TEJEDOR DINÁMICO DE ASPECTOS SOBRE LA PLATAFORMA .NET

Luis Vinuesa y Francisco Ortin

Universidad de Oviedo, Departamento de Informática
C\ Calvo Sotelo s/n – 33007 - Oviedo - España
{vinuesa,ortin}@lsi.uniovi.es
http://www.di.uniovi.es/~{vinuesa,ortin}

Abstract. Desde la aparición del Desarrollo de Software Orientado a Aspectos - Aspect Oriented Software Development- (AOSD) ha surgido un conjunto de herramientas con el fin de obtener los beneficios del AOSD en el desarrollo de software. En determinadas ocasiones es necesario que una aplicación sea capaz de adaptarse en tiempo de ejecución ante nuevos requerimientos que puedan surgir. Ésta es la razón por la que han aparecido algunas herramientas que permiten el tejido dinámico de aspectos. Sin embargo, estas herramientas tienen algunas limitaciones, como la dependencia del lenguaje y un conjunto limitado de puntos de enlace, las cuales restringen su utilidad.
Este artículo muestra la investigación que estamos llevando a cabo en el campo del tejido dinámico de aspectos. Hemos seleccionado el sistema .NET como sistema sobre el que desarrollamos nuestro tejedor dinámico, por su independencia de lenguaje y plataforma. Aplicando técnicas de reflectividad computacional al nivel de la máquina virtual conseguimos tejido dinámico independiente del lenguaje, haciendo posible que las aplicaciones se adapten dinámicamente a contextos surgidos en tiempo de ejecución.

1 Introducción

En muchos casos, es difícil expresar de una forma modular incumbencias -*concerns*- importantes de una aplicación software. Ejemplos de esto pueden ser la seguridad, sincronización o persistencia. El código que trata estas competencias se encuentra a menudo diseminado sobre muchas partes de la aplicación, con el inconveniente de estar enmarañado y ser más difícil de mantener, entender y reutilizar.

La Ingeniería del Software ha utilizado el principio de la separación de incumbencias (o separación de competencias) (SoC) -*Separation of Concerns*- [1][2] para gestionar la complejidad del desarrollo de software mediante la separación de las funcionalidades principales de la aplicación de otras partes con un propósito específico (típicamente ortogonales a la funcionalidad principal) – p.ej. autenticación, administración, rendimiento, logging, etc. La aplicación final se construye con el código de las funcionalidades principales más el código de propósito específico.

Este principio se ha desarrollado siguiendo múltiples aproximaciones, AOSD[3] es una de ellas. El AOSD persigue como objetivo construir programas adaptables a las competencias -*concerns*-. La mayoría de las herramientas existentes carecen de adaptabilidad en tiempo de ejecución: una vez que la aplicación ha sido generada (tejida) no es capaz de adaptar sus competencias (aspectos) en ejecución. Es lo que se conoce como tejido estático de aspectos. Existen ciertos casos en los que la adaptación de la

aplicación debe ser realizada en tiempo de ejecución en respuesta a cambios en el entorno- p.ej. aspectos de distribución de carga. Otro ejemplo de uso de AOSD de forma dinámica es lo que recientemente se ha llamado software autónomo – *autonomic software*: software capaz de auto repararse, gestionarse, optimizarse o recuperarse.

Para solucionar las limitaciones de las herramientas con tejido estático de aspectos han surgido diferentes aproximaciones de tejido dinámico (p.ej. PROSE [4]). Sin embargo, están limitadas por el conjunto de puntos de enlace que ofrecen (en comparación con las herramientas de tejido estático), restringiendo así el modo en que las aplicaciones pueden ser adaptadas en tiempo de ejecución. Además, ambas (estáticas y dinámicas) utilizan un lenguaje de programación fijo, por lo que los aspectos no son reutilizables de forma independiente de su lenguaje.

La reflectividad es una técnica que consigue la adaptación dinámica de una aplicación, por lo que se puede utilizar para conseguir la adaptación de aspectos en tiempo de ejecución. Como trabajo previo de investigación [5][6][7] hemos aplicado reflectividad computacional a una máquina virtual, obteniendo una adaptación del programa en ejecución sin restricciones y de forma neutral al lenguaje y a la plataforma.

Actualmente estamos aplicando [8] los conceptos que hemos aprendido a la plataforma Microsoft .NET que es independiente de lenguaje y plataforma y que ofrece un buen rendimiento en ejecución, siendo al mismo tiempo neutral respecto al lenguaje y hardware. Estamos desarrollando un tratamiento reflectivo de las aplicaciones .NET ofreciendo un marco de trabajo -*Framework*- que ofrece tejido dinámico de aspectos.

Cuando una aplicación compilada se va a ejecutar, el sistema transforma su código (manipulándolo en su lenguaje intermedio, *Microsoft Intermediate Language*, MSIL), añadiendo el código necesario para permitir a otras aplicaciones el acceso y adaptación de la primera. Para conseguir esto el sistema tiene un servidor (registro de aplicaciones) en el cual se registran todas las aplicaciones que se están ejecutando. Una vez que la aplicación se está ejecutando, si otra aplicación necesita adaptarla, el servidor le ofrece la capacidad de hacerlo en ejecución siguiendo el principio de AOSD.

2 Inyector de Puntos de Enlace

La arquitectura del sistema se muestra en la Fig.1 y Fig.2 . Antes de ejecutar una aplicación ya compilada, se procesa y se modifica por parte del sistema. El *Join Point Injector* (JPI) (inyector de puntos de enlace) del sistema inserta rutinas de reflectividad computacional basadas en MOP siguiendo los siguientes pasos:

1. La aplicación se carga en memoria mediante `System.Reflection`. Se utiliza este *namespace* para, a partir de los ficheros binarios, cargar la representación en MSIL del programa en memoria. Una vez allí el sistema manipulará este código como si se tratase de datos, localizando los puntos de enlace posibles.
2. Se insertan rutinas de control por medio de la herramienta RAIL (*Runtime Assembly Instrumentation Library)* [9]. En cada punto de enlace localizado anteriormente se insertan rutinas que manejan una tabla con referencias a los futuros aspectos que deben ser avisados con el fin de poder invocar a sus advices (código). En un principio este código inyectado no invoca ninguna rutina, pero insertando referencias

en la tabla posteriormente su comportamiento puede ser extendido – como ocurre en la mayoría de los sistemas de MOP.
RAIL es una herramienta que permite, de una manera sencilla, manipular e instrumentar código MSIL antes de su ejecución. Entre las acciones que permite realizar se encuentran la sustitución de accesos a campos, propiedades y métodos, la adición de código al principio y/o final de los mismos, o sustituir un método por otro. Esto permite al sistema contemplar un amplio abanico de puntos de enlace, similar al de los sistemas estáticos. Al trabajar sobre código MSIL es totalmente independiente del lenguaje en que se hubiese implementado la aplicación a manipular.

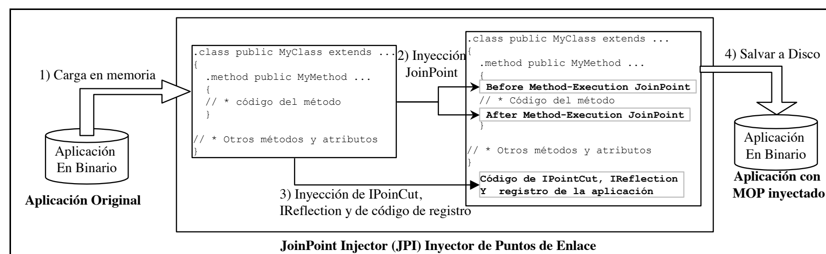

Fig. 1. Arquitectura del sistema A

3. Se inyecta el código de los interfaces `IPointCut` e `IReflection` así como el código necesario para que la aplicación se registre en el servidor. `IPointCut` provee el interfaz para definir diferentes puntos de enlace que se usarán en las adaptaciones futuras. Esta funcionalidad es la misma que los *pointcut designators* (o simplemente *pointcut*) de AspectJ.
   Mediante el uso de `IPointCut`, los aspectos, a través del servidor, podrán seleccionar un conjunto de puntos de enlace específicos de una aplicación para ser notificados cada vez que se alcance alguno de ellos (en la invocación de un método, acceso a un campo, etc.). Esto se consigue insertando en las tablas de las rutinas inyectadas en el segundo punto referencias a los aspectos que han solicitado ser notificados, provocando por tanto las invocaciones al código de los aspectos (advice).
   El interfaz `IReflection` permite a otras aplicaciones el acceso a la información de la aplicación (es decir, introspección intra aplicaciones, *outward*) –por razones de seguridad, en la plataforma .NET una aplicación sólo se puede inspeccionar a sí misma, no puede inspeccionar a otra aplicación. Cuando se están desarrollando los aspectos, normalmente es necesario acceder a la estructura de la aplicación que se va a adaptar: ésta es exactamente la principal funcionalidad ofrecida por `IReflection`. Este interfaz hace uso principalmente de `System.Reflection`, permitiendo que los aspectos puedan tener acceso a la información de la aplicación, puedan invocar un método, o incluso puedan crear un objeto.
   Por último, se inyecta el código necesario para conseguir que la aplicación se registre por sí misma en el servidor en el momento de arranque (el funcionamiento se explica más adelante).
4. Se guarda en disco la aplicación modificada

3 Adaptación Dinámica de Aplicaciones

Al iniciar la ejecución, la aplicación modificada se registra a sí misma, gracias al código inyectado anteriormente, con un GUID (identificador único global, *Globally unique identifier*) en el servidor, con el objetivo de permitir a los aspectos que interactúen con ella. Para adaptar una aplicación se siguen los siguientes pasos:

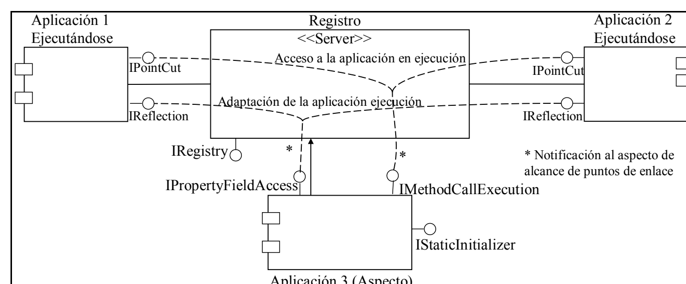

Fig. 2. Arquitectura del sistema B

1. El aspecto accede al servidor, le pasa una referencia a sí mismo y le informa acerca de los puntos de enlace de la aplicación destino (aquélla a adaptar) en los que está interesado, con el fin de ser notificado cuando se alcancen. La especificación de estos puntos de enlace (puntos de corte) se hace mediante un archivo XML que describe cómo debe hacerse el tejido de código. La sintaxis del archivo se valida contra un esquema XML que es una simplificación del utilizado en Weave.Net [10].
   La especificación se realiza de forma similar a los puntos de corte de AspectJ, pudiendo el aspecto especificar explícitamente los puntos de enlace que desea controlar (p.ej. la ejecución del método foo) o mediante reglas (p.ej. la ejecución de cualquier método público que reciba un parámetro de un tipo determinado).
   Mediante este sistema se separa por completo el código del aspecto (advice) de las reglas de tejido, con lo que se logra que no sea necesario añadir ninguna extensión al lenguaje en el que se implementen los aspectos, así como que el lenguaje en el que se implementen sea irrelevante. Asimismo se pueden cambiar las reglas de tejido sin tener que modificar el código del aspecto, lo que ofrece una mayor flexibilidad y permite distintas formas de utilizar el mismo aspecto con la misma aplicación o con otras, simplemente cambiando las reglas de tejido en el archivo XML.
   El aspecto debe haber implementado uno o varios de los interfaces `IMethodCallExecution` (para ejecuciones y llamadas a métodos y constructores), `IPropertyFieldAccess` (para accesos de lectura o escritura a campos y propiedades) o `IStaticInitializer` (para la inicialización estática), los cuales permitirán a la aplicación destino poder notificar al aspecto, mediante las rutinas que se le inyectaron, cuando alcance los puntos de enlace solicitados.
2. El servidor, mediante el interfaz `IReflection`, examina la aplicación destino (la que va a ser adaptada) y en función de los puntos de corte recibidos en el archivo XML, solicita a la aplicación, mediante el interfaz `IPointCut`, que notifique al aspecto cuando se alcancen los puntos de enlace especificados. Lo que hace internamente este interfaz es insertar en las tablas inyectadas anteriormente una refe-

rencia al aspecto, de modo que cuando se alcancen los puntos de enlace, la aplicación sepa si debe notificar tal evento o no. Esto sirve para obtener un mejor rendimiento de ejecución (sólo los puntos de enlace solicitados generan llamadas).
Con estos dos primeros pasos el aspecto ha designado los puntos de corte.
3. Cuando se alcanza en la aplicación destino alguno de los puntos de enlace solicitados, ésta invoca al aspecto o aspectos cuyas referencias están en la tabla asociada, enviándole una referencia de sí misma.
4. Mediante el uso de la referencia recibida y del interfaz `IReflection`, implementado por la aplicación destino, el aspecto puede acceder a la aplicación, obtener su estructura, invocar código de la aplicación, o ejecutar sus propias rutinas.
5. Si un aspecto ya no se necesita más, informa al servidor de esta circunstancia para no seguir siendo notificado. En este caso el servidor, mediante el interfaz `IPointCut` de la aplicación, elimina las referencias al aspecto en las tablas correspondientes, con lo que la aplicación deja de informar al aspecto.
De este modo el servidor puede llevar un registro de qué aplicaciones fueron adaptadas, por parte de qué aspectos y de qué forma, y cuándo dejaron de ser adaptadas.

Como todo el trabajo se realiza sobre código en lenguaje intermedio, MSIL, se obtienen dos beneficios principales: independencia del lenguaje y de la plataforma en la aplicación que va a ser adaptada y en los aspectos que la adaptan. Además, no es necesario conocer en el momento del diseño si la aplicación va a ser adaptada o no.

Con este sistema, también es posible *destejer* (borrar, eliminar) un aspecto cuando ya no se necesite más. Un ejemplo sencillo pueden ser las métricas de tiempo de ejecución: 1) cuando se necesita obtener alguna medida en una aplicación que está ejecutándose se le añade un aspecto que realice esta operación, 2) una vez que el aspecto ha adaptado a la aplicación se pueden obtener las métricas deseadas, y 3) cuando ya no se necesita la métrica se puede *destejer* el aspecto con lo que la aplicación vuelve al estado inicial.

Otro importante beneficio que se consigue es que no es necesario el código fuente de una aplicación para poder modificarla, ya que todo el proceso se realiza partiendo del código binario que se traduce a MSIL. Además, el sistema no necesita modificar la máquina abstracta con lo que se puede usar con cualquier aplicación estándar.

4 Ventajas del Sistema y Conclusiones

El trabajo que estamos llevando a cabo establece que, mediante la aplicación de técnicas de reflectividad computacional sobre una máquina virtual independiente del lenguaje y plataforma, es factible implementar un tejedor realmente dinámico de aspectos con un amplio conjunto de puntos de enlace. Mediante el uso de la plataforma .NET este sistema presenta las siguientes ventajas:

1. Independencia del lenguaje. Tanto la aplicación a adaptar como los aspectos que la adaptan se pueden escribir en cualquier lenguaje, ya que todo el proceso se realiza sobre código traducido al lenguaje intermedio MSIL.
2. Los aspectos y la funcionalidad principal siguen ciclos de vida completamente independientes. Gracias a esto se obtiene una mayor reusabilidad de ambos y una mayor facilidad de depuración al no estar el código enmarañado.

3. Independencia de la plataforma. Al realizar todo el proceso al nivel de la máquina virtual se obtiene total independencia de la plataforma sobre la que se implementa.
4. No es necesario disponer del código fuente de una aplicación para poder adaptarla, puesto que todo el proceso se lleva a cabo sobre código MSIL. Esto puede ser muy útil en el caso de tener que modificar una aplicación de terceros.
5. Utilización de máquina virtual estándar. El sistema no necesita modificar la máquina virtual, por lo que cumple la especificación ECMA. Por tanto, se puede utilizar el sistema con cualquier aplicación que funcione sobre el CLR estándar.
6. El sistema ofrece un amplio conjunto de puntos de enlace. El mecanismo que utiliza el sistema no limita el rango de puntos de enlace que puede capturar cualquier aplicación. De este modo se pueden desarrollar aspectos de la misma forma que en AspectJ, pero tejiéndolos en tiempo de ejecución. En esta versión el sistema puede controlar ejecuciones y llamadas a métodos y constructores, accesos de cualquier tipo a campos y propiedades y la inicialización estática.
7. Un aspecto se puede adaptar por medio de otro aspecto. Los aspectos son aplicaciones normales que trabajan dentro del sistema, por lo que siguen los mismos pasos que el resto de las aplicaciones y pueden ser adaptados de igual manera.
8. Es posible destejer aspectos que ya no se necesiten. Cuando un aspecto deja de ser necesario informa al servidor de esto con lo que deja de ser notificado, no incurriendo en una penalización innecesaria en el rendimiento.

References

1. Parnas, D., 1972. On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM, Vol. 15, No. 12.
2. Hürsch, W.L., Lopes, C.V., 1995. Separation of Concerns, Technical Report UN-CCS-95-03, Northeastern University, Boston, USA.
3. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J., 1997. Aspect Oriented Programming. In: Proceedings of European Conference on Object-Oriented Programming (ECOOP), vol. 1241 of Lecture Notes in Computer Science.
4. Popovici, A., Gross, Th., Alonso, G., 2001. Dynamic Homogenous AOP with PROSE, Technical Report, Department of Computer Science, ETH Zürich, Switzerland.
5. Ortin, F., and Cueva, J. M. Building a Completely Adaptable Reflective System. 2001. European Conference on Object Oriented Programming ECOOP'2001. Workshop on Adaptive Object-Models and Metamodeling Techniques, Budapest, Hungary, June 2001.
6. Francisco Ortin and Juan Manuel Cueva. Non-Restrictive Computational Reflection. Elsevier Computer Software & Interfaces. Volume 25, Issue 3, Pages 241-251. June 2003.
7. Francisco Ortin, Juan Manuel Cueva. Implementing a real computational-environment jump in order to develop a runtime-adaptable reflective platform. ACM SIGPLAN Notices, Volume 37, Issue 8, August 2002
8. Luis Vinuesa, Francisco Ortin. A Dynamic Aspect Weaver over the .NET Platform. Revised Papers of Metainformatics International Symposium, MIS 2003. Graz, Austria, September 2003. Vol. 3002 of Lecture Notes in Computer Science.
9. Bruno Cabral, Paulo Marques, Luís Silva, "IL Code Instrumentation with RAIL", .NET Developers Journal, Vol. 2, #1, pp. 34, January-2004
10. Donal Lafferty, Vinny Cahill. Language-Independent Aspect-Oriented Programming. OOPSLA'03, October 26–30, 2003, Anaheim, California, USA.