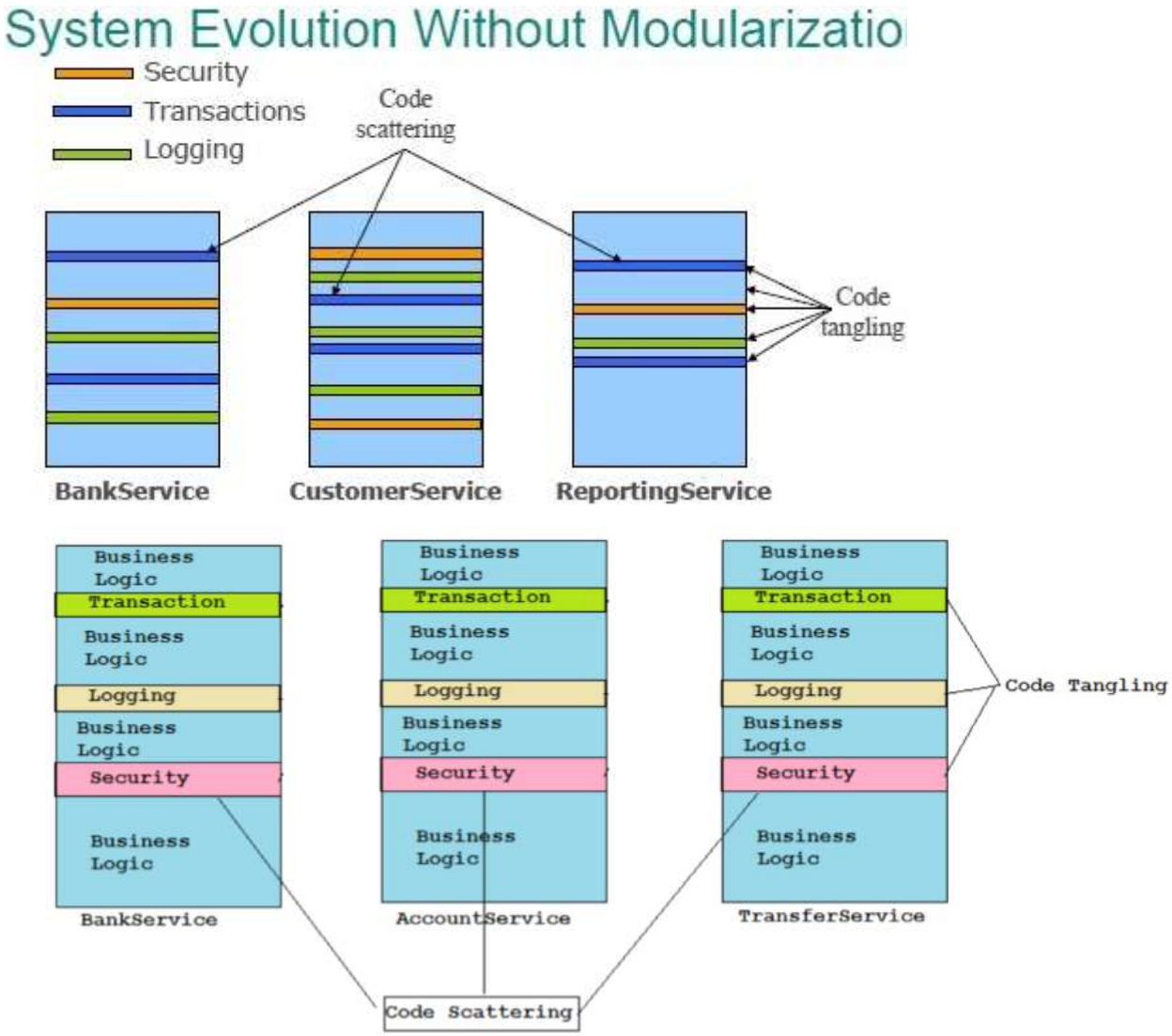
Лекция 7. Spring AOP

Spring AOP: принципы и основные понятия аспектно-ориентированного программирования.

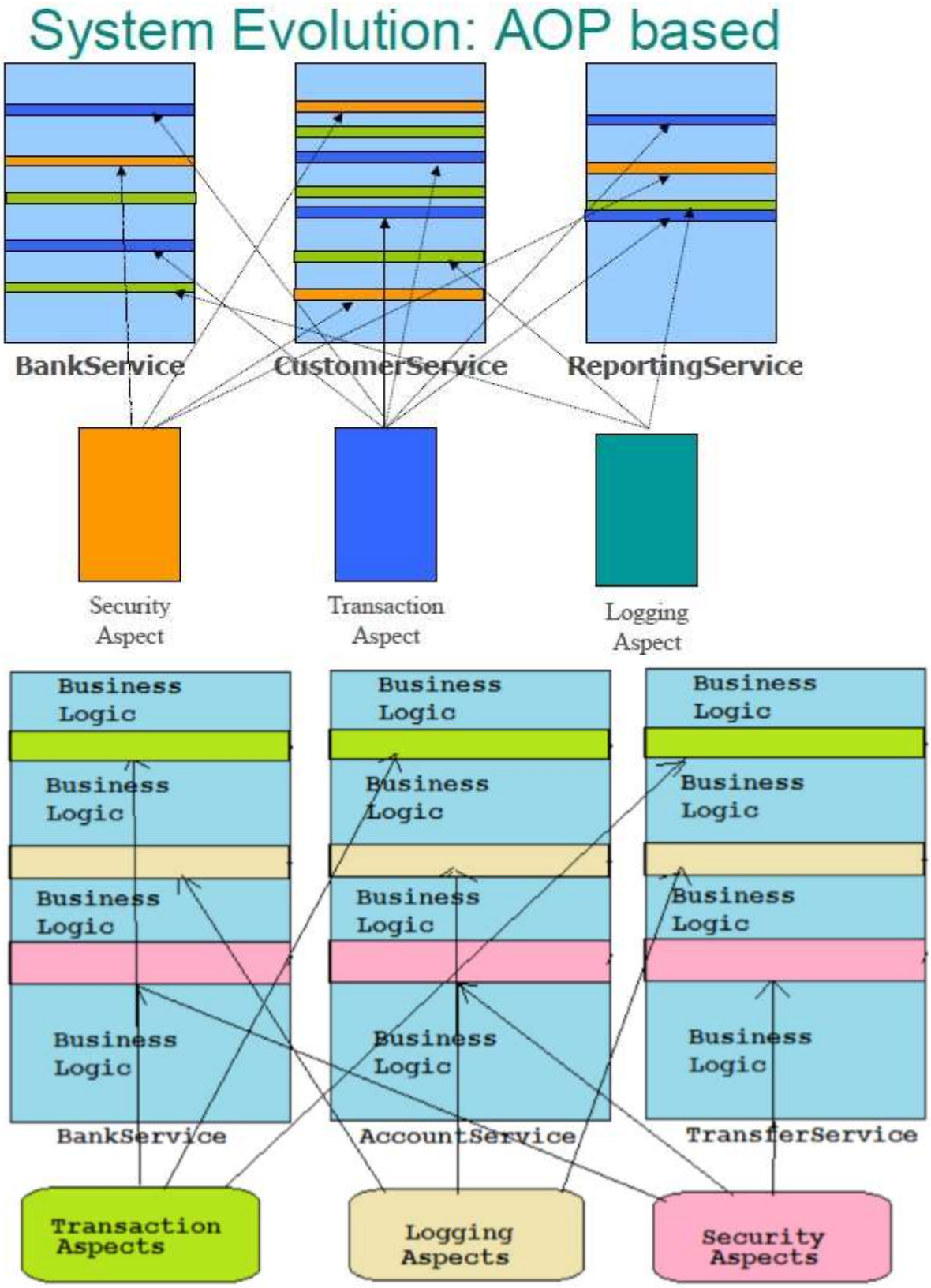
Введение

Система без AOP или модульности

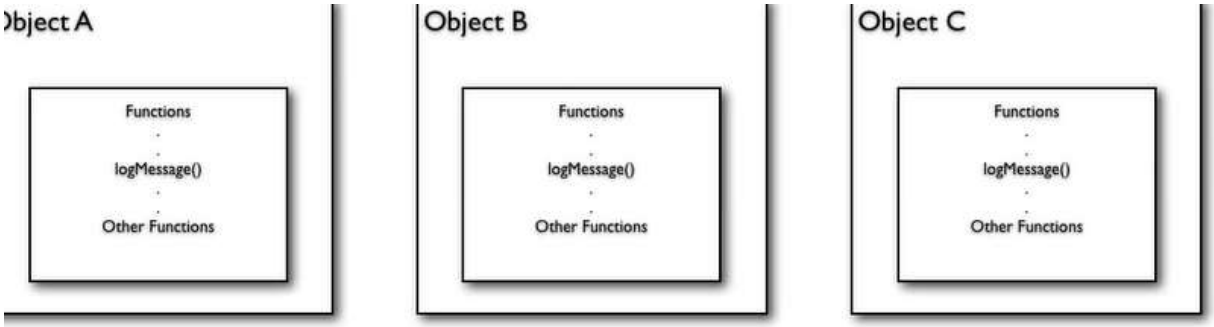


Есть две основные проблемы

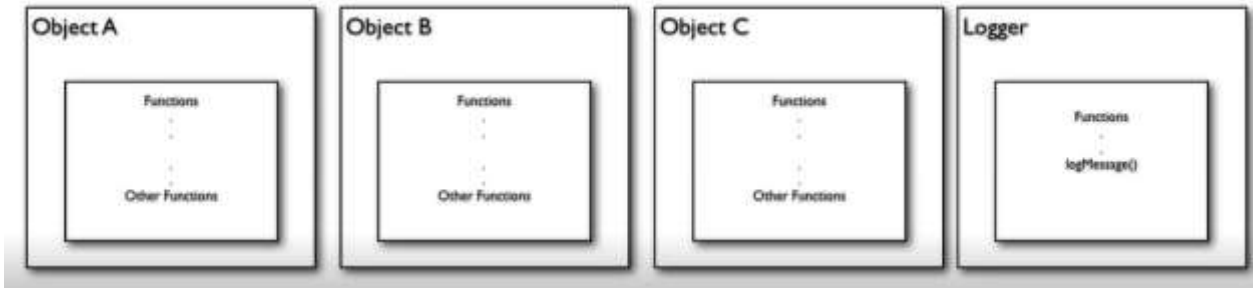
- Запутывание кода (tangling) (соединение проблем)
- Рассеяние кода (scattering) (проблема распространяется на разные модули)



Например надо писать лог.



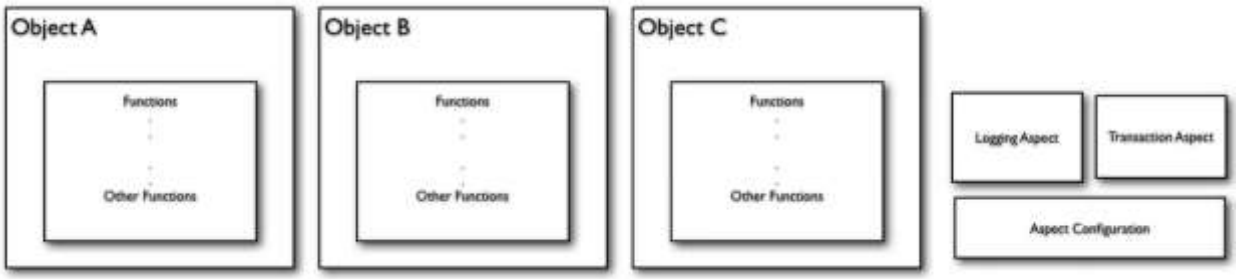
Чтобы избежать повторения можно сделать объект и вызывать из каждого класса. Если вынести эти функции в отдельные классы, то впоследствии все равно придется вызывать методы этого класса по нескольку раз.
Получается сильная связность.



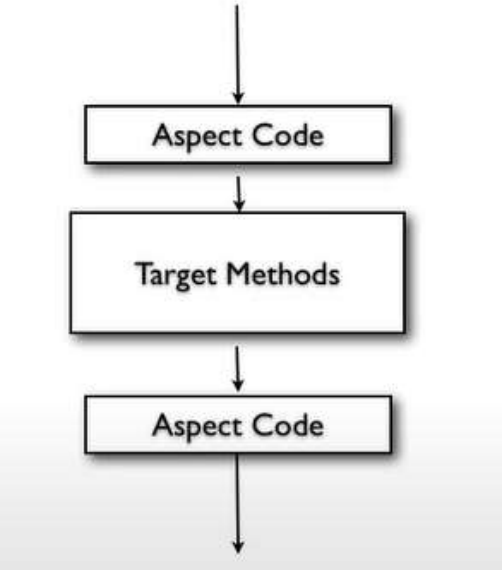
Но Logger не является важным типом. Он несет сквозную функциональность – это не бизнес объект. Проблемы:

1) Слишком много зависимостей с кросс функциональными объектами
2) Метод все равно надо вызывать
3) Нельзя все поменять за раз

Если использовать АОП. Определяются аспекты – классы. Но вместо вызова мы пишем конфигурацию аспектов, которая определяет какой метод в каком классе должен вызываться с использованием DI. Если нужны изменения – мы меняем конфигурацию или добавляем и это просто.



Например, пользуясь АОП, достаточно указать, что методы определенного класса должны вызываться до и после вызова каждого сервиса, вызываемого из приложения.



АОП дополняет ООП и дает гибкость.

Если вынести эти функции в отдельные классы, то впоследствии все равно придется вызывать методы этого класса по нескольку раз. А пользуясь АОП, достаточно указать, что методы определенного класса должны вызываться до и после вызова каждого сервиса, вызываемого из приложения.

Аспекто-ориентированном программировании (далее – АОП). АОП является одним из ключевых компонентов Spring. Смысл АОП заключается в том, что бизнес-логика приложения разбивается не на объекты, а на “отношения” (concerns).

Термином АОП нередко обозначают инструментальные средства для реализации сквозной функциональности (cross-cutting). Понятие сквозной функциональности имеет отношение к логике, которая не может быть отделена от остальной части приложения, что в конечном итоге приводит к дублированию кода и тесной связанности.

Или по другому можно сказать это общая функциональность, которая необходима во многих местах в вашем приложении

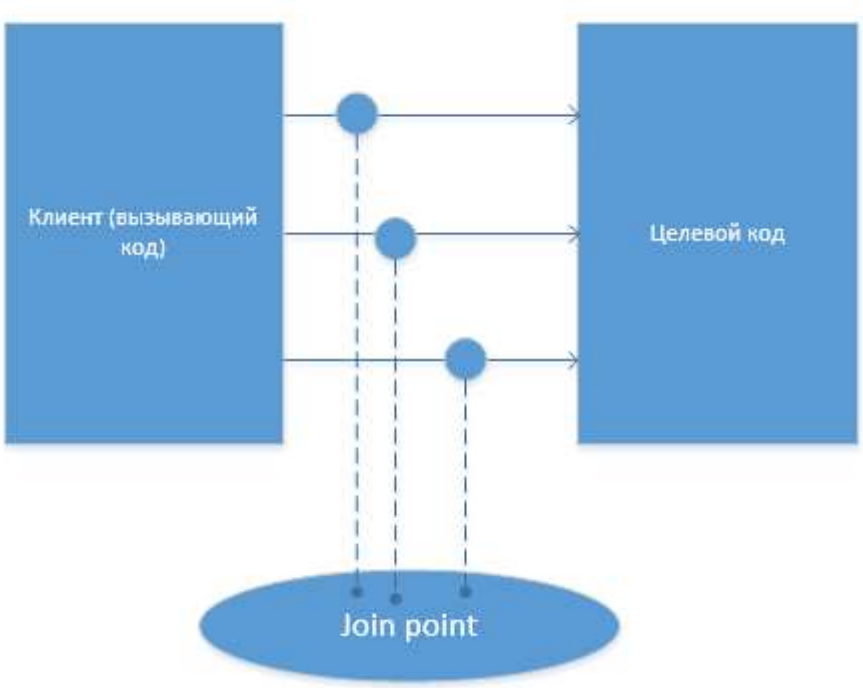
- Ведение журнала и трассировка
- Управление транзакциями
- Безопасность
- Кеширование
- Обработка ошибок
- Мониторинг производительности
- Пользовательские бизнес-правила

Основные понятия и терминология AOP

Ключевой единицей в ООП является объект, а ключевой единицей в АОП – аспект.

Точка соединения (Join point)- точка в приложении, где мы можем подключить аспект (как точка наблюдения). Другими словами, это место, где начинаются определённые действия модуля АОП в Spring.

Характерными примерами точек соединения служат вызов метода, обращение к методу, инициализация класса и получение экземпляра объекта.



Совет (Advice) - фактическое действие, которое должно быть предпринято до и/или после выполнения метода или набор инструкций выполняемых на точках среза (Pointcut). Это конкретный код, который вызывается во время выполнения программы.

Работа аспекта называется совет.

Существует несколько типов советов (advice):

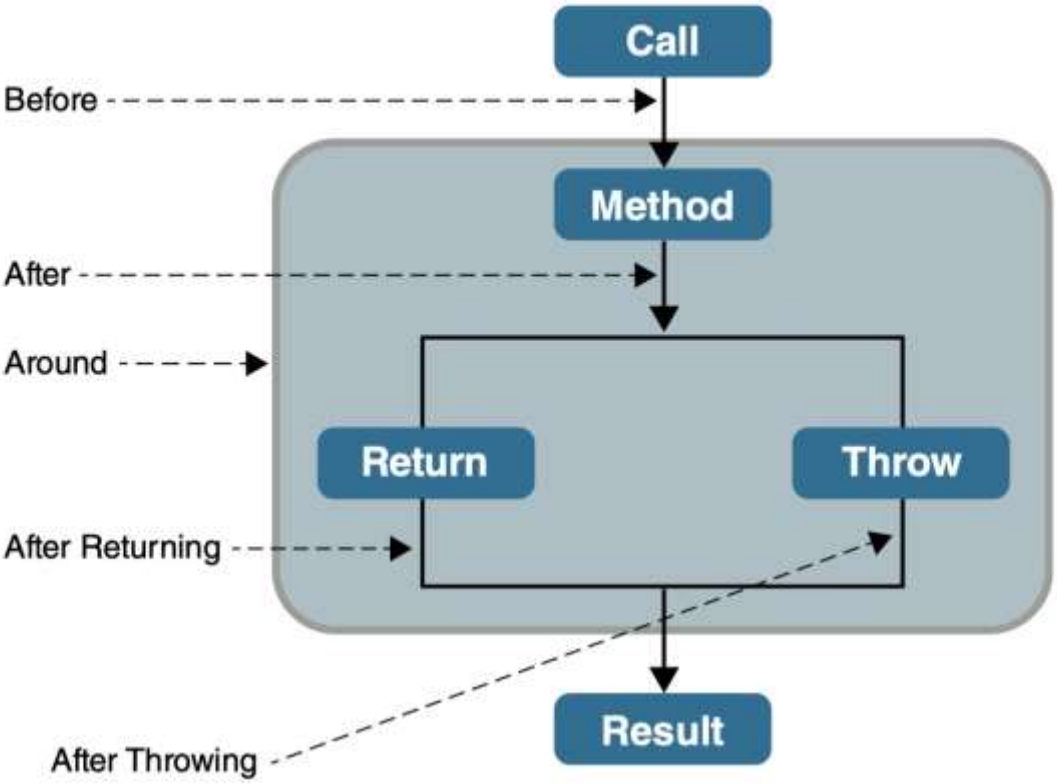
before - запускает совет перед выполнением метода. Подходит для контроля доступа, защиты и статистики

after - запускает совет после выполнения метода, независимо от результата его работы (кроме случая остановки работы JVM). Подходит для освобождения ресурсов

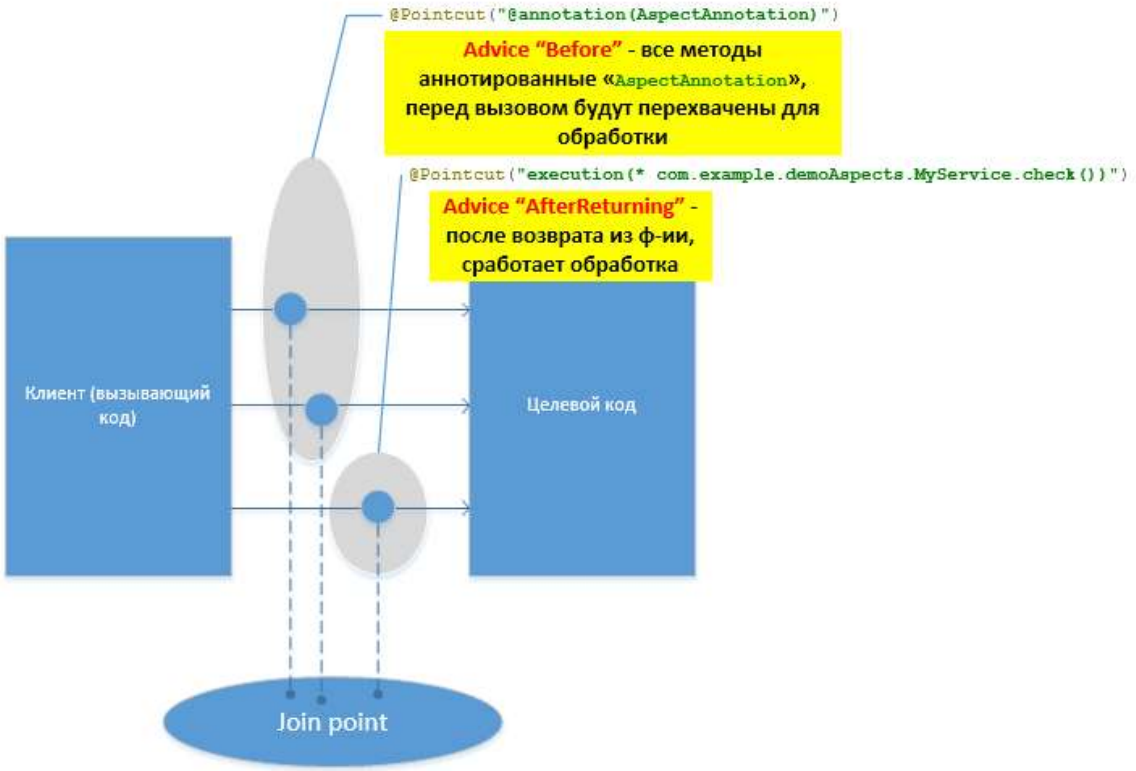
after-returning - запускает совет после выполнения метода, только в случае его успешного выполнения. Подходит для статистики и валидации

after-throwing - запускает совет после выполнения метода, только в случае, когда этот метод “бросает” исключение. Для обработки ошибок, рассылки сообщений с ошибками, восстановления после ошибок.

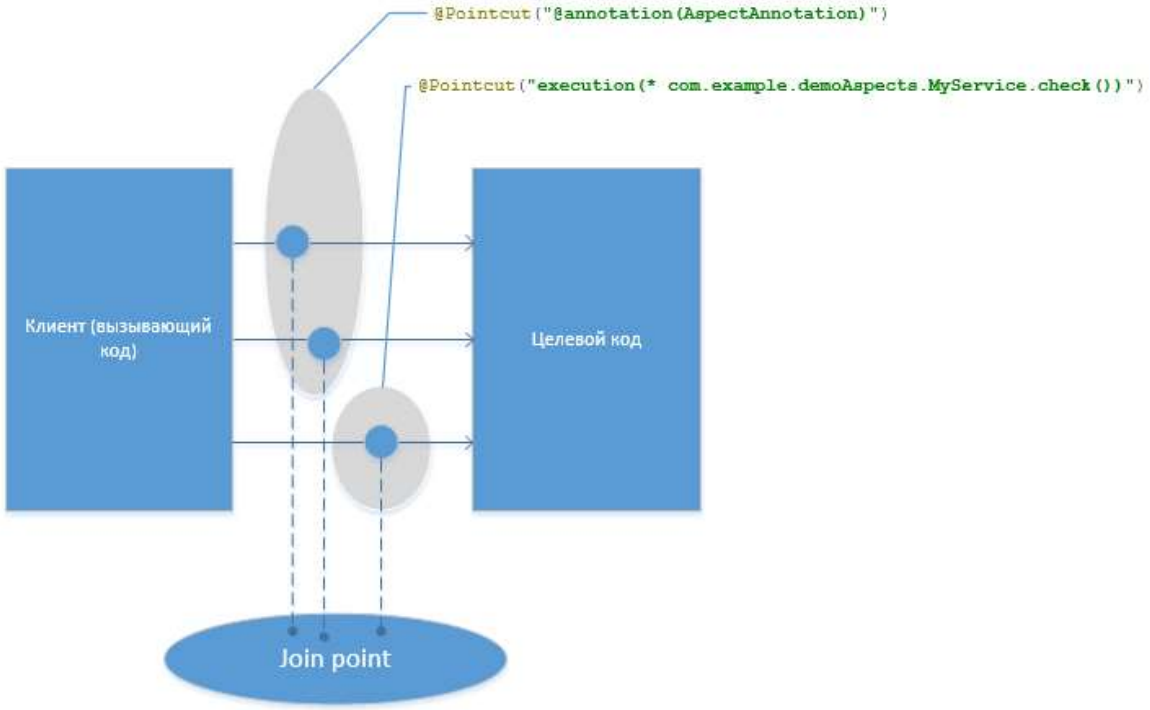
around - запускает совет до и после выполнения метода.




```java
@Aspect
public class Auditing {

    //Before transfer service
    @Before("execution(* TransferService.transfer(..))")
    public void validate() {
        System.out.println("bank validate your credentials before amount transferring");
    }

    //Before transfer service
    @Before("execution(* TransferService.transfer(..))")
    public void transferInstantiate() {
        System.out.println("bank instantiate your amount transferring");
    }

    //After transfer service
    @AfterReturning("execution(* TransferService.transfer(..))")
    public void success() {
        System.out.println("bank successfully transferred amount");
    }

    //After failed transfer service
    @AfterThrowing("execution(* TransferService.transfer(..))")
    public void rollback() {
        System.out.println("bank rolled back your transferred amount");
    }
```



Срез точек (Pointcut) - несколько объединённых точек (join points) или срез, в котором должен быть выполнен совет. Это предикат для идентификации точек.



Создавая срезы, можно приобрести очень точный контроль над тем, как применять совет к компонентам приложения. Типичной точкой

соединения служит вызов метода или же вызовы всех методов из отдельного класса.

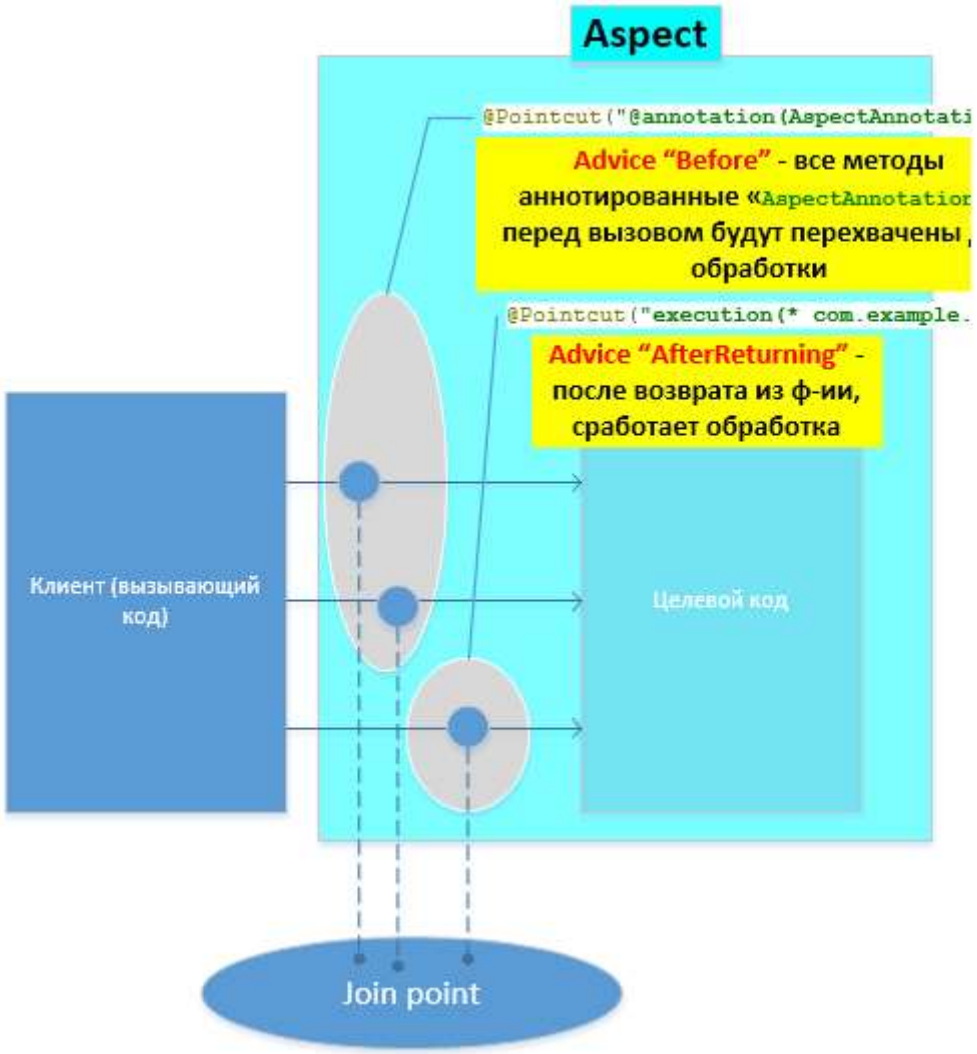
Целевой объект (Target object) - объект на который направлены один или несколько аспектов.

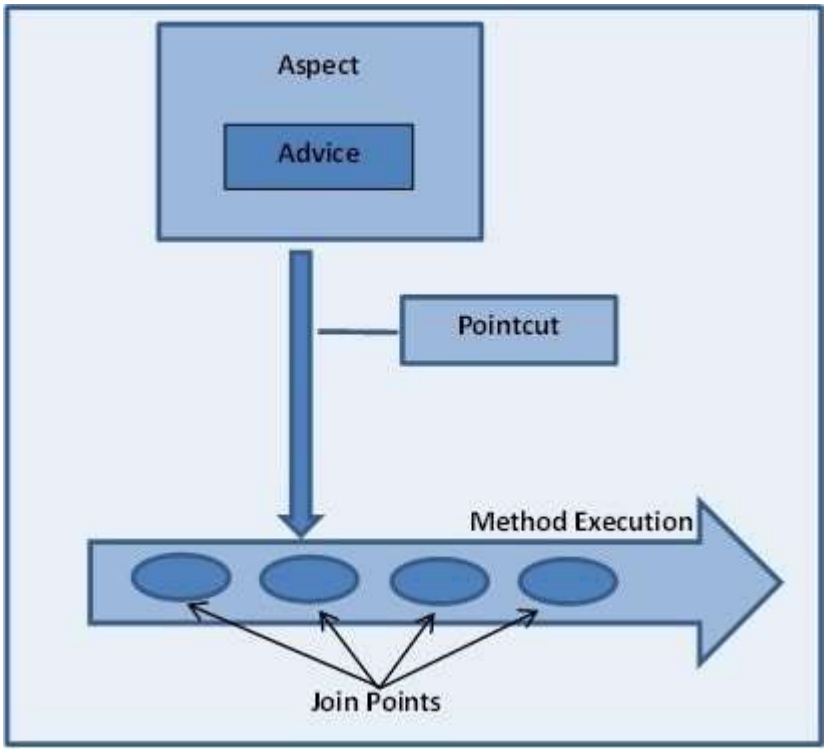
Зачастую целевой объект обозначается как снабженный советом объект.

Связывание или вплетение (Weaving) - процесс связывания аспектов с другими объектами приложения для создания совета. Может быть вызван во время компиляции, загрузки или выполнения приложения.

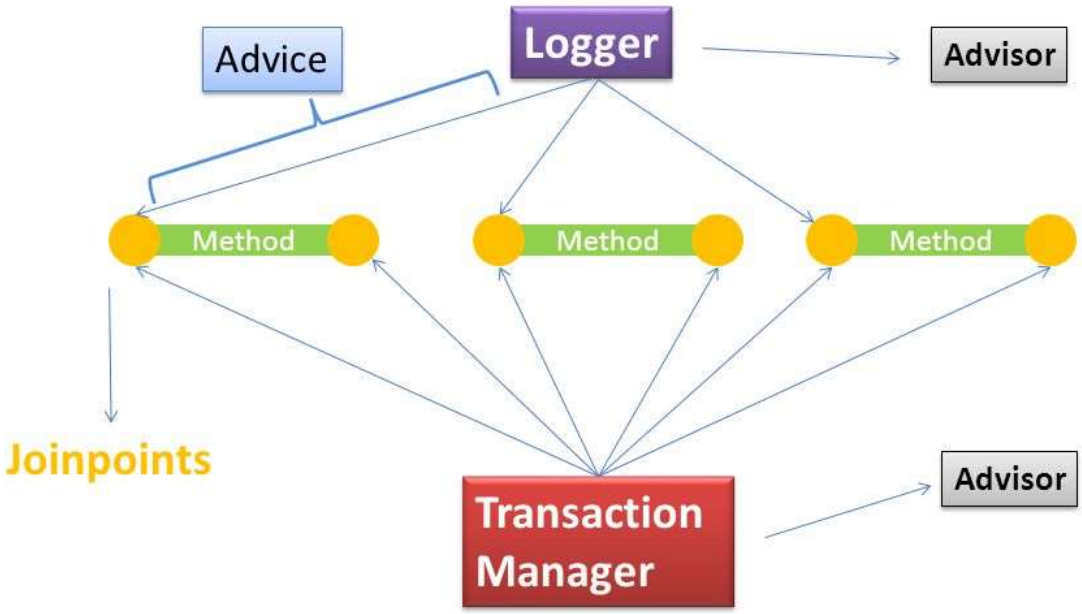
Аспект (Aspect) - модуль или класс, который имеет набор программных интерфейсов, которые обеспечивают сквозные требования (модуль в котором собраны описания Pointcut и Advice). К примеру, модуль логгирования будет вызывать АОП аспект для логгирования. В зависимости от требований, приложение может иметь любое количество аспектов.

Таким образом **Аспект** (aspect) - это сочетание совета и срезов, инкапсулированных в классе. Такое сочетание приводит в итоге к определению логики, которую следует внедрить в приложение, а также тех мест, где она должна выполняться.

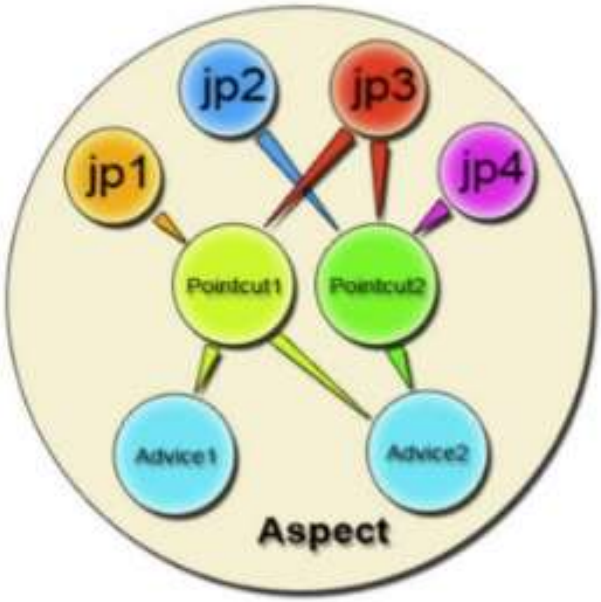

Aspect
@Pointcut("@annotation(AspectAnnotati
Advice "Before" - все методы аннотированные «AspectAnnotatior перед вызовом будут перехвачены , обработки
@Pointcut("execution(* com.example.
Advice "AfterReturning" - после возврата из ф-ии, сработает обработка
Клиент (вызывающий код)
Целевой код
Join point


Aspect
Advice
Pointcut
Method Execution
Join Points

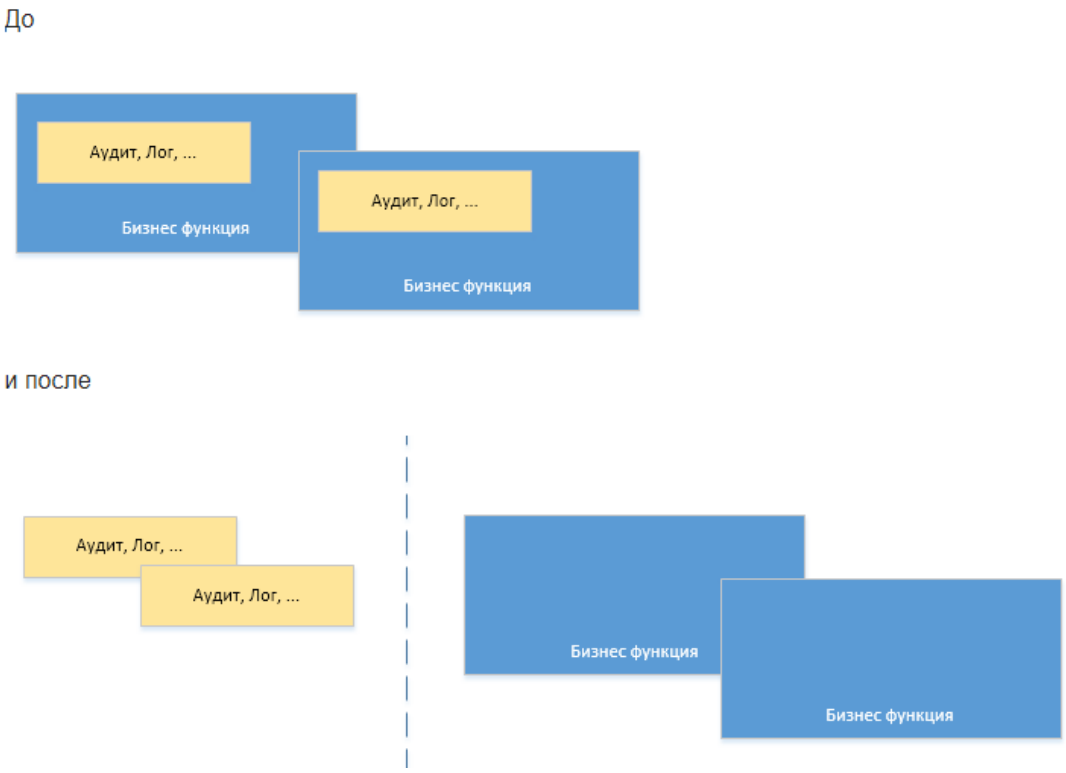
AOP – Definitions.



www.java9s.com



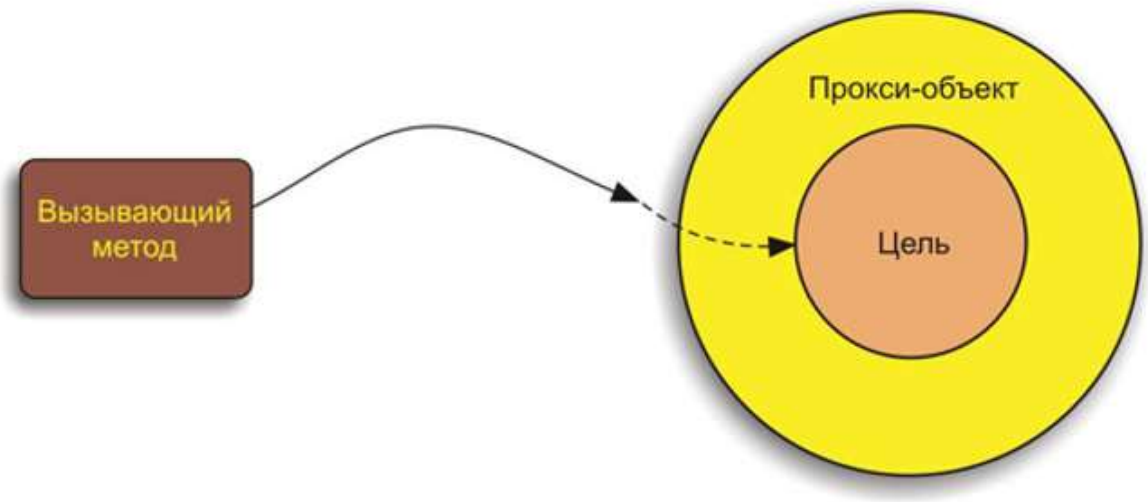
Архитектура АОП в Spring



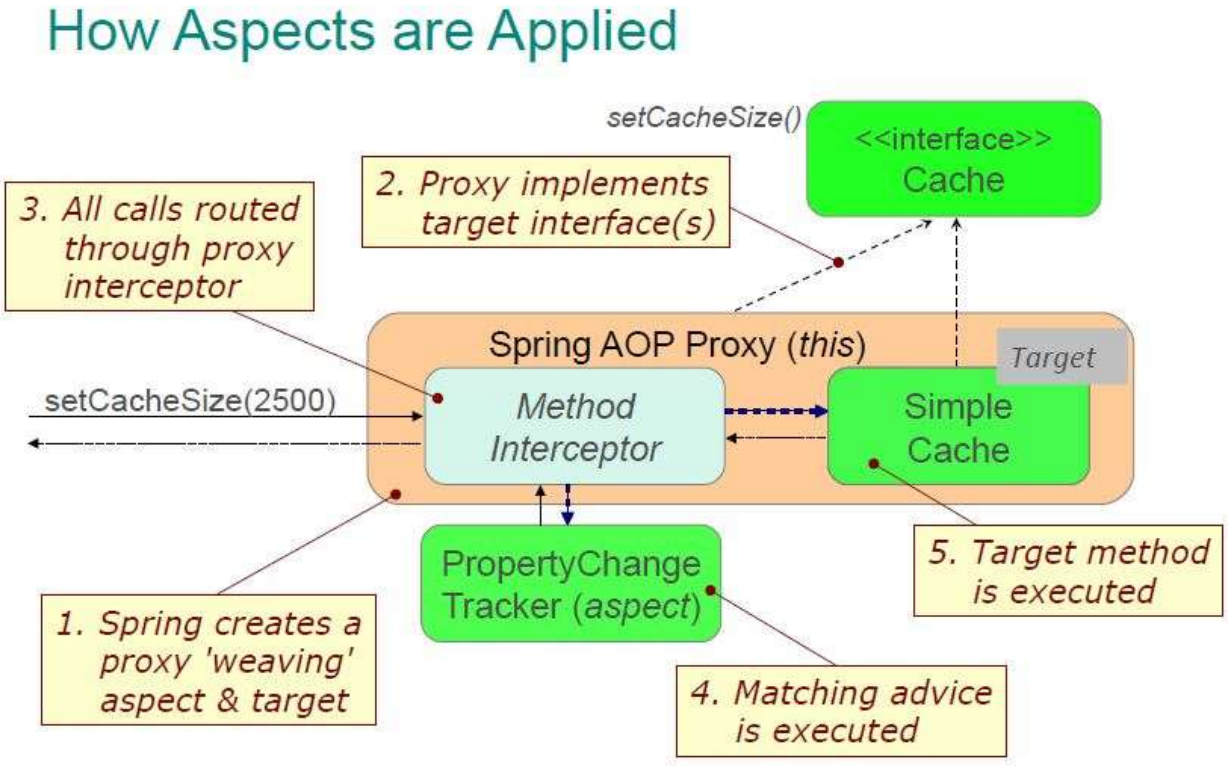
Модуль AOP в Spring обеспечивает нас такими сущностями, как “перехватчики” (interceptors) для перехвата приложения в определённые моменты. Например, когда выполняется определённый метод, мы можем добавить какую-то функциональность (к примеру, сделать запись в лог-файл приложения) как до, так и после выполнения метода.

В Spring поддерживаются 2 подхода для реализации АОП: **Основанный на XML** (применяется конфигурация с помощью конфигурационного XML-файла) и о**снованный на аннотациях @AspectJ** (применяется конфигурация с помощью аннотации)

Ключевая архитектура АОП в Spring основана на прокси. Когда вы хотите создать экземпляр класса, снабженный советом, то должны использовать класс ProxyFactory для создания прокси этого экземпляра, первым делом предоставив ProxyFactory со всеми аспектами, которые необходимо связать с прокси. Применение ProxyFactory - это чисто программный подход к созданию прокси АОП. По большей части использовать такой подход в своем приложении не обязательно; вместо этого можно положиться на механизмы декларативной конфигурации. Это класс ProxyFactory Bean, пространство имен aop и аннотации в стиле @AspectJ, которые обеспечат декларативное создание заместителей.

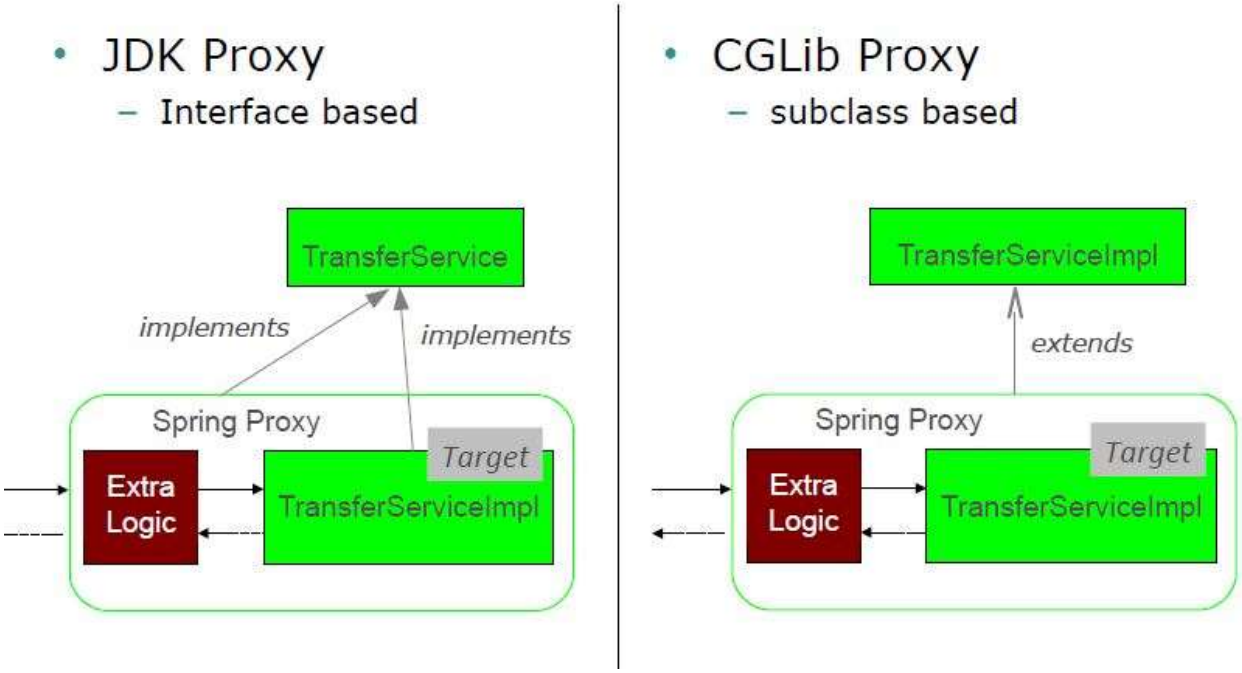


Во время выполнения платформа Spring анализирует сквозную функциональность, определенную для бинов в ApplicationContext, и динамически генерирует прокси-бины (которые являются оболочками для лежащих в основе целевых бинов). Вместо обращения к целевому бину напрямую вызывающие объекты внедряют прокси-бин. Прокси-бин затем анализирует текущие условия (т.е. точку соединения, срез или совет) и соответствующим образом связывает подходящий совет.



В самом каркасе Spring AOP поддерживаются две реализации proxy:

- динамические заместители JDK (**JDK Proxy**)
- заместители CGLIB (**CGLIB Proxy**).



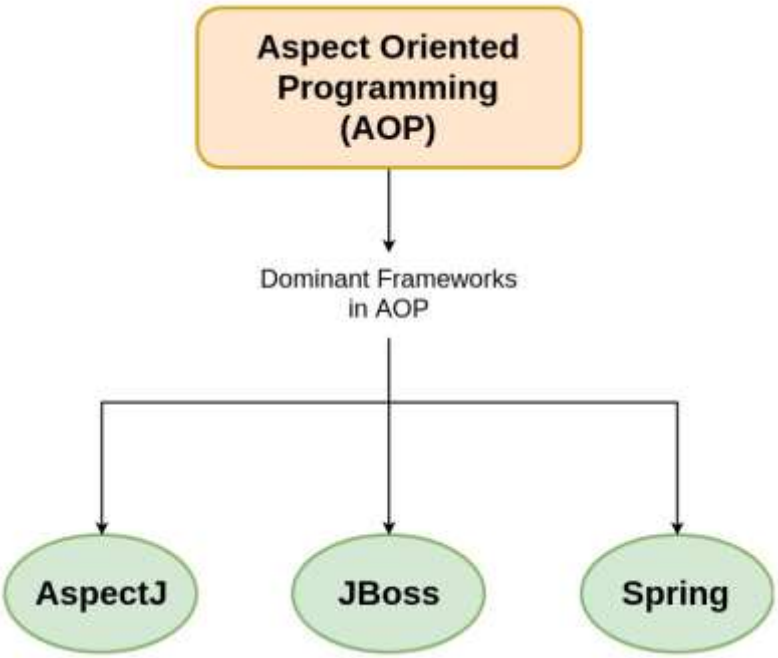
JDK Proxy: используется по умолчанию в Spring. Называется также динамическим прокси.Встроен в JDK, используется для интерфесов.

Есть интерфейс, который определяет метод объекта, который надо проксировать. Таким образом, конкретная реализация этого интерфейса обернута прокси. Поэтому, когда вызываем метод для своего объекта, вызываем его для его прокси. Вызов распознается перехватчиком метода, который в конечном итоге выполняет аспект, а затем выполняется вызванный метод.

CGLIB Proxy: прокси расширяет реализацию обернутого объекта, добавляя к нему дополнительные логические функции. Включен в Spring JAR. Это альтернативная реализация Spring AOP, чтобы использовать ее, мы должны добавить аннотацию @EnableAspectJAutoProxy (proxyTargetClass = true) в наш класс конфигурации.

По умолчанию, когда целевой объект, снабженный советом, реализует какой-нибудь интерфейс, для получения экземпляров заместителя целевого объекта в Spring будет использован динамический заместитель JDK. Но если целевой объект, снабженный советом, не реализует интерфейс (например, потому, что он представляет конкретный класс), то для получения экземпляров заместителей будет применяться библиотека CGLIB. И объясняется это в основном тем, что динамический заместитель JDK поддерживает создание заместителей только для интерфейсов.

AOP frameworks



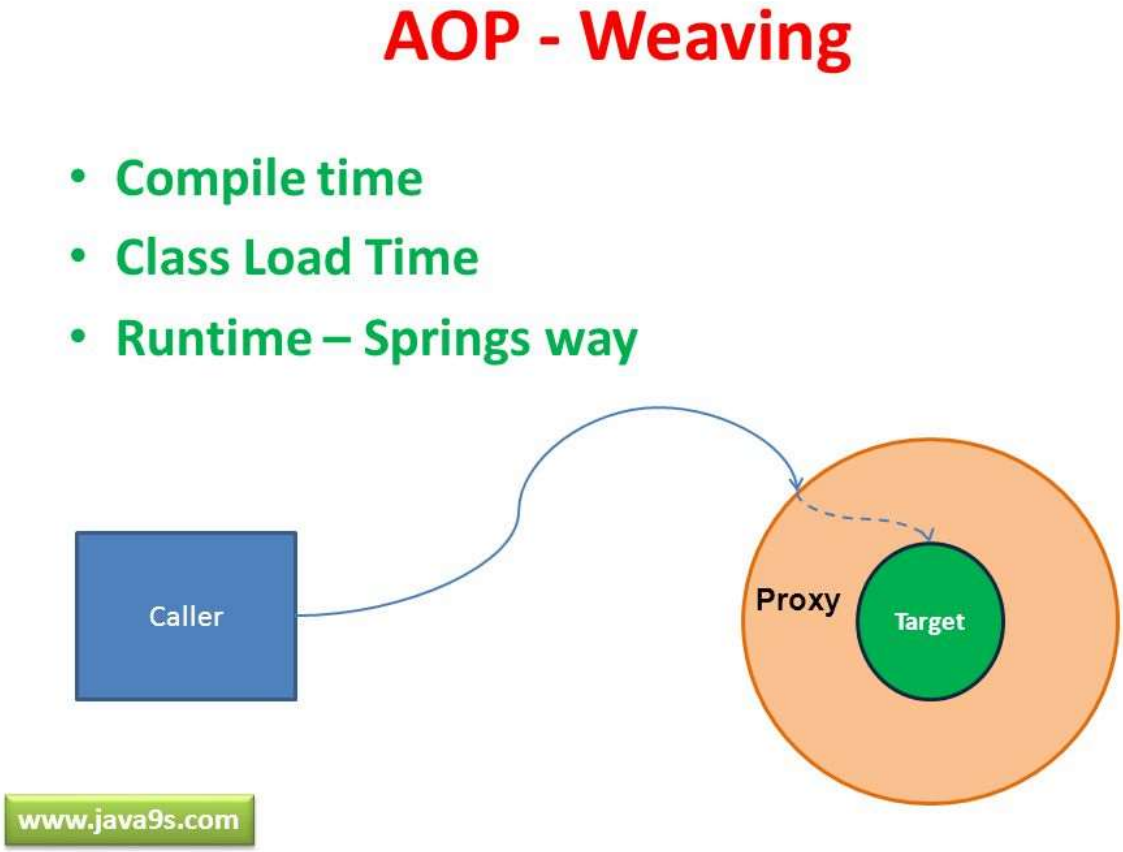
AspectJ - это простая в использовании и изучении Java-совместимая среда для интеграции сквозных реализаций. AspectJ был разработан в PARC. Сегодня это одна из известных сред AOP, благодаря своей простоте и возможности поддерживать модульность компонентов. Его можно использовать для применения AOP к статическим или нестатическим полям, конструкторам, методам, которые являются частными, общедоступными или защищенными.
Поддерживает следующие виды связывания:

Compile-time weaving: компилятор AspectJ принимает в качестве входных данных как исходный код нашего аспекта, так и нашего приложения, а в качестве выходных данных создает файлы классов.

Post-compile weaving: известно как бинарное плетение. Он используется для объединения существующих файлов классов и файлов JAR с нашими аспектами.

Load-time weaving: точно так же, как и в предыдущем переплетении, с той разницей, что переплетение откладывается до тех пор, пока загрузчик классов не загрузит файлы классов в JVM. Т.е. перехватывается базовый загрузчик классов виртуальной машины JVM и обеспечивается связывание с байт-кодом, когда загрузчик классов загружает его.

В Spring – динамические связывание.



Применяемый в Spring подход состоит в создании заместителя для всех целевых объектов, что дает возможность вызывать совет по мере надобности. Недостаток динамического АОП заключается в том, что оно обычно не выполняется так же хорошо, как и статическое АОП. А главным преимуществом динамических реализаций АОП является простота, с которой можно изменить целый ряд аспектов в приложении, не прибегая к повторной компиляции основного прикладного кода.

AspectWertz - еще одна легковесная мощная среда, совместимая с Java. Его можно легко использовать для интеграции как в новые, так и в существующие приложения. AspectWertz поддерживает как XML, так и написание и настройку аспектов на основе аннотаций. Начиная с AspectJ5, он был объединен с AspectJ.

JBoss AOP - поддерживает написание аспектов и целевых объектов динамического прокси. Его можно использовать для применения AOP к статическим или нестатическим полям, конструкторам, методам, которые являются частными, общедоступными или защищенными с помощью перехватчиков.

Dynaop - является фреймворком AOP на основе прокси.

CAESAR - это AOP-инфраструктура, совместимая с Java. Он поддерживает реализацию абстрактных компонентов, а также их интеграцию. -

Spring AOP - Это Java-совместимый простой в использовании фреймворк, который используется для интеграции AOP в Spring Framework. Это основанный на прокси фреймворк, который можно использовать при выполнении метода.
Но есть несколько ограничений, когда Spring AOP не может быть применен:

- Spring AOP не может быть применен к полям
- не можем применять другой аспект к существующему аспекту
- private и protected методы не могут быть снабжены советами
- Конструкторы не могут иметь советы

Spring AOP и AspectJ

У Spring AOP и AspectJ разные цели.

Spring AOP стремится обеспечить простую реализацию AOP в Spring IoC. Он не предназначен для использования в качестве полного решения AOP - его можно применять только к bean-компонентам, управляемым контейнером Spring.

С другой стороны, AspectJ является оригинальной технологией AOP, которая направлена на предоставление полного решения AOP. Он более надежен, но и значительно сложнее, чем Spring AOP. AspectJ может применяться ко всем объектам домена.

Spring поддерживает интеграцию AspectJ и Spring AOP. Но:

- Spring AOP основан на динамическом прокси, который поддерживает только точки соединения методов, но AspectJ может применяться к полям, конструкторам, даже если они являются private, public or protected.
- Spring AOP нельзя использовать для метода, который вызывает методы того же класса или который является статическим или final, но AspectJ может.
- AspectJ не нужен контейнер Spring для управления компонентом, в то время как Spring AOP может использоваться только с компонентами, которые управляются контейнером Spring.
- Spring AOP поддерживает связывание во время выполнения на основе шаблона прокси, а AspectJ поддерживает связывание во время компиляции, которое не требует создания прокси.
- Spring AOP легко реализовать, аннотируя класс с помощью аннотации @Aspect или с помощью простой конфигурации. Но чтобы использовать AspectJ, нужно создавать файлы * .aj.

Joinpoint	Spring AOP Supported	AspectJ Supported
Method Call	No	Yes
Method Execution	Yes	Yes
Constructor Call	No	Yes
Constructor Execution	No	Yes
Static initializer execution	No	Yes
Object initialization	No	Yes
Field reference	No	Yes
Field assignment	No	Yes
Handler execution	No	Yes
Advice execution	No	Yes

Простой класс без final, static методов - Spring AOP, в противном случае - для написания аспектов AspectJ.

Краткое резюме

Spring AOP	AspectJ
Implemented in pure Java	Implemented using extensions of Java programming language
No need for separate compilation process	Needs AspectJ compiler (ajc) unless LTW is set up
Only runtime weaving is available	Runtime weaving is not available. Supports compile-time, post-compile, and load-time Weaving
Less Powerful – only supports method level weaving	More Powerful – can weave fields, methods, constructors, static initializers, final class/methods, etc...
Can only be implemented on beans managed by Spring container	Can be implemented on all domain objects
Supports only method execution pointcuts	Support all pointcuts
Proxies are created of targeted objects, and aspects are applied on these proxies	Aspects are weaved directly into code before application is executed (before runtime)
Much slower than AspectJ	Better Performance
Easy to learn and apply	Comparatively more complicated than Spring AOP

Пример

Перед тем, как погрузиться в детали реализации АОП в Spring, давайте обратимся к примеру. Мы возьмем простой класс, выводящий сообщение "World", и с применением АОП трансформируем экземпляр этого класса во время выполнения, чтобы он выводил сообщение "Hello World!". Базовый класс MessageWriter

```
public class MessageWriter {
    public void writeMessage() {
        System.out.print("World");
```

```
    }
}
```

Имея реализованный метод вывода сообщения, давайте добавим к этому классу совет, чтобы writeMessage () взамен выводил "Hello World!". Чтобы сделать это, нам необходимо перед выполнением существующего тела метода выполнить один код (для вывода строки "Hello "), а после выполнения тела - другой код (для вывода "!"). В терминах АОП нам требуется совет вокруг, который выполняется вокруг точки соединения. В данном случае точкой соединения является вызов метода writeMessage ().

```
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

public class MessageDecorator implements MethodInterceptor {
    public Object invoke(MethodInvocation invocation) throws
Throwable {
        System.out.print("Hello ");
        Object retVal = invocation.proceed();
        System.out.println("!");
        return retVal;
    }
}
```

Интерфейс Methodinterceptor - это стандартный интерфейс Альянса АОП для реализации совета "вокруг" для точек соединения вызовов методов. Объект Methodinvocation представляет вызов метода, снабжаемый советом, и с помощью этого объекта мы управляем тем, когда вызову метода разрешено продолжаться. Поскольку это совет "вокруг", мы способны выполнять действия перед вызовом метода и после его вызова, но до того, как произойдет возврат из метода.

Финальный шаг этого примера заключается в связывании совета MessageDecorator (а точнее - метода invoke ()) с кодом. Для этого мы создаем экземпляр MessageWriter, т.е. **цель**, и затем создаем прокси этого экземпляра, инструктируя фабрику прокси относительно связывания с советом MessageDecorator.

```
import org.springframework.aop.framework.ProxyFactory;

public class HelloWorldAOPExample {
    public static void main(String[] args) {
        MessageWriter target = new MessageWriter();

        ProxyFactory pf = new ProxyFactory();
        pf.addAdvice(new MessageDecorator());
        pf.setTarget(target);

        MessageWriter proxy = (MessageWriter) pf.getProxy();

        target.writeMessage();
        System.out.println("");
```

```
        proxy.writeMessage();
    }
}
```

Важным моментом, который следует отметить в коде, является использование класса ProxyFactory для создания прокси целевого объекта и одновременного его связывания с советом. Совет MessageDecorator передается в ProxyFactory с помощью вызова addAdvice (), а цель связывания указывается посредством вызова setTarget (). После того, как цель установлена, и некоторый совет добавлен к ProxyFactory, с помощью вызова getProxy () мы генерируем прокси. Наконец, мы вызываем writeMessage () на исходном целевом объекте и на прокси-объекте. Выполнение кода дает в результате следующий вывод:

World

Hello World !

Как видите, вызов метода writeMessage () на незатронутом целевом объекте приводит к стандартному обращению без выдачи на консоль дополнительной информации. Однако при вызове прокси выполняется код в MessageDecorator, генерируя желаемый вывод сообщения "Hello World!".

В приведенном примере целевой класс не имеет никаких зависимостей от Spring или от интерфейсов Альянса АОП.
Советом можно снабдить почти любой класс, даже если этот класс создавался без учета АОП.

Конфигурации Spring AOP

Есть три способа объявления конфигурации Spring AOP

- ProxyFactoryBean,
  namespace AOP,
- аннотации в стиле @ AspectJ.

1. Выберите сквозную задачу, которая будет реализована
2. Напишите аспект.
3. Зарегистрируйте аспект как компонент в контексте Spring.
4. Напишите конфигурацию аспекта

Определение срезов

Основная структура выражения pointcut состоит из двух частей:

- обозначение точки
- шаблон, который выбирает точки соединения определенного указателем pointcut.

как вы помните Spring AOP поддерживает только точки соединения выполнения метода для bean-компонентов, объявленных в своем контейнере IoC. В противном будеь IllegalArgumentException.

Указатель AspectJ	Описание
`args()`	Ограничивает срез точек сопряжения вызовами методов, чьи аргументы являются экземплярами указанных типов
`@args()`	Ограничивает срез точек сопряжения вызовами методов, чьи аргументы аннотированы указанными типами аннотаций
`execution()`	Соответствует точкам сопряжения, которые являются вызовами методов
`this()`	Ограничивает срез точек сопряжений точками, где ссылка на компонент является ссылкой на прокси-объект указанного типа
`target()`	Ограничивает срез точек сопряжений точками, где целевой объект имеет указанный тип
`@target()`	Ограничивает срез точек сопряжений точками, где класс выполняемого объекта снабжен аннотацией указанного типа
`within()`	Ограничивает срез точек сопряжений точками только внутри указанных типов
`@within()`	Ограничивает срез точек сопряжений точками внутри указанных типов, снабженных указанной аннотацией (в Spring AOP соответствует вызовам методов в указанном типе, отмеченных указанной аннотацией)
`@annotation`	Ограничивает срез точек сопряжений точками, помеченными указанной аннотацией

Использование сигнатуры метода

Сигнатура метода может использоваться для определения точек на основе доступных точек соединения с использованием следующего синтаксиса:

```
execution( [scope] [ReturnType] [FullClassName].[MethodName]
([Arguments]) throws [ExceptionType])
```

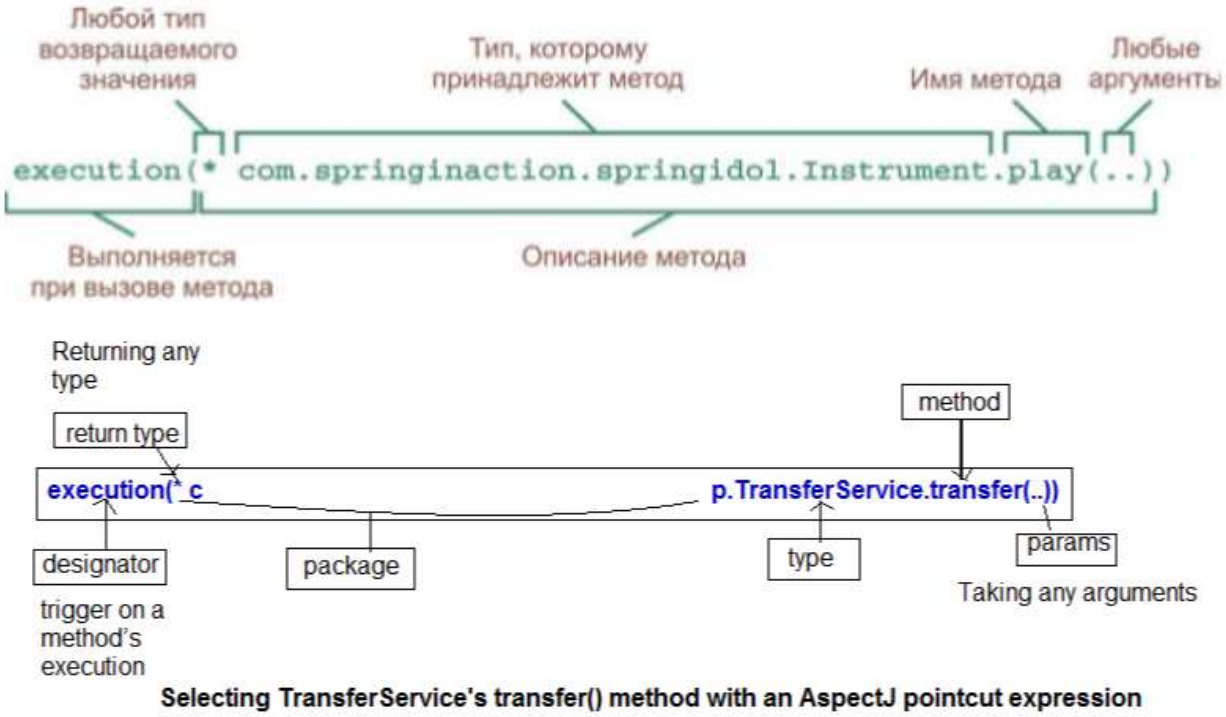
Java поддерживает private, public, protected и default как область действия метода, но Spring AOP поддерживает только public методы при написании выражений pointcut.
Список параметров используется, чтобы указать, какие типы данных будут учитываться при сопоставлении сигнатуры метода. Можно использовать две точки (..). Например:

expression(com.package.MyClass.*(..)*
все методы MyClass из пакета com.package

expression(public int com.packag.MyClass.(..)*
все методы возвращающие integer из MyClass с com.package

expression(public int com.package.MyClass.(int,..)*

expression(MyClass.*(..)*
все методы любой сигнатуры из MyClass



Selecting TransferService's transfer() method with an AspectJ pointcut expression

Использование типа

Для фильтрации методов по их типам - таким как интерфейсы, имена классов или имена пакетов. Мы можем использовать следующий синтаксис для указания типа:

```
within(<type name>)
```

тип может быть пакетом или именем класса

within(com.pack.)*
все методы класса из указанного пакета

within(com.pack..)*
все методы класса из указанного пакета и его подпакетов

within(com.pack.MyClass)

within(MyInterface+)
все методы всех классов реализующие интерфейс MyInterface**.**

Аналогично, чтобы объединить указатели логической операцией «ИЛИ», можно было бы использовать оператор || . А чтобы инвертировать смысл указателя – использовать оператор !. Поскольку в языке разметки XML амперсанды имеют специальное значение, при определении срезов множества точек сопряжения в конфигурационном XML-файле вместо оператора && можно использовать оператор and. Аналогично можно использовать операторы or и not вместо || и ! (соответственно).

Использование bean

Указатель bean() принимает идентификатор или имя компонента в виде аргумента и ограничивает срез множества точек сопряжения, оставляя в нем только точки, соответствующие указанному компоненту.

bean(name_of_bean)

*bean(*Component)*

определяет совпадающие точки соединения, принадлежащие компоненту, имя которого заканчивается на *Component*. Выражение нельзя использовать с аннотациями AspectJ.

Target

Цель используется для сопоставления точек соединения, где целевой объект является интерфейсом указанного типа. Он используется, когда Spring AOP использует создание прокси на основе JDK. Цель используется только в том случае, если целевой объект реализует интерфейс.

```
package com.pack;
class MyClass implements MyInterface{
 // method declaration
}
```

target(com.pack.MyInterface)
или
this(com.pack.MyClass)

Среди поддерживаемых указателей только execution фактически выполняет сопоставление – все остальные используются для ограничения множества совпадений. Это означает, что execution является основным указателем, который должен использоваться во всех определениях срезов множества точек сопряжения. Остальные указатели применяются только для ограничения точек сопряжения в срезе.

Подстановочные знаки : + * ..
.. Этот подстановочный знак соответствует любому количеству аргументов в определениях методов и соответствует любому количеству пакетов в определениях классов.
+ соответствует любым подклассам данного класса.
* соответствует любому количеству символов.

Объявление аспектов в XML

Элемент настройки AOP	Назначение
<aop:advisor>	Определяет объект-советник
<aop:after>	Определяет AOP-совет, выполняемый после вызова метода (независимо от успешности его завершения)
<aop:after-returning>	Определяет AOP-совет, выполняемый после успешного выполнения метода
<aop:after-throwing>	Определяет AOP-совет, выполняемый после возбуждения исключения
<aop:around>	Определяет AOP-совет, выполняемый до и после выполнения метода
<aop:aspect>	Определяет аспект
<aop:aspectj-autoproxy>	Включает поддержку аспектов, управляемых аннотациями, созданными с применением аннотации @AspectJ
<aop:before>	Определяет AOP-совет, выполняемый до выполнения метода

<aop:config>	Элемент верхнего уровня настройки механизма AOP
<aop:declare-parents>	Внедряет в объекты прозрачную реализацию дополнительных интерфейсов
<aop:pointcut>	Определяет срез точек сопряжения

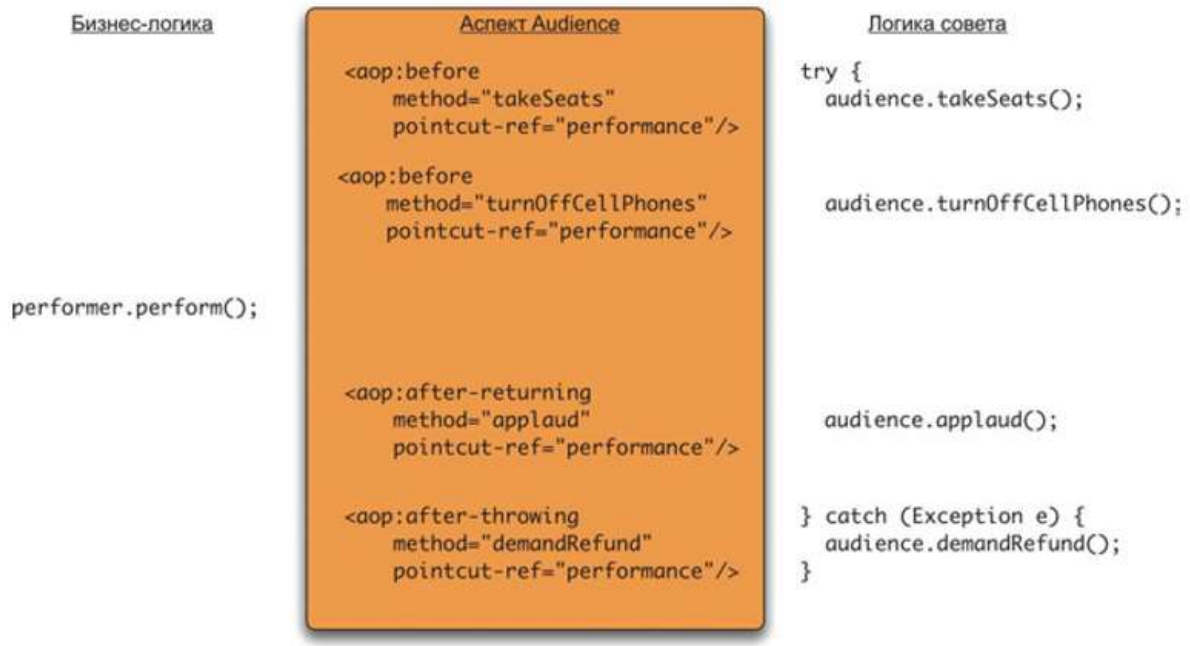
Пусть есть класс аспекта

```
public class Audience {
    public void takeSeats() { // Перед выступлением
        System.out.println("The audience is taking their
seats.");
    }
    public void turnOffCellPhones() { // Перед выступлением
        System.out.println("The audience is turning off their
cellphones");
    }
    public void applaud() { // После выступления
        System.out.println("CLAP CLAP CLAP CLAP CLAP");
    }
    public void demandRefund() { // После неудачного выступления
        System.out.println("Boo! We want our money back!");
    }
}
```

Регистрируем в виде компонента в контексте приложения Spring

<bean id="audience" class="com.springinaction.springidol.Audience" />

Логику советов вплетаем в основную логику работы приложения



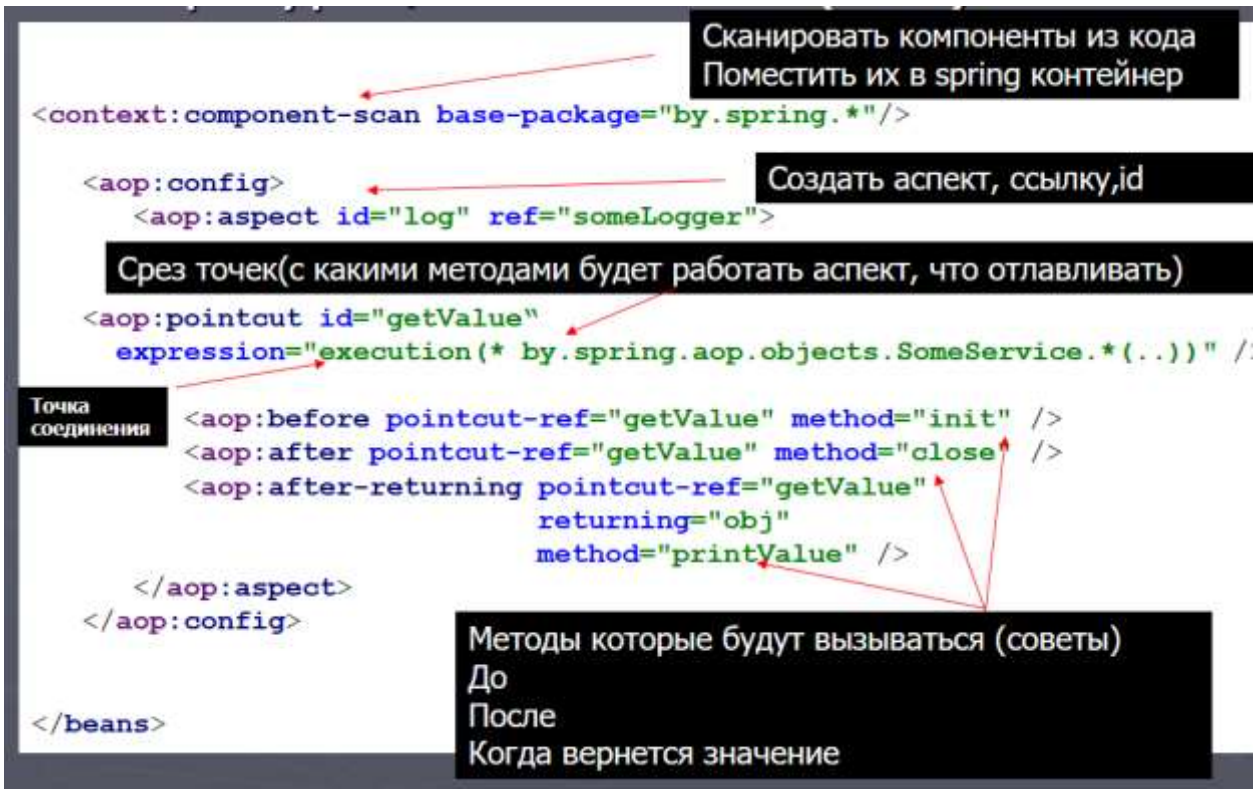
Делаем аспект

```
<aop:config>
    <aop:aspect ref="audience">
    <!-- Ссылка на компонент audience -->
    <aop:before pointcut=   "execution(*
com.springinaction.springidol.Performer.perform(..))"
                method="takeSeats" />
  <!-- Перед выступлением -->
    <aop:before pointcut=   "execution(*
com.springinaction.springidol.Performer.perform(..))"
                method="turnOffCellPhones" />
 <!-- Перед выступлением -->
    <aop:after-returning pointcut=   "execution(*
com.springinaction.springidol.Performer.perform(..))"
                         method="applaud" />
   <!-- После выступления -->
    <aop:after-throwing pointcut=   "execution(*
com.springinaction.springidol.Performer.perform(..))"
                        method="demandRefund" />
  <!-- После неудачного выступления -->
</aop:aspect>
</aop:config>
```

Определение именованного среза множества точек (исключение дублирования)

```
<aop:config>
    <aop:aspect ref="audience">
    <aop:pointcut id="performance" expression=
            "execution(*
com.springinaction.springidol.Performer.perform(..))"        />
        <!-- Определение среза множества точек сопряжения -->
    <aop:before             pointcut-ref="performance"
                method="takeSeats" />

    <aop:before             pointcut-ref="performance"
                    method="turnOffCellPhones" />

    <aop:after-returning   pointcut-ref="performance"
                    method="applaud" />

    <aop:after-throwing    pointcut-ref="performance"
                    method="demandRefund" />
</aop:aspect>
</aop:config>
```

Определение аспекта и советов – конфигурация контекста (XML)



Пример

Рассмотрим на примере нашего проекта.

У нас есть следующий контроллер. Как видите здесь использовалось логгирование.

```
@Slf4j
@RestController
@RequestMapping
public class PersonController {


    private  final PersonService personService;

    @Value("${welcome.message}")
    private String message;

    @Value("${error.message}")
    private String errorMessage;

    @Autowired
    public PersonController(PersonService personService) {
        this.personService = personService;
      //  this.personRepository = personRepository;
    }

    @GetMapping(value = {"/", "/index"})
    public ModelAndView index(Model model) {
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.setViewName("index");
        model.addAttribute("message", message);
```

```
        log.info("index was called");
        return modelAndView;
    }

    @GetMapping(value = {"/personList"})
    public ModelAndView personList(Model model) {
        List<Person> persons = personService.getAllPerson();
        log.info("person List" + persons);
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.setViewName("personList");
        model.addAttribute("persons", persons);
        log.info("/personList was called");
        return modelAndView;
    }


    @GetMapping(value = {"/addPerson"})
    public  ModelAndView showAddPersonPage(Model model) {
        ModelAndView modelAndView = new ModelAndView("addPerson");
        NewPersonDto personForm = new NewPersonDto();
        model.addAttribute("personForm", personForm);
        log.info("/addPerson - GET  was called" + personForm);
        return modelAndView;
    }
  //  @PostMapping("/addPerson")
    //GetMapping("/")
    @PostMapping(value = {"/addPerson"})
    public ModelAndView savePerson(Model model, //
                                   @Valid @ModelAttribute("personForm")
NewPersonDto personDto,
                                   Errors errors) {
        ModelAndView modelAndView = new ModelAndView();
        log.info("/addPerson - POST  was called" + personDto);
       if (errors.hasErrors()) {
            modelAndView.setViewName("addPerson");
        }
       else {
            modelAndView.setViewName("personList");
            Long id = personDto.getPersonId();
            String firstName = personDto.getFirstName();
            String lastName = personDto.getLastName();
            String street = personDto.getStreet();
            String city = personDto.getCity();
            String zip = personDto.getZip();
            String email = personDto.getEmail();
            Date birthday = (Date)personDto.getBirthday();
            String phone = personDto.getPhone();

            Person newPerson = new Person(id, firstName, lastName, street,
city, zip, email, birthday, phone);
             personService.addNewPerson(newPerson);

            model.addAttribute("persons",  personService.getAllPerson());
            log.info("/addPerson - POST  was called");
            return modelAndView;

        }
       return modelAndView;
    }

    @RequestMapping(value = "/editPerson/{id}", method = RequestMethod.GET)
    public ModelAndView editPage(@PathVariable("id") int id) throws
NoSuchEntityException {
        Person person = personService.getById(id).orElseThrow(()-> new
```

```
NoSuchEntityException("Person not found") );
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.setViewName("editPerson");
        modelAndView.addObject("person", person);
        return modelAndView;
    }

    @RequestMapping(value = "/editPerson", method = RequestMethod.POST)
    public ModelAndView editPerson( @Valid @ModelAttribute("person") Person
person,
                                    Errors errors) {
        log.info("/editPerson - POST  was called"+ person);
        personService.addNewPerson(person);

        // personService.editPerson(person);
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.setViewName("redirect:/personList");

        return modelAndView;
    }


    @RequestMapping(value = "/deletePerson/{id}", method = RequestMethod.GET)
    public ModelAndView deletePerson(@PathVariable("id") Long id) throws
NoSuchEntityException {
        Person person = personService.getById(id).orElseThrow(()-> new
NoSuchEntityException("Person not found") );
        personService.deletePerson(person);
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.setViewName("redirect:/personList");
        return modelAndView;
    }
}
```

Логгирование использовалось для отладки

```
zip=220353, email=pershkov@tut.by, birthday=null, phone=1534352672)
08:22:02 - /addPerson - POST  was calledNewPersonDto(personId=null, firstName=Nikita, lastName=Peshkov, street=Oktabarskaia,
zip=220353, email=pershkov@tut.by, birthday=null, phone=15343526724)
08:22:07 - /addPerson - POST  was calledNewPersonDto(personId=null, firstName=Nikita, lastName=Peshkov, street=Oktabarskaia,
zip=220353, email=pershkov@tut.by, birthday=null, phone=153435267245)
08:22:15 - /addPerson - POST  was calledNewPersonDto(personId=null, firstName=Nikita, lastName=Peshkov, street=Oktabarskaia,
zip=220353, email=pershkov@tut.by, birthday=null, phone=298745362)
08:22:15 - /addPerson - POST  was called
08:22:17 - /addPerson - GET  was calledNewPersonDto(personId=null, firstName=null, lastName=null, street=null, city=null,
mail=null, birthday=null, phone=null)
08:22:25 - /addPerson - POST  was calledNewPersonDto(personId=null, firstName=olga, lastName=Николаева, street=Oktabarskaia,
zip=220353, email=44@tut.by, birthday=null, phone=1534352672)
08:22:29 - /addPerson - POST  was calledNewPersonDto(personId=null, firstName=olga, lastName=Николаева, street=Oktabarskaia,
zip=220353, email=44@tut.by, birthday=null, phone=153435267223)
08:22:33 - /addPerson - POST  was calledNewPersonDto(personId=null, firstName=olga, lastName=Николаева, street=Oktabarskaia,
zip=220353, email=44@tut.by, birthday=null, phone=123456)
08:22:37 - /addPerson - POST  was calledNewPersonDto(personId=null, firstName=olga, lastName=Николаева, street=Oktabarskaia,
zip=220353, email=44@tut.by, birthday=null, phone=123456789)
```

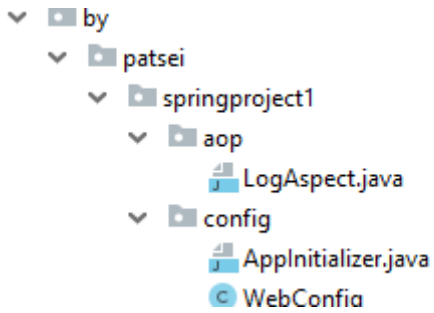
Но это как раз и есть сквозные функции, которые надо вынести из контроллера в аспекты.

1) Подключим зависимости, если вы не выбирали функцию AOP в Spring Boot.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
```

```
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

2) Создадим пакет aop с классом LogAspect



в нем опишем Pointcut и Advice.

```
@Aspect
@Component
@Slf4j
public class LogAspect {

    @Pointcut("execution(public *
by.patsei.springproject1.controller.PersonController.*(..))")
    public void callAtPersonController() {
    }

    @Before("callAtPersonController()")
    public void beforeCallMethod(JoinPoint jp) {
        String args = Arrays.stream(jp.getArgs())
                .map(a -> a.toString())
                .collect(Collectors.joining(","));
        log.info("before " + jp.toString() + ", args=[" + args + "]");

    }

    @After("callAtPersonController()")
    public void afterCallAt(JoinPoint jp) {
        log.info("after " + jp.toString());
    }
}
```

Здесь определены все public методы PersonController с любым типом возврата * и количеством и типом аргументов (..)

```
  @Pointcut("execution(
public * by.patsei.springproject1.controller.PersonController.*(..))")
```

В советах Before и After есть ссылка на Pointcut (`callAtPersonController()`)

Здесь мы создали условие выборки наших методов, для которых мы решаем задачу логгирования. Тут вариантов задать условие - много. Необязательно выражение для выборки задавать отдельно в *PointCut*, можно сразу в A*dvice.*

Итак @Pointcut -Тип возвращаемого значения метода должен быть пустым, а параметры метода должны соответствовать параметрам точки среза. Нет необходимости определять тело метода, потому что оно будет опущено.

Посмотрим на методы:

```
    @Before("callAtPersonController()")
    public void beforeCallMethod(JoinPoint jp)
...
 @After("callAtPersonController()")
    public void afterCallAt(JoinPoint jp) {
```

В аргументе *JoinPoint* есть полезная информация о методе.

3) Уберем из PersonController логи

   Обратите внимание в котроллере появились маркеры связанные с аспектами, нажав по ним можно перейти на совет

```
@GetMapping(value = {"/personList"})
public ModelAndView personList(Model model) {
    List<Person> persons = personService.getAllPerson();
    ModelAndView modelAndView = new ModelAndView();
    modelAndView.setViewName("personList");
    model.addAttribute( s: "persons", persons);
    return modelAndView;
}

@GetMapping(value = {"/addPerson"})
public  ModelAndView showAddPersonPage(Model model) {
    ModelAndView modelAndView = new ModelAndView( viewName:
    NewPersonDto personForm = new NewPersonDto();
    model.addAttribute( s: "personForm", personForm);
    return modelAndView;
}

@PostMapping(value = {"/addPerson"})
public ModelAndView savePerson(Model model, //
                               @Valid @ModelAttribute("per
                               Errors errors) {
    ModelAndView modelAndView = new ModelAndView();
    if (errors.hasErrors()) {
        modelAndView.setViewName("addPerson");
    }
```

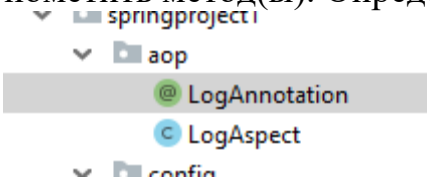
4) Запустим проект, выполним действия и посмотрим на консоль

```
08:33:16 - before execution(ModelAndView by.patsei.springproject1.controller.PersonController.index(Model)), args=[[]]
08:33:16 - after execution(ModelAndView by.patsei.springproject1.controller.PersonController.index(Model))
08:33:18 - before execution(ModelAndView by.patsei.springproject1.controller.PersonController.personList(Model)), args=[[]]
08:33:18 - after execution(ModelAndView by.patsei.springproject1.controller.PersonController.personList(Model))
08:33:22 - before execution(ModelAndView by.patsei.springproject1.controller.PersonController.editPage(int)), args=[6]
08:33:22 - after execution(ModelAndView by.patsei.springproject1.controller.PersonController.editPage(int))
08:33:28 - before execution(ModelAndView by.patsei.springproject1.controller.PersonController.editPerson(Person,Errors)),
n(id=6, firstName=Nikita, lastName=Peshkov, street=Oktabarskaia, city=Minsk, zip=220353, email=pershkov@tut.by, birthday=null,
5360),org.springframework.validation.BeanPropertyBindingResult: 1 errors
in object 'person' on field 'birthday': rejected value []; codes [typeMismatch.person.birthday,typeMismatch.birthday,typeMismatch
Date,typeMismatch]; arguments [org.springframework.context.support.DefaultMessageSourceResolvable: codes [person.birthday,
arguments []; default message [birthday]]; default message [Failed to convert property value of type 'java.lang.String' to
pe 'java.util.Date' for property 'birthday'; nested exception is org.springframework.core.convert.ConversionFailedException:
onvert from type [java.lang.String] to type [@javax.persistence.Column java.util.Date] for value ''; nested exception is
llegalArgumentException]]
08:33:28 - after execution(ModelAndView by.patsei.springproject1.controller.PersonController.editPerson(Person,Errors))
08:33:28 - before execution(ModelAndView by.patsei.springproject1.controller.PersonController.personList(Model)), args=[[]]
08:33:28 - after execution(ModelAndView by.patsei.springproject1.controller.PersonController.personList(Model))
```

Таким образом в котроллере нет никакого упоминания про запись в лог, а все логирование сосредоточено в отдельном пакете (классе).

5) Изменение среза

Если советы надо применять к разным методам контроллеров, сервисов и т.п., то можно сделать запрос по аннотации. Аннотация – это удобный способ пометить метод(ы). Определим аннотацию




```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface LogAnnotation {
}
```

И заменим Pointcut.

```
@Aspect
@Component
@Slf4j
public class LogAspect {

    @Pointcut("@annotation(LogAnnotation)")
 //   @Pointcut("execution(public *
by.patsei.springproject1.controller.PersonController.*(..))")
    public void callAtPersonController() {
    }
```

Теперь выборка методов будет связана с аннотацией.

```
    }
    @LogAnnotation
    @GetMapping(value = {"/personList"})
    public ModelAndView personList(Model model) {
        List<Person> persons = personService.getAllPerson();
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.setViewName("personList");
        model.addAttribute( s: "persons", persons);
        return modelAndView;
    }


    @GetMapping(value = {"/addPerson"})
    public  ModelAndView showAddPersonPage(Model model) {
        ModelAndView modelAndView = new ModelAndView( viewName: "addPersor
        NewPersonDto personForm = new NewPersonDto();
        model.addAttribute( s: "personForm", personForm);
        return modelAndView;
    }

    @PostMapping(value = {"/addPerson"})
    public ModelAndView savePerson(Model model, //
                                   @Valid @ModelAttribute("personForm") I
                                   Errors errors) {
        ModelAndView modelAndView = new ModelAndView();
       if (errors.hasErrors()) {
            modelAndView.setViewName("addPerson");
        }
```

Запустите и проверьте.

Аспекты можно сделать для проверки прав доступа.

https://docs.spring.io/spring/docs/3.0.x/spring-framework-reference/html/aop.html

https://www.eclipse.org/aspectj/