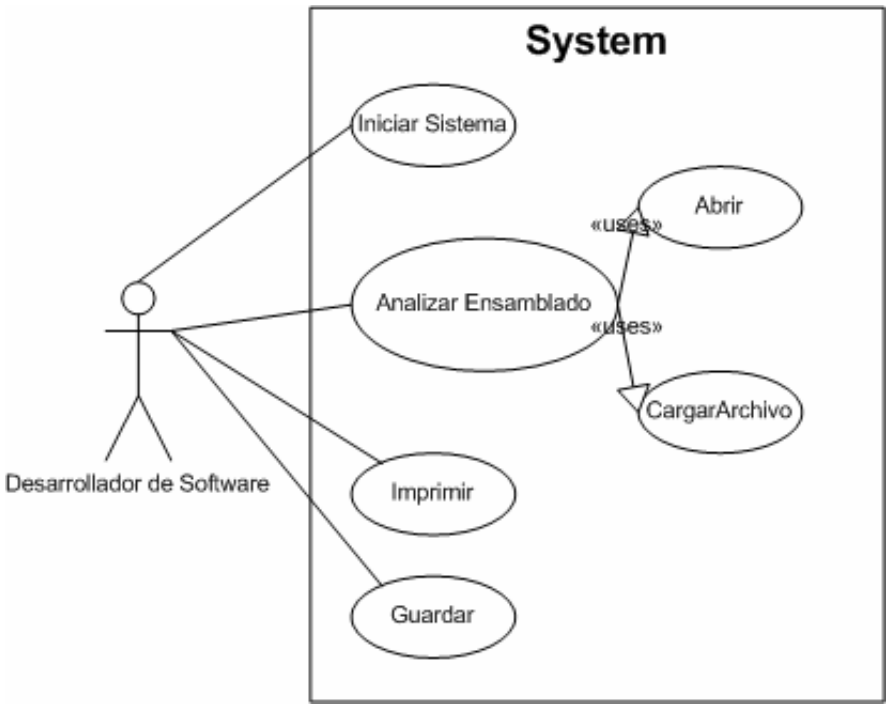

Figura 4. Diagrama de casos de uso de AEMS

La figura 5 muestra un diagrama con las actividades principales que realizará AEMS.

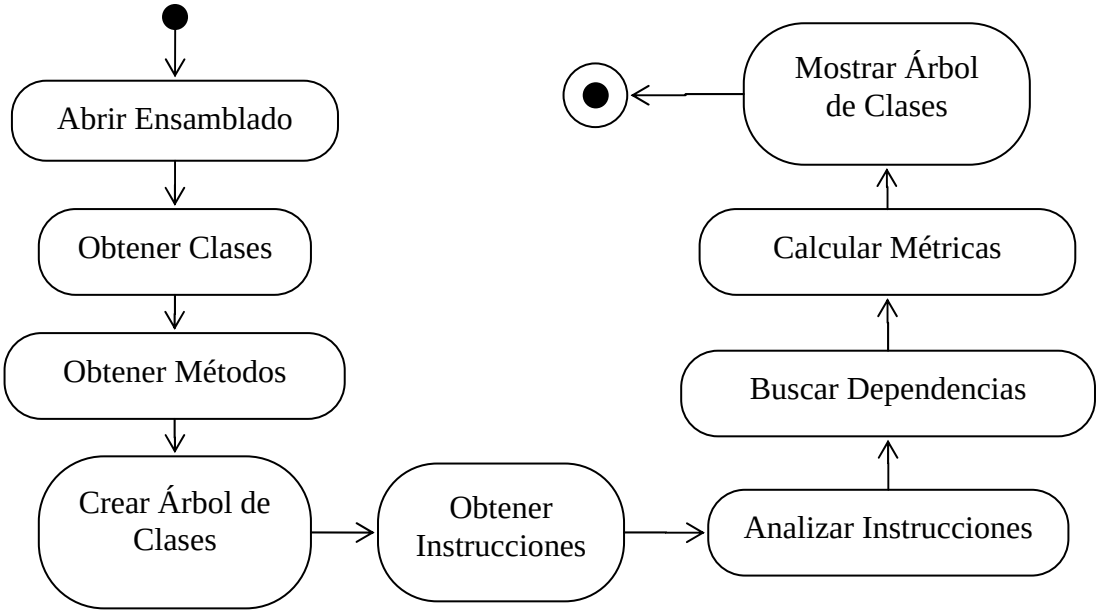

Figura 5. Diagrama de casos de uso de AEMS

Un diagrama de dependencia permitirá ver los métodos que dependen de una sola función, logrando así, informar a la persona que quiere darle mantenimiento al sistema de que funciones se verán afectadas.

En lo que respecta a la interfaz del usuario, se usaron diagramas semejantes a los diagramas de clase, en donde se muestran las funciones dependientes resaltadas con distinto color de fondo y unidas a la clase de la cual dependen por una flecha punteada, figura 6.

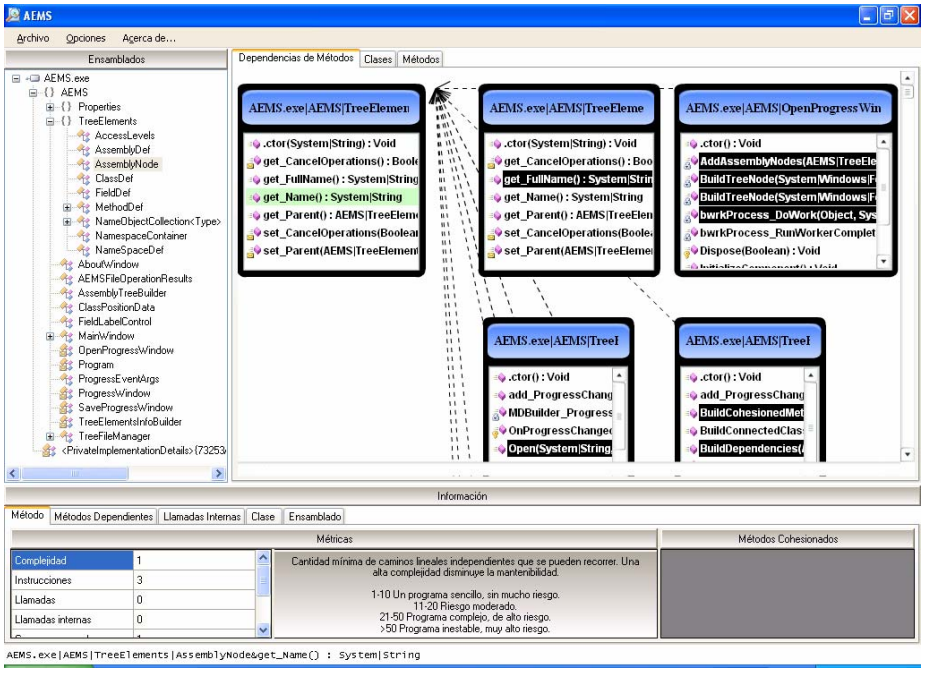

Figura 6. Interfaz de AEMS

Básicamente, el usuario sólo necesita abrir un archivo de ensamblado, es decir un archivo compilado para la plataforma .Net, el cual puede ser un archivo con extensión .exe o .dll (no todos los archivos con estas extensiones son ensamblados del .Net). A continuación el sistema analizará la estructura del archivo y la mostrará en forma de árbol para que el usuario pueda examinar las clases que contenga. La información de cada clase será desplegada junto con las funciones que contenga. La interfaz permite a el usuario seleccionar interactivamente una función distinta, arrastrar la clase a una nueva posición o seleccionar una nueva clase desde un árbol de clases. Cada vez que se selecciona un elemento, el sistema muestra las métricas correspondientes del mismo.

Conclusión

Como se ha mencionado anteriormente, el mantenimiento es una parte presente en todos los sistemas de una forma u otra. Hay varias razones por la cuales se debe de dar mantenimiento al mismo, ya sea por nuevos requerimientos de la empresa, para mejorar el rendimiento del mismo o para corregir errores que se presenten en él, entre otras. Sin embargo, sin importar el motivo, cuando se da mantenimiento a algún sistema se pueden ocasionar errores que no hayan sido tomados en cuenta, tal es el caso de la modificación en un método. Para ayudar a minimizar este error, se analizaron dos propuestas. En la primera que se analizó, la cual fue la creación de un *parser* utilizando la gramática del lenguaje del sistema a analizar, se presentaron varios problemas difíciles de resolver. Debido a esto se decidió abandonar esta solución. Aunque este fracaso era un aparente impedimento para la finalización y éxito del proyecto, se encontró otra forma de solucionar el problema, esto fue mediante *PERWAPI*. Gracias a esta librería se pudo plantear una nueva solución, adaptándola a las necesidades y requerimientos del sistema. Una vez planteada esta nueva solución, se pudo completar el proyecto con más ventajas de las pensadas inicialmente. El sistema a analizar puede ser desarrollado en cualquier lenguaje soportado por la plataforma .Net de Microsoft, tal como C#, C++ o Visual Basic, entre otros. El trabajo en extenso se encuentra en Galindo-Vargas 2007

Referencias y bibliografía

Aivosto Oy, Aivosto, *Cohesion metrics*, http://www.aivosto.com/project/help/pm-oo-cohesion.html

Bañuelos Pedro, 2006, *Taxonomía para procesos de desarrollo de software*, Universidad Autónoma de Ciudad Juárez Instituto de Ingeniería y Tecnología, Departamento de Ingeniería Eléctrica y Computación, proyecto de titulación.

Galindo J., Vargas A., 2007, *Analizador de Ensamblados para el Mantenimiento de Software*, Reporte Técnico de Investigación, Universidad Autónoma de Ciudad Juárez Instituto de Ingeniería y Tecnología, Departamento de Ingeniería Eléctrica y Computación, proyecto de titulación.

Piattini Mario, 2001 et al., *Mantenimiento del Software. Modelos, técnicas y métodos para la gestión del cambio*, Alfaomega, México, D.F.

Refactoring: http://www.refactoring.com/index.html

Samudra Gupta, Java(TM) Boutique, *Metrics for Object Oriented Software Development*, http://javaboutique.internet.com/tutorials/codemetrics/

SEI 2007, Software Engineering Institute of Carnegie Mellon, *Cyclomatic Complexity*, 6 de mayo de 2007, http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html