

# Аспектно-Оrientированное Программирование

# Содержание

Аспектно-Ориентированное Программирование.....	1
Тезисы:.....	3
1 Понятие об объектно-ориентированном проектировании.....	3
2 понятие аспектов и типовые примеры аспектов.....	4
3 аналоги в других технологиях программирования (темплэйты, инструментальные классы, trigger ы, view, декларативное программирование, функциональное программирование).....	5
Динамические прокси заменяют АОП.....	5
4 декларативное изменение объектов-классов как единый аспект.....	6
5 общие термины pointcut, joint points, advice.....	7
6 примеры на с++/java/C#.....	8
7 связь с декларативным программированием на макросах.....	9
8 проблемы (тестовое покрытие, оптимизация, анализ кода).....	10
9 связь с программированием по контракту .....	11
10 Примеры самых распространённых систем (Spring) .....	12
Контрольные вопросы:.....	13

## **Тезисы:**

### ***1 Понятие об объектно-ориентированном проектировании.***

**Объектно-ориентированное проектирование (ООП)** — это часть [ОО методологии](#), которая предоставляет возможность [программистам](#) оперировать [понятием объект](#), нежели [процедур](#) при разработке своего [кода](#). Объекты содержат [инкапсулированные данные](#) и [процедуры](#), [сгруппированные](#) вместе, отображая т.о. сущность объекта. «[Интерфейс объекта](#)», описывает взаимодействие с объектом, то, как он определен. [Программа](#), полученная при реализации объектно-ориентированного исходного кода, описывает взаимодействие этих объектов.

## 2 понятие аспектов и типовые примеры аспектов

— [парадигма программирования](#), основанная на идее разделения функциональности, особенно сквозной функциональности, для улучшения разбиения программы на [модули](#).

Существующие парадигмы программирования, такие как [процедурное программирование](#) и [объектно-ориентированное программирование](#), предоставляют некоторые способы для разделения и выделения функциональности, например, [функции](#), [объекты](#), [классы](#), пакеты, но некоторую функциональность с помощью предложенных методов невозможно выделить в отдельные сущности. Такую функциональность называют сквозной, так как её реализация разбросана по различным модулям программы. Сквозная функциональность приводит к рассредоточенному и запутанному коду. Запутанным называется такой код, в котором одновременно реализована различная функциональность.

### [Аспектно-ориентированное программирование](#)

[Трассировка](#) — типичный пример сквозной функциональности. Другие примеры: контрактное программирование, в частности проверка пред- и пост-условий, обработка ошибок, реализация систем безопасности.

AspectJ

[w:AspectJ](#) — аспектно-ориентированное расширение [Джавы](#), предложенное [Xerox PARC](#). Представим его на примере программы для журналирования событий, реализованной на AspectJ. Этот пример взят из системы с открытым кодом Cactus, упрощающей тестирование Джава-компонентов на стороне сервера. Каркас Cactus разработан для поддержки процесса отладки с помощью трассировки вызовов всех методов. Версия 1.2 Cactus 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 вызовы `logger` в начало и в конец каждого метода. Кроме того, рекомендовалось заносить в журнал 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 *advise*.

[\[править\]](#) Конструкция *pointcut* и точки соединения

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 таковых в классе *Logger*. (Регистрация *log* методов привела бы в результате к бесконечной рекурсии).

[\[править\]](#) Конструкция *advise*

Теперь, после того, как в аспекте определены точки, нужно использовать конструкцию *advise*, 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 регистрировать». В нашем примере первый *advise* объявлен следующим образом:

```

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

```

Этот *advise* 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 исполняется точка соединения. В данном, используемом *Cactus*-ом аспекте, *advise* 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[])

```

[\[править\]](#) Advice типа *around*

В примере *Cactus* определены *advise* типа *before()* и *after()*. *Advice* третьего типа *around()* дает разработчику аспектов управлять передачей управления на точку соединения. При этом используется специальный синтаксис *proceed()*. Следующий *advise* 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.");
    }
}

```

[\[править\]](#) 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 ајс компилятором. ајс 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 – ајс 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 AOP-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.

[\[править\]](#) Обзор возможностей AspectJ

[\[править\]](#) 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.

[\[править\]](#) Пример ситуации, когда необходим механизм *introduction*

Предположим, что у нас есть объект, представляющий кэш непрерывно изменяющихся 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:

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*).

[\[править\]](#) Наследование в стиле mix-in

AspectJ позволяет добавлять члены в интерфейсы (аналогично добавлению в классы), что относится к наследованию в стиле mix-in а 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+ означает любой подкласс 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.

[править] 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 ајс 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 ајс обнаруживает его только на этапе компиляции (большинство pointcut могут срабатывать на основе информации времени выполнения).

[править] Обработка ошибок

Есть множество обрабатываемых исключений в языке [w: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 *advice* 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.

[\[править\]](#) Инструментальная поддержка

Хероx 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 графический 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. 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*.

[\[править\]](#) 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. статью [Through the Looking Glass](#)). С ним вполне можно согласиться. Сфера действия AOP 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 AOP 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.

### 3 аналогии в других технологиях программирования (темплэйты, инструментальные классы, trigger ы, view, декларативное программирование, функциональное программирование)

#### Динамические прокси заменяют АОП

*Реальность: АОП предоставляет более простое решение, чем динамические прокси*

Динамические прокси (реализация шаблона проектирования Проху (Заместитель)) кажутся хорошей альтернативой АОП. В конце концов, вам нужно только написать метод `invoke()`, который выглядит имитацией `advise`. Сильная сторона такого подхода состоит в том, что он заключает пересекающееся поведение в объекты, инициирующие вызов (`invocation object`). Слабые стороны решения с динамическими прокси заключаются в следующем:

- Вынуждает использовать сложную основанную на отражениях программную модель;
- Ослабляет статическую проверку типов, поскольку в обработчике инициатора вызова типом объектов является только *Object*. Вы можете встретить этот недостаток в основанных на прокси АОП-реализациях;
- Требуется создания объектов через фабрику объектов (пока вы не замените каждый оператор `new` кодом для создания динамических прокси – это тоже пересекающаяся задача). Вы можете встретить этот недостаток в основанных на прокси АОП-реализациях, хотя он может быть скрыт внутри системы;
- Создает проблему тождественности объектов: прокси и внутренний объект являются двумя различными сущностями, логически представляющими один объект. То есть, нужно ли рассматривать эти два объекта как одинаковые (например, при реализации метода `equals()`)? Ответ зависит от цели проверки на равенство и может не подходить для всех случаев.

Отметим, что существуют основанные на прокси системы АОП; наиболее известная - Spring AOP. Приведенные здесь соображения касаются главным образом прямого использования динамических прокси, а не основанных на прокси АОП-систем. Такие АОП-системы проявляют характеристики, более близкие к АОП, нежели к прокси, поскольку они скрывают прокси внутри системы, используя их просто как технологию реализации.

Аспектно-ориентированное решение, реализующее идентичную функциональность, проще, чем динамические прокси. Все что вам нужно – `advise`, который добавляет необходимое динамическое поведение. Поскольку не создаются дополнительные объекты, фабрика объектов не нужна, и проблемы тождественности объектов исчезают.

***4 декларативное изменение объектов-классов как единый аспект***

## **5 общие термины *pointcut*, *joint points*, *advice***

- **Точка выполнения** ([англ. JoinPoint](#)) — определенная точка выполнения программы.
- **Срез** ([англ. PointCut](#)) — набор точек выполнения программы.
- **Применение** ([англ. Advice](#)) — состоит из условий применения и реализации функциональности. Условия применения определяют до, после или вместо какого среза надо вставить требуемую функциональность.
- **Аспект** ([англ. Aspect](#)) — модуль AspectJ.
- **Представление** ([англ. Introduction](#)) — метод изменения структуры класса путем введения новых полей и методов, а также изменения иерархии наследования.

## 6 примеры на c++/java/C#

Существует множество реализации АОП на различных языках программирования, так для Java самыми распространенными являются

AspectJ и Spring AOP

для C++ - AspectC++

пример:

```
aspect Trace {  
    pointcut virtual functions() = 0;  
    advice execution(functions()) : around() {  
        cout << "before " << JoinPoint::signature() << "(";  
        for (unsigned i = 0; i < JoinPoint::ARGS; i++)  
            cout << (i ? ", " : "") << JoinPoint::argtype(i);  
        cout << ")" << endl;  
        tjp->proceed();  
        cout << "after" << endl;  
    }  
};
```

для C# - Aspect.NET, часть Spring.NET

*Каждая АОП-реализация использует свой «новый» язык*

Ну ладно, АОП – это хорошо и даже полезно. Но действительно ли нам нужен новый язык для создания АОП-реализаций? Разработчики, которые поверхностно знакомы с основанными на XML или на аннотациях АОП-реализациями (например, Spring AOP, JBoss AOP или AspectWerkz), обычно отвечают "нет". На высоком уровне кажется, что эти реализации обеспечивают поддержку АОП без нового языка.

На самом деле каждая АОП-реализация использует новый язык и единственным отличием является разновидность используемого языка. Если традиционный синтаксис AspectJ использует прямые расширения базового языка, то другие реализации используют язык, основанный на XML или на аннотациях. Рассмотрим таблицу 1, в которой вы можете увидеть определение pointcut, написанное для разных АОП-реализаций.

**Таблица 1. Определение pointcut в АОП-реализациях**

Стиль	АОП-реализация	Pointcut
Расширение языка	AspectJ	pointcut logOp() : execution(* Account.*(..)); /**
Основанный на аннотациях	AspectWerkz (использует Javadoc)	* @Expression execution(* Account.*(..)) */
Основанный на	AspectJ (использует	public Pointcut logOp; @Pointcut("execution(*

аннотациях	аннотации Java 5)	Account.*(..)) ") public void logOp() {}
Основанный на XML	Spring 2.0 (использует язык pointcut AspectJ)	<aop:pointcut id="logOp" expression="execution(* Account.*(..)) "/>
Основанный на XML	JBoss AOP	<pointcut name="logOp" expr="execution(* Account->*(..))"/>

Беглый взгляд показывает, что различия в языке pointcut нескольких АОП-реализаций довольно незначительны. Это же истинно и для других конструкций, таких как композиция pointcut и advice. По существу, нет возражений относительно того, что АОП-конструкции должны быть как-то выражены. Однако вы должны выбрать разновидность языка для этого.

## **7 связь с декларативным программированием на макросах**

программа «декларативна», если она написана на исключительно [функциональном](#), [логическом](#) или [константном языке программирования](#). Выражение «декларативный язык» иногда употребляется для описания всех таких языков программирования как группы, чтобы подчеркнуть их отличие от [императивных](#) языков.

### **Макросы в программировании**

[В языке ассемблера](#), а также в некоторых других [языках программирования](#), макрос — символьное имя, заменяемое при обработке [препроцессором](#) на последовательность программных инструкций.

Макросы программирования являются своего рода альтернативой для АОП, так как позволяют также провести некоторые действия, связанные с определенной функцией, однако в случае использования макросов, нам каждый раз вместо прямого обращения к функции необходимо будет вызывать макрос.

## 8 проблемы (тестовое покрытие, оптимизация, анализ кода)

### Аспекты затрудняют отладку

*Реальность: АОП упрощает отладку пересекающейся функциональности*

Отладка требует применения правильных инструментальных средств – независимо от того, есть аспекты, или нет аспектов. Это факт. С процедурным программированием все понятно, поскольку вы можете видеть ход выполнения программы, который прямо отображается в элементах программы. Объектно-ориентированные программы, особенно хорошо написанные, требуют дополнительной работы для отображения хода выполнения. Например, вы можете видеть DAO-объект (объект доступа к данным), но, возможно, понадобится исследовать, какая была использована реализация при инициализации объекта (например, основанная на JDBC или на Hibernate). Это может потребовать некоторых поисковых действий: определить, как реализована ссылка, затем переходить по ссылкам выше, возможно в файл конфигурации (например, если вы используете интегрированную среду ИОС, скажем, Spring).

Несмотря на нарушение планов, большинство разработчиков пришли к пониманию того, что эта абстракция в конечном итоге во благо – мы рассматривали это в предыдущем разделе. Создание уровня абстракции обеспечивает ясность и улучшает ваше понимание того, что является важным в данном контексте. Вот где показывает себя отладчик. Отладчик, понимающий *точный* тип объекта и *точный* ход выполнения программы, предоставляет линейное направление для навигации между различными сущностями.

Хотя аспекты делают ход выполнения программы еще менее явным, они также повышают уровень абстракции. С соответствующим отладчиком вы можете исследовать точное взаимодействие между классами и аспектами. Объектно-ориентированные пуристы тоже посматривают свысока на ясность, которую вносят аспекты в отладку. Средства отладки терпят неудачу, когда определенная пересекающаяся функциональность не ведет себя так, как ожидается. Если ваш код исключительно объектно-ориентированный, требуются огромные усилия, чтобы отследить все места неисправностей в разрозненной реализации и исправить их. Но, благодаря последним улучшениям в подключаемом модуле Eclipse AJDT, отладка аспектно-ориентированных программ почти такая же простая, как и отладка объектно-ориентированных программ. Думая таким образом, не будет преувеличением сказать, что АОП на самом деле упрощает отладку пересекающейся функциональности.



## 9 связь с программированием по контракту

Концепцию контрактного программирования (*Design by Contract*) ввел Бертран Майер (Bertrand Meyer).<sup>4</sup> Согласно этому принципу разработчик класса и пользователь этого класса имеют общие допущения по реализации класса. Контракт включает инварианты, предварительные условия и постусловия. Контрактное программирование позволяет разработчику классов сконцентрироваться на логике, реализующей функциональность класса, не беспокоясь о достоверности аргументов. Разумеется, если контракт содержит предварительные условия для аргументов. Контрактное программирование позволяет избежать создания дополнительного кода и улучшить производительность до тех пор, пока все клиенты класса придерживаются контракта.

Однако, если создается к примеру, библиотека, которая может быть использована сторонними разработчиками, контрактное программирование теряет свои преимущества, так как нельзя быть уверенным в валидности параметров.

Эту проблему можно решить, если вынести реализацию проверок в отдельный аспект, производящий валидацию данных, проверку пред-условий и пост-условий.

## 10 Примеры самых распространённых систем (Spring)

### IoC контейнер

В основе Spring лежит паттерн Inversion of control. Применительно к легковесным контейнерам, основная идея этого паттерна заключается в устранении зависимости компонентов или классов приложения от конкретных реализаций вспомогательных интерфейсов и делегировании полномочий по управлению созданием нужных реализаций IoC контейнеру. Рассмотрим UML диаграмму.

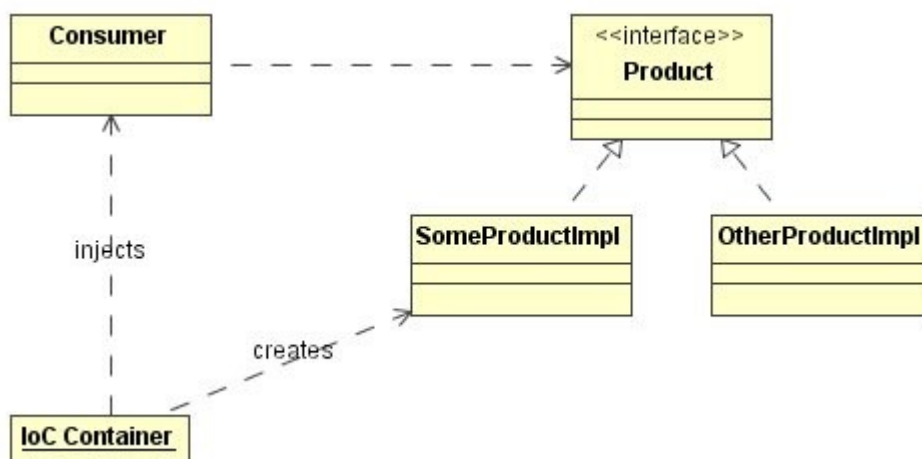


Рис. 2

IoC контейнер отвечает за создание нужной реализации Product для Consumer. При использовании класса Consumer в других проектах мы сможем заменить реализацию интерфейса Product на более подходящую, не внося изменений в код.

Основные преимущества IoC контейнеров:

1. управление зависимостями
2. упрощение повторного использования классов или компонентов
3. упрощение unit-тестирования
4. более "чистый" код (Классы больше не занимаются инициализацией вспомогательных объектов. Не стоит, конечно "перегибать палку", управляя созданием абсолютно всех объектов через IoC. В IoC контейнер лучше всего выносить те интерфейсы, реализация которых может быть изменена в текущем проекте или в будущих проектах.)

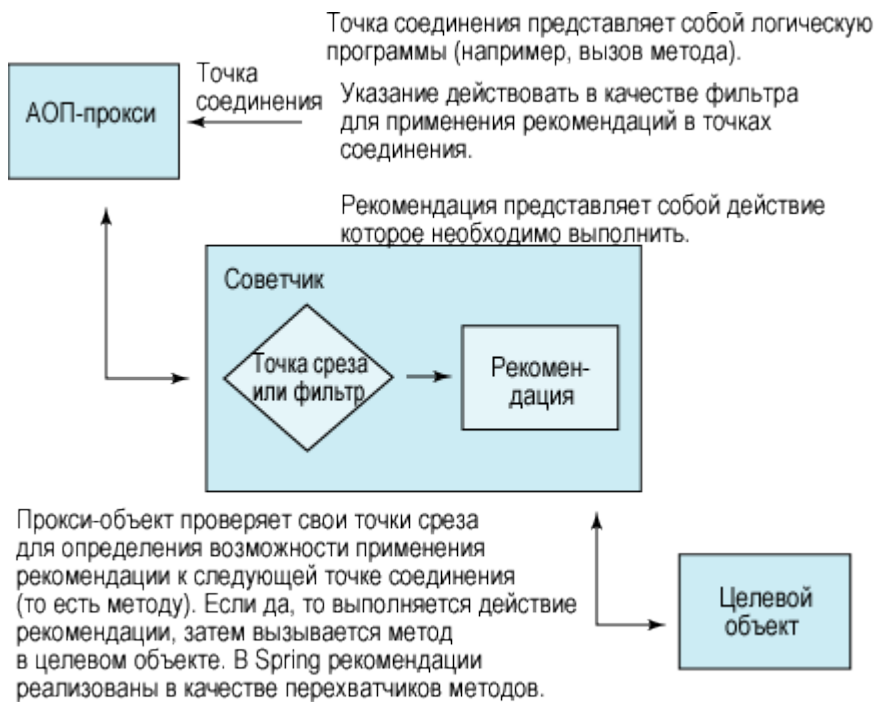
Среда Spring Framework в настоящий момент поддерживает некоторые понятия АОП, начиная со срезов точек и до рекомендаций.

- Аспекты реализованы как использование Spring в виде советников или перехватчиков;
- В Spring AOP точка соединения всегда соответствует вызову метода;
- Spring AOP предоставляет реализации для всех рекомендаций АОП в виде объектов-перехватчиков;
- Spring AOP поддерживает декларативное определение срезов точек при помощи регулярных выражений;

- В Spring AOP-прокси представляет собой динамический прокси JDK или прокси cglib.

На [рисунке 1](#) показана поддержка Spring AOP определенных здесь понятий.

**Рисунок 1. Понятия AOP, представленные в Spring AOP**



## Контрольные вопросы:

1. На каком этапе обрабатываются аспекты? (компиляция, выполнение)?
2. Можно ли при аспекте завести поле с данными и в каком месте это поле будет тогда храниться?
3. Как согласуются template ы (generic и) и АОП?
4. В каком порядке применяются advice ы?
5. Можно ли применять advice ы к методам Аспекта?
6. Можно ли каким либо образом получить исходники программы без аспектов?
7. Как тестировать программы на АОП? (с advice ми, без них, с выбранным набором)
8. Как пересекаются аспекты с уровнями видимости (private, public, protected)?
9. Каким образом можно следовать аспекто-ориентированной парадигме при программировании в обычной ООП среде?
10. Можно ли накладывать advice ы при загрузке dll? (в процессе линковки)?  
*Нет*
11. Можно ли в определении точки выполнения использовать условия на конкретные данные?
12. Можно ли тестировать отдельно каждый аспект?
13. Как точки выполнения ведут себя при работе с виртуальными методами? Можно ли прицепиться к методу конкретного класса или во всем его потомкам отдельно?
14. Как подстраховаться от того, что кто то переименует метод, к которому была прицеплена точка подключения?
15. Перечислите какие-нибудь из классических паттернов программирования, которые могут быть реализованы существенно по новому с использованием поддержки аспектов.
16. Могу ли я прицепить точку присоединения к вызову конкретного делегата?
17. Сработает ли точка присоединения если вызвать метод через делегат?
18. Как пересекаются аспекты и сериализация? Если мне при аспекте нужны какие то данные, как я буду их сериализовать?
19. Можно ли использовать точки присоединения к методам интерфейса?  
*да*
20. Можно ли выпускать библиотеки аспектов как коммерческий продукт и в каком виде они должны в этом случае поставляться?
21. Можно ли используя аспекты нарушить "контракт программирования" определённый

через interface

нет.

22. Можно ли продавать аспекты написанные для framework а Spring как отдельные продукты, не предоставляя исходников?

*Согласно лицензии Apache, под которой выпускается Springframework, пользователь имеет право свободно изменять, распространять копии и не требует неизменности лицензии, можно выпустить основанный на нем продукт под коммерческой лицензией.*

23. Расширяет ли использование аспектов во framework e Spring синтаксис языка Java?

*Нет, Spring использует описание на xml.*

24. Для чего нужен Inversion of Control (Dependency Injection)

*IoC нужен для устранения зависимостей между конкретными реализациями объектов.*

25. Для чего классы в Spring, работающие с аспектами должны наследовать interface?

*Для реализации возможностей IoC*