Компьютерные инструменты в образовании, 2016
№ 6: 5–19
УДК: 004.4'2
http://ipo.spb.ru/journal

ИНЖЕНЕРИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

РАСШИРЕНИЕ MS VISUAL STUDIO ДЛЯ БЕСШОВНОГО АСПЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

Григорьев Д.А.[1], Григорьева А.В.[1], Зотов М.А.[1], Макаров С.А.[1]

[1]СПбГУ, Санкт-Петербург, Россия

Аннотация

Аспектно-ориентированный подход позволяет отделить бизнес-логику приложения от сквозной функциональности. В данной работе описывается реализация системы Aspect.NET, которая интегрирована в Microsoft Visual Studio 2013 и позволяет создавать аспектно-ориентированные программы на платформе Microsoft.NET. Сквозная функциональность представляется в виде статических методов класса аспекта, а их интеграция в целевую сборку производится отдельной консольной программой — компоновщиком на этапе пост-компиляции с помощью библиотеки Mono.Cecil. Данный подход позволяет на этапе компиляции интегрировать новую функциональность в целевой проект без его модификации.

Ключевые слова: *аспектно-ориентированное программирование, аспекты, расширение Visual Studio, статическое внедрение аспектов.*

Цитирование: Григорьев Д.А., Григорьева А.В., Зотов М.А., Макаров С.А. Расширение MS VISUAL STUDIO для бесшовного аспектно-ориентированного программирования // Компьютерные инструменты в образовании, 2016. № 6. С. 5–19 .

1. ВВЕДЕНИЕ

Аспектно-ориентированное программирование (АОП) — парадигма программирования, в основе которой лежит идея выделения сквозной функциональности в отдельные модули, называемые аспектами. В таких модулях описываются действия, выполняемые в определенных точках программы и правила их «вплетения» в эти точки. К сквозной функциональности относятся: протоколирование, кэширование, сбор диагностической информации, авторизация и т. д. Инкапсуляция сквозной функциональности в отдельных модулях повышает качество архитектуры программного проекта в соответствии с принципом единственной ответственности SRP [1]. При этом основное приложение будет сосредоточено на бизнес-логике, в то время как аспекты на нефункциональных требованиях.

Для АОП разработки применяется множество инструментов, самыми популярными из которых являются AspectJ [2] и PostSharp [3]. Однако, проведенный в процессе написания работы, анализ предметной области показал, что популярность АОП-инструментов уступает альтернативным технологиям повышения качества программного обеспечения, таким, например, как рефакторинг [4] и IOC-контейнеры [5]. Существенной проблемой при изменении поведения программы с помощью современных АОП-инструментов

является отсутствие бесшовной интеграции аспектов и целевого исходного кода системы. Термин «бесшовная интеграция» применительно к аспектам и целевому коду можно сформулировать следующим образом — это такое расширение функциональности целевого кода, при котором не требуется вносить изменения в целевой проект, включая его код, файлы настроек и свойства проекта. При бесшовной интеграции в этом смысле у пользователя должна быть возможность отдельно компилировать, запускать и тестировать аспектную и целевую сборки, без необходимости иметь их исходный код. Данное определение отличается от «бесшовности» применения ряда АОП-инструментов (PostSharp, FuseJ [6] и др.), когда целевой код не зависит от аспектов, однако они тесно интегрированы в его проект, что препятствует тестированию бизнес-логики и затрудняет возможный отказ от применяемого АОП-инструмента.

В то время как пользователи AspectJ для Java-платформы имеют возможность создавать свои аспекты и правила внедрения в одном модуле и внедрять их в целевые сборки на бинарном уровне, то у платформы MS .NET до сих пор нет полноценной поддержки бесшовного АОП-программирования. Были разработаны несколько экспериментальных АОП-инструментов, которые позволяют бесшовно внедрять аспекты, например, проекты SheepAspect [7] и AOP.NET [8]. Оба эти проекта предлагают хранить определения аспектов вместе с правилами их внедрения и вставлять их в целевую сборку на этапе посткомпиляции, однако ни один из них не вышел из стадии альфа-версии и их развитие приостановлено.

Основные преимущества АОП проявляются при введении в процесс разработки с самого начала. Однако парадигма АОП позволяет проводить и АОП-рефакторинг [9] унаследованного кода (то есть повторяющиеся куски кода, пригодные для повторного использования, выносятся в отдельный «аспектный» модуль и связываются с целевым кодом через инструмент АОП). Кроме возможности повторного использования кода со сквозной функциональностью, данный подход позволяет улучшить метрики качества целевых классов, когда вместо одной сложной сущности с появляются несколько более простых. При наличии инструмента бесшовной интеграции сквозная функциональность выносится в отдельный проект (solution), который зачастую не имеет ссылок на целевой проект и может быть затем повторно использован простым переконфигурированием правил внедрения аспектов.

Осталось упомянуть еще одно возможное применение АОП-инструментов в виде тонкой прослойки между целевым проектом и сторонней библиотекой, реализующей сквозную функциональность. С учетом того, что компания Microsoft выпустила библиотеку Enterprise Library (EL) [10], которая позиционируется как набор готовых функциональных блоков со сквозной функциональностью (Caching Application Block, Validation Application Block, Logging Application Block и т. п.), роль АОП-инструмента можно упростить и свести к вставке вызовов соответствующих функциональных блоков в нужных местах целевой программы.

В данной работе описывается подход к реализации АОП-инструмента с бесшовной интеграцией аспектов для платформы MS .NET, разработанного авторами в рамках проекта Aspect.NET [11] на базе библиотеки Mono.Cecil [12]. Получившийся инструмент предоставляет минимально необходимую функциональность по созданию аспектов, но, в то же время, обладает высокой производительностью результирующей сборки и позволяет повышать метрики качества исходного проекта, выделяя сквозную функциональность в отдельные проекты. Таким образом, система Aspect.NET может быть использована, например, при создании web-приложений [13], АОП-рефакторинге [14] и бесшовной интеграции с библиотекой MS Enterprise Library [15].

2. ОБЗОР АОП-ИНСТРУМЕНТОВ

Все реализации АОП можно разделить по типу внедрения аспектов в целевую сборку:

1. Во время выполнения целевой программы (IOC-контейнеры, Spring.NET [16], Microsoft Unity [5] и т. п.)
2. Во время инициализации и загрузки целевой программы (Rapier- Loom.NET [17]).
3. На этапе компиляции (PostSharp, Aspect.NET, AspectJ и пр.).

В первом случае, АОП-инструмент во время выполнения целевой программы конструирует прокси-класс над целевым [18]. Далее объект этого класса подставляется вместо целевого объекта. Таким образом, перехват вызова целевых методов происходит в одноименных методах прокси-класса, которые выполняют вызовы действий аспектов и делегируют остальную функциональность агрегированной ссылке на целевой объект.

К преимуществам такого динамического внедрения аспектов можно отнести простоту реализации и применения, так как используются только стандартные конструкции языка программирования. Кроме того, становится возможным менять конфигурацию аспектов динамически, прямо во время выполнения программы. Однако необходимость генерировать низкоуровневый код для прокси-класса во время выполнения целевой программы приводит к низкой производительности результирующей сборки.

Применить аналогичный подход, но конструировать прокси-классы во время запуска целевой программы предлагается во втором случае. Это повышает производительность выполнения целевой программы, но теряется возможность динамической конфигурации аспектов.

Наконец, статическое внедрение аспектов в целевую сборку происходит на этапе пост-компиляции (postbuild). Специальная утилита, компоновщик аспектов, инструментирует исходные тексты целевой программы (или ее собранную сборку) и вставляет в нужные места вызовы к библиотеке с аспектами:

Данный подход позволяет создавать результирующие сборки с наибольшим быстродействием, перекладывая затраты на внедрение аспектов на время компиляции. Также данный подход предоставляет аспектам весь контекст точки внедрения, начиная от текущего this-объекта, аргументов целевого метода и заканчивая объектом произошедшего исключения. Минусом данного подхода является невозможность гибкого переключения аспектов во время выполнения программы.

Самым популярным АОП-инструментом является AspectJ [2], первые работы по которому относятся к 1997 году. AspectJ представляет собой расширение языка Java аспектно-ориентированными конструкциями [19]: аспект, точка внедрения (joinpoint), ее спецификация в потоке выполнения программы (pointcut), действие аспекта (advice), расширение (introduction) целевого класса. Иными словами, программа на AspectJ содержит базовую часть (base code) с обычными конструкциями класса и интерфейсов для описания основной функциональности и аспектную (aspect code), которая моделирует сквозную функциональность. В последних версиях AspectJ добавлена возможность описывать аспектные конструкции в виде атрибутов Java (@Annotations).

Аспект — это новая сущность языка программирования, дополняющая сущность класса:

- Обозначается ключевым словом aspect, аналогично объявлению класса в Java через идентификатор class;
- Может быть создан (instantiate), унаследован (sub aspect), содержать члены (state) и методы;

- Содержит правила внедрения, определяющие как он будет скомбинирован с целевым классом;
- Имеет возможность дополнить целевой класс новыми методами, членами и реализацией интерфейса. Эти новые элементы могут быть как видимы всем классам в целевом проекте (public introduction), так и только аспекту (private introduction).

Аспекты комбинируются с целевыми классами в точках внедрения: вызов какого-либо метода или конструктора, обращение к его полю (member), обработчиках исключений и пр. Спецификация таких точек внедрения может быть составлена (designate) из других спецификаций. В действиях аспекта описывается код, который будет вызван перед (before), после (after) или вместо (around) целевого кода в точке внедрения.

Следует отметить, что в AspectJ используются три модели компоновки аспектов с целевым кодом:

- Во время компиляции (compile-time weaving). Специальный компилятор ajc обрабатывает исходные тексты аспектов и целевого проекта, а затем компилирует их в одну сборку.
- Пост-компиляция (post-compile weaving). Вначале целевой и аспектные исходные тексты компилируются штатным javac компилятором, а затем ajc сливает их на уровне байт-кода. Именно такой способ обеспечивает бесшовную интеграцию аспектов.
- Во время загрузки классов в JVM (load-time weaving). Вызовы действий аспектов встраиваются в байт-код целевых классов при их загрузке в JVM.

Подводя итог, можно сказать, что AspectJ обладает всеми АОП-функциями для вынесения сквозной функциональности в аспекты на платформе Java. Большинство АОП-инструментов могут только лишь приблизиться к его возможностям. Однако, далеко не все эти возможности необходимы для повседневного АОП-программирования. Большое число возможностей ведет к повышению сложности для программиста и, до сих пор, AspectJ используется лишь небольшой долей Java-программистов. Таким образом, при реализации нового АОП-инструмента (например, для платформы .NET), стоит сконцентрироваться на его упрощении.

Инструмент PostSharp [3] появился в 2007 году на всплеске интереса к АОП и с тех пор активно развивается. Его активное развитие обусловлено коммерциализацией продукта, что позволяет компании-разработчику SharpCrafters контролировать и управлять всем процессом разработки. Однако это одновременно является препятствием для его применения в академических, государственных и проектах с открытым кодом.

PostSharp имеет тесную интеграцию с Microsoft Visual Studio, в том числе шаблоны для быстрого добавления аспектов и графический интерфейс для навигации по ним и точкам внедрения. Аналогично Aspect.NET внедрение аспектов происходит на этапе пост-компиляции (статически), однако если первый позволяет создавать аспекты в отдельном проекте, то второй устанавливается через менеджер пакетов Nuget прямо в целевой проект, добавляя в него свои зависимости. В этом смысле PostSharp не может полностью обеспечить бесшовность расширения целевого проекта.

В какой-то степени по своим возможностям PostSharp превосходит AspectJ, например, возможностью вызывать действия аспектов во время компиляции проекта и предоставлять все доступную информацию времени построения. Таким образом, принимать решение о внедрении в заданную точку целевого кода, аспект может на этапе компиляции, проанализировав целевой код или настройки. Более того, аспекты могут исследовать архитектуру целевого проекта и выдавать рекомендации по ней (architecture validation), аналогично продукту по статическому анализу FxCop [20].

Аспекты могут применяться к целевым методам, обращениям к полю или свойству, исключениям или событиям. Сами аспекты реализуются в виде сериализуемых классов — наследников от соответствующих атрибутов. Применение аспекта заключается в:

- Пометке атрибутом целевого класса, поля или метода. Этот способ применяется для точечного указания места внедрения.
- Указанию в свойствах сборки маски названия целевого класса или метода. Таким образом аспект применяется к нескольким местам (multicast).
- Определение конкретных целевых классов и методов в XML-файле PostSharp. Так можно определять точки внедрения, не меняя исходных текстов приложения. Требование бесшовной интеграции при этом не выполняется в полной мере, так как PostSharp остается тесно интегрированным в целевой проект.

Как и в AspectJ, к возможностям PostSharp также относится механизм добавления к целевому классу новых полей, методов и реализации интерфейсов. Однако, несмотря на полную поддержку АОП, благодаря наличию готовых аспектов, применение PostSharp не требует специальных навыков. На своем сайте [3] разработчики PostSharp позиционируют его в качестве расширения стандартного компилятора паттернами (в виде аспектов), которые после применения к целевому классу автоматически реализуют следующую сквозную функциональность: уведомление об изменении свойств (паттерн INotifyPropertyChanged), функциональность Undo/Redo значений свойств, проверку входных аргументов на корректные значения (Code Contracts), протоколирование (logging), обеспечение потоко-безопасности (thread safety) и обнаружение взаимных блокировок (deadlocks).

3. ОСНОВЫ ASPECT.NET

Aspect.NET — это легковесный инструмент АОП для платформы .NET, требования и спецификации к которому были сформулированы профессором В.О. Сафоновым в 2004 году [21]. При создании Aspect.NET были поставлены следующие задачи:

- Минимизировать время программиста на изучение системы.
- Хранить код и правила внедрения аспектов отдельно от целевого проекта (для поддержки их бесшовной интеграции).
- Дать возможность аспектам влиять на целевой код.
- Снизить накладные расходы на вызов аспектов.

Как уже было сказано, PostSharp имеет такие исчерпывающие возможности по созданию аспектов, как перехват вызовов методов (MethodInterceptionAspect), доступа к полю (LocationInterceptionAspect), инстанцирование аспектов во время компиляции (CompileTimeInitialize) и передача дополнительной информации при создании аспекта в режиме выполнения (RuntimeInitialize), однако, в рамках простой переадресации вызовов к сторонним службам сквозной функциональности, они приводят к усложнению кода и дополнительным накладным расходам. В тех случаях, когда перед вызовом целевого достаточно вызвать статический метод аспекта, который вызовет стороннюю службу с нужными параметрами, PostSharp будет вынужден сериализовать (при компиляции) и десериализовать (во время выполнения) состояние аспекта из памяти для возможности его применения.

Также стоит упомянуть проблему тестирования целевого кода, к которому на этапе компиляции применяется аспект PostSharp. Хотя на уровне исходного кода бизнес-логика и аспект разделены по разным классам, но содержатся они в одном проекте и такие популярные инструменты модульного тестирования как NUnit [22] применяют тесты

к результирующей сборке, в которой целевой код и аспект уже сцеплены друг с другом. Это делает затруднительным тестирование самой бизнес-логики и приходится выключать PostSharp в свойствах целевого проекта на время прогона тестов. При наличии сложной системы непрерывной интеграции (continous integration) программистам, применяющим PostSharp, приходится искать нетривиальные решения [23]. В свою очередь, бесшовная интеграция аспектов Aspect.NET лишена таких проблем: целевой проект ничего не знает про аспекты, его компиляция и тестирование проходит как обычно, а интеграция с аспектами осуществляется во время компиляции аспектного проекта.

С помощью Aspect.NET можно определять аспекты в отдельных классах, а затем вплетать вызовы их методов в заданные места целевой сборки (joinpoints). Определения аспектов не зависят от конкретного языка, а разрабатывать их можно в любой среде разработки, поддерживающей платформу .NET.

Аспектом может быть любой класс, производный от класса Aspect (предопределенного в библиотеке Aspect.NET). Реализация аспекта осуществляется статическими методами («действиями»), которые затем будут вставлены компоновщиком в заданные точки внедрения (joinpoints) в сборке целевого приложения. Требуемое множество точек внедрения задается в пользовательском атрибуте AspectAction() своего действия аспекта. 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. Также внутри действия аспекта можно использовать свойства базового класса Aspect, предоставляющие доступ к контексту точки внедрения [24]:

- *SourceFileLine* — это строка, представляющая номер строчки в исходном коде файла, где расположен вызов целевого метода.
- *SourceFilePath* — это строка, представляющая путь к целевому файлу исходного кода, в котором расположен вызов целевого метода.
- *TargetObject* — это ссылка типа *System.Object* на объект в целевой программе, к которому применяется целевой метод в точке присоединения. Например, если вызов целевого метода в точке присоединения имеет вид p.M(), то *TargetObject* обозначает ссылку на объект p. Если целевой метод статический, *TargetObject* равно *null*.
- *TargetMemberInfo* — это ссылка типа *System.Reflection.MemberInfo* на объект, представляющий метаданные целевого метода.
- *WithinMethod* — это ссылка типа *System.Reflection.MethodBase* на метаданные, представляющие метод, в котором расположен вызов целевого метода.
- *WithinType* — это ссылка типа *System.Type* на определение класса, в котором расположен вызов целевого метода.
- *RetValue* — это ссылка типа *System.Object* на результат, возвращаемый целевым методом. Для всех действий кроме %after имеет значение *null*.
- *This* — это ссылка типа System.Object на объект в целевой программе, внутри метода которого оказалось вставлено данное действие аспекта. Например, если вызов p.M() содержится внутри метода объекта X, то X является ссылкой This для действия аспекта, применяемого к p.M().

В дополнение к свойствам контекста, компоновщик обеспечивает «захват» аргументов, передавая их от целевого метода в аргументы действия аспекта.

```
int Sum(int a, int b, int c){..} //Целевой метод
...
/* Действие аспекта с первым и третьим захваченными у целевого метода аргументами*/
[AspectAction("%after %call *Sum(int,int,int) && %args(arg[0],arg[2]))]
public static AspectAfterSum(int a, int c) {..}
```

Подробно синтаксис «захвата» аргументов рассмотрен в [24]. Небходимо лишь упомянуть, что данный подход «точечной» передачи нужных аргументов выигрывает по накладным расходам по сравнению со способом, когда весь набор аргументов целевого метода упаковывается в коллекцию объектов, и действие аспекта вынуждено перебором выбирать нужные (например, PostSharp). С другой стороны, явное указание нужных аргументов усиливает связь между аспектом и конкретным целевым методом. Это может быть оправдано в задачах АОП-рефакторинга, но не при написании универсальных аспектов. Поэтому авторы не исключают в будущем реализацию подобной упаковки аргументов в массив объектов.

4. РЕАЛИЗАЦИЯ КОМПОНОВЩИКА АСПЕКТОВ В ASPECT.NET

Aspect.NET вставляет действия на уровне MSIL-инструкций после этапа компиляции целевой сборки (аналогично PostSharp), что влечет повышение производительности целевого приложения по сравнению с IOC-контейнерами. Более того, такая «постобработка» дает возможность выбирать конкретные места применения действий аспектов. В то время, как IOC-контейнер перехватывает все вызовы целевого метода на этапе выполнения программы, а PostSharp предлагает сформулировать фильтрующий метод, вызываемый во время компиляции, то в Aspect.NET применение аспектов происходит в два этапа: вначале в целевой сборке находятся все точки внедрения, удовлетворяющие правилам, затем пользователь в специальном окне имеет возможность отключить ненужные. Отфильтрованный список точек внедрения подается на этап внедрения аспектов, где инструментирование MSIL-кода производится с помощью библиотеки Mono.Cecil [12]. Стоит все же отметить, что при необходимости переконфигурирования аспектов прямо во время выполнения программы, альтернативы IOC-контейнерам нет.

Основной критерий, которому должно удовлетворять проектное решение — это высокая производительность, иначе трудно убедить рядовых разработчиков потратить свое время на освоение новой технологии. Рассмотрим целевой код из работы [25]:

```
Program p = new Program();
p.Method(); // Целевой код
```

И действие аспекта:

```
[AspectAction("%before %call *Method")]
static public void BeforeAction() { // Действие аспекта
   Console.WriteLine(Aspect.SourceFilePath);
   Console.WriteLine(Aspect.This);
}
```

Компоновшик аспектов имеет дело с бинарными сборками и кодом MSIL. Сканируя целевую сборку, компоновщик находит точки внедрения и вставляет туда требуемый код загрузки контекста и вызов действия аспекта:

```
  IL_0001:  newobj  instance void Program::.ctor()
  IL_0006:  stloc.0              // Аналог Program p = new Program()
  IL_0011:  ldstr       ".../Program.cs"
  IL_0016:  call void AspectDotNet.Aspect::InternalSetSourceFilePath(string)
            // Аналог Aspect.InternalSetSourceFilePath(".../Program.cs")
  IL_001b:  ldloc.0
  IL_001c:  call void AspectDotNet.Aspect::InternalSetTargetObject(object)
            // Аналог Aspect.InternalSetTargetObject(p)
```

```
IL_0021:  call void MyAspectClass::BeforeAction()
          //Аналог MyAspectClass.BeforeAction()
IL_0026:  ldloc.0
IL_0035:  callvirt  instance void %Program::Method()
         //Аналог p.Method()
```

Как можно видеть, каждая необходимая переменная для контекста передается в специальные методы класса Aspect, начинающиеся с InternalSet..., которые сохраняют ее в соответствующих полях Aspect. Затем get-свойства их просто возвращают действию аспекта. Также видно, что отладочные данные SourceFileLine и SourceFilePath напрямую записываются в строковые ресурсы. Далее в строке 26 видно, что перед вызовом каждого нестатического метода, компилятор загружает в стек его целевой объект (инструкция ldloc.0) и он будет первым аргументом вызываемого метода. Если вызываемый метод обратиться к своему первому аргументу через инструкцию ldarg.0, то получит ссылку на текущий (this) объект. Поэтому, когда компоновщик аспектов встречает MSIL-инструкцию ldloc.0, она дублируется и передается методу InternalSetTargetObject(), а копия ldarg.0 будет загружена в InternalSetThisObject(). Таким образом, в аспекте будут проинициализированы свойства TargetObject и This.

Отдельно стоит отметить, что компоновщик анализирует каждое действие аспекта и определяет те свойства доступа к контексту, которые реально используются и инициализирует только их. Подобная оптимизация есть только в коммерческой версии PostSharp.

Рассмотрим загрузку свойств TargetMemberInfo и WithinMethod. Можно было бы воспользоваться стандартным способом и получить их через метод рефлексии Type.GetMethod(). В этом случае, на этапе выполнения программы среда выполнения .NET будет просматривать все методы заданного типа, сопоставлять их с условиями и определять нужный. Авторами был предложен другой вариант, когда нужный метод определяется в виде System.RuntimeMethodHandle на этапе внедрения аспектов, загружается в стек инструкцией ldtoken и передается в InternalSetTargetMethod() или InternalSetWithinMethod(). Внутри get-свойства TargetMemberInfo компоновщик аспектов преобразует RuntimeMethodHandle в MethodInfo с помощью вызова MethodInfo.GetMethodFromHandle(RuntimeMethodHandle). Аналогичного механизма на C# нет, поэтому производительность полученной сборки в ряде случаев может быть лучше, чем вставка действий аспектов в исходном коде целевого проекта вручную.

```
IL_0001:  ldtoken method instance void DemoAspect.Program::Method1()
IL_0006:  call void Aspect::InternalSetWithinMethodHandle(valuetype
[mscorlib]System.RuntimeMethodHandle)
IL_000b:  ldtoken  method instance void DemoAspect.Program::Method2()
IL_0010:  call void Aspect::InternalSetTargetMethodHandle(valuetype
[mscorlib]System.RuntimeMethodHandle)
```

Реализация свойства WithinType сложности не представляет — это просто WithinMethod.DeclaringType.

Последнее свойство — RetValue. Результатом метода в .NET может быть либо ссылочный тип (object), либо простой тип, например, int. Для простоты реализации было принято решение, что свойство RetValue имеет всегда тип object. Тогда перед его инициализацией переменной простого типа будет необходимо произвести упаковку (boxing) простого типа в object. После вызова целевого метода переменная с этим результатом будет находится на вершине стека, откуда ее можно взять, провести при необходимости

упаковку (boxing), передать в метод InternalSetRetValue(object), распаковать обратно, положить на вершину стека и продолжить выполнение. Для результата ссылочного типа все аналогично, но без упаковки-распаковки.

```
IL_0008:  callvirt   instance int32 Program::Method()
         // Аналог вызова int i = p.Method();
IL_000d:  box [mscorlib]System.Int32
IL_0012:  call  object Aspect::InternalSetRetValue(object)
IL_0017:  unbox.any  [mscorlib]System.Int32
IL_001c:  call void MyAspectClass::AfterAction()
```

Далее рассмотрим реализацию захвата аргументов на приведенном выше примере целевого метода Sum. Допустим, что необходимо сделать вызов Sum(100,200,300) в целевом коде и применить после него действие AspectAfterSum, захватив первый и третий аргументы. Компоновщик сгенерирует следующий код:

```
IL_0008:  ldc.i4.s   100
IL_000a:  ldc.i4     0xc8
IL_000f:  ldc.i4     0x12c
IL_0014:  callvirt   instance void Program::Sum(int32,int32,int32)
          //Аналог вызова Program.Sum(100, 200, 300)
IL_0019:  ldc.i4.s   100
IL_001b:  ldc.i4     0x12c
IL_0020:  call  void MyAspectClass::AfterSumAction(int32, int32)
          // Аналог вызова MyAspectClass.AfterSumAction (100, 300)
```

Для передачи списка аргументов в целевой метод, компилятор C# загружает их поочередно в стек (8–14 строки). Теперь становится очевидно, что компоновщику аспектов будет достаточно продублировать соответствующие ldc инструкции и это автоматически приведет к подстановке нужных аргументов целевого метода в параметры действия аспекта.

Проиллюстрируем работу аспекта на примере работы с простым классом банковского счета, который поддерживает операцию снятия средств withdraw(float):

```
public class BankAccount {
      float account;
      public float withdraw(float amount){/*...*/}
}
```

Если вынести проверку прав на совершение операции в аспект, то он будет выглядеть таким образом:

```
public class BankManagement: Aspect {
      [AspectAction("%instead %call *BankAccount.withdraw(float) && args(..)")]
      public static float WithdrawWrapper(float amount) {
              BankAccount acc = (BankAccount) TargetObject;
              if (isLicenceValid(base.TargetMemberInfo.Name))
                 return acc.withdraw(amount);
              Console.WriteLine("Withdraw operation is not allowed");
              return acc.Balance;
      }
      public static float isLicenceValid(string operation) {/*...*/}
 }
```

Данное действие аспекта определяет «заглушку», которая будет вставлена вместо целевого метода withdraw. Внутри действия происходит проверка прав на выполнение операции withdraw и вызывается исходная операция с тем же набором параметров, если авторизация прошла успешно. Доступ к целевому объекту и имени операции осуществляется через свойства TargetObject и TargetMemberInfo контекста точки внедрения.

5. РЕАЛИЗАЦИЯ ЗАМЕЩАЮЩЕГО НАСЛЕДНИКА В ASPECT.NET

Кроме возможности вставлять действия аспектов перед, после или вместо вызова целевого метода, Aspect.NET предлагает концепцию «замещающего наследника». Современное программирование стремится к использованию различных каркасов (frameworks), которые предлагают общую архитектуру для ряда задач, а программист лишь уточняет поведение программы в методах обратного вызова (callbacks). В качестве примера можно привести платформу для создания веб-приложений MS ASP.NET. Предположим, у нас есть веб-страница Default.aspx, на которой расположена кнопка LogButton. При ее нажатии необходимо записать сообщение в функциональный блок EL 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");
        }
}
```

Как уже упоминалось, протоколирование относится к сквозной функциональности, и его имеет смысл выносить в аспект. В этом случае метод LogButton_Click вызывается средой ASP.NET и у компоновщика нет доступа к вызывающему коду, чтобы обернуть его вызовом аспекта. В этом случае PostSharp предлагает применять правило внедрения OnMethodBoundaryAspect, при котором действия аспекта вставляются внутри тела целевого метода. Однако, если потребуется получить доступ к защищенным полям или методам целевого класса, придется использовать рефлексию .NET. Более оптимальным решением здесь было бы создать наследника класса Default с переопределенным целевым методом LogButton_Click:

```
//Проект с замещающим аспектным наследником
[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) {}
}
```

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.
4. Объединить с помощью инструмента ILRepack [26] (из проекта Mono.Cecil) аспектную сборку и целевую.
5. Присвоить какое-нибудь служебное имя базовому целевому классу, а его первоначальное имя — замещающему наследнику из аспекта.

Подобное создание декораторов (но во время выполнения программы) лежит в основе IOC-контейнеров [23]. Преимуществами такого подхода 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 теряется возможность декларативно вводить интерфейсы в целевые классы, однако появляется возможность использовать их члены в аспекте.

6. РАСШИРЕНИЕ MS VISUAL STUDIO ДЛЯ ПОДДЕРЖКИ ASPECT.NET

При использовании Aspect.NET, решение (solution) в MS Visual Studio (VS) представляет из себя набор проектов, один из которых — целевой проект, другие — проекты аспектов. С помощью средств программирования среды MS Visual Studio SDK (VS SDK) было создано расширение, которое добавляет в панель навигации меню, которое, в свою очередь, позволяет выполнять все действия по внедрению аспектов (см. рис. 1).

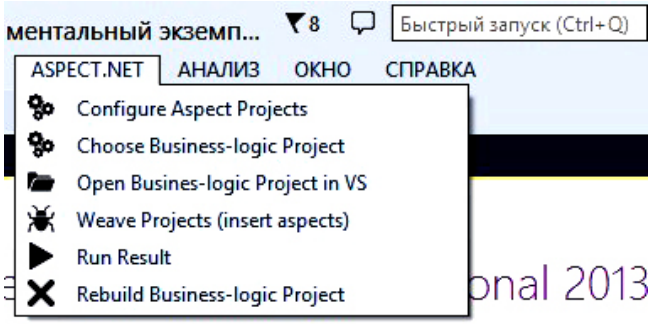

Рис. 1

Таким образом, программист избавляется от обязанности самостоятельно формировать события «пост-компиляции» и прописывать нужные пути. При проектировании аспектно-ориентированного приложения в рамках работы с системой Aspect.NET, необходимо создавать проекты аспектов. В любом таком проекте присутствуют:

– ссылка на библиотеку AspectDotNet.dll;
– ссылка на проект бизнес-логики (если в коде аспектов имеются обращения к классам, определенным в проекте бизнес-логики);
– статические методы действий с правилами внедрения.

В связи с этим был реализован шаблон проекта-аспекта для MS VS, в котором уже имеется ссылка на библиотеку AspectDotNet.dll и написаны примеры кода создания аспектов с тремя действиями (%before, %after, %instead) и правилами внедрения. Разработчик, основываясь на имеющемся коде, может писать собственные аспекты или использовать уже созданные (см. рис. 2).

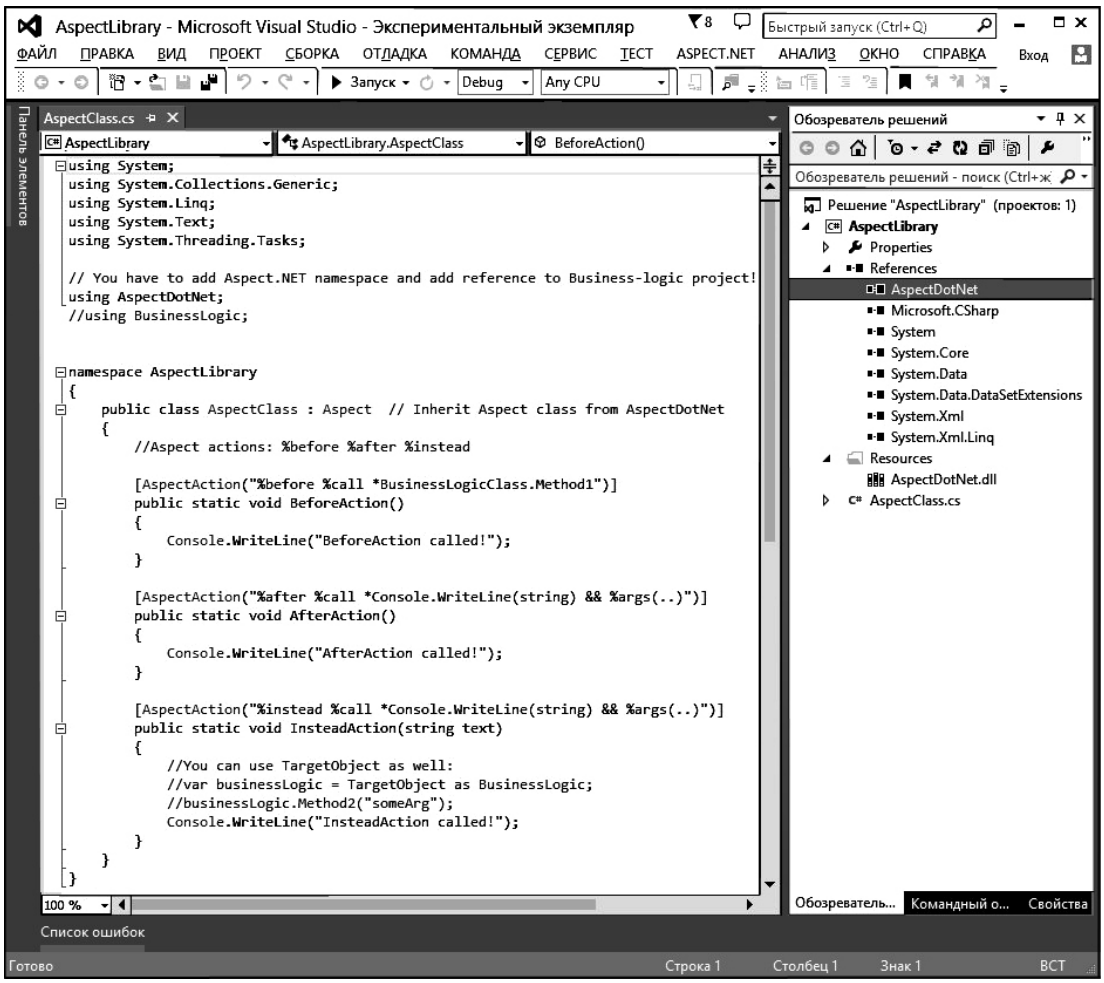

Рис. 2

7. ЗАКЛЮЧЕНИЕ

АОП позволяет разделять бизнес-логику и сквозную функциональность. Ее выделение в аспекты повышает связность целевых классов и улучшает метрики качества кода. АОП-системы подобные PostSharp имеют гибкие возможности для создания своих аспектов, но не устраняют целиком зависимость от них целевого проекта. После этого заменить АОП-инструмент (или отказаться от него полностью) становится затруднительно.

В свою очередь, уже существуют сторонние библиотеки и службы, которые берут на себя реализацию сквозной функциональности (например, MS EL). Для их бесшовного подключения к целевому проекту может применяться разработанная авторами система Aspect.NET для MS VS на основе библиотеки Mono.Cecil. Программирование аспектов в Aspect.NET осуществляется в привычном программном и языковом окружении. Расширение поведения обеспечивается за счет вставки необходимых действий перед, после или вместо целевого метода, а также возможности создать наследника целевого класса и подменить им исходный. Связывание проекта аспекта и бизнес-логики производится программистом в специальном расширении MS VS, доступном для установки в галерее расширений MS VS.

Компоновка аспектов с бизнес-логикой производится на этапе компиляции аспектного проекта и с минимальными накладными расходами. Проведенные в работе [27] измерения производительности, внедренных с помощью Aspect.NET аспектов, показали эффективность по времени сравнимую или даже меньшую, чем вызов той же функциональности в коде напрямую. Однако у предложенного подхода есть и недостатки — аспект имеет сильную связанность с целевым классом, что снижает накладные расходы, но может препятствовать созданию обобщенных, универсальных аспектов.

Список литературы

1. *Мартин Р., Мартин М.* Принципы, паттерны и методики гибкой разработки на языке C#. СПб.: Символ, 2011.
2. *Miles R.* AspectJ Cookbook. Cambridge, USA: O'Reilly, 2004.
3. Сайт проекта PostSharp / http://www.sharpcrafters.com/ (дата обращения: 01.04.2016).
4. *Фаулер М.* Рефакторинг. Улучшение существующего кода. СПб: Символ, 2004.
5. *Эспозито Д.* Аспектно-ориентированное программирование, перехват и Unity 2.0 // MSDN Magazine, 12.2010 / http://msdn.microsoft.com/ru-ru/magazine/gg490353.aspx (дата обращения: 01.04.2016).
6. *Suvée D. et al.* Evaluating FuseJ as a web service composition language // Web Services, 2005. ECOWS 2005. Third IEEE European Conference on. IEEE, 2005.
7. Сайт проекта SheepAspect / http://sheepaspect.codeplex.com (дата обращения: 01.04.2016]).
8. Сайт проекта AOP.NET / https://sourceforge.net/projects/aopnet/ (дата обращения: 01.04.2016).
9. *Hannemann J.* Aspect-oriented refactoring: Classification and challenges // Workshop on Linking Aspect Technology and Evolution (LATE'06). 5th International Conference on Aspect-Oriented Software Development (AOSD'06). 2006.
10. Сайт проекта MS Enterpise Library / http://msdn.microsoft.com/en-us/library/ff648951.aspx (дата обращения: 01.04.2016).
11. Сайт проекта Aspect.NET / http://aspectdotnet.org/ (дата обращения: 01.04.2016).
12. Сайт проекта Mono.Cecil / http://www.mono-project.com/docs/tools+libraries/libraries/Mono.Cecil/ (дата обращения: 01.04.2016).
13. *Нгуен В.Д., Сафонов В.О.* Применение аспектно-ориентированного подхода и системы ASPECT .NET к разработке web-приложений // Компьютерные инструменты в образовании, 2010. №. 5. С. 3–11.
14. *Григорьева А.В.* Аспектно-ориентированный рефакторинг облачных приложений MS Azure с помощью системы Aspect.NET // Компьютерные инструменты в образовании, 2012. № 1. С. 21–30.
15. Григорьев Д.А., Григорьева А.В., Сафонов В.О. Бесшовная интеграция аспектов в облачные приложения на примере библиотеки Enterprise Library Integration Pack for Windows Azure и Aspect.NET // Компьютерные инструменты в образовании, 2012. № 4. С. 3–15.
16. Spring.NET Reference Documentation, 2011 / http://www.springframework.net/doc-latest/reference/pdf/spring-net-reference.pdf (дата обращения: 01.04.2016).

17. *Rasche A., Schult W., Polze A.* Self-adaptive multithreaded applications: a case for dynamic aspect weaving // Proceedings of the 4th workshop on Reflective and adaptive middleware systems. ACM, 2005. С. 10.
18. *Wen W., Zhang S.* Research And Implementation Of AOP Technology In. NET Framework // Enterprise Systems Conference (ES), 2014. IEEE, 2014. С. 97–101.
19. *Iwamoto M., Zhao J.* Refactoring aspect-oriented programs // 4th AOSD Modeling With UML Workshop, UML. 2003.
20. Сайт проекта FxCop 10.0 / https://www.microsoft.com/en-us/download/details.aspx?id=6544 (дата обращения: 01.04.2016).
21. *V.O. Safonov.* Aspect.NET: concepts and architecture. .NET Developer's Journal, 2004, № 10.
22. *Хант Э., Томас Д.* Pragmatic Unit Testing in C# with NUnit // The Pragmatic Bookshelf, Raleigh, 2004.
23. *Groves M.* AOP in .NET. Practical Aspect-Oriented Programming // Manning Publications, 2013.
24. *Григорьев Д.А.* Реализация и практическое применение аспектно-ориентированной среды программирования для Microsoft .NET // СПб. Научно-технические ведомости, СПбГПУ. 2009. № 3. С. 225–232.
25. *Григорьев Д.А., Григорьева А.В., Сафонов В.О.* Реализация механизма доступа к динамическому контексту в точках применения аспектов для системы Aspect.NET // СПб.: СПИСОК-2014. Материалы Всероссийской научной конференции по проблемам информатики. 2014 г. С. 142–147.
26. Страница проекта ILRepack / https://github.com/gluck/il-repack (дата обращения 01.04.2016).
27. *Safonov V.O.* Using aspect-oriented programming for trustworthy software development. Wiley Interscience. John Wiley & Sons, 2008.

Поступила в редакцию 09.11.16, окончательный вариант — 13.12.16.

Computer tools in education, 2016
№ 6: 5–19
http://ipo.spb.ru/journal

IMPLEMENTATION OF THE MS VISUAL STUDIO ADD-IN FOR SEAMLESS ASPECT-ORIENTED PROGRAMMING

Grigoriev, D.A.[1], Grigorieva A.V.[1], Zotov, M.A.[1], Makarov, S.A.[1]

[1]SPbSU, Saint-Petersburg, Russia

Abstract

Aspect-oriented approach is helpful to simplify the business logic of an application, due to explicit separation of its cross — cutting concerns. The goal of the Aspect.NET project described in this paper is to create an aspect — oriented programming tool for Microsoft .NET which would be integrated to the latest Microsoft software development environment — Visual Studio .NET 2013. Cross — cutting concerns are incapsulated in static methods of aspect class. Applying of aspects into target assembly is made by console tool (named "weaver") on the post — build stage. To handle .NET assemblies, Aspect.NET uses Mono.Cecil library. This approach allows to weave new functionality during the post-build stage without any modifications of target project.

Keywords: *aspect-oriented programming, aspects, Visual Studio add-in, statical weaving.*

Citation: Grigoriev, D., Grigorieva A., Zotov, M. & Makarov, S., 2016. “Rasshirenie MS VISUAL STUDIO dlya besshovnogo aspektno-orientirovannogo programmirovaniya” [“Implementation of the MS VISUAL STUDIO add-in for Seamless Aspect-oriented Programming”], *Computer tools in education*, no. 6, pp. 5–19.

Received 09.11.16, the final version — 13.12.16.

Dmitry A. Grigoriev, department of computer science, Mathematics & Mechanics Faculty, St. Petersburg State University, associate professor, Universitetskaya emb., 7-9, St. Petersburg, Russia, 199034, gridmer@mail.ru
Anastasia V. Grigorieva, department of computer science, Mathematics & Mechanics Faculty, St. Petersburg State University, post-graduate student, nastya001@mail.ru
Mikhail A. Zotov, department of computer science, Mathematics & Mechanics Faculty, St. Petersburg State University, student, zotov1994@mail.ru
Sergey A. Makarov, department of computer science, Mathematics & Mechanics Faculty, St. Petersburg State University, student, makar93@mail.ru

Григорьев Дмитрий Алексеевич,
кандидат физико-математических наук,
доцент кафедры информатики
математико-механического факультета
СПбГУ; 199034, Россия, Санкт-Петербург,
Университетская наб., д. 7–9,
gridmer@mail.ru

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

Зотов Михаил Анатольевич,
студент кафедры информатики
математико-механического факультета
СПбГУ,
zotov1994@mail.ru

Макаров Сергей Алексеевич,
студент кафедры информатики
математико-механического факультета
СПбГУ,
makar93@mail.ru