инженерия программного обеспечения

Григорьев Дмитрий Алексеевич,
Григорьева Анастасия Викторовна,
Сафонов Владимир Олегович

УДК 004.4”2

БЕСШОВНАЯ ИНТЕГРАЦИЯ АСПЕКТОВ В ОБЛАЧНЫЕ ПРИЛОЖЕНИЯ НА ПРИМЕРЕ БИБЛИОТЕКИ ENTERPRISE LIBRARY INTEGRATION PACK FOR WINDOWS AZURE И ASPECT.NET

Аннотация

Библиотека Enterprise Library Integration Pack for Windows Azure – это решение компании Microsoft для выделения «сквозной функциональности» при разработке облачных приложений. Использование этой библиотеки подразумевает модификацию исходного кода целевого приложения. На практике возникают ситуации, когда изменение исходного кода нежелательно. Данная статья описывает методику бесшовной интеграции аспектов и целевого проекта с помощью Aspect.NET, которая позволяет не менять исходный код целевого приложения.

Ключевые слова: Enterprise Library, MS Azure, аспектно-ориентированное программирование, бесшовная интеграция, Aspect.NET.

Любой программист сталкивался с задачей сопровождения кода. Как правило, для изменения поведения программы приходится вносить изменения в ее исходный код. Однако в ряде случаев было бы удобно добавлять новую функциональность в систему бесшовным образом, не затрагивая исходный текст проекта. Предположим, например, что заказчик просит временно добавить новый тип отчета, либо один из пользователей сообщил о проблеме с производительностью на его нестандартной машине. Другим примером служит ситуация, когда разным пользователям требуется, чтобы одна и та же функциональность была реализована различным образом. Вместо создания множества разных веток кодовой базы в системе контроля версий, имеет смысл сохранить только одну, а желаемое поведение обеспечивать бесшовной интеграцией.

Как известно, все пользовательские требования к программному продукту подразделяются на функциональные и нефункциональные [1]. Примером функционального требования для Интернет-магазина может являться интеграция базы товаров с поисковой машиной, вычисление скидки для конкретной группы покупателей или система построения отчетов и т. п. В то же время, к нефункциональным требованиям относятся различные требования безопасности, кэширование, протоколирование, управление производительностью (autoscaling) и пр.

Объектно-ориентированная методология программирования (ООП) существенно облегчает разработку функциональных требований и бизнес-логики, позволяя идентифицировать в предметной области объекты, выстраивать их иерархию и распределять между ними обязанности. В свою очередь, нефункциональные требования относятся к «сквозной функциональности» (cross-cutting concerns), которая горизонтально рассредоточена по иерархиям объектов предметной области, то есть множество объектов вовлечены в реализацию этого требования, например кэширования. Использование ООП в этом случае приводит к необходимости решать одну и ту же задачу отдельно для каждого уровня абстракции архитектуры. Исходный код такой системы обладает высокой связанностью, его сложно декомпозировать и сопровождать [2].

Аспектно-ориентированное программирование (АОП) предлагает решение данной проблемы – выделение «сквозной функциональности» в один модуль, аспект [15]. По своей сути реализация «сквозной функциональности» состоит в том, чтобы выполнить некие действия в определенных точках целевой программы или же расширить существующие классы новыми методами и полями. Все такие действия (advices) инкапсулируются в аспекте и снабжены описанием своих точек внедрения (joinpoints). Кроме того, чтобы влиять на поведение целевой программы, действия аспекта имеют доступ к контексту своих точек внедрения, например к ссылкам на объекты, окружающим данную точку.

Существенной проблемой при изменении поведения программы с помощью современных АОП-инструментов является отсутствие бесшовной интеграции аспектов и целевого исходного кода системы. Причиной этого является распространенное мнение, что АОП наиболее эффективно только в том случае, когда целевая система изначально разрабатывалась с его помощью, например все зависимости между интерфейсами и реализующими их классами разрешаются через АОП-инструмент, а система декомпозирована таким образом, что все действия аспектов идеально соответствуют своим точкам внедрения. К сожалению, например для унаследованного кода этот принцип не работает, тем более при наличии требования бесшовной интеграции.

Данная статья описывает методику бесшовной интеграции аспектов и целевого кода, а также раскрывает механизмы ее реализации на базе АОП-инструмента Aspect.NET [4]. Методика поясняется на примере интеграции в исходный код проекта сервисов библиотеки Microsoft Enterprise Library Integration Pack for Windows Azure [5]. Для простоты воспользуемся MS Visual Studio 2012 и примерами из лабораторных работ (Hands-On Labs) данной библиотеки [6], однако прежде рассмотрим альтернативные АОП-технологии.

Наиболее популярным АОП-инструментом для платформы MS .NET является PostSharp [7]. Аспекты в нем определяются в виде пользовательских атрибутов, а точками внедрения могут являться вызовы конкретных методов, доступ и запись значений свойств, полей класса, генерация события, либо исключения. Слияние аспектов и целевого кода производится компилятором PostSharp на этапе пост-компиляции.

Для добавления аспектов необходимо добавить в целевой проект ссылки на служебные сборки PostSharp, вставить исходные файлы аспектов в проект и пометить специальными атрибутами точки внедрения в коде целевого проекта. Также точки внедрения аспекта можно указать объявлением маски целевого класса или метода в файле AssemblyInfo.cs, например

```
[assembly: Trace(AttributeTargetTypes=
"AdventureWorks.BusinessLayer.*")]
```

К сожалению, все это требует хранения и определений аспектов и правил их внедрения вместе с исходными файлами целевого проекта, что приводит к их высокой связанности (high coupling). В дальнейшем если приоритеты команды изменятся и потребуется изъять PostSharp, либо сменить его на другой АОП-инструмент, то сделать это будет затруднительно. Любые способы бесшовной интеграции целевого проекта со сборками сторонних библиотек в PostSharp неофициальны и недокументированы [8].

Наконец, PostSharp – коммерческий продукт, и это затрудняет его применение в ряде академических проектов.

Библиотека Microsoft Enterprise Library [9] представляет собой набор наиболее удачных образцов решения стандартных проблем и также предназначена для реализации «сквозной функциональности». В нее входят следующие функциональные блоки:

- Caching Application Block для поддержки кэширования;
- Cryptography Application Block для поддержки шифрования;
- Data Access Application Block для поддержки работы с базами данных;
- Exception Handling Application Block для реализации стратегий обработки исключений;
- Logging Application Block для поддержки протоколирования;
- Security Application Block для поддержки авторизации и безопасности приложений;
- Validation Application Block для поддержки механизмов валидации данных бизнес-объектов.

В 2011 году компания Microsoft выпустила расширение Enterprise Library Integration Pack for Windows Azure [5], которое содержит функциональные блоки для управления производительностью (Auto-scaling Application Block) и ограничения функциональности под нагрузкой (Transient Fault Handling Block). Программист может реализовать ту или иную «сквозную функциональность», если в исходном коде своего проекта вызовет методы из набора классов соответствующего функционального блока. С точки зрения АОП очевидно, что местоположение этих вызовов в исходном коде целевого приложения является совокупностью точек внедрения для действий соответствующего аспекта.

Чтобы облегчить внедрение (weaving) функциональных блоков в целевое приложение, компания Microsoft представила Unity Application Block – IOC-контейнер с ограниченной поддержкой АОП [2]. Его механизмы позволяют во время исполнения целевой программы перехватывать вызовы заданных методов и передавать управление цепочке методов – «перехватчиков» (interceptors). Цепочка «перехватчиков» может вызываться как перед вызовом целевого метода, так и после него, имея возможность обратиться к результату выполнения данного целевого метода.

Кроме того, Unity позволяет разрывать зависимости между конкретным классом и его интерфейсом. Иными словами, за создание конкретного класса отвечает специальная фабрика Unity, что дает возможность задавать правила сопоставления интерфейса и класса не в исходном коде, а в конфигурационном файле. К сожалению, говорить в этом случае о бесшовной интеграции также не приходится, так как процесс применения Unity включает в себя создание Unity-контейнера, а затем обращение к нему за конкретным классом в том месте исходного кода, где требуется получить реализацию для интерфейса.

Проект Aspect.NET, разрабатываемый коллективом авторов с 2004 г. в СПбГУ, также представляет собой аспектно-ориентированную среду разработки программ для платформы Microsoft.NET. Аспекты определяются на метаязыке Aspect.NET ML [3], либо с помощью пользовательских атрибутов в отдельных проектах MS Visual Studio, а их слияние с целевым кодом происходит на уровне сборок статически, то есть после этапа компиляции. Вначале компилируется сборка с аспектами, из которой компоновщик аспектов (weaver) извлекает правила внедрения каждого действия аспекта, содержащиеся в его пользовательском атрибуте AspectAction(). Затем компоновщик анализирует MSIL-код откомпилированной сборки целевого проекта, находит соответствующие места внедрения (сканирование) и вставляет туда действия из аспектной сборки. Операции сканирования и внедрения действий аспектов разделены, что позволяет пользователю просматривать и фильтровать точки внедрения. Весь анализ и модификация .NET сборок производится с помощью сервисов отражения (reflection) и библиотеки MS Phoenix [15], которая декомпилирует сборку и представляет ее в виде набора высокоуровневых инструкций.

Любое действие можно вставлять перед (ключевое слово %before), после (%after) или вместо (%instead) вызова заданного целевого метода. Название целевого метода задается с помощью регулярных выражений относительно его сигнатуры. Местонахождение точки внедрения внутри конкретного метода или класса можно фильтровать по ключевому слову %within.

Внутри действий можно использовать свойства базового класса Aspect, предоставляющие доступ к контексту точки внедрения, например «this» и «target» объектам целевого метода, его метаданным (типа MethodInfo), отладочной информации, результату выполнения и пр. Причем компоновщик вставляет в итоговую сборку MSIL-код для передачи контекста только в том случае, если действие аспекта использует его. Аргументы целевого метода передаются через аргументы действия. Таким образом, аспект – это класс производный от служебного базового класса Aspect, а его действиями будут открытые статические методы, помеченные атрибутом AspectAction(). В случае, когда определение аспекта задается на Aspect.NET ML, специальный конвертер переводит его в определение с пользовательскими атрибутами.

Несмотря на то, что PostSharp обладает более высокой выразительной мощностью в определении аспектов, к преимуществам Aspect.NET можно отнести:

– более высокую в ряде случаев производительность результирующего кода;

– возможность для пользователя фильтровать вручную требуемые точки внедрения;

– компактное и лаконичное определение аспектов с помощью метаязыка;

– совместимость с академической версией .NET – MS Rotor [10].

В свою очередь, MS Enterprise Library имеет множество готовых функциональных блоков и не требует никаких сторонних инструментов для своей работы, но возможности Unity по их аспектно-ориентированному применению серьезно ограничены. Перед авторами была поставлена задача заменить Unity и реализовать бесшовную интеграцию этих функциональных блоков с помощью Aspect.NET. Это позволило бы объединить сильные стороны двух инструментов: большой набор готовых аспектов, приемлемую скорость результирующего кода и независимость целевого проекта от АОП-инструмента. Объектом исследования были выбраны Autoscaling Application Block и Transient Fault Handling Block как наиболее полезные для разработки облачных приложений на платформе MS Azure. По мнению авторов, фокусирование на решении проблем сквозной функциональности в области разработки облачных приложений более перспективно, так как здесь возможности применения аспектно-ориентированного программирования еще не до конца изучены [11].

Для практического ознакомления с EL Integration Pack for Windows Azure компания Microsoft предлагает серию лабораторных работ (Hands-on Labs). Программисту предоставляется исходный проект и методические указания по его пошаговому изменению. Полученный результат можно сравнить с эталонным конечным проектом, в котором теперь задействуется тот или иной функциональный блок EL Integration Pack for Windows Azure. Если на основе этих методических указаний составить аспект и применить его с помощью Aspect.NET к начальному проекту, то результирующая сборка будет обладать функциональностью соответствующего эталонного проекта, но без модификации кода исходного проекта. Однако стоит учитывать, что изменения в конфигурационные файлы исходного проекта необходимо все же вносить вручную, поскольку АОП позволяет удалять сквозную функциональность лишь из программного кода.

Рассмотрим упражнение «Hands-on Lab 1: Using the Logging Application Block with Windows Azure Storage», где путем добавления ссылок на сборки EL производится подключение функционального блока протоколирования к исходному проекту, а затем вызов его метода для передачи сообщения в облачное хранилище диагностической информации WAD (листинг 1). Это дает возможность настраивать параметры сбора и хранения отладочных сообщений через графический интерфейс Logging Application Block, либо через его конфигурационные файлы.

Листинг 1

```
//Веб-роль, на странице которой тестируется Logging Application Block
public partial class Default : System.Web.UI.Page {
     //Сообщение отсылается в обработчике щелчка мыши по кнопке страницы
    protected void LogButton_Click(object sender, EventArgs e) {
         Microsoft.Practices.EnterpriseLibrary.Logging.
           Logger.Write("Message from the Logging Application Block");
     }
}
```

Итак, наша задача заключается в том, чтобы перенести все зависимости от EL и вызовы методов протоколирования в отдельный проект с аспектом. Применив затем с помощью Aspect.NET данный аспект к исходному проекту, мы получим его бесшовную интеграцию с Logging Application Block.

Традиционно, в системе Aspect.NET подобные задачи решаются размещением кода протоколирования в действии аспекта и вставкой его перед, после или вместо вызова целевого метода в исходном проекте. В нашем случае целевой метод – это обработчик события щелчка мыши LogButton_Click() класса веб-страницы Default, причем созданием объекта этого класса и отправкой ему событий занимается среда ASP.NET и сервер IIS. Это означает, что код вызова нашего целевого метода располагается вне сборки исходного проекта и недоступен Aspect.NET.

По мнению авторов, перехват вызовов методов, которые реагируют на внешние события, может быть осуществлен через наследование классов. Если в аспектном проекте создать класс, который наследует от целевого класса, а затем подменить им свой базовый класс в сборке исходного проекта, то требуемый перехват можно осуществить в переопределенном виртуальном методе (см. листинг 2). Специальный пользовательский атрибут [ReplaceBaseClass] предписывает компоновщику Aspect.NET заменить целевой класс своим аспектным наследником:

1. Заменить в исходной сборке все вызовы методов базового целевого класса (в том числе и конструкторы) на вызовы методов его наследника в аспектной сборке.

2. Принудительно объявить виртуальными те методы целевого класса, которые переопределены в замещающем его наследнике. Если они закрыты (private), то сделать их защищенными (protected).

3. Если вызов этих методов в исходной сборке производится с помощью MSIL-инструкции call или ldftn, заменить их на callvirt и ldvirtftn соответственно.

Листинг 2

```
//Проект с замещающим аспектным наследником
[AspectDotNet.ReplaceBaseClass]
public class AspectClass : Default {
    protected void LogButton_Click(object sender, EventArgs e) {
        Microsoft.Practices.EnterpriseLibrary.Logging.
            Logger.Write("Message from the Logging Application Block");
         base.LogButton_Click(sender, e);
     }
}

//Исходный проект, после отделения зависимости от Logging Application Block
public partial class Default : System.Web.UI.Page {
    protected void LogButton_Click(object sender, EventArgs e) {}
}
```

4. Объединить с помощью инструмента ILRepack (из проекта Mono.Cecil [12]) сборки с аспектом и исходную.

5. Присвоить какое-нибудь служебное имя базовому целевому классу, а его первоначальное имя – замещающему наследнику из аспекта.

Преимуществами такого алгоритма является простота подмены классов для пользователя, а также использование только штатного синтаксиса языка .NET. Теперь с помощью аспекта можно уточнять поведение любого метода целевого класса, реализовывать в нем дополнительные интерфейсы, накладывать различные пользовательские атрибуты и т. п. По сравнению с обобщенным введением интерфейса в PostSharp, мы теряем возможность декларативно вводить интерфейсы в целевые классы, однако получаем возможность использовать их члены в аспекте. Вычислительная сложность такого алгоритма O(N), где N – количество MSIL-инструкций в исходной сборке. Однако в компоновщике аспектов Aspect.NET эти операции объединены с операциями вставки аспектов (также сложностью O(N)), поэтому общая асимптотическая сложность компоновки аспектов не увеличилась.

Следует отметить, что данное решение все же не типично для АОП, где аспект описывается максимально изолированно от целевого класса, в то время как наследование предполагает сильную связь. Одновременно выразительные средства АОП решают ограниченный круг задач и, если есть возможность решить задачу более элегантным способом, стоит им воспользоваться. Косвенно это подтверждается действиями команды PostSharp, которая взяла курс на создание инструментов (PostSharp Toolkits [13]), предназначенных для решения специфичных задач (например многопоточности и диагностики). Применение для тех же целей исключительно аспектов PostSharp вызывает практические затруднения.

Рассмотрим теперь «Hands-on Lab 6: Implementing Throttling Behavior», где иллюстрируется ограничение функциональности под нагрузкой с использованием сервисов функционального блока Autoscaling Application Block. Отдельный компонент Autoscaler занимается мониторингом диагностической информации и, в зависимости от текущей нагрузки на облако, устанавливает свойство ThrottlingMode в файле конфигурации исходного проекта. В зависимо-

Листинг 3

```
//Веб-роль, на странице которой тестируется Autoscaling Application
//Block
public partial class Default : System.Web.UI.Page {
   protected override void OnPreRenderComplete(EventArgs e) {
       base.OnPreRenderComplete(e);
       string throttlingMode = RoleEnvironment.
                    GetConfigurationSettingValue("ThrottlingMode");
       switch (throttlingMode)
        {
          case "HighActivity":
            this.ThrottlingLabel.Text = "Работа при высокой активности…";
            break;
           default:
             this.ThrottlingLabel.Text = "Работа при обычной
                                                    активности…" ;
            this.DoSomeUsualWork();
            break;
        }
    }
   private void DoSomeUsualWork() {/*…*/}
}
```

сти от значения этого свойства, какие-то из методов класса веб-страницы могут изменять свое поведение (листинг 3).

Данный метод можно перенести в аспект, и тогда целевой класс будет сконцентрирован только на решении своей задачи, в то время как компонент Autoscaler и бесшовная интеграция с аспектом обеспечит реакцию на повышенную нагрузку. Задачу можно было бы решить аналогично предыдущему примеру, но здесь есть препятствие в виде вызова закрытого в целевом классе метода DoSomeUsualWork(). Для того чтобы он стал доступным замещающему наследнику, компоновщик аспектов мог бы принудительно сделать этот метод защищенным. Однако это нарушит инкапсуляцию целевого класса, и единственный способ сохранить ее – использовать рефлексию .NET. Закрытые члены целевого класса становятся полями его аспектного наследника, которые инициализируются в конструкторе. Также предположим, что в замещающем наследнике целевого класса нам понадобится вызвать метод OnPreRenderComplete следующего по иерархии базового класса System.Web.UI.Page. Защищенные и открытые методы целевого класса используются в его замещающем аспектном наследнике без ограничений. Итоговый аспект представлен в листинге 4.

Листинг 4

```
using System.Reflection;
[AspectDotNet.ReplaceBaseClass]
public class AspectClass : _Default {
    MethodInfo DoSomeUsualWork;

    public AspectClass() {
       Type BaseType = this.GetType().BaseType;
       //Получение ссылки на закрытый метод целевого класса _Default
        DoSomeUsualWork = BaseType.GetMethod("DoSomeUsualWork",
             BindingFlags.NonPublic | BindingFlags.Instance);
       //Ссылка на метод базового класса System.Web.UI.Page
        PageOnPreRenderComplete = base.GetType().BaseType.
             GetMethod("OnPreRenderComplete",
             BindingFlags.NonPublic | BindingFlags.Instance);
   }

    protected override void OnPreRenderComplete(EventArgs e) {
       //Вызываем метод базового класса System.Web.UI.Page
       PageOnPreRenderComplete.Invoke(this, new object[] { e });

       string throttlingMode = RoleEnvironment.
                        GetConfigurationSettingValue("ThrottlingMode");
       switch (throttlingMode) {
            case "HighActivity":
               //Использование в аспекте члена целевого класса _Default
             this.ThrottlingLabel.Text = "Работа при высокой
                                                          активности...";
                break;
           default:
               this.ThrottlingLabel.Text = "Работа при обычной
                                                          активности...";
                //Вызов закрытого члена целевого класса _Default
               DoSomeUsualWork.Invoke(this, null);
                break;
        }
   }
}
```

Следующий пример бесшовной интеграции основан на примере «Hands-on Lab 11: Transient Fault Handling». Здесь задача заключается в том, чтобы добавить в целевой код работы с базой данных стратегию обработки исключительных ситуаций. Сама стратегия отделена от кода, работающего с базой данных, и конфигурируется средствами EL. Например, для любого запроса к базе данных можно составить стратегию вида: попытаться совершить 4 последовательных запроса, если каждый из предыдущих совершается неудачно. При этом между вторым и третьим запросом должна быть пауза в 5 сек. Исходный код целевого класса приведен в листинге 5.

Запрос к базе данных производится с помощью вызова метода SqlCommand.ExecuteReader(), который может выбросить исключение при сбое соединения с базой данных. Чтобы применить к нему стратегию обработки исключительных ситуаций, необходимо выполнить данный блок в рамках метода ExecuteAction() класса Microsoft.Practices.TransientFaultHandling.RetryPolicy<T>, где T – класс, реализующий стратегию. Получить объект этого класса можно через специальный менеджер TransientFaultHandling.RetryManager, который должен быть инициализирован библиотекой EL и передан нам в конструкторе целевого класса. Для этого требуется создать объект нашего замещающего наследника с помощью средств EL (см. листинг 6). Средства АОП-программирования, в том числе и Aspect.NET, предоставляют механизмы замены вызова целевого метода на действие аспекта. Для подмены создания класса замещающего наследника мы этим и воспользовались, однако в вышеупомянутом случае ситуация осложняется тем, что «целевым» является весь блок using. Для таких ситуаций и предназначен наш механизм замены целевого класса замещающим наследником. Результирующий код аспекта в сокращенном виде приведен в листинге 7.

Далее будут рассмотрены некоторые инженерные проблемы адаптации Aspect.NET

Листинг 5

```
public class Main : Form {
    private void ExecuteQueryButton_Click(object sender, EventArgs e) {
         //…
          try {
            using (var connection = new
                  SqlConnection(ConfigurationManager.ConnectionStrings
                                  ["Northwind"].ConnectionString)) {
               connection.Open();
               var command = new SqlCommand("dbo.GetProductDetails",
                                                              connection)
                           {CommandType = CommandType.StoredProcedure};
               // …Заполнение параметров command…
               using (var reader = command.ExecuteReader()) {
                   while (reader.Read()) {
                         //…Обрабатываем результат успешного запроса
                     }
                }
            }
          } catch (Exception ex) {
             //…
          }
    }
}
```

Листинг 6

```
//Все аспекты с действиями в Aspect.NET должны наследоваться от Aspect
class ChangeRunAspect : AspectDotNet.Aspect {
        [AspectDotNet.AspectAction("%instead %call *.Application.Run")]
         static public void ReplaceAction() {
            Application.Run(EnterpriseLibraryContainer.Current.
                                                GetInstance<AspectMain>());
          }
}
```

для реализации методики бесшовной интеграции. Авторы уверены, что их изучение позволит читателям лучше понять процесс создания подобных инструментов, трансформирующих целевые сборки на уровне MSIL-инструкций.

Листинг 7

```
[AspectDotNet.ReplaceBaseClass]
public partial class AspectMain : Main {
     private RetryManager retryManager;

     public AspectMain(RetryManager retryManager) {
            this.retryManager = retryManager;
    }

     private void ExecuteQueryButton_Click(object sender, EventArgs e) {
          //…
          try {
             using (var connection = new
                   SqlConnection(ConfigurationManager.ConnectionStrings
                                        ["Northwind"].ConnectionString)) {
               connection.Open();
               var command = new SqlCommand("dbo.GetProductDetails",
               connection){CommandType = CommandType.StoredProcedure};
               // …Заполнение параметров command…
               var policy = this.retryManager.GetRetryPolicy
               <HolSqlTransientErrorDetectionStrategy>("HOL Strategy");
               policy.Retrying += (s, a) =>
                   Log.Invoke(this,new object[] {"Попытка нового
                                                        соединения…"});
               policy.ExecuteAction(() => {
                   using (var reader = command.ExecuteReader()) {
                        while (reader.Read()) {
                          //…Обрабатываем результат успешного запроса
                        }
                   }
               });
             }
          } catch (Exception ex) {
               //…
          }
    }
}
```

Современная разработка на MS Azure подразумевает использование MS Visual Studio (MS VS) 2012, в которой серьезно упрощено развертывание облачных приложений. Программисту достаточно сделать несколько щелчков мышью, чтобы его приложение отправилось в облако. Последняя опубликованная версия Aspect.NET 2.2 совместима лишь с MS Visual Studio 2008, так как именно с ее COM-библиотекой по работе с отладочной информацией (msdia90.dll) сконфигурирован MS Phoenix. Установка MS VS более новой версии приводит к тому, что MS Phoenix автоматически пытается использовать обновленную msdia90.dll и завершается с ошибкой. К сожалению, с 2008 года развитие кросс-платформенной среды построения оптимизирующих компиляторов MS Phoenix прекратилось. При этом его широкие возможности все еще используются в академической среде, и несовместимость с современными версиями MS VS является серьезным препятствием для таких проектов. Решение заключается в том, чтобы в директории компоновщика аспектов находилась библиотека с нужной версией, а также с ее манифестом msdia90.manifest (см. листинг 8).

Следующей проблемой становится публикация средствами MS VS результирующей сборки с внедренными аспектами в облако MS Azure. Например, для облачного веб-приложения, после выбора в свойствах проекта пункта контекстного меню Publish, процесс публикации заключается в компиляции проекта, упаковке его сборки вместе с используемыми библиотеками и другими служебными файлами в специальный архив, а затем передаче его службам MS Azure. В свою очередь, компоновщик аспектов в Aspect.NET – это консольное приложение, которое в виде параметров командной строки принимает пути к сборкам: аспектной, целевой и результирующей:

```
weaver -aspects MyAspect.dll
       -in HelloWorld.exe
        -out AspectHelloWorld.exe
```

Следовательно, если бы удалось выполнить эту команду сразу после этапа компиляции и подменить результирующей сборкой целевую, то MS VS отправит на публикацию приложение с внедренными аспектами. Более того, теперь запуск на выполнение целевого проекта приведет к запуску проекта с внедренными аспектами, что облегчает аспектно-ориентированную разработку программ. В итоге был разработан следующий алгоритм разработки и бесшовной интеграции аспектов в MS VS 2012 с помощью Aspect.NET

Вход: Проект с исходными текстами целевого приложения, к которому нужно применить аспекты.

1. Создаем отдельный проект для аспекта (типа Class Library) и объединяем его с целевым проектом в рамках общего решения (solution).

2. Чтобы обеспечить интеграцию аспекта с целевой сборкой, в свойствах его проекта на вкладке Build Events добавляем соответствующий скрипт (листинг 9).

3. Выключаем процесс применения аспектов при неудачной компиляции целевого проекта путем выбора «When the build

Листинг 8

```
<?xml version="1.0" encoding="utf-8"?>
<asmv1:assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-
com:asm.v1" xmlns:asmv1="urn:schemas-microsoft-com:asm.v1" >
<assemblyIdentity type="win32" name="msdia90" version="9.0.30729.4947"
                        language="neutral" processorArchitecture="x86" />
<file name="msdia90.dll">
     <comClass clsid="{B86AE24D-BF2F-4ac9-B5A2-34B14E4CE11D}"
                                                  threadingModel="both"/>
</file>
</asmv1:assembly>
```

updates the project output» в опции «Run the post-build event».

4. Настраиваем порядок сборки проектов (Project Build Order в контекстном меню аспектного проекта) так, чтобы первым собирался целевой проект, затем аспектный.

5. Устанавливаем целевой проект для запуска (в его контекстном меню «Set as StartUp Point»).

Выход: Решение (solution) с аспектным и не модифицированным целевым проектом. При компиляции аспектного проекта будет происходить внедрение аспектов, а при запуске в MS VS – запуск результирующей сборки. В свою очередь, компиляция и запуск целевого проекта приведет к запуску исходного проекта без внедренных аспектов.

При бесшовной интеграции аспектов с целевой сборкой желательно иметь возможность применить отладчик MS VS при тестировании результирующей сборки. Вся информация о связи исходного кода и точки останова (breakpoint) в исполняемом файле содержится в pdb-файлах, которые производит компилятор MS VS. После применения компоновщика создается исполняемый файл с интегрированными аспектами, однако отладочные pdb-файлы аспектного и целевого проекта не соответствуют ему. Это делает невозможным установку точек останова в аспектах и пошаговую отладку результирующей сборки.

Итак, наша последняя задача заключается в том, чтобы создать отладочный pdb-файл, соответствующий результирующей сборке. В процессе своей работы компоновщик вставляет в целевую сборку дополнительные MSIL-инструкции для вызова действий аспектов, а также передачи в них контекста. Эти операции производится с помощью MS Phoenix, а Mono.Cecil отвечает за этапы 2, 4 и 5 в вышеприведенном алгоритме по подстановке замещающего наследника. Как MS Phoenix, так и Mono.Cecil при преобразовании целевой сборки автоматически поддерживают соответствие с отладочной информацией. Иными словами, при соответствующих параметрах этих инструментов отладочный pdb-файл для результирующей сборки будет создан автоматичес-

Листинг 9

```
:: Укажем папку со сборкой целевого проекта
set TargetAssemblyDir=C:\HelloWorld\HelloWorld\bin\Debug\
:: Название его сборки
set TargetAssembly=HelloWorld
:: Ее расширение
set TargetAssemblyExt=.exe
:: Для каждого нового аспекта или целевого проекта необходимо менять лишь
:: вышеуказанные переменные


:: Зададим путь к директории Aspect.NET
set AspectDotNetDir=C:\AspectDotNet

set
TargetAssemblyPath=%TargetAssemblyDir%%TargetAssembly%%TargetAssemblyExt%
set TargetAssemblyName=%TargetAssembly%%TargetAssemblyExt%


cd %AspectDotNetDir%
weaver -aspects $(TargetPath) -in %TargetAssemblyPath% -out
                                                    %TargetAssemblyName%
:: Подмена сборки в целевом проекте результирующей
move /Y %TargetAssemblyName% %TargetAssemblyPath%
```

ки. Все, что нам теперь остается,– это средствами MS Phoenix добавить отладочную информацию для новых MSIL-инструкций. С учетом того, что их добавление не влияет на исходный текст целевого проекта, необходимо лишь скопировать в новые MSIL-инструкции отладочную информацию от предшествующих им инструкций в целевой сборке. Это реализуется копированием свойства DebugTag в классе Phx.IR.Instruction – высокоуровневого представления инструкций в MS Phoenix. Теперь отладчик за один шаг сможет выполнить новые инструкции и перейти внутрь действия аспекта, а далее будет задействована отладочная информация аспектной сборки, которая уже была создана компилятором .NET.

Бесшовная интеграция аспектов в целевые приложения дает возможность эффективно решать проблемы сопровождения проектов, когда необходимо добавлять новую функциональность без изменения исходного кода. Представленная методика позволяет применить сторонние библиотеки для устранения сквозной функциональности, избегая при этом зависимости от них в исходном коде целевого проекта. В ходе дальнейшей разработки можно безболезненно отказаться от использования выбранной библиотеки, либо заменить ее на другую. Таким образом, существенно снижаются риски того, что неудачно выбранное АОП-решение или библиотека повлекут за собой существенную переделку целевого проекта. Реализация данной методики на базе Aspect.NET позволяет программистам использовать привычную среду разработки MS Visual Studio 2012, а также простой процесс создания аспектов, предоставляющих доступ к сервисам сторонних библиотек. Все упомянутые аспекты доступны на сайте проекта Aspect.NET [4].

Литература

1. *Ларман К.* Применение UML и шаблонов проектирования. 2-е издание. // М.: Изд-во Вильямс, 2004.

2. *Эспозито Д.* Аспектно-ориентированное программирование, перехват и Unity 2.0 // MSDN Magazine, 12.2010 // Режим доступа [проверено 01.09.2012]: http://msdn.microsoft.com/ru-ru/magazine/gg490353.aspx .

3. *Григорьев Д.А.* Реализация и практическое применение аспектно-ориентированной среды программирования для Microsoft .NET // Научно-технические ведомости // СПб.: Изд-во СПбГПУ, 2009. № 3. 225 с.

4. *Сафонов В.О.* Аспектно-ориентированное программирование // Учебное пособие // СПб.: Изд-во СПбГУ, 2011. 28 с.

5. Сайт проекта Enterprise Library 5.0 Integration Pack for Windows Azure // Режим доступа [проверено 01.09.2012]: http://entlib.codeplex.com/wikipage?title=EntLib5Azure .

6. Сайт проекта Hands-On Labs for Enterprise Library 5.0 Integration Pack for Windows Azure // Режим доступа [проверено 01.09.2012]: http://www.microsoft.com/en-us/download/details.aspx?id=28785 .

7. Сайт проекта PostSharp // Режим доступа [проверено 01.09.2012]: http://www.sharpcrafters.com/ .

8. *Davis D.* Applying aspects to 3rd party assemblies using PostSharp // Режим доступа [проверено 01.09.2012]: http://programmersunlimited.wordpress.com/2011/07/27/applying-aspects-to-3rd-party-assemblies-using-postsharp/ .

9. Сайт проекта MS Enterpise Library // Режим доступа [проверено 01.09.2012]: http://msdn.microsoft.com/en-us/library/ff648951.aspx .

10. Сайт проекта MS SSCLI 2.0 // Режим доступа [проверено 01.09.2012]: http://www.microsoft.com/en-us/download/details.aspx?id=4917 .

11. *Григорьева А. В.* Аспектно-ориентированный рефакторинг облачных приложений MS Azure с помощью системы Aspect.NET // Компьютерные инструменты в образовании, 2012. № 1. С. 21–30.

12. Сайт проекта Mono.Cecil // Режим доступа [проверено 01.09.2012]: http://www.mono-project.com/Cecil .

13. Сайт проекта PostSharp Toolkits // Режим доступа [проверено 01.09.2012]: https://github.com/sharpcrafters/PostSharp-Toolkits/wiki .

14. Сайт проекта Aspect.NET // Режим доступа [проверено 01.09.2012]: http://aspectdotnet.org/ .
15. Сайт проекта MS Phoenix // Режим доступа [проверено 01.09.2012]: http://research.microsoft.com/en-us/collaboration/focus/cs/phoenix.aspx .

Abstract

The Enterprise Library Integration Pack for Windows Azure is a solution by Microsoft for separation of cross-cutting concern in developing cloud applications. Using this library implies modification of the source code of the target application. In practice, there appear situations when any update of the source code is undesirable. The paper covers a method of seamless integration of aspects and the target project with Aspect.NET that allows us to avoid changing the source code of the target application.

Keywords: Enterprise Library, MS Azure, aspect-oriented programming, seamless integration, Aspect.NET.

Григорьев Дмитрий Алексеевич,
кандидат физико-математических наук, доцент кафедры информатики математико-механического факультета Санкт-Петербургского государственного университета,
gridmer@mail.ru,

Григорьева Анастасия Викторовна,
аспирант кафедры информатики математико-механического факультета СПбГУ,
nastya001@mail.ru,

Сафонов Владимир Олегович,
доктор технических наук, профессор кафедры информатики математико-механического факультета СПбГУ,
vosafonov@gmail.com