ISSN 1870-4069

Intercesión en invocaciones de métodos con la reflexión de Java

Andoni Rodríguez-Díaz, Ulises Juárez-Martínez, Silvestre Gustavo Peláez-Camarena, Hilarión Muñoz-Contreras, Beatriz A. Olivares-Zepahua

Instituto Tecnológico de Orizaba,
División de Estudios de Posgrado e Investigación, Orizaba,
Veracruz, México

{arodriguezd,ujuarez}@ito-depi.edu.mx,sgpelaez@yahoo.com.mx,
hmunoz@itorizaba.edu.mx,bolivares@ito-depi.edu.mx

Resumen. Los sistemas de software requieren de mantenimiento para realizar cambios a los mismos, esto implica que se detenga la ejecución del sistema para aplicar las nuevas mejoras. La reflexión permite obtener la información de un sistema en ejecución, lo que da la posibilidad de adaptar el software sin interrupciones. La reflexión del lenguaje Java en cuanto a intercesión se limita únicamente a la modificación de los valores de los campos, por lo que no se ofrece el soporte para manipular los comportamientos del sistema. Este trabajo presenta un esquema de intercesión que hace uso de la información que ofrece la reflexión para aplicar nuevas funcionalidades al sistema.

Palabras clave: Reflexión, adaptación de software, intercesión.

Intercession on Methods' Invocations with Java Reflection

Abstract. Software systems require maintenance to modify them, this involves stopping the system's execution and apply the changes. Reflection allows to get the executing system's information and let adapt the system without any interruption. The intercession in Java programming language is limited only in writing into fields, therefore it is not possible to modify the program's behavior. This paper features an intercession scheme which gets information from reflection to add new functionalities into the existing system.

Keywords: Reflection, adapting software, intercession.

1. Introducción

De acuerdo al estándar 14764 de IEEE [1], los sistemas de software requieren del mantenimiento para añadir soporte a los cambios en el entorno de ejecución, adición de

nuevas funcionalidades y mejoras en las ya existentes. Existen situaciones en las que el código fuente no está disponible o la documentación existente no satisface las cuestiones del equipo de desarrollo. Existen enfoques para modificar el sistema en su representación binaria o en código intermedio, entre ellos están los marcos de trabajo para la manipulación de *bytecodes* como ASM y Javassist; sin embargo, se requiere que los desarrolladores tengan el conocimiento completo de la estructura del sistema; además los cambios se aplican en tiempo de compilación, es decir, es necesario detener el sistema e implica el riesgo de fallas en cualquier parte de la ejecución.

Las capacidades de reflexión permiten obtener la información precisa del sistema en tiempo de ejecución como, por ejemplo, el tipo real de un objeto o acceder a los elementos que están ocultos por sus modificadores. La reflexión del lenguaje de programación en Java limita la capacidad para realizar cambios a una clase únicamente a la modificación de valores de los campos; el uso de *proxies* en Java para la intercesión de métodos sólo es compatible con las interfaces que se especifican durante su construcción y reduce el acceso a otros elementos de una clase concreta.

Los trabajos relacionados presentan enfoques para controlar el uso de la reflexión durante la ejecución de los sistemas y para la implementación de soluciones utilizando dicha técnica.

Este trabajo presenta un esquema de intercesión para la invocación de métodos de un sistema mediante la creación de clases en Java para reemplazar la funcionalidad existente con una alterna; se dispone de AspectJ para interceptar las ejecuciones de cada método y recuperar la información necesaria para aplicar la intercesión.

Este artículo se compone de la siguiente forma: la sección 2 se describen los trabajos relacionados en cuanto a intercesión. En la sección 3 se analizan los fundamentos de reflexión y las herramientas que implementan esta técnica. En la sección 4 se analizan los enfoques de intercesión y sus desventajas. En la sección 5 se presenta y se describe el esquema de intercesión propuesto. En la sección 6 muestra un ejemplo aplicando el esquema de intercesión. En la sección 7 se analizan los resultados. En la sección 8 se dan a conocer las conclusiones y el trabajo a futuro.

2. Trabajos relacionados

En [2] se presentó un análisis estático para los flujos de control implícitos que existen en la reflexión de Java y en los intentos en Android, lo cual permite determinar qué procedimientos se llaman y qué datos se pasan; para el caso de reflexión se analizan las llamadas reflexivas y conocer qué clases se manipulan. En [3] se desarrolló Mirror, una biblioteca para añadir las capacidades de reflexión al lenguaje de programación C++ y de esta forma implementar el patrón de diseño Factory para la generación de nuevas instancias. En [4] se presentó un modelo que contempla cinco dimensiones de control en el meta-nivel que se reportan en otras literaturas relacionadas: control espacial y temporal, control de colocación, control de nivel y control de identidad; este modelo se implementó en el entorno de programación Pharo. En [5] resaltó que aplicar la reflexión hacia los elementos privados de una estructura de datos rompe la encapsulación en objetos; se propuso una política de control que determina si alguna acción reflexiva se considera haber roto dicha encapsulación. En [6] se presentó DroidRA, un enfoque para controlar las llamadas reflexivas en las aplicaciones Android que complementa a las

herramientas de análisis estáticos. En [7] se demostró que la técnica de "cadenas de despachado" trata el problema en el desempeño al aplicar las técnicas de meta-programación, sobre todo las operaciones de reflexión; esta técnica es aplicable tanto para generación de compiladores just-in-time como para las evaluaciones parciales.

3. Reflexión

La reflexión es la capacidad de un programa para analizar y modificar a sí mismo y a su entorno en tiempo de ejecución. La reflexión permite realizar dos acciones: introspección e intercesión. La introspección es la acción de analizar la estructura de un tipo de dato, la composición de estados y comportamientos, y los valores actuales de cada estado; la intercesión permite modificar la estructura del tipo de dato: añadir nuevos estados, modificar su comportamiento y modificar su jerarquía. La información que describe a cada tipo de dato, así como también la de cada elemento que componen al primero, se conocen como meta-objetos; al modificar un meta-objeto sus efectos se reflejan en la ejecución del programa [8].

3.1. Reflexión en Java

El lenguaje de programación Java ofrece la característica de reflexión mediante la importación de clases en el espacio de nombres {java.lang.reflect}, la descripción de cada elemento o meta-objeto se representa como un objeto de tipo del elemento que lo identifica (ej. Method). Por ejemplo, la clase java.lang.Object es una instancia de tipo java.lang.Class, por lo tanto, toda clase que se carga a la máquina virtual de Java es una instancia de tipo Class; el tipo de dato Class, de forma redundante es una instancia de sí mismo, entonces Class es una meta-clase. Desde una instancia de tipo Class, es posible obtener otros meta-objetos en relación, como los campos, los métodos y los constructores que componen a una clase, sin importar el nivel de acceso de cada uno, además permite la creación de instancias detallando la información del meta-objeto del constructor. Desde un meta-objeto de un campo se obtiene la información acerca de sus características, como por ejemplo el tipo de dato y sus modificadores, y permite recuperar o modificar el valor que contiene. Desde un meta-objeto de un método, se obtiene la misma información a la de un campo, y permite la invocación correspondiente. Desde reflexión es posible obtener información que no está disponible en tiempo de compilación como, por ejemplo, conocer el verdadero tipo de dato de una instancia en una variable de tipo Object [9].

3.2. Proxy en Java

La reflexión en Java ofrece la capacidad para crear elementos que intervienen en la interacción entre dos objetos, la generación de *proxies* dinámicos. La clase java.lang.reflect.Proxy cuenta con métodos para la creación tanto de clases como instancias de este tipo en tiempo de ejecución. Para la creación de una clase proxy se hace uso de un cargador de clase y una lista de interfaces; para la generación de instancias se dispone de los dos elementos anteriores y una instancia de tipo InvocationHandler, que contiene la implementación para intervenir en la interacción de

un objeto y realizar las invocaciones al mismo con reflexión. Como resultado, la invocación a un método de un objeto se realizará a través de la instancia InvocationHandler y, posteriormente, al primer elemento; además la instancia de proxy es compatible con los tipos que se especificaron en la lista de interfaces.

3.3. AspectJ

AspectJ es un lenguaje orientado a aspectos que extiende al lenguaje de programación Java para la creación e implementación de aspectos. Se establecen cortes en puntos, que se componen de puntos de unión que identifican en qué partes del código del programa principal se aplicarán los cortes y la funcionalidad del aspecto, esto último mediante el uso de avisos. AspectJ trabaja con los mecanismos de reflexión de Java en lo que respecta a la información dinámica y estática de un punto de unión. Por ejemplo, es posible determinar, por la parte dinámica, el objeto donde se realizó la invocación al método y consultar los valores de la lista de parámetros; mientras que por la parte estática se describe las propiedades del punto de unión desde el *bytecode* [10,11].

4. Intercesión

La intercesión de la reflexión en Java se limita en la modificación en los valores de los campos, no tiene soporte para la manipulación en la estructura de una clase ni en el comportamiento de la misma. Un enfoque para la manipulación del comportamiento en una clase es la implementación de *proxies*; Java ofrece la característica para generar *proxies* en tiempo de ejecución o *proxies* dinámicos, con la capacidad de establecer funcionalidad antes y después de la invocación al método de un objeto. Sin embargo, un objeto proxy sólo se aplica para las interfaces que se especifiquen durante su construcción. El uso de *proxies* implica generar un objeto de este tipo por cada objeto que compone al sistema. Otra solución es la transformación de clases en su representación en *bytecode* mediante el uso de biblioteca, tales como ASM y Javassist; pero estas modificaciones se realizan en tiempo de compilación, lo que requiere detener el sistema para aplicar los nuevos cambios y que los desarrolladores tengan los conocimientos acerca de la composición de cada clase del sistema.

5. Esquema de intercesión propuesto

En esta sección se presenta un esquema de intercesión en métodos en tiempo de ejecución, lo que evita la necesidad de detener el sistema. En la figura 1 se presenta el diagrama de clases del modelo para la intercesión.

El diagrama en la figura 1 muestra la interfaz Intercesión con los métodos intercede() y override() que aplican la ejecución alterna; interrupt() indica si la funcionalidad original se reemplazará con el objeto de intercesión; getDescriptor() devuelve una referencia a un objeto de tipo TargetDescriptor que describe en qué parte del sistema se aplicará la intercesión. En la clase TargetDescriptor se especifica la clase, el nombre del método y la lista de los tipos de los argumentos.

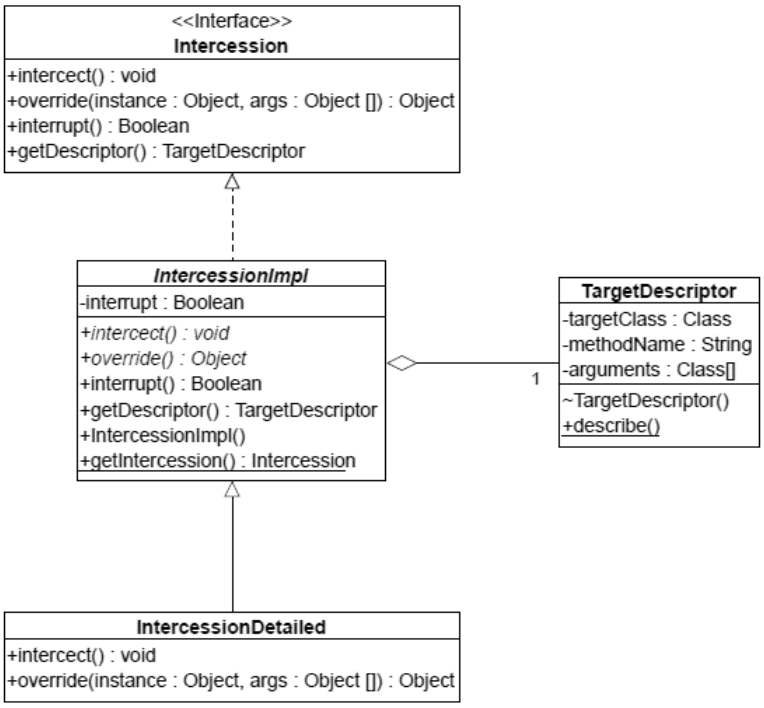

Fig. 1. Diagrama de clase del esquema de intercesión.

```
1  public aspect Intercect {
2      Object around() : execution(!static * Object+.*(..)) {
3          boolean interrupt = false;
4          Intercession intercession = IntercessionImpl.getIntercession(
5                  joinpoint.getTarget().getClass(),
6                  joinpoint.getSignature().getName(),
7                  ((CodeSignature)joinpoint.getSignature()).getParameterTypes()
                       );
8
9          if(intercession != null) {
10             intercession.intercede();
11             interrupt = intercession.interrupt();
12         }
13
14         if(!interrupt) {
15             return proceed();
16         }
17
18         return intercession.override(joinpoint.getTarget(), joinpoint.getArgs
               ());
19     }
20 }
```

Fig. 2. Intercesión de métodos.

La clase IntercesiónImpl es la responsable de recuperar el objeto de intercesión de un conjunto de este tipo. El desarrollador implementa la nueva funcionalidad mediante una clase concreta que redefine los métodos intercede() y override().

En el código Fig. 1, el aspecto Intercect realiza los cortes a todos los métodos de alcance de instancia, su aviso obtiene la información del punto de unión que corresponde a la invocación de un método (línea 2). El aspecto invoca a IntercesiónImpl.getIntercesión() para recuperar un objeto que coincida con el punto de unión (línea 4-7); si el objeto existe (línea 9) y la variable interrupt es verdadera (línea 14), entonces se reemplaza la funcionalidad original por la que se definió en Intercesión.override() (línea 18), si el objeto no existe se procede de forma normal con la ejecución (línea 15).

6. Ejemplo de intercesión

En la sección anterior se mostró el uso de AspectJ para permitir la intercesión mediante clases en Java. El ejemplo (Fig. 2) presenta una clase que simula un dispositivo, el cual reporta el porcentaje de procesamiento y el consumo de memoria.

```
1  public class Machine implements Runnable {
2      private boolean operating = false;
3      private byte cpuUsage;
4      private int memoryUsed;
5      @Override
6      public void run() {
7          turnOn();
8      }
9      public void turnOn() {
10         operating = true;
11         while(operating) {
12             cpuUsage = (byte)((Math.random() * 100) % 100 + 1);
13             memoryUsed = (int)((Math.random() * 1024) % 1024 + 1);
14             Thread.sleep(1000);
15         }
16     }
17     public void turnOff() {
18         if(operating)
19             operating = false;
20     }
21 }
```

Fig. 3. Clase Machine (máquina).

La clase MachineRunIntercesión (Fig. 3) hereda de IntercesiónImp (línea 1) e implementa los métodos intercede() y override() (líneas 5-12); su funcionalidad se aplica en la invocación al método Machine.turnOn() (línea 3) y mostrará en pantalla al usuario un mensaje de que la máquina inició su ejecución (línea 7).

La clase MachineEndedIntercesión (Fig. 4) comparte las mismas características al código anterior. En este caso se reemplazará la funcionalidad del método Machine.turnOff() mostrando un mensaje de que la máquina no detendrá su ejecución (líneas 12-16).

```
1  public class MachineRunIntercession extends IntercessionImpl {
2      public MachineRunIntercession() {
3          super(TargetDescriptor.describe(Machine.class, "turnOn", new Class
               [0]));
4      }
5      @Override
6      public void intercede() throws RuntimeException {
7          JOptionPane.showMessageDialog(null, "This machine has started on " +
               Thread.currentThread().toString());
8      }
9      @Override
10     public Object override(Object instanceSource, Object[] args) throws
           RuntimeException {
11         return null;
12     }
13 }
```

Fig. 4. Clase de intercesión al invocar el método Machine.turnOn().

```
1  public class MachineEndedIntercession extends IntercessionImpl {
2      public MachineEndedIntercession() {
3          super(TargetDescriptor.describe(Machine.class, "turnOff", new Class
               [0]));
4          super.interrupt = false;
5      }
6
7      @Override
8      public void intercede() throws RuntimeException {
9          JOptionPane.showMessageDialog(null, "This has ended.");
10     }
11
12     @Override
13     public Object override(Object instanceSource, Object[] args) throws
           RuntimeException {
14         JOptionPane.showMessageDialog(null, "Lies. This will continue.");
15         return null;
16     }
17 }
```

Fig. 5. Clase de intercesión al invocar el método Machine.turnOff().

```
1  public class DisplayFahrenheit extends reflect.IntercessionImpl {
2      public DisplayFahrenheit() {
3          super(TargetDescriptor.describe(weather.util.display.DisplayManager.
               class, "displayTemperature", new Class[] { int.class }));
4      }
5      public void intercede() throws RuntimeException {
6          super.interrupt = true;
7      }
8      public Object override(Object instanceSource, Object[] args) throws
           RuntimeException {
9          int value = (int)args[0];
10         int result = (int)(((float)value * (9f / 5f)) + 32f);
11         ui.getTxtTemp().setText(Integer.toString(result) + " F");
12         return null;
13     }
14 }
```

Fig. 6. Clase para representar la temperatura en grados Fahrenheit.

7. Resultados

El esquema de intercesión se aplicó en un sistema de reporte de las condiciones del clima en tiempo real, para representar y visualizar los datos en otras unidades de medición. Se generaron las clases para reemplazar la ejecución original. AspectJ realizó los cortes en todos los métodos del sistema, se analizó con la reflexión la información de cada método para determinar la existencia alguna ejecución alterna. Por ejemplo, el código 5 (Fig. 5) contiene la ejecución para representar la temperatura en grados Fahrenheit; reemplazando la definición original del método intercesión displayTemperature() de la clase intercesión DisplayManager.

Este esquema permitió modificar el sistema sin detenerlo lo que evitó la pérdida de la información, a diferencia de aplicar los cambios en tiempo de compilación.

8. Conclusión y trabajo a futuro

Se presentó un esquema de intercesión para las invocaciones de métodos sin detener la ejecución del programa y se implementó en un sistema de reporte de las condiciones del clima en tiempo real, para representar y visualizar los datos en otras unidades de medición; se aplicó la reflexión para obtener la información de la invocación en cada método y se trabajó con AspectJ para aplicar los cortes en todos los métodos del sistema.

El uso de la reflexión permite obtener la información del sistema que no está disponible en tiempo de compilación, lo que da la posibilidad de realizar adaptaciones seguras sin afectar en forma negativa la integración del sistema.

Como trabajo a futuro se considera trabajar con los cargadores de clase dinámicos para permitir la carga e intercambio de clases que implementen intercesiones con el propósito de realizar el mantenimiento al sistema en tiempo de ejecución.

Agradecimientos. Este trabajo cuenta con apoyo por parte del Consejo Nacional de Ciencia y Tecnología (CONACYT).

Referencias

1. IEEE, ISO/IEC: Norma IEEE Std 14764-2006 Revision of IEEE Std 1219-1998, IEEE std, IEEE (2006)
2. Barros, P., Just, R., Millstein, S., Vines, P., Dietl, W., d'Amorim, M., Ernst, M.D.: Static analysis of implicit control flow: Resolving Java reflection and Android intents. In: ASE 2015: Proceedings of the 30th Annual International Conference on Automated Software Engineering, Lincoln, NE, USA, pp. 669–679 (2015)
3. Chochlik, M.: Implementing the factory pattern with the help of reflection. Computing and Informatics (2015)
4. Papoulias, N., Denker, M., Ducasse, S., Fabresse, L.: Reifying the reectogram: Towards explicit control for implicit reflection. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing, SAC '15, New York, NY, USA, ACM, pp. 1978–1985 (2015)

5. Teruel, C., Ducasse, S., Cassou, D., Denker, M.: Access control to reflection with object ownership. SIGPLAN Not, 51(2), pp. 168–176 (2015)
6. Li, L., Bissyandé, T. F., Octeau, D., Klein, J.: Droidra: Taming reflection to support whole-program analysis of android apps. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, New York, NY, USA, ACM, pp. 318–329 (2016)
7. De Wael, M., Marr, S., Van Cutsem, T.: Fork/join parallelism in the wild: Documenting patterns and anti-patterns in java programs using the fork/join framework. In: Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, PPPJ '14, pp. 39–50 (2014)
8. Forman, I. R., Forman, N.: Java Reflection in Action (In Action Series). Manning Publications Co., Greenwich, CT, USA (2004)
9. Oracle-Corporation: Trail: The reflection API. https://docs.oracle.com/javase/tutorial/reflect/
10. Eclipse-Foundation: Aspectj documentation. https://eclipse.org/aspectj/docs.php
11. Laddad, R.: AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications Co., Greenwich, CT, USA (2003)