Раздел II. Информационные технологии, управление

УДК 004.416.6

С.Ю. Калинин, И.А. Колоколов, А.Н. Литвиненко

ПРИМЕНЕНИЕ КОНЦЕПЦИЙ АОП В РАЗРАБОТКЕ РАСШИРЯЕМЫХ ПРИЛОЖЕНИЙ

Рассматриваются проблемы разработки и сопровождения программных проектов больших приложений. Показывается необходимость введения специальной единицы модульности проекта – слоя. Описываются идеи аспектно-ориентированного программирования, на которых основан механизм слоёв. Проводится сравнительный анализ механизма слоёв с AspectJ – аспектно-ориентированным расширением языка Java.

Аспектно-ориентированное программирование; AspectJ; слой.

S.Y. Kalinin, I.A. Kolokolov, A.N. Litvinenko

APPLYING AOP CONCEPTS TO THE DEVELOPMENT OF EXTENDABLE APPLICATIONS

The paper deals with the problems of large program projects development and maintenance. The necessity of introduction of the special unit of project's modularity (called layer) is shown. The ideas of aspect-oriented programming on which the layer technique is based upon are described. The comparative analysis of the layer technique and AspectJ (aspect oriented extension to Java) is conducted.

Aspect-oriented programming; AspectJ; layer.

Формулировка одной из самых главных проблем разработки «больших» приложений звучит банально: объём программного кода увеличивается и его становится сложно контролировать. С каждым вмешательством в проект программный код становится всё менее и менее читабельным, всё больше и больше времени тратится на то, чтобы выяснить, кем и для чего были сделаны некоторые изменения. Безболезненное изменение программного кода становится всё более сложной задачей («Метод изменения содержимого программного фонда называется безболезненным, если его применение не может нарушить работоспособность отлаженных ранее версий программы» [1. С. 12]). Для эффективного решения задач сопровождения и расширения приложения предлагается использовать специальный механизм слоёв, который основан на концепции АОП (аспектно-ориентированного программирования).

Недостаточность существующих решений. Не секрет, что модуляризация – один из важнейших инструментов создания "хорошего" приложения. Вслед за монографией [1] под модулем будем понимать выделенную по тем или иным мотивам часть первичного материала программного фонда [1. С. 42]. Модульность современного мейнстримового программирования основана на понятии объект. ООП с редкими примесями АОП, по сути, позиционируются сегодня как универсальный инструмент для решения практически любых задач программирования. Однако современный класс ООП не всегда хорошо играет роль модуля многократного использования. Существенный минус такого подхода заключается в том, что он по-

зволяет выделять модули только в исходных файлах, написанных на конкретном объектно-ориентированном языке. В то же время программный проект может включать код на языках, не являющихся объектно-ориентированными. К примеру, программный проект некоторого СУБД-приложения обычно содержит файлы с кодом на некотором диалекте SQL, текстовые файлы с описаниями на естественном, "человеческом" языке, конфигурационные XML-файлы и т.д. В этом случае модуляризация, предлагаемая ООП, не обеспечивает адекватной изменяемости и многократного использования. Здесь нужна модульность иного уровня. Нужно получить возможность выделять в модуль функциональность, которая может быть разнесена по различным файлам, содержащих код на различных языках. Такой возможности не дают и современные реализации АОП. Хотя они и позволяют оформить сквозную функциональность [2. С. 2] в виде аспекта, но остаются привязанными к конкретному языку: аспект в них является просто особым видом объекта ООП и должен содержать код только на фиксированном языке. Поэтому в качестве единицы построения программного проекта рассматривается модуль, который мы будем называть слоем. Это некоторый аналог аспекта, объединяющий в себе элементы любых файлов проекта, а не только файлов с кодом на конкретном языке. (Понятие «слой», которое во многом схоже с рассматриваемым в статье, введено в монографии А.Л. Фуксмана [4].)

АОП (аспектно-ориентированное программирование). Объектно-ориентированное программирование предоставляет большой набор инструментов для оформления функциональности в виде модулей (к примеру, функции, классы), но некоторую функциональность с помощью предложенных методов принципиально невозможно выделить в отдельные модули. Такую функциональность обычно называют сквозной, так как её реализация разбросана по различным модулям программы [2. С. 2]. Типичные задачи, приводящие к появлению сквозной функциональности, – это задачи журнализации и обработки ошибок. Вызовы функций, реализующих журнализацию и обработку ошибок, обычно разбросаны по различным частям проекта, и нет практически никакой возможности манипулировать всей логикой журнализации (или обработки ошибок) как единым модулем. АОП призвано решить эту проблему. С точки зрения АОП разработка приложения распадается на две задачи:

1) выделение функциональных компонентов, т.е. классов, функций;

2) разработка механизмов взаимодействия этих компонентов, т.е. определение структур, объединяющих функциональные компоненты.

Объектно-ориентированные языки решают лишь первую задачу. Для решения второй задачи АОП предлагает использовать специальный вид модуля – аспект.

Аспект – это набор функциональных компонентов и правила их взаимодействия с другими функциональными компонентами. Другими словами, аспект есть собранная воедино сквозная функциональность, а также правила, определяющие, как и где эта функциональность в проекте должна работать. При аспектно-ориентированной разработке выделяются традиционные модули (классы, функции) и аспекты. Либо на этапе выполнения, либо на этапе компиляции функциональность аспектов на основе указанных в них правил "вплетается" в традиционные модули. Этот процесс принято называть вплетением (weaving), а механизм, который реализует вплетение, – интегратором аспектов (aspect weaver). В итоге получается код, предназначенный для выполнения. Будем называть полученный код переплетённым представлением.

AspectJ как реализация идей АОП. Рассмотрим в качестве примера современной реализации идей АОП расширение языка Java под названием AspectJ [3]. AspectJ является не только самой распространённой на данный момент реализаци-

ей идей АОП, но и очень типичной – большинство прочих реализаций чрезвычайно похожи на AspectJ.

Ключевые понятия AspectJ:

- JoinPoint – строго фиксированная точка выполнения программы, к примеру, вызов метода, доступ к полю класса, возникновение исключения;
- PointCut – набор точек JoinPoint, удовлетворяющих некоторому критерию отбора;
- Advice – набор инструкций языка (в данном случае java), выполняемых до, после или вместо каждой точки выполнения JoinPoint из некоторого среза PointCut;
- Introduction – способность аспекта изменять структуру класса.

Аспектом в AspectJ является особый вид объекта, который содержит элементы PointCut и Advice. То есть аспект AspectJ – объект с некоторой функциональностью и условиями, при которых эта функциональность должна выполняться. Такие условия определяются двумя факторами:

- для каких точек выполнения программы (JoinPoint) функциональность применяется;
- когда (по отношению к инструкциям точки JoinPoint) эта функциональность должна отработать.

Первая часть условия задаётся с помощь конструкции PointCut. Например,

call (public void MyMethod (int))

является конструкцией PointCut, которая соответствует вызовам public-метода MyMethod, принимающего в качестве параметра переменную типа int и возвращающего переменную типа void. В конструкциях PoitCut можно применять групповые символы (wildcards) и знаки логических операций &&, || и !. К примеру PointCut вида

*call (public * * (..)) && ! call (public void MyMethod (int))*

соответствует вызову всех public-методов с любыми параметрами (первый символ "*" означает произвольный возвращаемый тип, второй – произвольное имя метода, символы ".." в списке параметров – произвольные параметры) кроме public-метода MyMethod с описанной сигнатурой.

Правила, определяющие, когда набор инструкций выполняется, задаются следующими ключевыми словами:

- before. Функциональность, описанная в advice, выполняется до функциональности точки выполнения JoinPoint;
- afterreturning. Функциональность, описанная в advice, выполняется после возвращения значения из точки выполнения JoinPoint;
- afterthrowing. Функциональность, описанная в advice, выполняется после возникновения исключительной ситуации в точке выполнения JoinPoint;
- after. Функциональность, описанная в advice, выполняется после функциональности точки выполнения JoinPoint;
- around. Функциональность, описанная в advice, выполняется вместо функциональности точки выполнения JoinPoint.

Рассмотрим конкретный пример применения AspectJ. Пусть в рамках соглашения по созданию проекта разработчику в начало каждого public-метода класса LogMe необходимо добавлять вызов метода logger.log, который осуществляет журнализацию. Очевидно, что без применения технологий аспектного программирования следование этим правилам неоправданно усложнит каждый метод класса LogMe:

```
public void doMethod()
{
        logger.log();
        /* основной код метода */
}
```

Для решения этой задачи механизмами AspectJ необходимо создать следующий аспект:

```
aspect AutoLog
{
        pointcut publicMethods(LogMe s):target(s) && call(public * *(..));
        before(LogMe s): publicMethods(s)
        {
                Logger.log();
        }
}
```

Интегратор аспектов построит (на этапе выполнения либо на этапе компиляции) переплетённое представление на основе этого аспекта. В итоге при выполнении полученного кода перед каждым public-методом класса LogMe гарантированно будет выполняться метод logger.log.

Слои. Изложенная выше реализация принципов АОП содержит существенный минус. Он заключается в том, что аспект должен состоять из фрагментов, написанных на одном языке. Для AspectJ, к примеру, этим языком является Java, и оформить аспект, который кроме кода на Java содержал бы фрагменты на SQL, XML или других языках, не представляется возможным.

Слой как единица модульности проекта. Главным мотивом при выделении каких-либо модулей является следующий: модуль должен наиболее полно соответствовать некоторой логически единой сущности. То есть модуль, описанный "компьютерным языком", должен наиболее естественно соответствовать некоторой "человеческой концепции". Однако при выделении модулей в объектно-ориентированном программировании играет роль и "неестественный" в некотором смысле мотив модуляризации: сделать модуль (функцию, объект) функциональным самостоятельно. Такой мотив делает модуль смещённым в сторону условностей компьютерных концепций и менее прозрачным с точки зрения концепций человеческих. Данная статья предлагает разделять программный проект на набор модулей, каждый из которых соответствует некоторой логически единой задаче, человеческой концепции, бизнес-объекту. При этом модуль может состоять из фрагментов кода, написанных на различных языках. Модули можно помечать произвольными метками (о метках речь пойдёт ниже) и связывать в иерархии. Такой модуль вслед за монографией [4] будем называть слоем. Таким образом, проект будет распадаться на набор слоёв. При этом весь проект целиком (для формальной строгости) можно считать некоторым базовым слоем, который стоит на вершине иерархии слоёв. Установившееся в практике программирования деление СУБД-приложения на 3 части (слой данных (Data Access Layer – DAL), слой бизнес-логики (Business Logic Layer – BLL) и слой представления (Presentation Layer – PL)) есть частный случай выделения слоёв и хорошо иллюстрирует предлагаемую концепцию.

Слой, по сути, является мультиязыковым аспектом. Основная задача слоя – собрать воедино весь текст, относящийся к некоторому бизнес-объекту, в едином модуле. Под текстом далее будем понимать не только собственно программный код, но и комментарии, описания, документацию.

Можно дать следующее простое функциональное определение слоя: слой – это модуль (вообще говоря, мультиязыковый), содержащий весь текст, отвечающий за реализацию некоторого бизнес-объекта.

Слой может находиться в двух представлениях:

- весь текст, относящийся к данному бизнес-объекту, рассредоточен по проекту. Такое представление будем называть рассредоточенным представлением слоя. Код слоя в рассредоточенном представлении смешан с остальным кодом проекта и компилируется вместе с ним;
- весь текст, относящийся к данному бизнес-объекту, собран в виде единого модуля (слоя). Такое представление будем называть сосредоточенным представлением слоя. Код слоя в сосредоточенном представлении отделён от остального кода проекта и является временной некомпилируемой структурой. Механизм, который переводит слой из рассредоточенного представления в сосредоточенное и обратно, будем называть интегратором слоёв.

Если проводить аналогию с аспектами AspectJ, то кардинальное отличие слоёв заключается в следующем. Схема добавления нового аспекта в AspectJ такова:

1) программист определяет аспект как класс;

2) интегратор аспектов (aspect weaver) строит переплетённое представление.

Ключевой момент здесь в том, что первичным объектом для работы является аспект, а переплетённое представление вторично и обычно не предназначено для непосредственного редактирования.

Схема создания нового слоя отличается от схемы добавления нового аспекта:

1. Программист, работая с кодом проекта, добавляет вручную код, относящийся к слою. То есть первоначально формируется рассредоточенное представление слоя.

2. Запуская механизм интегратора слоёв, программист получает сосредоточенное представление слоя.

3. В дальнейшем, внося изменения в проект, программист работает с сосредоточенным представлением соответствующего слоя. Снова запуская интегратор слоёв, программист переводит обновлённое сосредоточенное представление в рассредоточенное представление, тем самым внося изменения в рабочий код проекта.

Метки. Для того чтобы программист мог относить некоторый текст к определённому слою, используется механизм меток. Начало блока кода, который программист решает включить в некоторый слой, должно быть отмечено открывающей меткой, а конец – закрывающей. Можно дать следующее определение слоя, используя понятие меток: слой – это набор (вообще говоря, мультиязыковых) фрагментов кода, помеченных парными метками и относящихся к реализации некоторого бизнес-объекта. Текст, заключенный в метки, будем называть фрагментом слоя. Метки всегда парные: каждой открывающей метке должна соответствовать закрывающая.

Если говорить о формальном описании общего вида меток (без привязки к конкретному языку), то открывающая метка является строкой, которая состоит из следующих элементов (подстрок), разделённых пробелами:

<символ однострочного комментария>, <открывающий символ метки>, <имя слоя>, <номер фрагмента>, <тип фрагмента>, <дополнительная информация>.

Закрывающая метка имеет симметричный вид:

<символ однострочного комментария>, <закрывающий символ метки>, <имя слоя>, <номер фрагмента>, <тип фрагмента>, <дополнительная информация>.

Здесь *<символ однострочного комментария>* – это символ однострочного комментария в том языке, на котором пишется данный фрагмент слоя. К примеру,

если фрагмент слоя относится к коду на Microsoft C#, то этим символом будет //, если к коду на Microsoft FoxPro, то *. Этот символ нужен для того, чтобы метки воспринимались компилятором как комментарий. <*открывающий символ метки*> и <*закрывающий символ метки*> – это произвольные символы, которые нужны для того, чтобы отличать открывающую метку от закрывающей. <*имя слоя*> задаёт имя слоя, к которому данный фрагмент относится. <*номер фрагмента*> задаёт номер данного фрагмента, который позволяет однозначно определить фрагмент. Пары <*имя слоя*>, <*номер фрагмента*> должны быть уникальными, для того чтобы метка однозначно определяла фрагмент текста. Такая уникальность должна контролироваться редактором среды разработки. Удобно, если такой редактор при создании очередной метки подсказывает посредством Intellisense следующий доступный номер фрагмента слоя. <*тип фрагмента*> задаёт тип фрагмента слоя. Авторы данной работы предлагают разделять фрагменты на три типа: фрагменты определяющего вхождения, фрагменты использующего вхождения и метаданные слоя (подробнее типы фрагментов рассматриваются ниже). Наконец <*дополнительная информация*> – это некоторая дополнительная информация о фрагменте слоя, которая может указываться программистом.

К примеру, вот так может выглядеть пара меток в языке Microsoft FoxPro:

**(LOGGER 2 DEF журналирование при сохранении,*

**) LOGGER 2 DEF журналирование при сохранении.*

Здесь "*" – однострочный комментарий языка FoxPro, "(" и ")" – открывающий и закрывающий символы метки, "LOGGER" – имя слоя, "2" – номер фрагмента слоя, "DEF" – тип фрагмента, т.е. константа, указывающая, что данный фрагмент относится к определяющему вхождению, "журналирование при сохранении" – дополнительная информация о назначении фрагмента слоя.

Типы фрагментов слоя. Будем выделять 3 типа фрагментов слоя:

1) фрагменты определяющего вхождения слоя;

2) фрагменты использующего вхождения слоя;

3) фрагменты метаданных слоя.

Такое разделение не требуется для корректной работы механизма слоёв, а вводится лишь для удобства манипулирования слоем.

Фрагмент определяющего вхождения слоя – это фрагмент слоя, который содержит определения конструкций языка программирования (переменных, функций, процедур, интерфейсов, классов, XML-структур и т.п.), относящихся непосредственно к данному слою. Такие конструкции могут быть описаны в произвольных файлах проекта и должны заключаться в метки с фиксированным типом фрагмента, скажем, "DEF". В тексте проекта фрагментов определяющего вхождения может быть произвольное количество. К примеру, если мы описываем слой, отвечающий за журнализацию в системе, то функцию, которая эту журнализацию реализует, нужно описать во фрагменте определяющего вхождения слоя:

```
*( LOGGER 1 DEF
PROCEDURE LOGGER
* код процедуры журнализации
ENDPROC
*) LOGGER 1 DEF
```

Все фрагменты, помеченные в коде проекта парой меток с типом "DEF" (в примере **(LOGGER 1 DEF* и **) LOGGER 1 DEF*), после интеграции слоя попадут в блок кода, который будем называть *определяющим вхождением слоя*.

Таким образом, после интеграции слой (в сосредоточенном его представлении) будет включать блок кода, содержащий все определения конструкций, относящихся к данному слою. Просмотр такого блока кода является чрезвычайно информативным и сильно проясняет логику работы слоя.

Фрагменты использующего вхождения слоя – это, как правило, фрагменты слоя, содержащие код проекта, в котором используются конструкции, описанные в определяющем вхождении. Кроме того, сюда относятся фрагменты такого кода, который не имеет смыла оформлять (или вообще нельзя оформить) в виде конструкции определяющего вхождения, но который также непосредственно относится к слою. К примеру, вызов функции журнализации в коде проекта нужно оформить в виде фрагмента использующего вхождения:

```
*( LOGGER 2 журнализация при сохранении;
DO LOGGER && вызвать процедуру журнализации;
*) LOGGER 2 журнализация при сохранении.
```

Поскольку фрагменты использующего вхождения используются чаще других, в их метках удобно тип не указывать вообще и считать по умолчанию фрагментом использующего вхождения фрагмент с меткой без типа.

Предположим также, что при включении журнализации необходимо изменить заголовок некоторой экранной формы. Вряд ли имеет смысл оформлять такой код в виде отдельной процедуры. Но фрагмент кода, который изменяет свойство Caption экранной формы, очевидно, непосредственно относится к слою журнализации, и его также необходимо оформить в виде фрагмента использующего вхождения:

```
*( LOGGER 3 изменить заголовок формы,
ThisForm.Caption = ThisForm.Caption + " Журнализация",
*) LOGGER 3 изменить заголовок формы.
```

Все такие фрагменты (с метками без типа) после интеграции слоя попадут в блок кода, который будем называть использующим вхождением слоя.

То есть после интеграции слой (в сосредоточенном его представлении) будет включать блок кода, содержащий все обращения к конструкциям определяющего вхождения (и прочий код, который не оформлен в виде определяющего вхождения).

Фрагменты метаданных слоя – это фрагменты с некоторыми дополнительными данными о слое. Сюда включаются описания слоя на уровне пользователя и более детальное и глубокое описание слоя на уровне программиста. Метаданные слоя могут содержать некоторые произвольные тэги, маркирующие слой. С помощью таких тэгов можно, к примеру, организовывать быстрый поиск. Подобно фрагментам определяющего и использующего вхождений, фрагменты метаданных слоя могут быть описаны в произвольных файлах проекта и должны заключаться в метки с фиксированным типом фрагмента, скажем, "DESC". Таких фрагментов может быть произвольное количество. К примеру, описание процедур журнализации нашего гипотетического слоя можно поместить в следующих фрагментах:

```
*( LOGGER 4 DESC описание журнализации для пользователя
* Журнализация всегда включена при редактировании документов.
* Пользователь может просмотреть журнал изменения данных,
* запустив экранную форму "Журнализация" из меню "Сервис"...
*) LOGGER 4 DESC описание журнализации для пользователя
*( LOGGER 5 DESC описание журнализации для разработчика
* Ключевые моменты журнализации реализованы в процедуре LOGGER.
* Алгоритм её работы следующий...
*) LOGGER 5 DESC описание журнализации для разработчика
```

Все фрагменты, помеченные в тексте проекта метками с типом "DESC", после интеграции слоя попадут в блок *метаданных слоя*. Ясно, что иметь под рукой блок с собранными воедино комментариями относительно работы слоя чрезвычайно удобно и информативно.

Общая структура слоя и схема работы механизма слоёв. В итоге после запуска интегратора слоёв, все фрагменты слоя LOGGER будут собраны в следующее сосредоточенное представление:

```
*( LOGGER
 СЛОЙ: LOGGER
 *( ФАЙЛЫ СЛОЯ
 C:\Projects\Test\PLUGIN\test.prg
 *) ФАЙЛЫ СЛОЯ
 *( ФАЙЛЫ, ЦЕЛИКОМ ОТНОСЯЩИЕСЯ К СЛОЮ
C:\Projects\Test\PLUGIN\test.prg
 *) ФАЙЛЫ, ЦЕЛИКОМ ОТНОСЯЩИЕСЯ К СЛОЮ
 *( ###TAG MetaData
*( LOGGER 4 DESC описание журнализации для пользователя
* Журнализация всегда включена при редактировании документов.
* Пользователь может просмотреть журнал изменения данных,
* запустив экранную форму "Журнализация" из меню "Сервис"...
*) LOGGER 4 DESC описание журнализации для пользователя
*( LOGGER 5 DESC описание журнализации для разработчика
* Ключевые моменты журнализации реализованы в процедуре LOGGER.
* Алгоритм её работы следующий...
*) LOGGER 5 DESC описание журнализации для разработчика
*) ###TAG MetaData
 *( ###TAG Definition
*( LOGGER 1 DEF  {\PLUGIN\test.prg}
PROCEDURE LOGGER
* код процедуры журнализации
ENDPROC
*) LOGGER 1 DEF
 *) ###TAG Definition
 *( ###TAG Usage
*( LOGGER 2 журнализация при сохранении {\PLUGIN\test.prg}
DO LOGGER   && вызвать процедуру журнализации
*) LOGGER 2 журнализация при сохранении
*(LOGGER 3 изменить заголовок формы {\PLUGIN\test.prg}
ThisForm.Caption = ThisForm.Caption + " Журнализация"
* ) LOGGER 3 изменить заголовок формы
*) ###TAG Usage
*) LOGGER
```

Блок кода внутри меток специального вида **(###TAG MetaData* и **(###TAG MetaData* содержит метаданные слоя. Метки **(###TAG Definition* и **) ###TAG Definition* ограничивают определяющее вхождение слоя, **(###TAG Usage* и **) ###TAG Usage* – использующее вхождение. Кроме этих основных блоков кода в начале слоя содержится список файлов, в которых встретились фрагменты данного слоя, а также список файлов, содержащих фрагменты только данного слоя.

В дальнейшем, когда программисту необходимо будет что-то изменить в логике работы журнализации, ему не придётся выискивать в проекте строки кода, реализующие журнализацию. После запуска интегратора слоёв он будет вносить изменения непосредственно в сосредоточенное представление слоя, которое для удобства манипулирования разбито на три части.

Сосредоточенное представление слоя, вообще говоря, является временной структурой, которая не компилируется. Если она хранится в некотором файле (от-

дельном или нет), то перед редактированием сосредоточенное представление нужно "пересобрать", т.е. запустить интегратор слоёв и получить актуальное сосредоточенное представление, со всеми изменениями, которые появились в файлах проекта. После того как программист внёс необходимые ему изменения в сосредоточенное представление слоя, он снова запускает интегратор слоёв, который "вплетает" изменения в файлы проекта. Пусть, к примеру, в приведённом выше сосредоточенном представлении слоя программист изменил код фрагмента **(LOGGER 3*. Чтобы эти изменения были внесены в проект, программист запускает интегратор слоёв. Интегратор ищет в коде проекта фрагмент, помеченный такой же меткой (точнее меткой, с таким же именем слоя и таким же номером фрагмента). После этого код найденного фрагмента заменяется кодом, который содержится в сосредоточенном представлении во фрагменте с меткой **(LOGGER 3*, т.е. тем кодом, который программист менял. В итоге изменения, внесённые в сосредоточенное представление слоя, попадают в код проекта.

Детали реализации механизма слоёв. Если говорить о конкретных деталях реализации, то авторы пользуются интегратором слоёв, реализованным в виде плагина для свободно распространяемого текстового редактора Notepad++ (http://notepad-plus.sourceforge.net/ru/site.htm). Кроме того, в виде плагина к Notepad++ реализованы визуальные средства для отображения списка слоёв проекта и манипулирования ими. Программист имеет возможность совершать следующие основные действия со слоями:

- получить список слоёв проекта;
- получить сосредоточенное представление некоторого слоя из списка;
- "вплести" изменения, внесённые в сосредоточенное представление слоя в файлы проекта;
- переходить от фрагментов проекта к фрагментам сосредоточенного представления слоя и наоборот;
- отключить (закомментировать) и включить слой.

Сам редактор Notepad++ чрезвычайно удобно настроить так, чтобы он позволял использовать "сворачивание" (folding) фрагментов слоя. Это делает просмотр кода слоя куда более комфортным и информативным.

Сравнительный анализ AspectJ и механизма слоёв. Проведём сравнительный анализ реализации идей АОП в AspectJ и в механизме слоёв. В первую очередь отличаются цели применения аспекта AspectJ и слоя. Грубо говоря, цель применения слоя - собрать в виде единого модуля всё, что относится к некоторому бизнес-объекту в проекте. Цель применения аспекта AspectJ – гарантированно выполнить некоторый код после (до или вместо) определённых точек JointPoint проекта, не затрагивая код самого проекта (кроме, собственно, кода самого аспекта). Как уже отмечалось выше, отличается алгоритм добавления нового слоя и нового аспекта AspectJ в проект. При использовании AspectJ роль переплетённого представления кода вторична и оно не предназначено для просмотра и редактирования. При использовании же слоёв переплетённое представление кода вторичным не является. Как и аспект AspectJ, слой (точнее его сосредоточенное представление) состоит из двух элементов:

- функциональность, относящаяся к некоторой логически единой задаче;
- правила "вплетения" этой функциональности в файлы проекта.

Однако в случае слоя правила вплетения очень простые и не требуют структур вроде PointCut. "Вплетение" происходит благодаря меткам, расставленным в коде проекта. В AspectJ конструкции advice можно привязывать только к ограниченному набору точек JointPoint, которыми могут быть вызов метода, доступ к полю класса, возникновение исключения. Привязать выполнение некоторых инст-

рукций к произвольному месту в коде невозможно. Кроме того, действие аспекта AspectJ распространяется только на программный код, написанный на Java. Механизм слоёв позволяет привязывать некоторые инструкции к произвольному месту в коде проекта, причём независимо от языка, на котором этот код написан. Однако такая гибкость достигается путём дополнительных усилий программиста по расстановке меток в коде проекта.

В случае же аспекта AspectJ достигается полная безболезненность изменений, поскольку программист вносит изменения лишь в код аспекта и не затрагивает отлаженного кода проекта.

Заключение. Посредством меток слои можно связывать в иерархии. Это приводит к тому, что код программного проекта, разделанный на слои, перестаёт быть просто текстом. Он обретает свойства структуры, которая оптимизирована для задач сопровождения. Такая структура даёт возможность взглянуть на проект с необходимой точки зрения и получить в удобном виде весь текст, относящийся к некоторому бизнес-объекту. В простейшем случае, если нам нужно внести изменения, скажем, в логику журнализации, достаточно получить сосредоточенное представления необходимого слоя и всё, что отвечает за журнализацию, будет перед нашими глазами. Причём в удобном виде: отдельным блоком все комментарии (и прочие метаданные), отдельно все определения необходимых в слое структур и отдельно все использования этих структур. В более сложном случае можно пойти ещё дальше – реализовать гибкие механизмы для выборки и писать специальные запросы (аналогичные SQL) к коду проекта и получать слои и фрагменты слоёв, удовлетворяющие некоторому критерию. Применение такого механизма видится чрезвычайно перспективным. А простейший механизм работы со слоями уже оправдывает себя, сокращая затраты труда по сопровождению реально работающих СУБД-приложений.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Горбунов-Посадов М.М.* Расширяемые программы. – М.: Полиптих, 1999. – 336 с.
2. *Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J., Irwin J.* Aspect-oriented programming. Published in preceedings of the European Conference on Object-Oriented Programming (ECOOP), Jyvaskyla, Finland, 1997 URL: http://people.cs.ubc.ca/~gregor/papers/kiczales-ECOOP1997-AOP.pdf (дата обращения 15.09.2009).
3. *Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W.* An Overview of AspectJ. Published in preceedings of the European Conference on Object-Oriented Programming (ECOOP), Budapest, Hungary, 2001. URL: http://people.cs.ubc.ca/~gregor/papers/kiczales-ECOOP2001-AspectJ.pdf (дата обращения 15.09.2009).
4. *Фуксман А.Л.* Технологические аспекты создания программных систем. – М.: Статистика, 1979. – 184 с.

Калинин Сергей Юрьевич
Южно-Российский региональный центр информатизации (ЮГИНФО) Федерального государственного образовательного учреждения высшего профессионального образования "Южный федеральный университет" в г. Ростове-на-Дону.
E-mail: the_distance@mail.ru.
344015, г. Ростов-на-Дону, ул. 339 стрелковой дивизии, 17.
Тел.: 89185961472.

Колоколов Иван Анатольевич
Федеральное государственное образовательное учреждение высшего профессионального образования "Южный федеральный университет" в г. Ростове-на-Дону.
E-mail: i__one@list.ru
344015, г. Ростов-на-Дону, ул. Еременко, 60/2.
Тел.: 89044456227.

Литвиненко Александр Николаевич
E-mail: litva@rsu.ru.

Kalinin Sergey Yurievich
South Russian Regional Center of Informatization (UGINFO) of the Federal State-Owned Educational Establishment of Higher Vocational Education "Southern Federal University".
E-mail: the_distance@mail.ru.
17, 339 rifle division street, Rostov-on-Don, 344015, Russia.
Phone: 89185961472.

Kolokolov Ivan Anatolevich
Federal State-Owned Educational Establishment of Higher Vocational Education "Southern Federal University".
E-mail: i__one@list.ru.
60/2, Eremenko street, Rostov-on-Don, 344015, Russia.
Phone: 89044456227.

Litvinenko Alexander Nikolaevich
E-mail: litva@rsu.ru.

УДК 681.325.3

В.В. Сарычев

ТЕЛЕМЕТРИЧЕСКАЯ СИСТЕМА НА БАЗЕ ИНТЕЛЛЕКТУАЛЬНЫХ ИНТЕРФЕЙСОВ

Предлагается аппаратное решение для оконечного устройства телеметрических систем, создаваемых на базе современных интерфейсов обмена данными. Децентрализация процессов преобразования сигналов дает новые возможности для формирования потоков данных в зависимости от динамических свойств первичных сигналов в каждом канале.

Дискретизация; частота дискретизации; информационно-измерительная система.

V.V. Sarychev

TELEMETERING SYSTEM ON THE BASIS OF INTELLECTUAL INTERFACES

Hardware solution for the terminal of the telemetering systems created on the basis of modern interfaces of data exchange is offered. Decentralisation of processes of conversion of signals gives new possibilities for creation of data flows depending on dynamic properties of primary signals in each channel.

Dgitization; sampling rate; informational-measuring system.

В процессе телеметрических измерений параметров объекта наиболее информативными считаются моменты смены состояний объекта, а также аварийные ситуации. Эти режимы сопровождаются пиком интенсивности потока данных и требуют предельных значений частот дискретизации сигналов с датчиков. Бóльшую часть периода контроля объект находится в штатном режиме, когда информативность данных низкая, а каналы связи, память, вычислительные ресурсы продолжают работать в режиме предельных нагрузок. Для сохранения общей работоспособности телеметрии предельные нагрузки снижают путем уменьшения частот дискретизации до значений, определяемых условиями штатного режима, в ущерб информативности для периодов критических состояний объекта [1]. В большей степени такое положение определяется традиционной архитектурой телеметрических систем: датчики – фильтры – коммутатор – АЦП – канал связи, которая подразумевает синхронный режим работы без учета изменения состояния объекта. Применение процедур программно-адресного опроса датчиков позволяет