Rastreo de la ejecución de sistema heredado usando programación orientada a aspectos para facilitar su mantenimiento

Tracing of a legacy system execution using aspect oriented programming to facilitate maintenance

Alejandro Domingo Velázquez-Cruz [1]*, Emmanuel Mendoza-Escobar [2], Antonio Rodríguez-Cabrales, [3] Ulises Jesús López-Maldonado [2], Sergio David Ixmatlahua-Díaz [2], Roque Manuel Rueda-Anastacio [2]

[1] Sensa Control Digital.
Av. Bravo Oriente 93, col. Centro. Torreón, Coahuila, México. CP 27000
[2] Departamento de Posgrado e Investigación, Instituto Tecnológico de Orizaba.
Av. Instituto Tecnológico 852, col. Emiliano Zapata. Orizaba, Veracruz, México. CP 94300
[3] Instituto Tecnológico de la Laguna,
Boulevard Revolución y Calzada Cuahutémoc, col. Centro.
Torreón, Coahuila, México. CP 27000
* Correo-e: lap.alejandro@hotmail.com

PALABRAS CLAVE:

sistemas heredados, rastreo, programación orientada a aspectos

RESUMEN

Este artículo revisa los beneficios de la programación orientada a aspectos para rastrear la ejecución de los procesos internos de un sistema heredado. Para poder alcanzar dicho propósito es necesario separar la implementación del rastro en una abstracción diferente aislando su código del resto del sistema. Esta aproximación permite enlazar el sistema heredado con las funcionalidades de monitoreo, y es necesario utilizar una capa intermedia que permita enlazarlos. Esta nueva abstracción llamada aspecto maneja solamente lo relacionado con el rastreo y no requiere modificar el código fuente del sistema, con lo cual se reducen costos y dinero para el análisis, diseño y desarrollo.

KEYWORDS:

aspect oriented programming legacy systems software reenginering

ABSTRACT

This work reviews the benefits of aspect oriented programming to enable to track the execution of a legacy system inner processes, in order to achieve such purpose is necessary to separate tracing handling into a separate abstraction isolating its code from the rest of the system. To bond the legacy system with monitor function a lities it is necessary to use an intermediate layer that allows to link them. This new abstraction call aspect handles only the tracing issue and it does not require to modify the source code of the system, reducing cost, as well as time to analysis, design and development.

1 INTRODUCCIÓN

En la actualidad algunas empresas poseen sistemas heredados desarrollados a través de largos y caros procesos que cubren parcialmente sus necesidades de información. Durante los ciclos de mantenimiento el rastreo de la ejecución del sistema se facilita utilizando técnicas de monitoreo que permiten analizar la interacción entre las diferentes partes que componen el sistema, reduciendo el esfuerzo necesario para identificar el segmento donde se encuentra un problema. Herramientas como *logging* y *tracing* son valiosas para facilitar la ubicación de la causa principal que ocasiona una desviación en la ejecución del sistema y en la actualidad son necesarias para reducir costos y tiempos en los procesos de mantenimiento de sistemas heredados. Incrementar la capacidad de monitorear la ejecución de un sistema reduce considerablemente el tiempo necesario para identificar un error y permite obtener información valiosa para evaluar su gravedad.

La programación orientada a aspectos (POA) permite desarrollar las herramientas de monitoreo de forma separada, evitando la necesidad de modificar el código fuente original de un sistema. Para su correcta implementación es necesario aislar la nueva funcionalidad en una nueva capa de abstracción, simplificando su análisis, diseño e implementación. Es de suma importancia tener las herramientas necesarias para el monitoreo de la ejecución del sistema separadas del resto del sistema sin modificar su comportamiento original, para tener la certeza de que el funcionamiento inadecuado no ha sido ocasionado por modificaciones realizadas para el monitoreo de la ejecución. Para vincular la nueva funcionalidad con la provista por el sistema heredado es necesario un puente que les permita interactuar. La POA provee el lazo necesario para unir la funcionalidad original con la nueva, en este caso con las herramientas de monitoreo que permiten observar el funcionamiento interno del sistema. AspectJ permite operar con elementos estáticos, así como con elementos dinámicos sin modificar el sistema original, proveyendo las herramientas necesarias para monitorear la invocación de métodos, los argumentos que reciben, el tipo de retorno, así como la lectura y escritura de campos. Para poder observar el comportamiento de variables locales es necesario un marco de trabajo más especializado como Énfasis [1, 2].

2 PROGRAMACIÓN ORIENTADA A ASPECTOS

En la actualidad la programación orientada a objetos (POO) es el paradigma más utilizado para la elaboración de sistemas, pero por sus características ocasiona en algunas situaciones la presencia de código disperso y enredado [3], el cual representa funcionalidades que no pueden ser encapsuladas en una sola unidad funcional, como es el caso del *logging* y el *tracing*. Al implementar aspectos como el *tracing* utilizando POA se crea un vínculo entre la funcionalidad original y la nueva y así se origina un objeto que contiene ambas [4] y puede representarse utilizando UML [5].

La POA provee de los mecanismos necesarios para separar los aspectos en un nuevo nivel de abstracción, el cual corta en aquellos puntos en que es necesario modificar el comportamiento original. Un proceso llamado *weaving* mezcla la funcionalidad original con la nueva generando el nuevo sistema. Esto permite crear nuevas funcionalidades y delegar en el compilador de AspectJ el proceso de incluirla en aquellos puntos de unión que se han identificado previamente [6]. De acuerdo con Laddad [4], los aspectos son requerimientos que debe proveer el sistema y existen dos tipos: aquellos que son vitales para el funcionamiento del mismo y deben conocerse y aquellos que proporcionan funcionalidades adicionales que simplifican aspectos como el mantenimiento; en ambos casos afectan a una considerable cantidad de segmentos de código.

3 TÉCNICAS DE MONITOREO

En la actualidad existen conceptos concretos que no pueden ser encapsulados con las metodologías de programación actuales dentro de una unidad funcional.

Agregar a un sistema el manejo de errores posterior al diseño del mismo requiere muchos y pequeños cambios y adiciones por todo el sistema debido a los diferentes contextos dinámicos que pueden llevar a una falla, y las diferentes políticas relacionadas con el manejo de la misma [7].

En general, los aspectos en un sistema que tengan que ver con el atributo *performance*, resultan diseminados por todo el sistema. Podemos afirmar entonces que las técnicas tradicionales no soportan de una manera adecuada la separación de las propiedades de aspectos distintos a la funcionalidad

básica y que esta situación tiene un impacto negativo en la calidad del *software*. Las técnicas de monitoreo permiten reunir y transmitir información de los eventos sistemáticos de manera cronológica en la ejecución de un sistema de *software*.

El *loging* o registro de los eventos de un sistema de *software* permite detectar los problemas de procesamiento en forma histórica, controlando las notificaciones a los usuarios de las condiciones de alarma que permitan tomar las medidas necesarias para la solución de los inconvenientes en las mismas. La información detallada de los datos recuperados para cada punto (evento) dentro de la ejecución del sistema proporciona las herramientas necesarias para establecer los procedimientos necesarios dependiendo de la información obtenida.

Por otro lado, el *tracing* o seguimiento permite seguir el funcionamiento del sistema paso a paso, para analizar si el comportamiento observado es el esperado o si se está haciendo algún proceso de forma inadecuada. Está técnica permite registrar el orden en que se ejecutaron los segmentos de código, así como los valores recibidos y devueltos para poder conocer más sobre el sistema y verificar su correcto funcionamiento.

La diferencia entre el *logging* y el *tracing* consiste en que el primero es un registro permanente de los aspectos más relevantes de la operación, para poder reconstruir un escenario en caso de un funcionamiento inadecuado del sistema con intención de depurarlo, mientras que el segundo es un seguimiento de todas las acciones que realiza el sistema, es más intenso y abarca toda la operación de sistema

4 IMPLEMENTACIÓN

En el caso de los sistemas heredados el rastro de la ejecución de la implementación suele complicarse porque en ocasiones se realiza a través de empleados independientes (*outsorcing*), porque en la empresa no se cuenta con personal que haya formado parte del equipo de desarrollo y que conozca el código fuente con el detalle suficiente para realizar un mantenimiento. Otro motivo es que para reducir los costos es común que se invierta poco tiempo o nada en lo absoluto para el desarrollo de herramientas que faciliten a futuro el mantenimiento del sistema.

También debe considerarse que el rastreo de la ejecución de un sistema afecta sensiblemente el desempeño al agregar instrucciones para guardar o mostrar información con una frecuencia considerable, por ello una solución óptima implicaría poder activar y desactivar el rastreo de la ejecución.

Si bien a través del uso de banderas es posible activar o desactivar el rastreo, implicaría una verificación del estado actual de la bandera por parte de todos aquellos puntos de interés que se encuentran dispersos a lo largo del código fuente, por lo que continúa afectando sensiblemente el desempeño del sistema.

Otra cuestión importante a tener en cuenta es que, en caso de que se haya tomado la precaución de implementar estrategias para el rastreo de la ejecución del sistema, la flexibilidad que presentan utilizando la orientación a objetos es poca. Para adaptar el rastreo a un nuevo formato, o para redirigir el flujo de la información hacia un nuevo destino (como podría ser un archivo o una base de datos). Es necesario modificar todos los segmentos de código que se encuentran dispersos a lo largo del sistema, lo que requiere una inversión de tiempo considerable. Adicionalmente es difícil asegurar la modificación de todos los puntos de rastreo.

Centralizar el manejo del rastreo en una clase resuelve parcialmente el problema al permitir obtener en una sola clase la responsabilidad del direccionamiento del flujo de información proporcionada por el rastreo. Pero en caso de que se requiera información adicional del contexto nuevamente surge la necesidad de modificar una cantidad considerable de segmentos de código.

Una mejor solución es proporcionada por la POA, que permite puntos de unión para identificar eventos en la ejecución de un programa y definir un corte para afectar una colección de eventos que comparten características similares y así obtener información sobre la ejecución, incluso del contexto, como valores de retorno y argumentos. Existen dos tipos de corte que pueden efectuarse y son:

- *Estático*. Modificar la estructura de las clases antes de la ejecución del sistema.
- *Dinámico*. Detectar eventos en tiempo de ejecución, como son invocaciones de métodos y manipulación de campos.

Una vez identificados los lugares donde se puede realizar un corte, se declara un aspecto que permita

alterar la ejecución normal del sistema. Éste puede actuar antes del punto de corte, después o bien reemplazar por completo ese segmento de código, en cuyo caso recibe la lista de argumentos que había sido enviada al método y tiene la obligación de devolver el mismo tipo que el método cuya ejecución esté sustituyendo. Los tipos de aspectos que existen son:

- *Before*. Permite insertar funcionalidad antes de que se ejecute el punto de unión.
- *After*. Permite insertar funcionalidad después de que se ejecuta el punto de unión.
- *Around*. Permite sustituir una funcionalidad proporcionando una implementación alternativa sustituyendo el punto de unión. Puede también acceder al método que está envolviendo e interactuar con él.

```
1public aspect LoggingAspect { }
```

Una vez definido el aspecto es necesario que se defina el corte que permitirá capturar la colección de eventos que contienen información sobre la ejecución que interesa. En este caso es necesario capturar la información de todos los puntos de unión que existen en el sistema, debido a que se requiere evaluar el funcionamiento completo del sistema. Para evitar que se busque dentro del aspecto ocasionando un ciclo infinito es necesario indicar que no debe tomar en cuenta los eventos que se generan en la clase actual TraceAspect, al negarlo con el operador ! se le indica que debe buscar fuera de la clase que se le pasa como argumento.

```
1pointcut tracePoints(): !within(-
TraceAspect);
```

Una vez definido el corte se define el tipo de aspecto a emplear. *Before* tiene un espectro más amplio de puntos de unión que *around* y *after*, con lo cual permite capturar un mayor número de eventos.

```
pointcuttracePoints() : !within(-
TraceAspect);
before() : tracePoints() { }
```

Una vez definido el aspecto por utilizar, así como los puntos en que se hará el corte para introducir la funcionalidad del rastreo, la palabra clave thisJoinPoint permite obtener información sobre cada uno de los puntos de unión en los que se ha efectuado el corte. El aspecto completo con salida a un archivo permite detectar los eventos y actuar para obtener información para rastrear la ejecución del sistema direccionando la información a otra clase, a un archivo, a una base de datos o a cualquier medio que facilite el análisis de la ejecución.

```
1  aspect TraceAspect {
2  private Filefile;
3  private FileWriter writer;
4
5  pointcuttracePoints() : !within(-
TraceAspect);
6
7  before() : tracePoints() {
8  try {
9  file = new File("./trace.txt");
10 writer = newFileWriter(file,true);
11 StringBufferparamBuffer = new
StringBuffer("\n\t[Objeto: ");
12paramBuffer.append(thisJoinPoint.
getThis());
13 Object[] arguments = thisJoinPoint.
getArgs();
14 paramBuffer.append("]\n\
t[Argumentos: (");
15 for (int length = arguments.length,
i = 0; i< length; ++i) {
16 Object argument = arguments[i];
17 paramBuffer.append(argument);
18 if (i != length-1) {
19 paramBuffer.append(',');
20 }
21 }
22 paramBuffer.append(")]");
23 writer.write(paramBuffer.toS-
tring()+"\n");
24 writer.close();
25 } catch (Exception e) {
26 e.printStackTrace();
27 }
28 }
29 }
```

Se define un archivo para volcar la información generada con la ejecución del sistema, y se obtiene la información necesaria sobre la instancia del objeto que está ejecutando la instrucción, así como la lista de argumentos, que se guardan en un arreglo de tipo *object*. Éste se recorre para poder exponer el tipo y valor de cada uno de los argumentos recibidos y se almacenan en un StringBuffer. Una vez obtenido el objeto y sus características, así como los argumentos

recibidos con su tipo y valor, se guardan en el archivo para su consulta posterior.

Una vez definido el aspecto de más amplio espectro de visibilidad y mayor grado de detalle, se seleccionan aquellos puntos de unión donde es necesario implementar un corte en aquellos puntos de interés que serán vigilados de manera permanente, de una forma no tan minuciosa, buscando solamente conservar la información elemental para poder comprobar el correcto funcionamiento del sistema.

```
1   public aspectLoggingAspect {
2   private Filefile;
3   private FileWriter writer;
4
5   pointcutloggingPoints()    :    !wi-
thin(LoggingAspect);
6
7   before() : loggingPoints() {
8   try {
9   file = new File("./log.txt");
10  writer = newFileWriter(file,true);
11  writer.write(thisJoinPoint.toS-
tring()+"\n");
12  writer.close();
13  } catch (Exception e) {
14  e.printStackTrace();
15  }
16  }
17  }
```

Ambos aspectos se encuentran relacionados con dos banderas de tipo *boolean* que se encuentran en el menú y permiten activar o desactivar las características de *tracing* y *logging* en tiempo de ejecución. En caso de que el sistema tenga una carga de trabajo muy intensa y sea necesario liberar recursos, el menú *item* es agregado de igual manera utilizando un aspecto, lo cual permite más adelante al usuario, en esa misma barra de herramientas, agregar nuevas características que puedan activarse o desactivarse.

```
1   public aspectRastreo {
2   JMenu menu;
3   JMenuItemmenuItem;
4
5   pointcutbarraMenu(Main main, JMenu-
BarmenuBar) :
6   call(void    simulator.Main.setJMenu-
Bar(JMenuBar))
7   && target(main)
8   &&args(menuBar);
9
10  before(final    Main    main,    JMenu-
BarmenuBar) :
11  barraMenu(main, menuBar) {
12  menu = newJMenu("Rastreo");
13
14  menuItem= newJMenuItem("Tracing");
15  menuItem.addActionListener(new    Ac-
tionListener() {
16  public           voidactionPerforme-
d(ActionEvent event) {
17  if(Simulador.tracing)
18  Simulador.tracing= false;
19  else
20  Simulador.tracing= true;
21  }
22  });
23  menu.add(menuItem);
24
25  menuItem= newJMenuItem("Logging");
26  menuItem.addActionListener(new    Ac-
tionListener() {
27  public           voidactionPerforme-
d(ActionEvent event) {
28  if(Simulador.logging)
29  Simulador.logging= false;
30  else
31  Simulador.logging= true;
32  }
33  });
34  menu.add(menuItem);
35
36  main.getBarraMenu().add(menu);
37  }
38  }
```

Las palabras reservadas *target* y *args* permiten obtener la referencia al objeto y el valor del argumento para, de esa manera, utilizar el método original getBarraMenu() y modificar el menú con los nuevos elementos definidos. Esto simplifica tareas como agregar una barra de herramientas que permita controlar, de manera dinámica en tiempo de ejecución, el rastreo de la ejecución que se lleva a cabo mediante un aspecto manteniendo un menú homogéneo sin poner en riesgo la estabilidad del sistema. Con esta aproximación se puede habilitar y deshabilitar el rastreo en tiempo de ejecución, según sea requerido sin necesidad de alterar el código original del sistema.

5 CONCLUSIÓN

La POA permite modificar el comportamiento de un programa al identificar un punto de unión que sea de interés y realizar un corte en él o en una colección de puntos para introducir nuevas funcionalidades

en el sistema. Esta característica permite incorporar nuevas funcionalidades sin modificar el código fuente adicional aislando los cambios del mismo. Su habilidad de poder separar los aspectos en unidades funcionales, a pesar de su alta dispersión, facilita el trabajo de mantenimiento de los sistemas heredados, ahorrando significativamente en tiempo y costo, proveyendo de la capacidad de aprovechar tanto como sea posible la funcionalidad actual de un sistema heredado y operar una serie de cambios orientados a mejorar su funcionalidad.

6 TRABAJO FUTURO

Existe la necesidad de implementar nuevas funcionalidades, aprovechando las facilidades que provee la POA, que tengan como principal característica una alta dispersión a lo largo del sistema y que no puedan ser encapsuladas en una unidad funcional. Se tiene contemplada la centralización del control de errores, la ejecución concurrente de una gran cantidad de equipos, así como la homologación y renovación de la apariencia y la implementación de *profiling*. Además de los aspectos anteriormente mencionados, se debe explorar la posibilidad de reforzar las políticas de desarrollo y redefinir la arquitectura del sistema para proveerla de una mayor flexibilidad utilizando POA.

RECONOCIMIENTOS

Al Consejo Nacional de Ciencia y Tecnología, por los recursos asignados para poder realizar una estancia en la industria.

A la Cámara Nacional de Manufacturas Eléctricas, por su constante esfuerzo para vincular las actividades académicas con las productivas.

A la empresa Sensa Control Digital, por brindarme la oportunidad de realizar una estancia en una empresa mexicana que busca innovar constantemente y mejorar sus procesos, deseo que más empresas mexicanas sigan su ejemplo.

REFERENCIAS

1. Juárez-Martínez, U., Olmedo-Aguirre, J. O. (2008). Énfasis: a model for local variable crosscutting". *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08. Nueva York, 261-265, ACM.
2. Juárez-Martínez, U. (2008). Énfasis: Programación orientada a aspectos de grano fino. Tesis de doctorado, Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional.
3. Aksit, M. (1996). Separation and composition of concerns in the object-oriented model.
4. Laddad, R. (2003). AspectJ in action: Practical aspect-oriented programming. Greenwich, CT, EU. Manning Publications Co.
5. Suzuki, J., Yamamoto, Y. (1999). Extending UML with aspects: Aspect support in the design phase. *Lecture notes in computer science*, Springer-Verlag.
6. Kiczales, G., Irwin, J., Lamping, J., Loingtier, J. M., Lopes, C. V. Maeda, C. Mendhekar, A. (1996). Separation and composition of concerns in the object-oriented model.
7. Lopes, C. V., Kiczales, G. (1998). Recent developments in aspectj. En *Proceedings of European Conference on Object-Oriented Programming-Workshop on Aspect-Oriented Programming*, Springer-Verlag, 398-401.

Acerca de los autores

Emmanuel Mendoza Escobar es maestro en Sistemas Computacionales (2012) e ingeniero en Sistemas Computacionales (2007), por el Instituto Tecnológico de Orizaba. Sus áreas de investigación son inteligencia artificial, robótica, mecatrónica, sistemas expertos, modelado y simulación en entornos hetereogenios, desarrollo ágil de proyectos de *software* y aplicaciones en web. Ha colaborado con diferentes instituciones de educación superior y ha trabajado en diferentes empresas del sector privado.

Antonio Rodríguez Cabrales es ingeniero en sistemas computacionales egresado del Instituto Tecnológico de la Laguna, cuenta con diferentes certificaciones: conceptos de programación, programación orientada a objetos, Java, entre otros. Actualmente labora en Integration Point, Inc. Empresa trasnacional de comercio internacional.

Ulises Jesús López Maldonado se graduó como ingeniero en Sistemas Computacionales en el Instituto Tecnológico de Orizaba, Veracruz. Sus intereses incluyen el desarrollo de sistemas desde el enfoque del paradigma orientado a componentes, programación orientada a aspectos, ingeniería de *software* y *software* educativo. Actualmente se desempeña como docente en el Instituto Tecnológico de Iztapalapa.

Sergio David Ixmatlahua Díaz es maestro en Sistemas Computacionales por el Instituto Tecnológico de Orizaba, Veracruz. Durante sus estudios de maestría realizó una estancia profesional en la Facultad de Informática de la Universidad Politécnica de Madrid, España. Ha participado en congresos internacionales y nacionales como ponente presentando artículos relacionados con la ingeniería de *software* y desarrollo web. Actualmente se desempeña como profesor en el Instituto Tecnológico de Iztapalapa en la Ciudad de México, DF.

Roque Manuel Rueda Anastacio es licenciado en Sistemas Computacionales Administrativos por la universidad Veracruzana. Actualmente se desempeña como *senior system engineer* en Infosys. Participa activamente en proyectos de desarrollo y mantenimiento de *software* enfocado en la mejora de procesos *software*. Ha realizado trabajo en el análisis de metodologías de desarrollo de *software* y mejores prácticas. Asimismo, en diseño arquitectónico de sistemas e implementación de nuevas tecnologías.

Alejandro Domingo Velázquez Cruz es licenciado en Administración Pública por la Universidad Abierta de San Luis Potosí y maestro en Sistemas Computacionales por el Instituto Tecnológico de Orizaba. Es docente del Instituto Tecnológico de Iztapalapa, se especializa en la formación de estudiantes en el área de ingeniería de *software*. Ha ocupado diversos cargos en la Secretaría de Desarrollo Social, el Instituto Mexicano del Seguro Social y la iniciativa privada.