

Аспектно-ориентированное программирование. Концепции и их реализация.

Аспектно-ориентированное программирование (АОР) – это новая технология, позволяющая программистам компоновать программы из взаимно пересекающихся блоков (crosscutting concerns). АОР вводит в действие аспекты (aspects). Они инкапсулируют особенности поведения взаимно влияющих друг на друга классов в составе повторно используемых модулей. В последней версии AspectJ от Xerox PARC Java-разработчики могут пользоваться всеми преимуществами модульной компоновки, предлагаемыми АОР. В этой статье читатель познакомится с AspectJ и интересными возможностями его использования в проектировании.

Аспектно-ориентированное программирование «выросло» из осознания того, что в типовых программах часто представлено поведение, которое не вмещается естественно в один или даже в несколько тесно связанных программных модулей. Пионеры аспектного подхода определили такое поведение термином *crosscutting*, поскольку при этом пересекаются друг с другом ответственности разработчиков программных модулей. В объектно-ориентированном программировании, например, естественной единицей модульности является класс, а отношение *crosscutting* охватывает несколько классов. Обычно отношение *crosscutting* встречается при организации журналов работы приложения (logging), контекстно-зависимой обработке ошибок, оптимизации выполнения программ, а также в шаблонах проектирования.

Если вы когда-нибудь работали над кодом с *crosscutting*-отношением, вам известны проблемы, связанные с ограничением модульности. Поскольку *crosscutting*-поведение реализуется разрозненно, разработчики находят такое поведение затруднительным в осмысливании, реализации и изменении. Код для ведения журналов, например, переплетается с кодом, отвечающим в основном за что-либо другое. В зависимости от масштаба и сложности *crosscutting*-отношения степень запутанности может быть более или менее значительной. Изменение политики регистрации активности приложения может потребовать многих сотен правок – тяжелейшая, если, вообще, выполнимая задача. С другой стороны известен пример следующего рода. Для оптимизации эффективности некую программу переписали – при этом из 768-строковой она превратилась в 35213 строк. С применением аспектно-ориентированной технологии этот код уменьшился до размера в 1039 строк, почти полностью сохранив при этом эффективность.

АОР дополняет объектно-ориентированное программирование, обогащая его другим типом модульности, который позволяет локализовать код реализации *crosscutting* логики в одном модуле. Такие модули обозначаются термином *аспекты*, от аспектно-ориентированного программирования. За счет отделения аспектно-ориентированного кода работа с *crosscutting*-отношениями упрощается. Аспекты в системе могут изменяться, вставляться, удаляться на этапе компиляции и, более того, повторно использоваться.

1. И сразу пример

Чтобы лучше почувствовать особенности аспектно-ориентированного программирования, рассмотрим AspectJ - аспектно-ориентированное расширение Java, предложенное Xerox PARC. В качестве примера рассмотрим фрагмент кода, предназначенный для регистрации событий в log-файле, реализованный на AspectJ. Этот пример взят из системы с открытым кодом Sactus, упрощающей тестирование Java-компонентов на стороне сервера. Каркас Sactus разработан для поддержки процесса отладки с помощью трассировки вызовов всех методов. Версия 1.2 Sactus была написана без AspectJ. Поэтому большинство методов выглядели, как показано ниже.

```
public void doGet(JspImplicitObjects theObjects) throws ServletException
{
    logger.entry("doGet(...)");

    JspTestController controller = new JspTestController();
    controller.handleRequest(theObjects);

    logger.exit("doGet");
}
```

Каждому разработчику, в рамках соглашений по созданию проектов, предлагалось включать log-вызовы в начало и в конец каждого метода. Кроме того, рекомендовалось заносить в log значения параметров каждого метода. Следование этим соглашениям требовало существенных усилий со стороны разработчика. Так в версии Cactus 1.2 содержится около 80 различных регистрационных вызовов, охватывающих 15 классов. В версии 1.3 эти 80 вызовов были заменены одним аспектом, который автоматически регистрирует и параметры, и возвращаемые значения наряду с входами и выходами метода. Упрощенная версия этого аспекта (опущена регистрация параметра и возвращаемого значения) представлена ниже:

```
Public aspect AutoLog{

    Pointcut publicMethods() : execution(public * org.apache.cactus..*(..));

    Pointcut logObjectCalls() :
        execution(* Logger.*(..));

    pointcut loggableCalls() : publicMethods() && ! logObjectCalls();

    before() : loggableCalls(){
        Logger.entry(thisJoinPoint.getSignature().toString());
    }

    after() : loggableCalls(){
        Logger.exit(thisJoinPoint.getSignature().toString());
    }
}
```

Проанализируем этот пример и посмотрим, какие действия осуществляет аспект. Первое, на что нужно обратить внимание – это объявление аспекта. Оно подобно объявлению класса и, так же как класс, определяет тип Java. Кроме того, аспект содержит конструкции *pointcut* и *advice*.

1.1 Конструкция *pointcut* и точки соединения (joint points)

Прежде всего, рассмотрим, что представляет собой *join point* (точка соединения). Точки соединения – это однозначно определенные точки при выполнении программы. Так под точками соединения в AspectJ подразумеваются: вызовы методов, точки обращения к членам класса и исполнение блоков обработчиков исключений и т.д. Точки соединения могут, в свою очередь, содержать другие точки соединения. Например, результат вызова метода может передаваться каким-то другим методам. А *pointcut* является языковой конструкцией, которая отбирает множество точек соединения на основании определенного критерия. В приведенном выше примере первый *pointcut* под именем `publicMethods` выбирает исполнения всех `public` методов в пакете `org.apache.cactus`. Подобно `int`, который является базовым типом Java, `Execution` является базовым *pointcut*. Он выбирает исполнения методов, соответствующих сигнатуре, заданной в скобках. Для сигнатур допустимо включение символов шаблонов: в приведенном примере оба *pointcut*-а содержат несколько таких символов. Второй *pointcut* с именем `logObjectCalls` выбирает все исполнения методов в классе `Logger`. Третий *pointcut* `loggableCalls`, объединяет два предыдущих, используя `&& !`, что означает выбор всех `public` методов из `org.apache.cactus` за исключением таких в классе `Logger`. (Регистрация `log` методов привела бы в результате к бесконечной рекурсии).

1.2 Конструкция *advice*

Теперь, после того, как в аспекте определены точки, нужно использовать конструкцию *advice*, чтобы выполнить текущую регистрацию. *Advice* – это фрагмент кода, выполняющийся до, после или в составе точки соединения. *Advice* определяется для *pointcut*, что представляет собой нечто наподобие указания “выполнить этот код после каждого вызова метода, который надо зарегистрировать”. В нашем примере первый *advice* объявлен следующим образом:

```
Before() : loggableCalls(){
    Logger.entry(thisJoinPoint.getSignature().toString());
}
```

Этот advice использует класс `Logger`, методы `entry` и `exit` которого выглядят следующим образом:

```
public static void entry(String message){
    System.out.println("entering method " + message);
}
```

В приведенном примере классу `logger` передается `String`, образованная от `thisJoinPoint`, специального объекта, разрешающего доступ к контексту времени выполнения, в котором выполняется точка соединения. В данном, используемом `AspectJ` аспекте, advice применяет этот объект для извлечения параметров метода, передающихся в каждый зарегистрированный вызов метода. «След» вызова метода (с применением аспекта) в `log`-файле выглядит следующим образом:

```
Entering method: void test.Logging.main(String[])
Entering method: void test.Logging.foo()
exiting method: void test.Logging.foo()
exiting method: void test.Logging.main(String[])
```

1.3 Advice типа *around*

В примере `AspectJ` определены advice типа `before()` и `after()`. Advice третьего типа `around()` дает возможность разработчику аспектов управлять передачей управления на точку соединения. При этом используется специальный синтаксис `proceed()`. Следующий advice вызывает (или не вызывает) исполнение метода `say` из класса `Hello` в зависимости от генерируемого случайного числа (`random`):

```
Void around(): call(public void Hello.say()){
    if(Math.random() > .5){
        proceed();//go ahead with the method call
    }
    else{
        System.out.println("Fate is not on your side.");
    }
}
```

2. Программирование в AspectJ

Теперь, когда мы представляем себе, что такое код аспекта, зададим вопрос “Как заставить приведенный выше код работать?”.

Чтобы аспекты могли оказывать воздействие на обычный, основанный на классах код, эти аспекты должны быть “вплетены” в модифицируемый ими код. Чтобы осуществить это в `AspectJ`, надо откомпилировать код класса и аспекта `ajc` компилятором. `ajc` может функционировать как компилятор или как прекомпилятор, генерируя или действующий код класса, или `.java` файлы, которые можно затем компилировать и запускать в любом стандартном окружении `Java` (со ссылкой на небольшой `run-time JAR`).

Для компиляции в `AspectJ` необходимо явно задать исходные файлы (и для аспектов, и для классов), подлежащие включению в данную компиляцию – `ajc` не использует `classpath`, в отличие от `javac`. Это имеет определенный смысл, поскольку каждый класс стандартного приложения `Java` является, в некотором смысле, изолированным компонентом. Для корректной работы классу требуется всего лишь присутствие других классов, на которые он ссылается. Аспекты же представляют совокупное поведение, перекрывающее множество классов. Поэтому АОР-приложение должно компилироваться, как модуль, а не по одному классу за один раз.

Задавая файлы для компиляции, можно также включать и отключать различные аспекты системы на этапе компиляции. Например, включая или исключая описанный ранее аспект для регистрации, разработчик приложения может добавлять или удалять трассировку метода системы Cactus.

Существенное ограничение текущей версии AspectJ состоит в том, что ее компилятор может вводить аспекты только в код, для которого есть исходный текст. Иными словами, невозможно использовать ајс для включения advice в уже откомпилированные классы. Разработчики AspectJ представляют это ограничение как временное, и на Web-сайте AspectJ можно найти подтверждение того, что в будущей версия (официально — версия 2.0) будут допустимы модификации на уровне байт-кода.

3. Обзор возможностей AspectJ

3.1. Introduction как средство воздействия на структуру класса

Pointcuts и advice позволяют влиять на динамику выполнения программы, *introduction* предоставляет аспектам возможность модифицировать статическую структуру программы. Используя *introduction*, аспекты могут добавлять новые методы и переменные в классы, объявлять класс как реализацию интерфейса, устанавливать или отменять проверку исключений.

3.1.1 Пример ситуации, когда необходим механизм *introduction*

Предположим, что у нас есть объект, представляющий кэш непрерывно возобновляющихся данных. Для оценки “свежести” данных, возможно, мы решили добавить к объекту поле `timestamp`, чтобы можно было легко определять, синхронизирован ли объект с той информацией, которая хранится во внешней памяти. Однако, поскольку объект представляет бизнес данные, очевидно, имеет смысл отделить эту механическую деталь от объекта. В AspectJ можно использовать синтаксис, представленный ниже, для добавления даты и времени к существующему классу:

Листинг 4. Добавление переменных и методов к существующему классу

```
public aspect Timestamp {  
  
    private long ValueObject.timestamp;  
  
    public long ValueObject.getTimestamp(){  
        return timestamp;  
    }  
  
    public void ValueObject.timestamp(){  
        // "this" refers to ValueObject class not Timestamp aspect  
        this.timestamp = System.currentTimeMillis();  
    }  
}
```

Мы объявляем внедряемые методы и переменные члены класса почти так же как для обычных членов класса, за исключением того, что нужно уточнять, для какого класса мы их определяем (отсюда `ValueObject.timestamp`).

3.1.2 Наследование в стиле *mix-in*

AspectJ позволяет добавлять члены в интерфейсы (аналогично добавлению в классы), что относится к наследованию в стиле *mix-in a la C++*.

Если мы хотим ввести в общее употребление аспект, рассмотренный в предыдущем разделе, таким образом, чтобы стало возможным многократное использование кода `timestamp` для множества объектов, следует определить интерфейс с именем `TimestampedObject`, и далее использовать *introduction* для добавления тех же самых членов и переменных в интерфейс вместо конкретного класса:

```
public interface TimestampedObject {  
    long getTimestamp();  
}
```

```

    void timestamp();
}

public aspect Timestamp {

    private long TimestampedObject.timestamp;

    public long TimestampedObject.getTimestamp(){
        return timestamp;
    }

    public void TimestampedObject.timestamp(){
        this.timestamp = System.currentTimeMillis();
    }
}

```

Теперь можно использовать синтаксис `declare parents`, чтобы заставить `ValueObject` реализовать новый интерфейс. Конструкция `declare parents`, как и другие выражения для типов в AspectJ, могут быть применены к нескольким типам одновременно.

```
declare parents: ValueObject || BigValueObject implements TimestampedObject;
```

После того как определены операции, поддерживаемые интерфейсом `TimestampedObject`, можно использовать `pointcut`-ы и `advise` для автоматического обновления меток времени (`timestamp`) при возникновении подходящих условий. Так, дополнение к `timestamp`, показанное ниже, регистрирует время каждого вызова setter метода:

```

pointcut objectChanged(TimestampedObject object) :
    execution(public void TimestampedObject+.set*(..) &&
        this(object);
/*TimestampedObject+ means any subclass of TimestampedObject*/

after(TimestampedObject object) : objectChanged(object){
    object.timestamp();
}

```

Заметьте, что `pointcut` определяет аргументы, используемые в `advise` типа `after()` – в данном случае, это `TimestampedObject`, имеющий метод установки `set`. `Pointcut this()` определяет все точки соединения, где исполняемый в настоящее время объект имеет тип, заданный в скобках. Несколько других типов значений могут быть связаны с аргументами `advise`, такие как аргументы метода, исключения при исполнении метода и результат вызова метода.

3.2 Настраиваемые ошибки компиляции

Возможность настройки ошибок компиляции можно считать одной из самых дерзких особенностей AspectJ. Предположим, мы хотим изолировать подсистему так, чтобы клиентскому коду пришлось бы использовать некую промежуточную функциональную обвязку вместо прямого обращения к рабочим объектам (такая ситуация имеет место в шаблоне проектирования Facade). Используя синтаксис `declare error` или `declare warning`, можно настроить реакцию компилятора ас на появление точки соединения в коде:

```

public aspect FacadeEnforcement {

    pointcut notThruFacade() : within(Client) && call(public * Worker.*(..));

    declare error : notThruFacade():
        "Clients may not use Worker objects directly.";
}

```

Pointcut `within` подобен `this()` за исключением того, что `ajc` обнаруживает его только на этапе компиляции (большинство pointcut могут срабатывать на основе информации времени выполнения).

3.3 Обработка ошибок

Есть множество обрабатываемых исключений в языке Java. Зачастую создаются такие методы, которые наверняка не должны вызывать исключений, и, возможно, они не будут происходить ни у кого из потенциальных пользователей метода. Мы не призываем игнорировать возможные исключения, но тяжело отслеживать присутствие исключений во всех вызовах метода. Есть разные искусные способы использования блоков `try/catch`, чтобы все-таки решить эту задачу, но самый элегантный – это `declare soft` в AspectJ. Рассмотрим пример работы с базой данных:

```
public class SqlAccess {  
  
    private Connection conn;  
    private Statement stmt;  
  
    public void doUpdate(){  
        conn = DriverManager.getConnection("url for testing purposes");  
        stmt = conn.createStatement();  
        stmt.execute("UPDATE ACCOUNTS SET BALANCE = 0");  
    }  
  
    public static void main(String[] args) throws Exception{  
        new SqlAccess().doUpdate();  
    }  
}
```

Если не использовать AspectJ или объявлять исключение в каждой сигнатуре метода, то пришлось бы встраивать блоки `try/catch` для обработки `SQLException`, генерируемого почти каждым методом JDBC API. Язык AspectJ позволяет использовать следующий внутренний аспект, чтобы автоматически транслировать все `SQLException` в `org.aspectj.lang.SoftException`.

```
private static aspect exceptionHandling{  
    declare soft : SQLException : within(SqlAccess);  
  
    pointcut methodCall(SqlAccess accessor) : this(accessor)  
        && call(* * SqlAccess.*(..));  
  
    after(SqlAccess accessor) : methodCall (accessor){  
        System.out.println("Closing connections.");  
        if(accessor.stmt != null){  
            accessor.stmt.close();  
        }  
        if(accessor.conn != null){  
            accessor.conn.close();  
        }  
    }  
}
```

Pointcut и `advise` закрывают соединение и оператор после каждого метода из класса `SqlAccess`, в любом случае, приводит ли он к исключению или завершается нормально. Возможно, это расточительно – использовать обрабатывающий ошибки аспект для одного метода, но если есть намерения добавить некоторые другие методы, использующие соединение и оператор, то такая методика обработки ошибок применялась бы и к ним. Такое автоматическое применение аспектов к новому коду является одним из ключевых проявлений устойчивости AOP: авторам нового кода не нужно знать о взаимно пересекающемся поведении для того, чтобы принимать в нем участие.

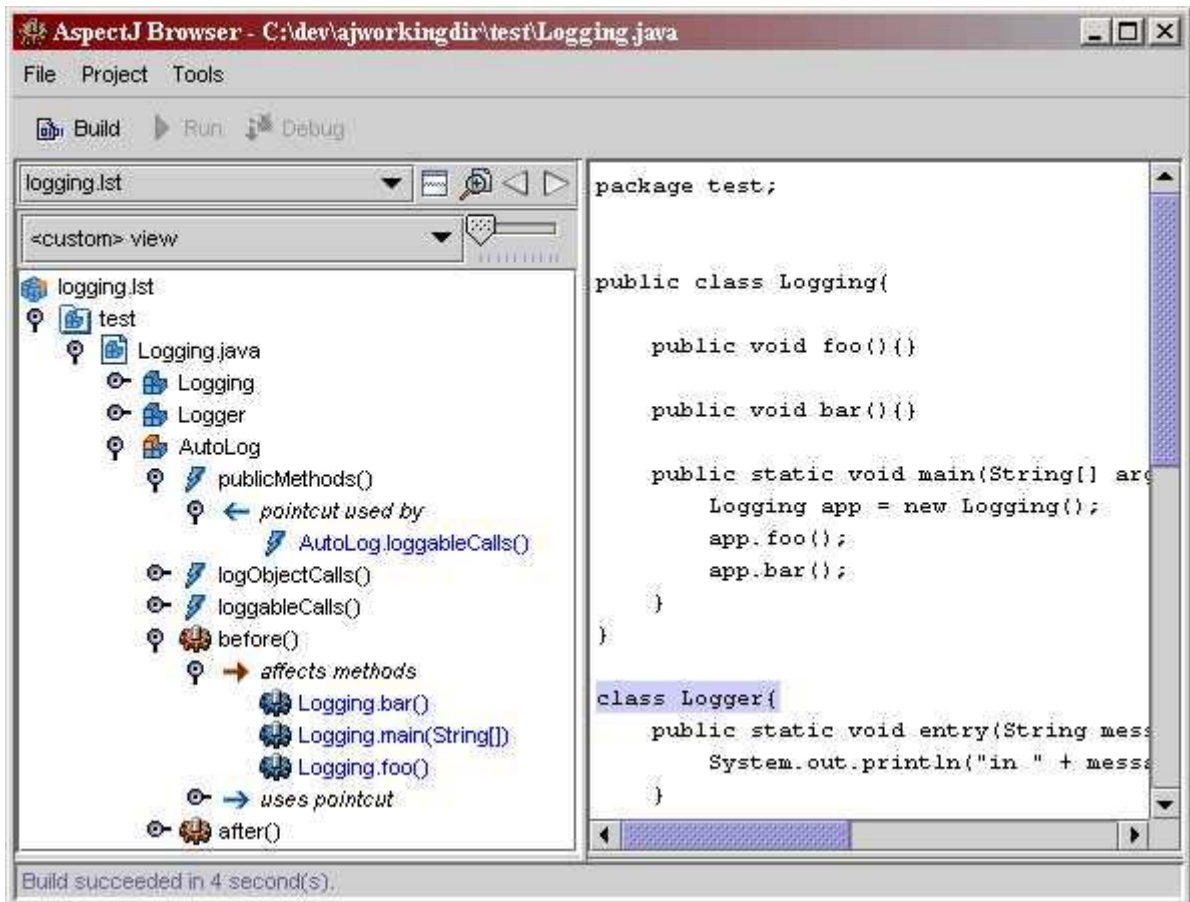
4. Инструментальная поддержка

Хехо подготовил AspectJ к использованию под Mozilla Public License, что является хорошей новостью для энтузиастов открытого кода. Это обрадует и тех, кто собирается остановить свой выбор на AspectJ в ближайшем будущем, поскольку продукт ничего не стоит, и при этом для пользователя сохраняется гарантированная возможность проверки исходного кода. Использование открытого кода означает также, что исходный код AspectJ был предметом серьезного общественного обсуждения, прежде чем появиться на рынке.

В AspectJ релиз включены несколько инструментальных средств. Это свидетельствует о твердых обязательствах авторов AspectJ в части создания дружественного по отношению к разработчикам средства. Инструментальная поддержка чрезвычайно важна для аспектно-ориентированных систем, поскольку программные модули могут зависеть от других модулей, о наличии которых они не знают.

Одним из наиболее важных инструментов в релизе AspectJ является графической структурный браузер, который позволяет быстро увидеть, как аспекты взаимодействуют с другими компонентами системы. Этот структурный браузер доступен и как plug-in для популярных IDE, и как самостоятельное средство. На рисунке 1 показано визуальное представление для обсуждавшегося ранее примера регистрации.

Рисунок 1. Структурный графический браузер для навигации и, в частности, чтобы определить, какие методы AutoLog используются в конструкции advice



В дополнение к структурному браузеру и основному компилятору можно загрузить с Web-сайта AspectJ отладчик для аспектов, инструмент javadoc, Ant task и plug-in Emacs.

5. Заключение

Стоит ли использовать AspectJ? Гради Буч описывает аспектно-ориентированное программирование как одно из трех направлений, которые в совокупности знаменуют начало фундаментальных изменений в способах проектирования и написания программного обеспечения (см. <http://www.sdmagazine.com/documents/s=843/sdm0107i/0107i.htm> - "Through the

Looking Glass", Software Development, July 2001). С ним вполне можно согласиться. Сфера действия АОР охватывает пространство проблем, непосильных для объектно-ориентированных и процедурных языков. Оно предлагает элегантные пути для реализации задач, решение которых сдерживалось из-за фундаментальных ограничений программирования. Было бы справедливо сказать, что АОР представляет собой одну из самых мощных абстракций в программировании с момента появления объектов.

Разумеется, для AspectJ есть некоторая “кривая обучения”. Как в любом языке или расширении языка программирования, в нем есть свои тонкости, которые необходимо освоить, прежде чем задействовать всю мощь этого средства. Однако, “кривая обучения” не слишком крутая – по прочтении руководства пользователя и после проработки нескольких примеров можно составлять полезные аспекты. AspectJ воспринимается естественно, поскольку скорее заполняет пробел в знаниях по программированию, чем придает им новое направление. Способность AspectJ к «модулированию немодулируемого» должна найти достойное применение. Если вы пока не готовы использовать AspectJ в полном объеме для разработки, его на первых порах можно легко применить в отладке, не упуская благоприятных возможностей, предоставляемых этим расширением.

6. Ссылки на ресурсы Internet

- www.aspectj.org - официальный ресурс AspectJ
- asod.net – портал поклонников аспектно-ориентированного программирования

Статью подготовили Рувинская Виктория и Беркович Леонид