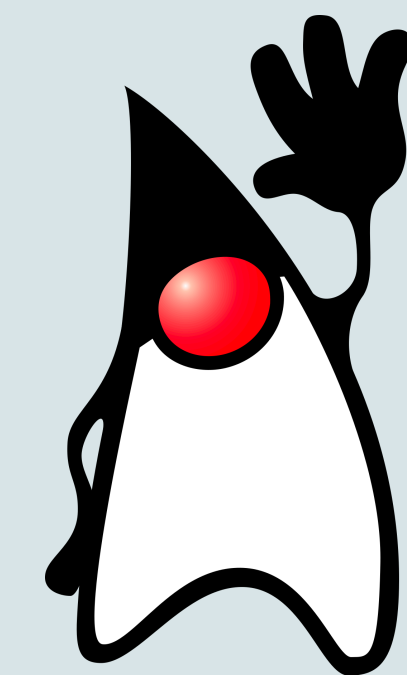




# СБЕРБАНК

---

Корпоративный  
университет



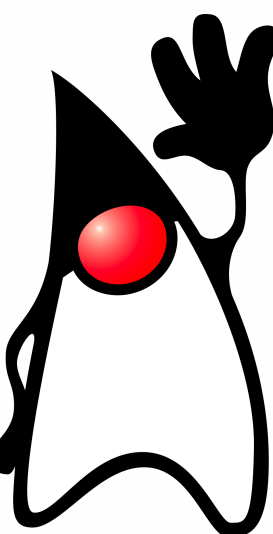
# Dependency Injection. Spring Framework. Начало

Занятие №14

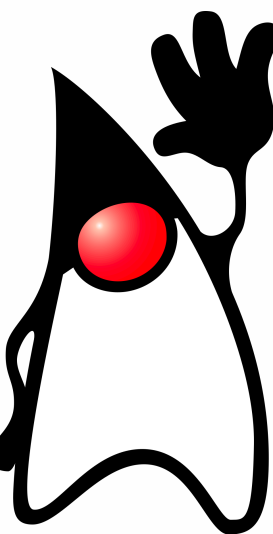


**СБЕРБАНК**  
Корпоративный  
университет

- Dependency Injection
- Spring Framework. Описание проекта
- Spring Framework. IoC. Классическая конфигурация.
- Spring AOP.

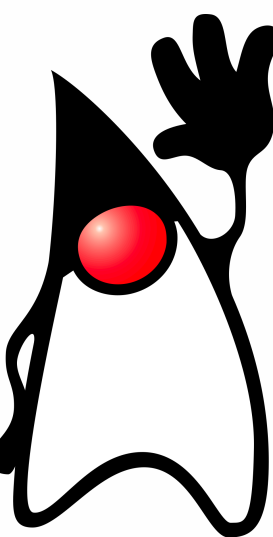


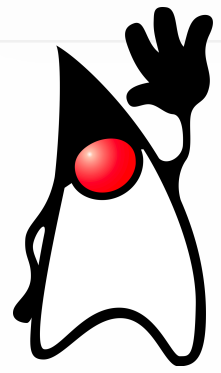
- Активно участвуем. Не стесняйтесь задавать вопрос.
- Но off-topic обсуждаем в Telegram @sb\_ku\_java\_2019\_10
- Не стесняйтесь просто спрашивать в telegram.
- ДЗ - работаем над библиотекой



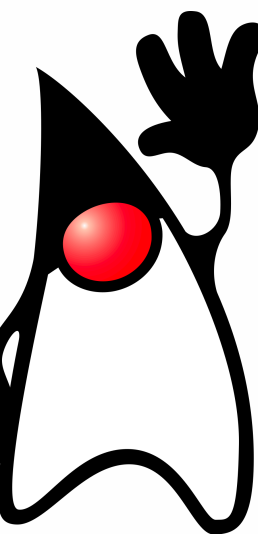
**Договорились?  
Поехали!**

- Не надо пытаться всё максимально засунуть в один класс. Это не лучшая практика
- Builder. Хороший сайт по шаблонам - <https://refactoring.guru/ru>
- **GOF. Шаблоны проектирования.** Сложно, научно, скучно. Но must read и ПОНЯТЬ
- Много хороших не сложных статей - Джошуа Блох. Effective Java





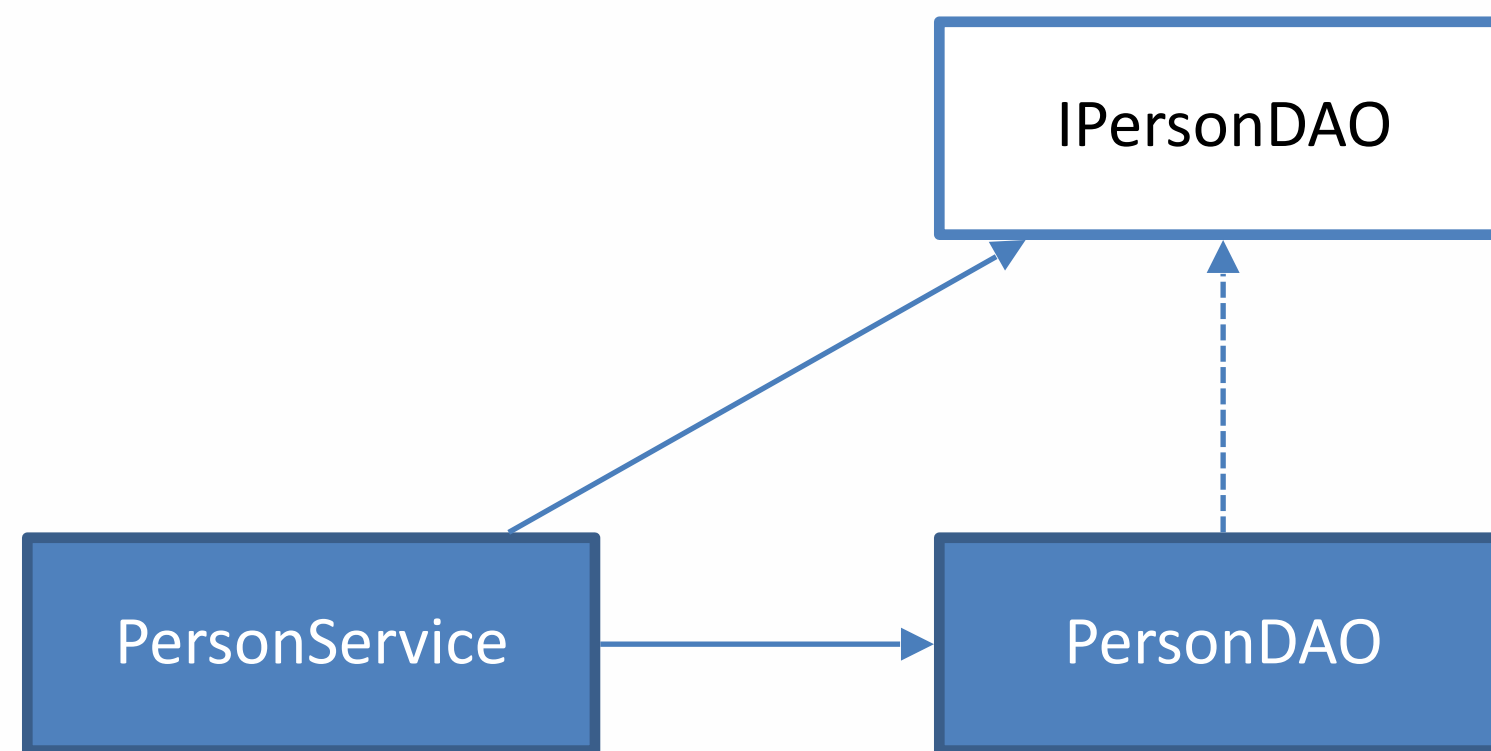
- **Dependency Injection**
- Spring Framework. Описание проекта
- Spring Framework. IoC. Классическая конфигурация.
- Spring AOP



01

# Dependency Injection



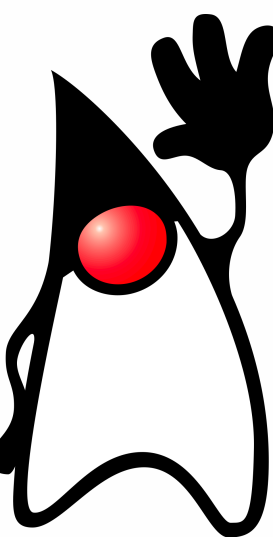


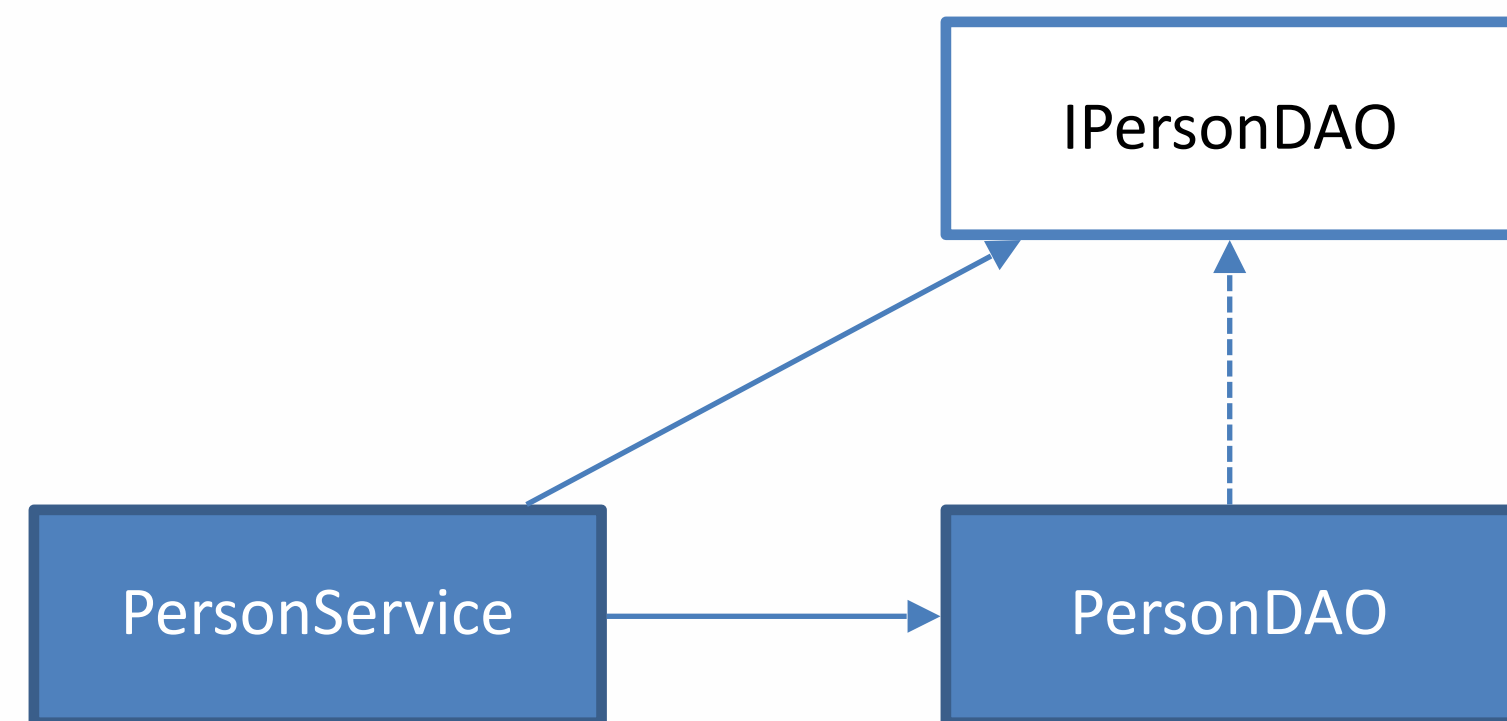
```
class PersonService {
    private IPersonDAO dao;

    public PersonService() {
        this.dao = new PersonDAO(
            "127.0.0.1:80"
        );
    }
}

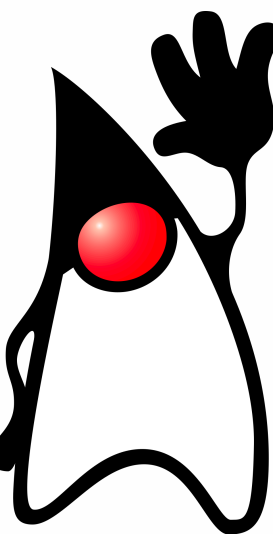
class PersonDAO implements IPersonDAO {
    private String url;

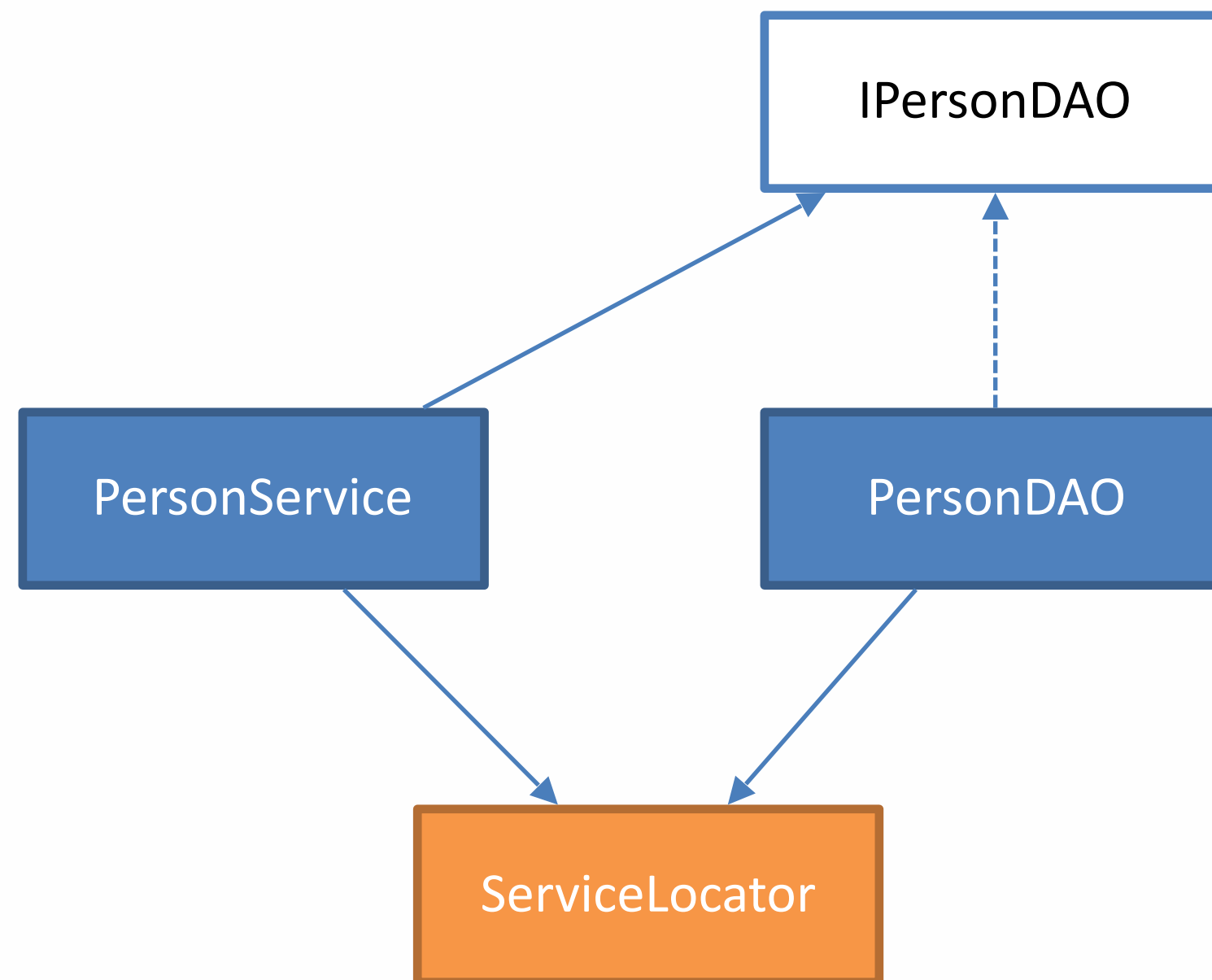
    public PersonDAO(String url) {
        this.url = url;
    }
}
```





- Класс PersonService напрямую зависит от PersonDAO
- Невозможно тестировать PersonService отдельно от PersonDAO
- Жизненный цикл классов связан напрямую
- Невозможно заменить PersonDAO на другую реализацию



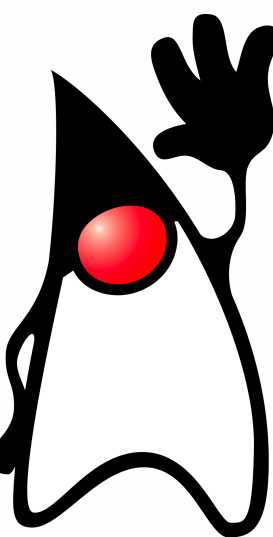


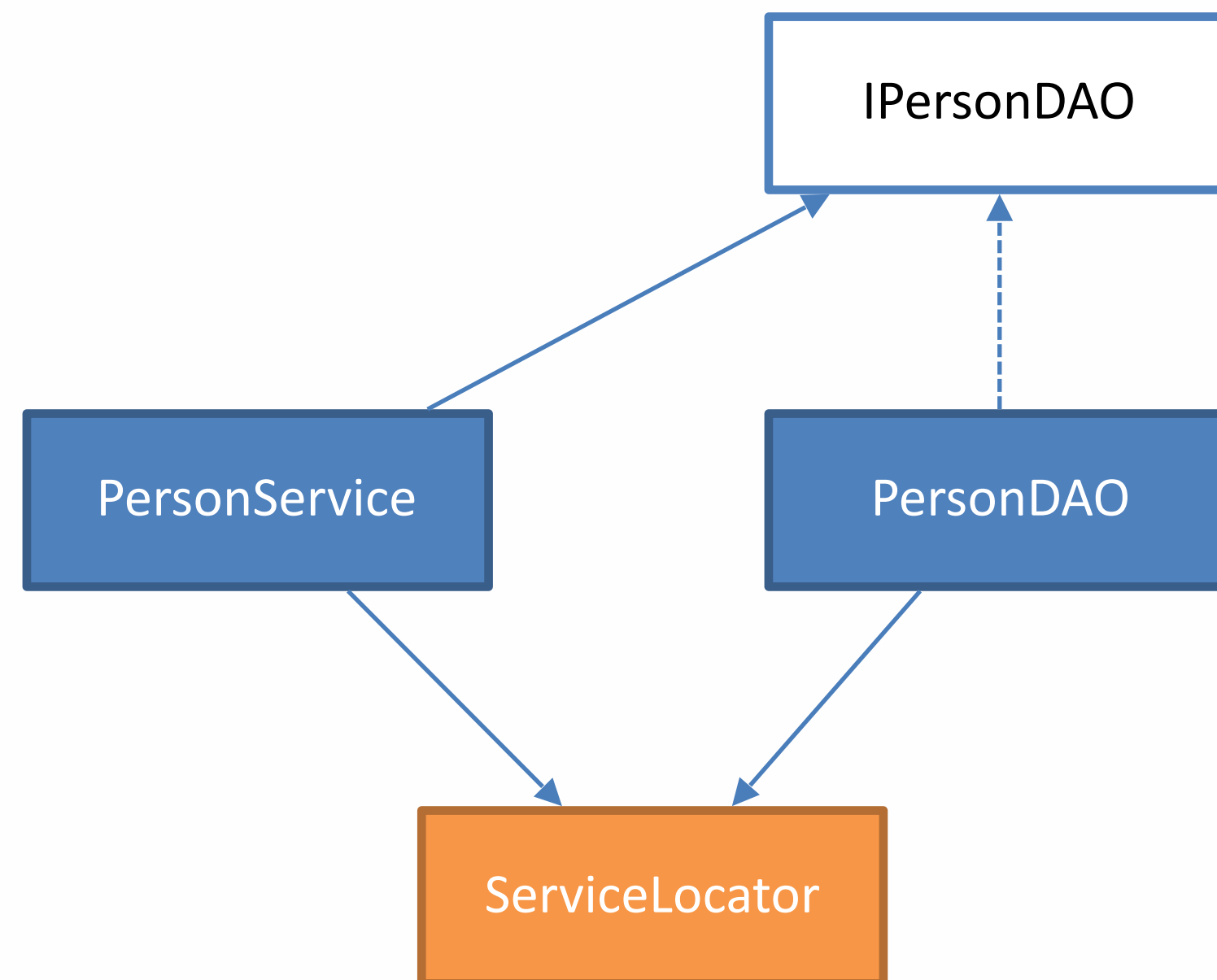
```
class PersonService {
    private IPersonDAO dao;

    public PersonService() {
        this.dao = ServiceLocator
            .getPersonDAO();
    }
}

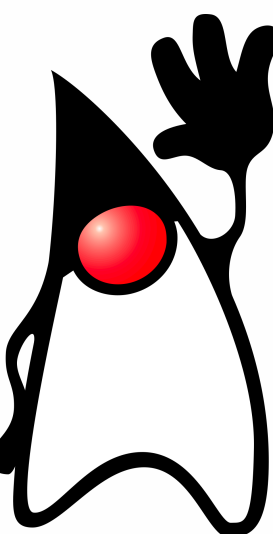
class PersonDAO implements IPersonDAO {
    private String url;

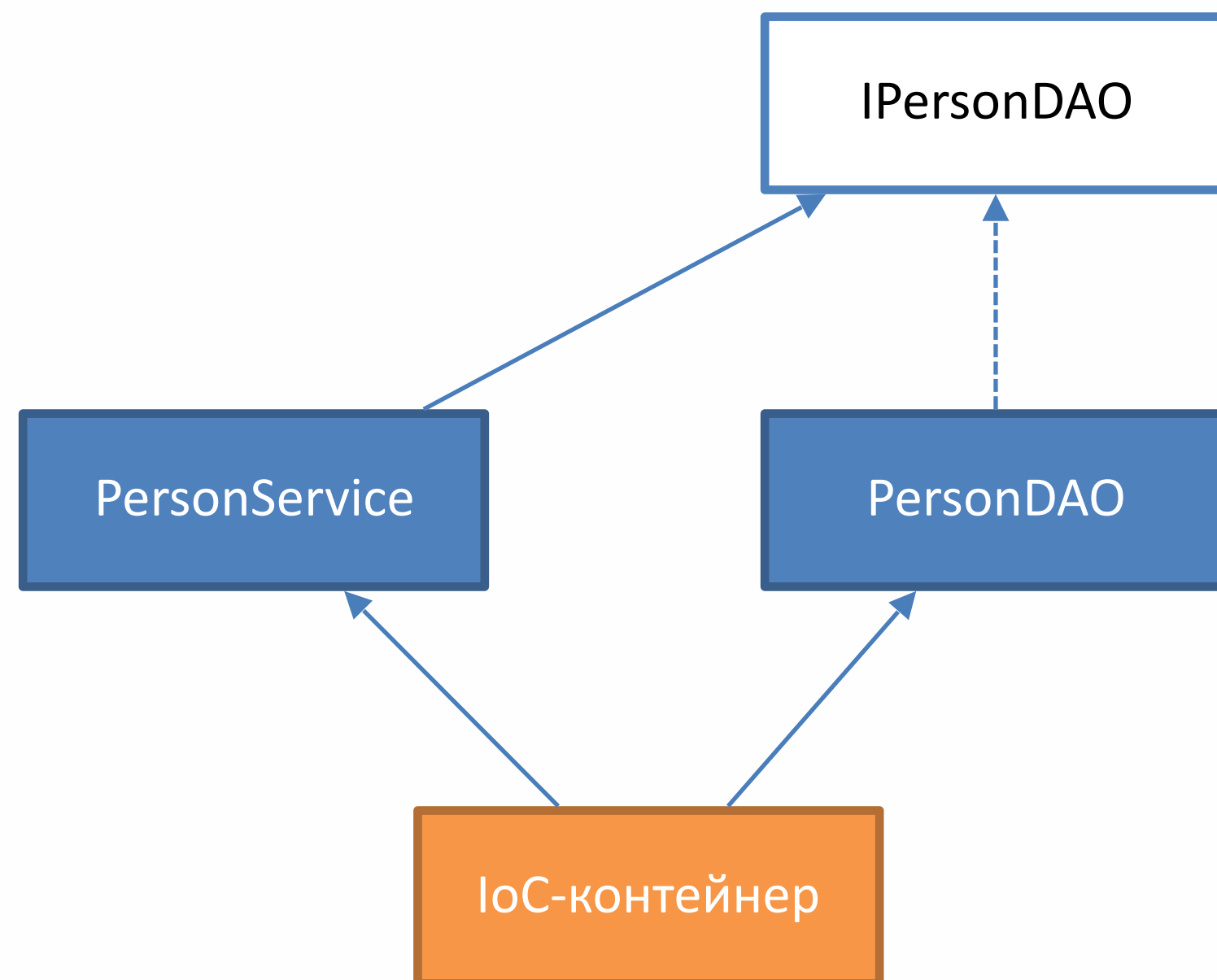
    public PersonDAO(String url) {
        this.url = url;
    }
}
```





- Класс **PersonService** не зависит от **PersonDAO**, но зависит от **ServiceLocator**-а
- Если постараться, то **PersonService** можно тестировать отдельно от **PersonDAO**
- **PersonDAO** можно заменить на другую реализацию



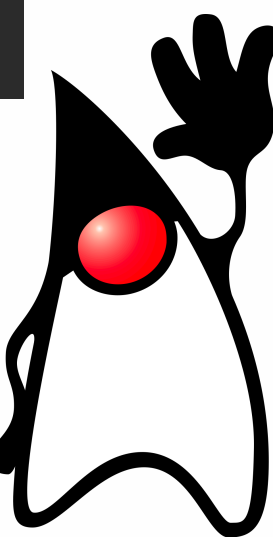


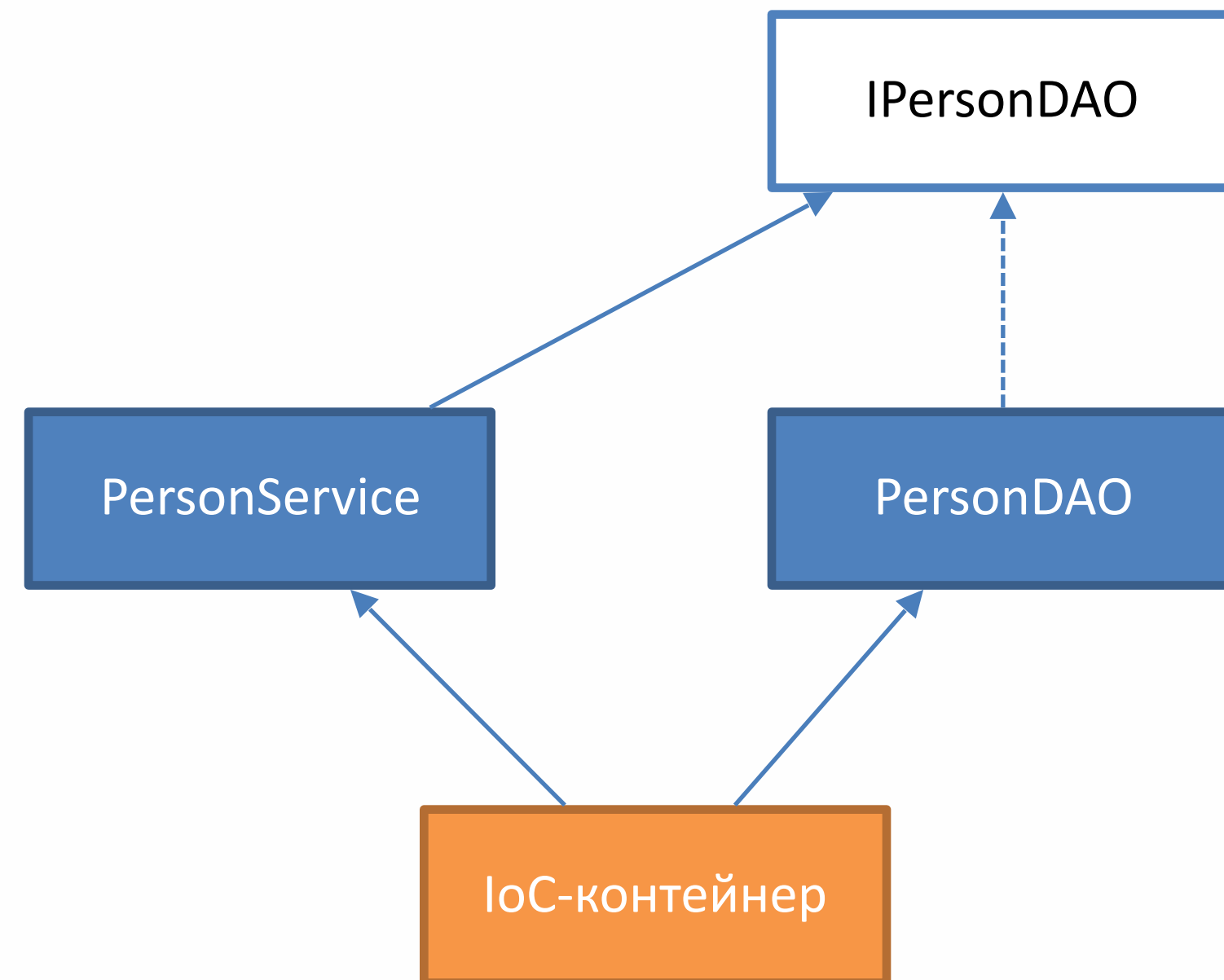
```
class PersonService {
    private IPersonDAO dao;

    public PersonService(IPersonDao d) {
        this.dao = d;
    }
}

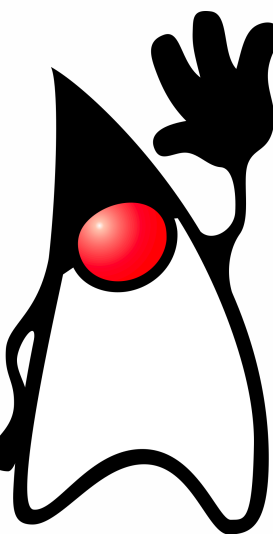
class PersonDAO implements IPersonDAO {
    private String url;

    public PersonDAO(String url) {
        this.url = url;
    }
}
```



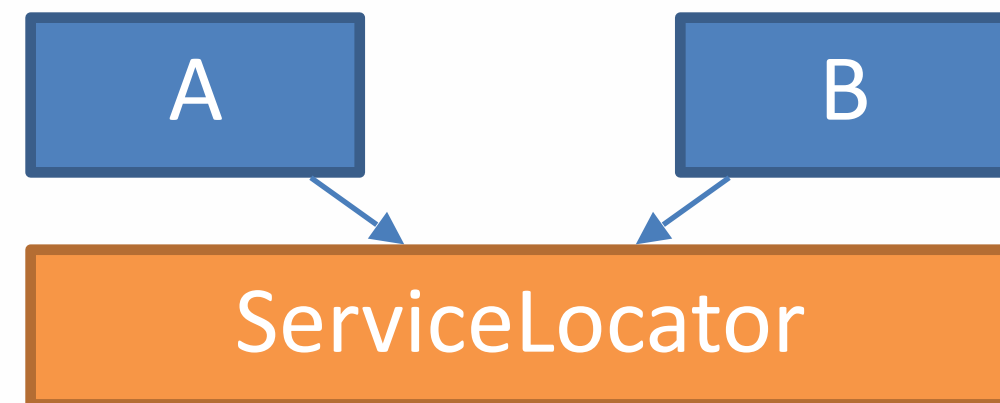


- Контейнер создаёт необходимые объекты и управляет жизненным циклом
- Класс PersonService не зависит от PersonDAO
- PersonService можно легко тестировать отдельно от PersonDAO
- PersonDAO можно заменить на другую реализацию

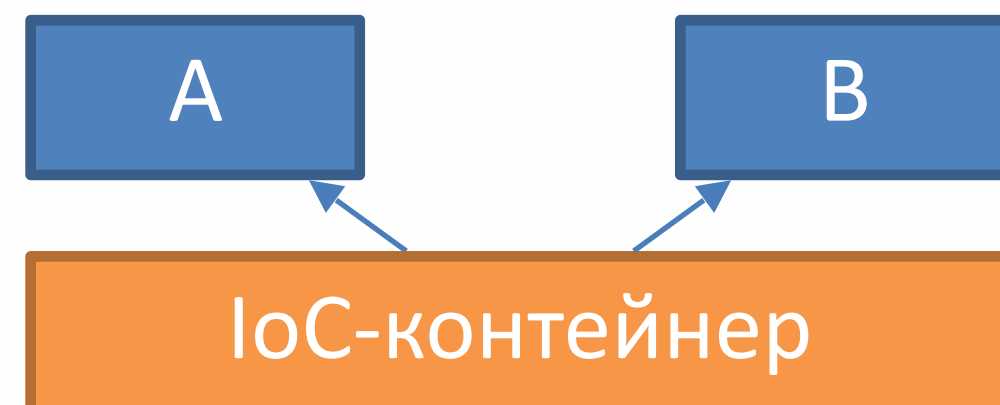




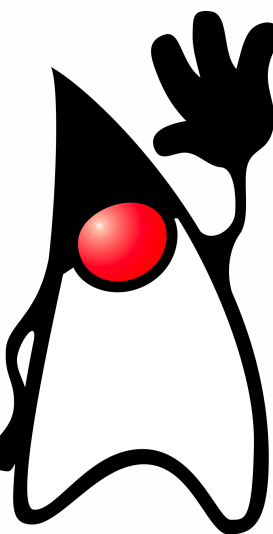
Классический подход



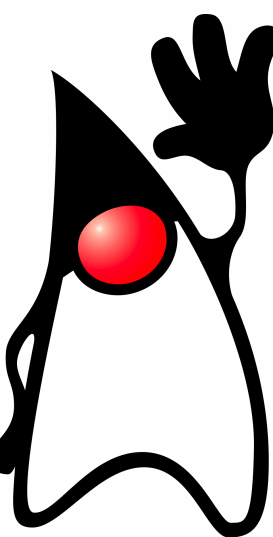
Service Locator



IoC-контейнер

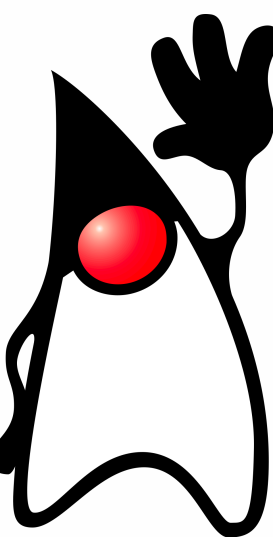


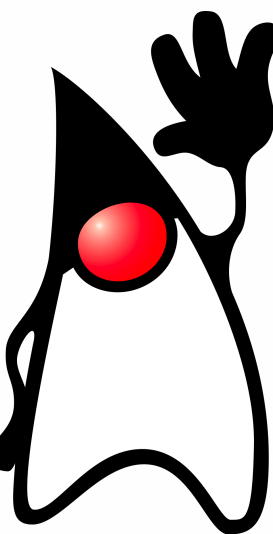
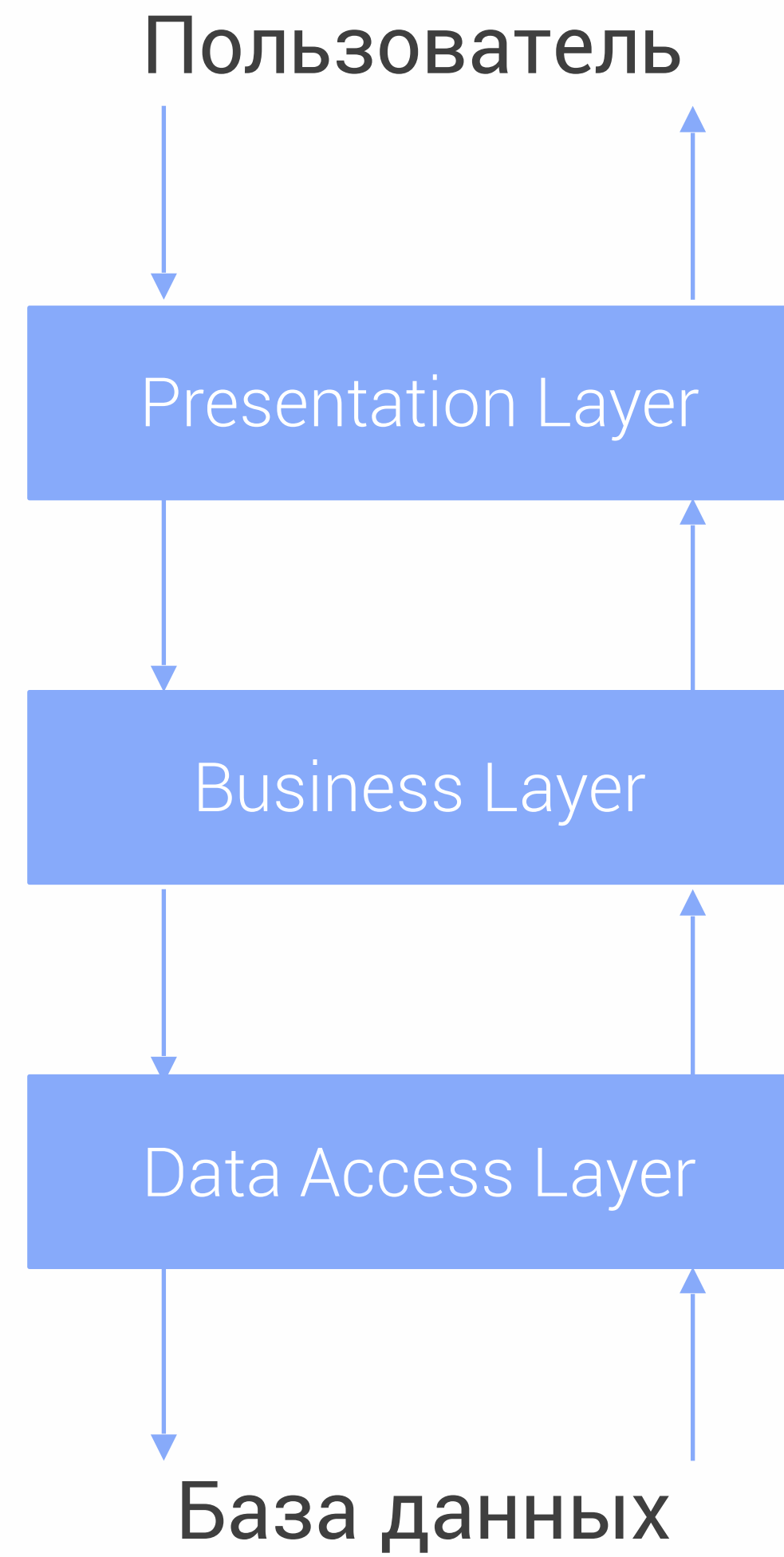
- IoC (Inversion of Control) – базовый принцип, на котором строится Spring
- «Голливудский принцип» (Hollywood Principle):  
«Не звоните мне, я сам Вам позвоню.»
- DI – Dependency Injection – имплементации интерфейсов передаются при создании (чаще всего) объекта.





- Упрощает reuse компонентов.
- Упрощает unit-тестирование
- Чистый код – только бизнес-логика, никакого конфигурационного кода

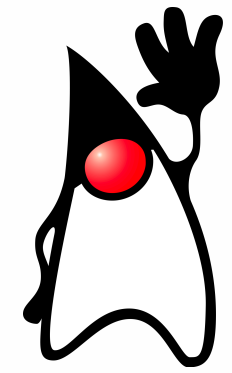




# Ваши вопросы?

Если что — их можно задать  
ПОТОМ

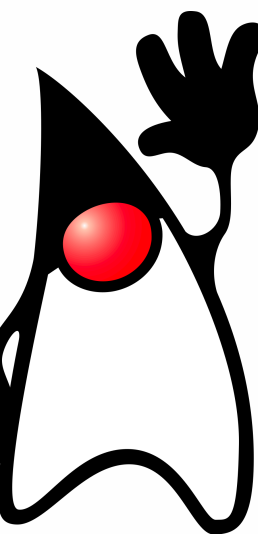
- Dependency Injection



- **Spring Framework. Описание проекта**

- Spring Framework. IoC. Классическая конфигурация.

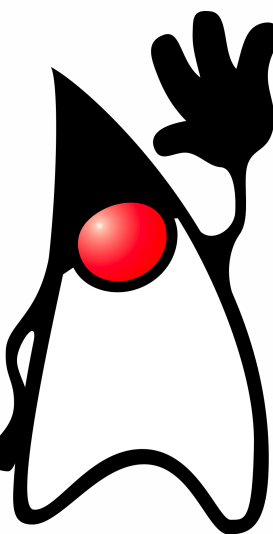
- Spring AOP



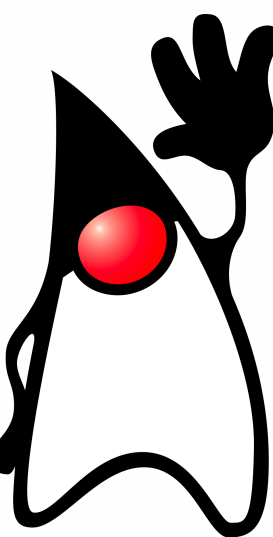
02

# Spring Framework

- Spring – индустриальный **стандарт**.
- Spring – это фреймворк уровня приложений, на нём можно построить целиком всё приложение.



- Но есть «**подфреймворки**» Spring (проекты Spring), которые отвечают только за один слой приложения.
- Да, Spring состоит из других различных **Spring-фреймворков** (проектов Spring).
- Spring – «**lightweight**» фреймворк (объём кода, который необходимо написать программисту – минимален).

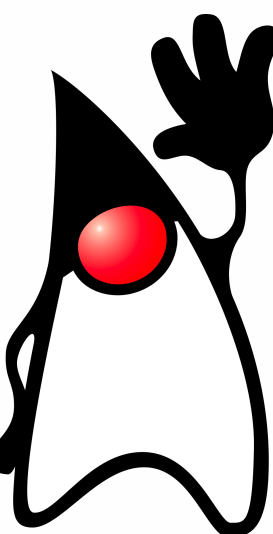


# Spring Projects (только некоторые!)

- Spring IoC + AOP = Context
- Spring JDBC
- Spring ORM,
- Spring Data, Spring Data JPA
- Spring MVC, Spring WebFlux
- Spring Security
- Spring Cloud \*
- Spring Boot

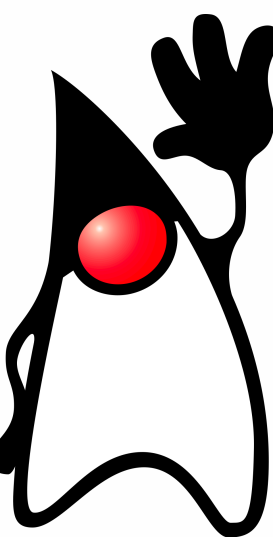


Spring Framework 5

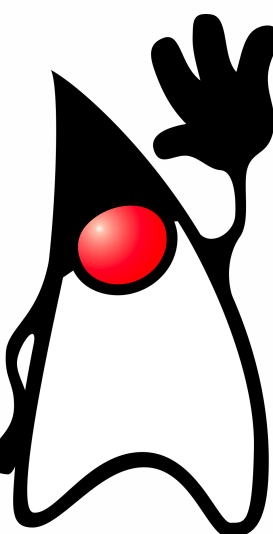




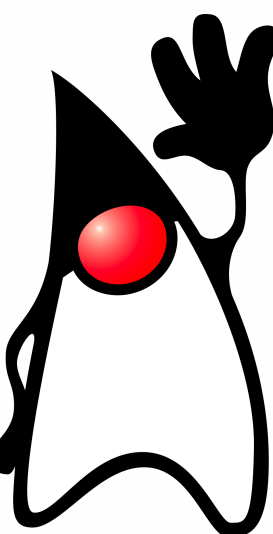
- Изначально создавался для облегчения разработки **JavaEE-приложений**.
- Сейчас на нём разрабатываются, как JavaEE, так и **JavaSE-приложения**.



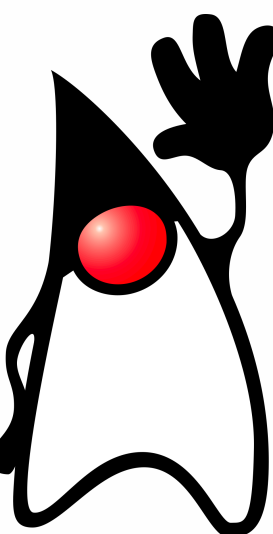
- **Spring Framework** – предоставляет базовую функциональность: IoC, AOP, SpEL и т.д.
- **Spring Boot** – максимально ускоряет разработку production -grade приложения от идеи до деплоя.
- **Spring Cloud** – для разработки приложений в облачной среде, в частности для систем на микросервисной архитектуре.



- **Spring Data** – позволяет создавать JDBC-репозитории, написав только интерфейс репозитория.
- **Spring Integration** – содержит реализацию Enterprise Integration Patterns и позволяет создавать приложения с Messaging архитектурой.



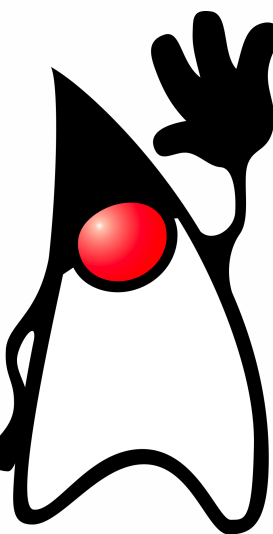
- **Spring Security** – один из самых популярных фреймворков для реализации авторизации и аутентификации в приложении.
- **Spring WebServices** – позволяет максимально быстро создавать как сами веб-сервисы, так и клиентов к ним.
- **Spring MVC** – позволяет создавать веб-приложения на основе MVC паттерна.



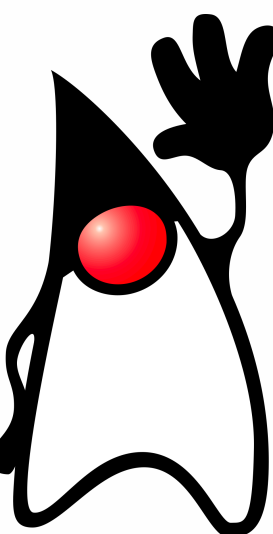


<http://spring.io>

- Документация
- Текущие версии библиотек
- Примеры и tutorials

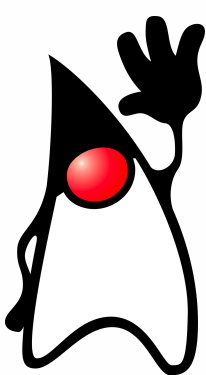


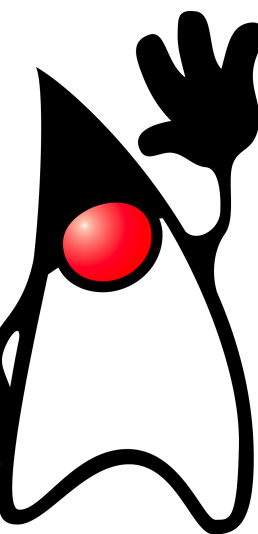
- <http://www.baeldung.com> - лаконичные примеры (не всегда эффективные);
- <http://www.mkyong.com/> – максимально лаконичные примеры (иногда старый спринг);
- <https://stackoverflow.com/questions/tagged/spring> – и другие разделы посвящённые Spring;
- <http://mvnrepository.com/> – при работе с Maven зависимостями.



# Ваши вопросы?

Если что — их можно задать  
ПОТОМ

- Dependency Injection
- Spring Framework. Описание проекта
-  • **Spring Framework. IoC. Классическая конфигурация**
- Spring AOP

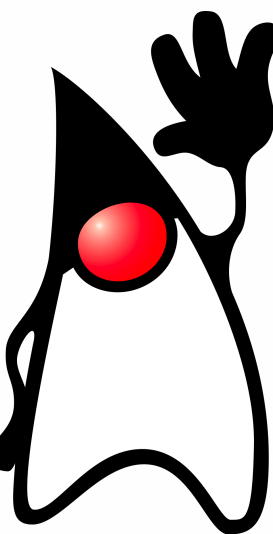




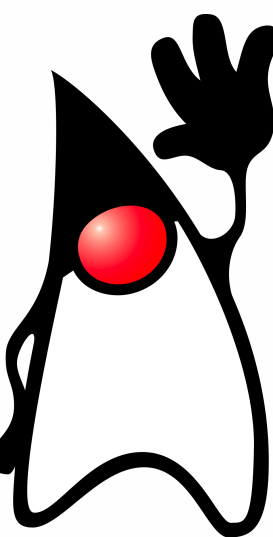
03

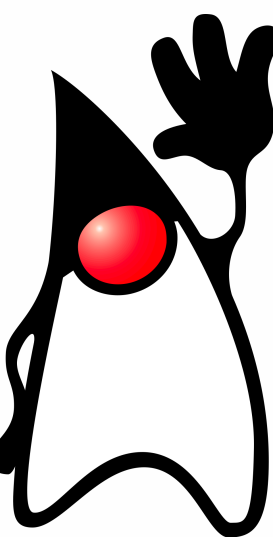
# Spring IoC. Классическая конфигурация

- Бывают разные.
- Конфигурируются разными способами.
- Конфигурируются разными языками XML, Java, Groovy.

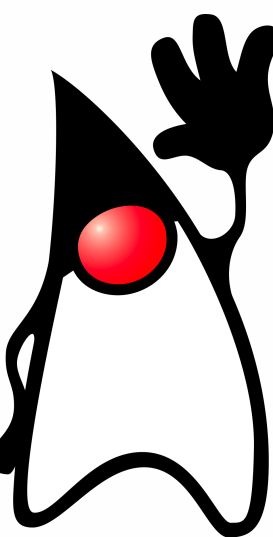


- **XML** является традиционным способом задания конфигурации контейнера, хотя существуют и другие способы задания метаданных (аннотации, Java код и т.д.).
- Во многих случаях проще и быстрее конфигурировать контейнер с помощью аннотаций. (это мы пройдем позже).
- **На старте** мы будем конфигурировать с помощью XML, дальше забудем как страшный сон.

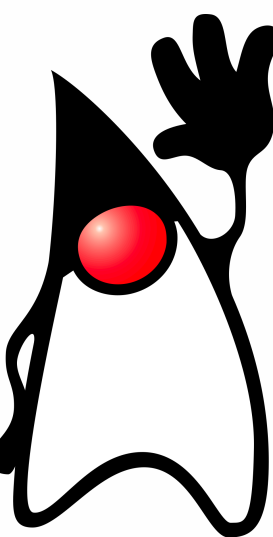




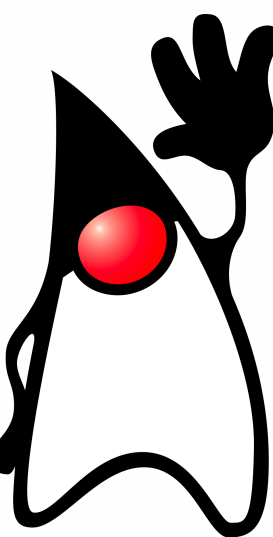
```
1 package ru.otus.spring01.service;
2
3 import ru.otus.spring01.dao.PersonDao;
4 import ru.otus.spring01.domain.Person;
5
6 public class PersonServiceImpl implements PersonService {
7
8     private PersonDao dao;
9
10    public PersonServiceImpl(PersonDao dao) {
11        this.dao = dao;
12    }
13
14    public Person getByName(String name) {
15        return dao.findByName(name);
16    }
17 }
18
```



```
<dependency>  
  <groupId>org.springframework</groupId>  
  <artifactId>spring-context</artifactId>  
  <version>5.1.3.RELEASE</version>  
</dependency>
```



```
public class Main {  
  
    public static void main(String[] args) {  
        ClassPathXmlApplicationContext context =  
            new ClassPathXmlApplicationContext("/context.xml");  
        PersonService s = context.getBean(PersonService.class);  
        s.getPerson();  
    }  
}
```

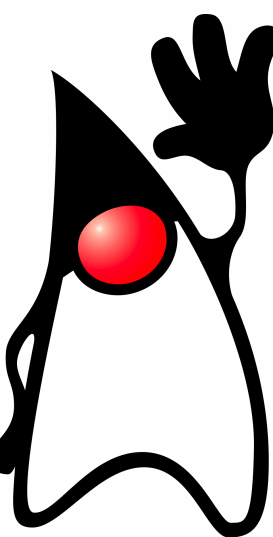


```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="http://www.springframework.org/schema/beans"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="personDao" class="ru.otus.spring01.dao.PersonDaoSimple">
    </bean>

    <bean id="personService" class="ru.otus.spring01.service.PersonServiceImpl">
        <constructor-arg name="dao" ref="personDao"/>
    </bean>

</beans>
```

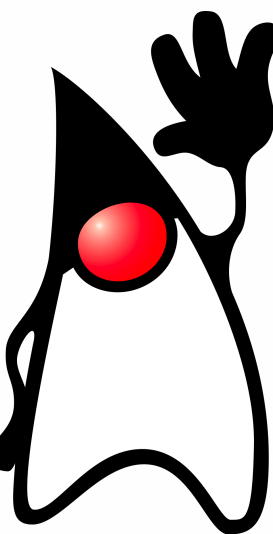




# Ваши вопросы?

Если что — их можно задать  
ПОТОМ

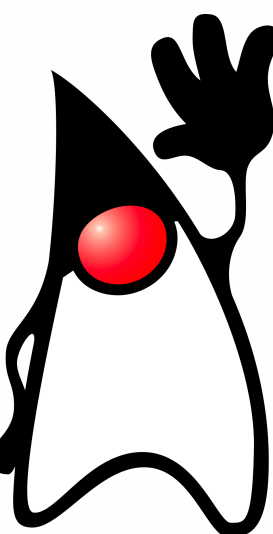
- Смотрим вместе репозиторий.



### Добавить Maven-зависимость spring-context в pom.xml

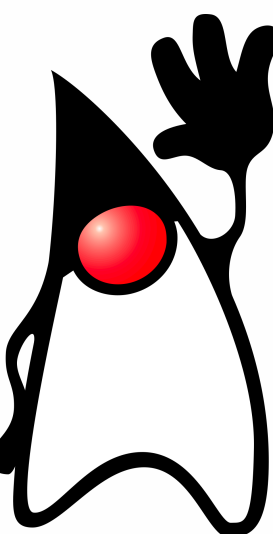
Текущую версию можно узнать на <http://mvnrepository.com>

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>????????????</version>
</dependency>
```



Создать контекст в main.

```
public class Main {  
  
    public static void main(String[] args) {  
        ClassPathXmlApplicationContext context =  
            new ClassPathXmlApplicationContext("/context.xml");  
        PersonService s = context.getBean(PersonService.class);  
        s.getPerson();  
    }  
}
```



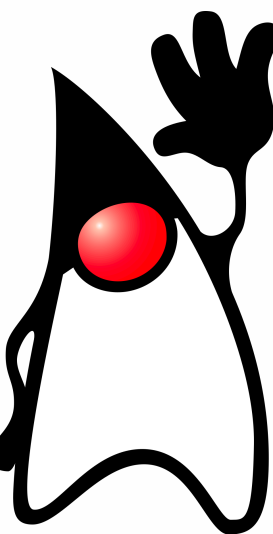
1. Добавить бины в контекст.
2. Настроить зависимость через <constructor-arg ref=.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="personDao" class="ru.otus.spring01.dao.PersonDaoSimple">
    </bean>

    <bean id="personService" class="ru.otus.spring01.service.PersonServiceImpl">
        <constructor-arg name="dao" ref="personDao"/>
    </bean>

</beans>
```



```
package ru.otus.spring01.service;

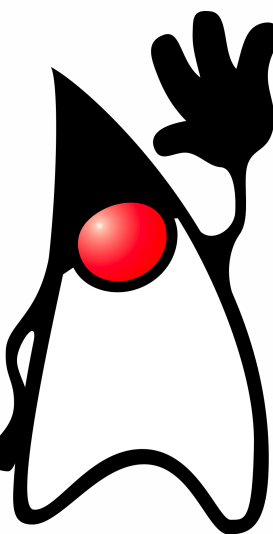
import ru.otus.spring01.dao.PersonDao;
import ru.otus.spring01.domain.Person;

public class PersonServiceImpl implements PersonService {

    private PersonDao dao;

    public void setDao(PersonDao dao) {
        this.dao = dao;
    }

    public Person getByName(String name) {
        return dao.findByName(name);
    }
}
```

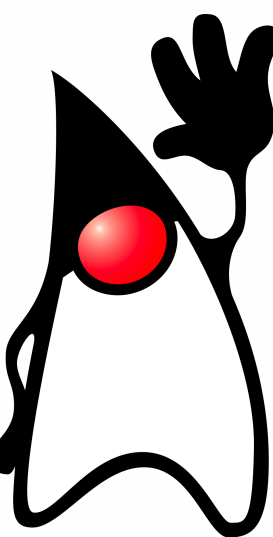


```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="personDao" class="ru.otus.spring01.dao.PersonDaoSimple">
    </bean>

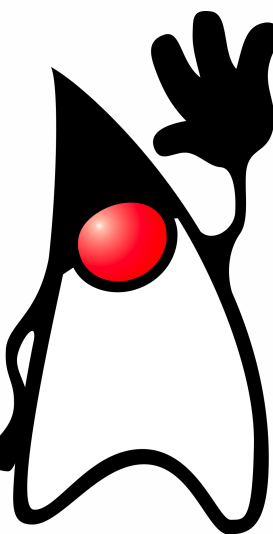
    <bean id="personService" class="ru.otus.spring01.service.PersonServiceImpl">
        <property name="dao" ref="personDao"/>
    </bean>

</beans>
```



# Упражнение 5

1. Добавить бины в контекст.
2. Настроить зависимость через `<property ref=`.



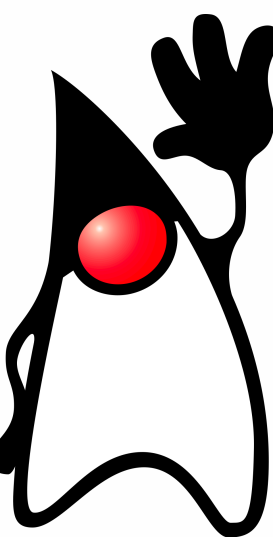


```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd">

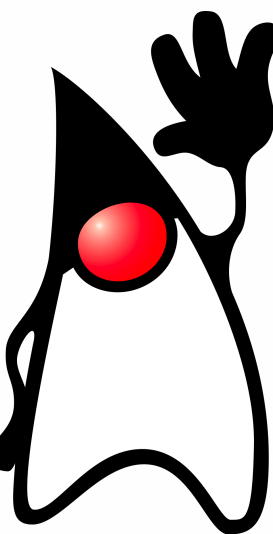
    <bean id="personDao" class="ru.otus.spring01.dao.PersonDaoSimple">
        <property name="defaultAge" value="28"/>
    </bean>

    <bean id="personService" class="ru.otus.spring01.service.PersonServiceImpl">
        <property name="dao" ref="personDao"/>
    </bean>

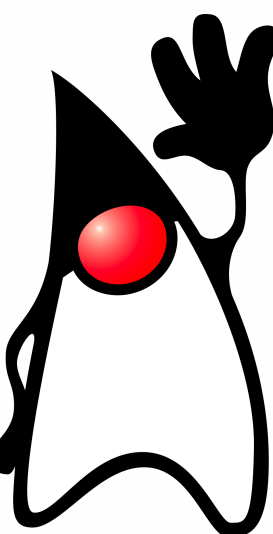
</beans>
```



- DI через конструктор – рекомендован.
- Но могут быть циклические зависимости (виноваты в этом – Вы).
- Вы всегда получаете готовый к работе класс.
- С property может быть NPE.

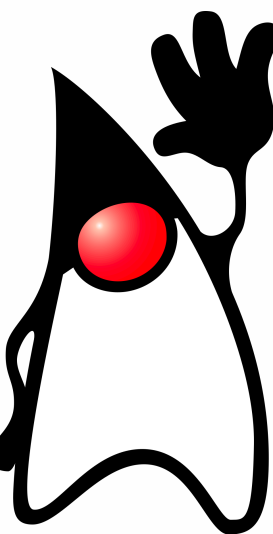


- Бизнес-сервисы (DAO, Services);
- Подключения к внешним системам;
- Мапперы/Маршаллеры/конвертеры;
- Служебные бины (PersistenceContextManager...);
- Бизнес-бины стратегий (Паттерн стратегия).



# Что не нужно класть

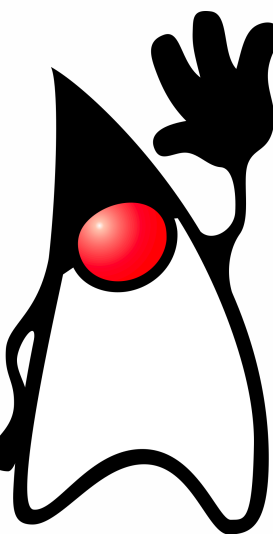
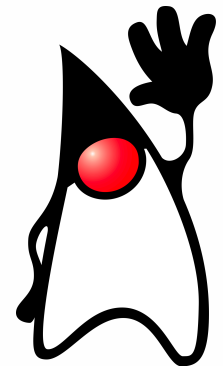
- Бизнес-объекты (бины, пользователи\*);
- Настройки, кроме пачек/файлов настроек\*;
- Объекты, которые понадобятся только один раз в один момент (временные).
- Стандартные классы (String, InputStream, Locale\*)
- Scanner в Вашем домашнем задании, которое сейчас задам 😊



# Ваши вопросы?

Если что — их можно задать  
ПОТОМ

- Dependency Injection
- Spring Framework. Описание проекта
- Spring Framework. IoC. Классическая конфигурация.
- **Spring AOP.**

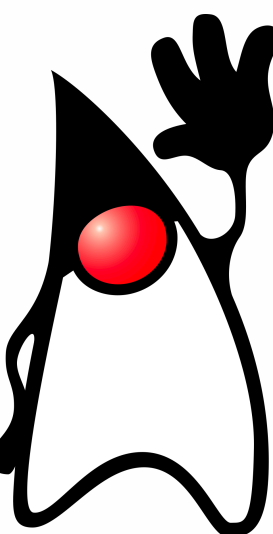


04

**Spring AOP**

Пара слов про тему AOP:

- Это будете использовать напрямую достаточно редко
- Но это базовая магия, которой нужно владеть
- Весь Spring основан на AOP
- Это такая же базовая функциональность как IoC-контейнер





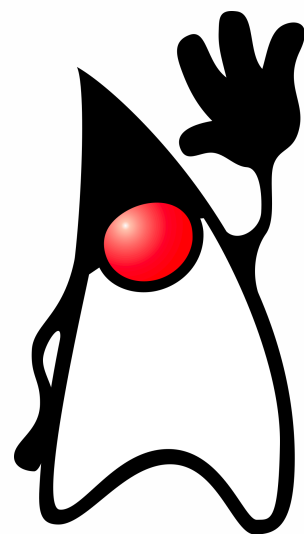
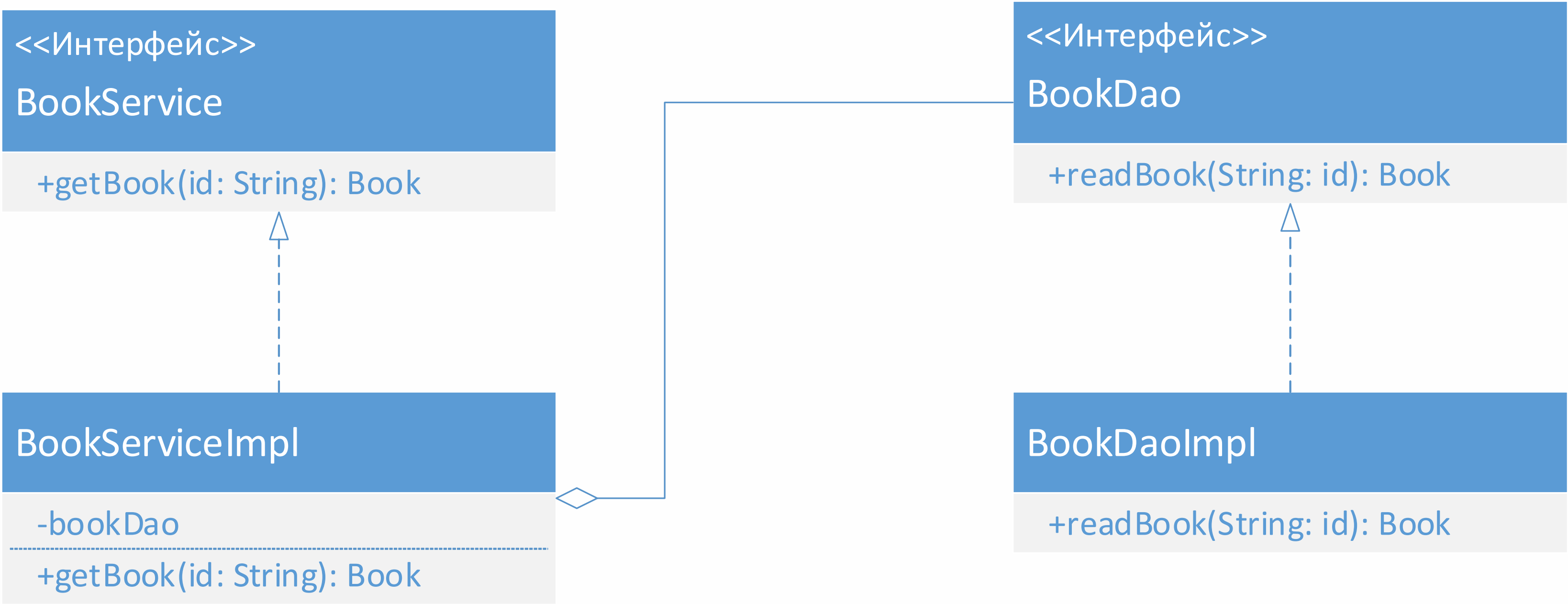
```
@Service
public class BookServiceImpl implements BookService {

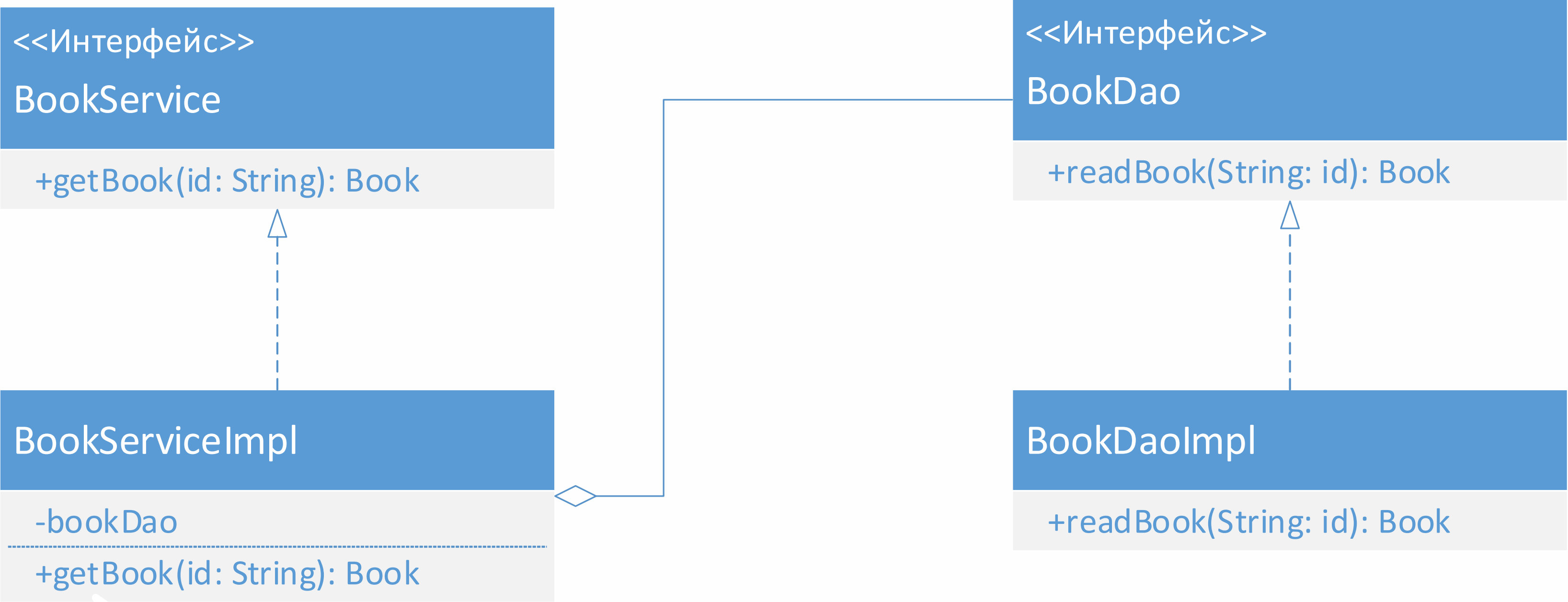
    private final BookDao bookDao;

    public BookService(BookDao bookDao) {
        this.bookDao = bookDao;
    }

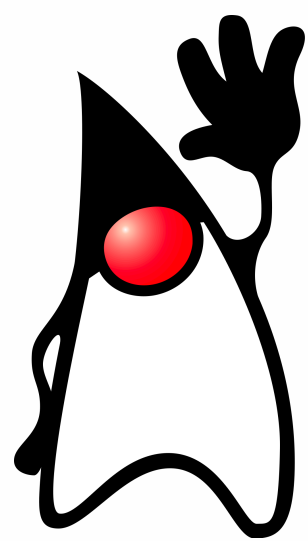
    @Override
    public Book getBook(Integer bookId) {
        Book book = bookDao.readBook(bookId);
        return book;
    }
}
```







Будем дальше писать  
BookServiceImpl.getBook



```
@Override  
public Book getBook(Integer bookId) {  
    Book book = bookDao.readBook(bookId);  
    return book;  
}
```



```
@Override
public Book getBook(Integer bookId) {
    LOG.debug("Call method getBook with id={}", bookId);

    Book book = bookDao.readBook(bookId);

    LOG.debug("Book is {}", book);
    return book;
}
```



```
@Override
public Book getBook(Integer bookId) {
    LOG.debug("Call method getBook with id={}", bookId);

    Book book = null;
    try {
        book = bookDao.readBook(bookId);
    } catch (SQLException e) {
        throw new ServiceException(e);
    }

    LOG.debug("Book is {}", book);
    return book;
}
```



```
@Override
public Book getBook(Integer bookId) {
    if (!SecurityContext.getUser().hasRight("GetBook"))
        throw new AuthException("Access Denied");

    LOG.debug("Call method getBook with id={}", bookId);

    Book book = null;
    try {
        book = bookDao.readBook(bookId);
    } catch (SQLException e) {
        throw new ServiceException(e);
    }

    LOG.debug("Book is {}", book);
    return book;
}
```



```
@Override
public Book getBook(Integer bookId) {
    if (!SecurityContext.getUser().hasRight("GetBook"))
        throw new AuthException("Access Denied");

    LOG.debug("Call method getBook with id={}", bookId);

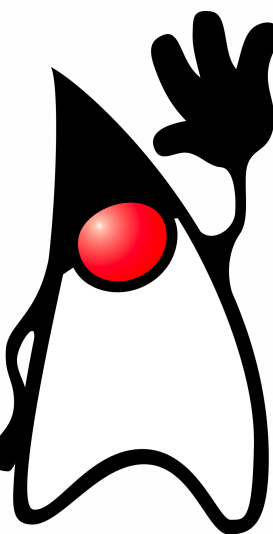
    String cacheKey = "getBook:" + bookId;
    Book book = null;
    try {
        if (cache.containsKey(cacheKey)) {
            book = (Book) cache.get(cacheKey);
        } else {
            book = bookDao.readBook(bookId);
            cache.put(cacheKey, book);
        }
    } catch (SQLException e) {
        throw new ServiceException(e);
    }

    LOG.debug("Book is {}", book);
    return book;
}
```

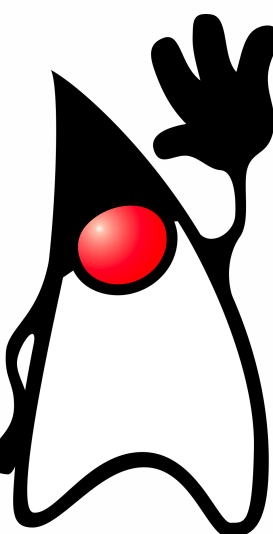


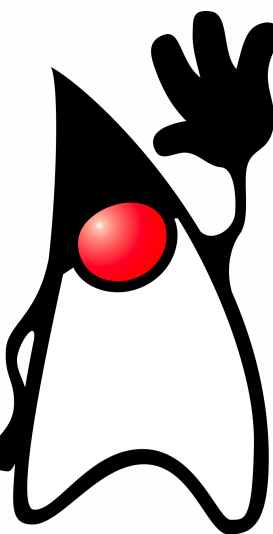
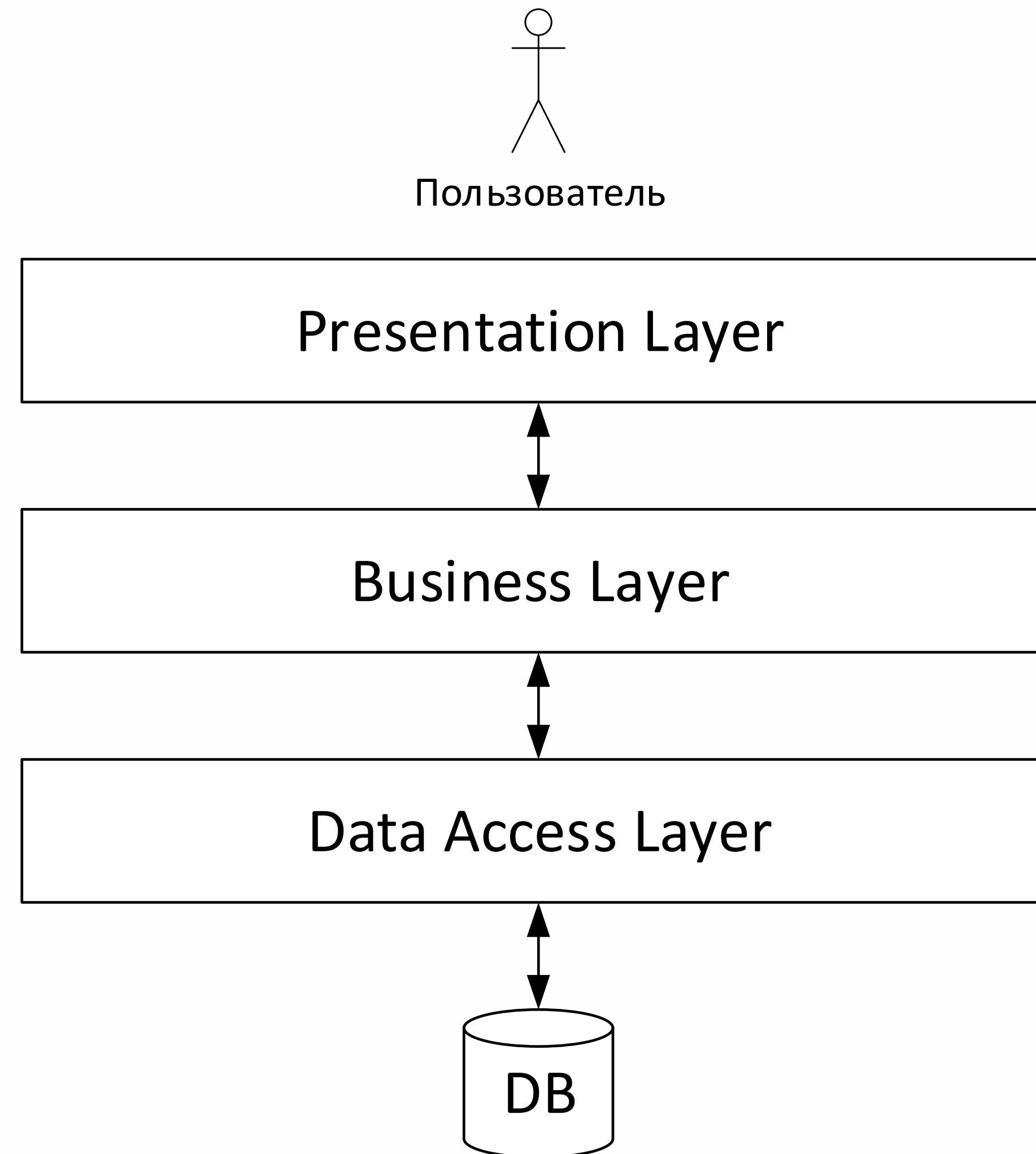


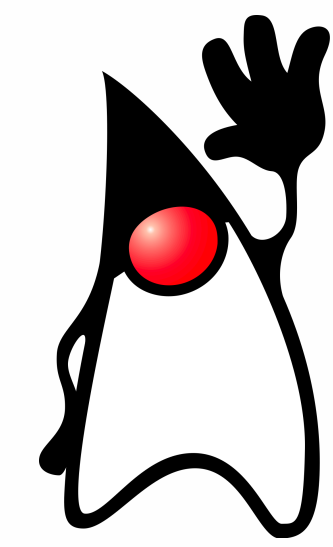
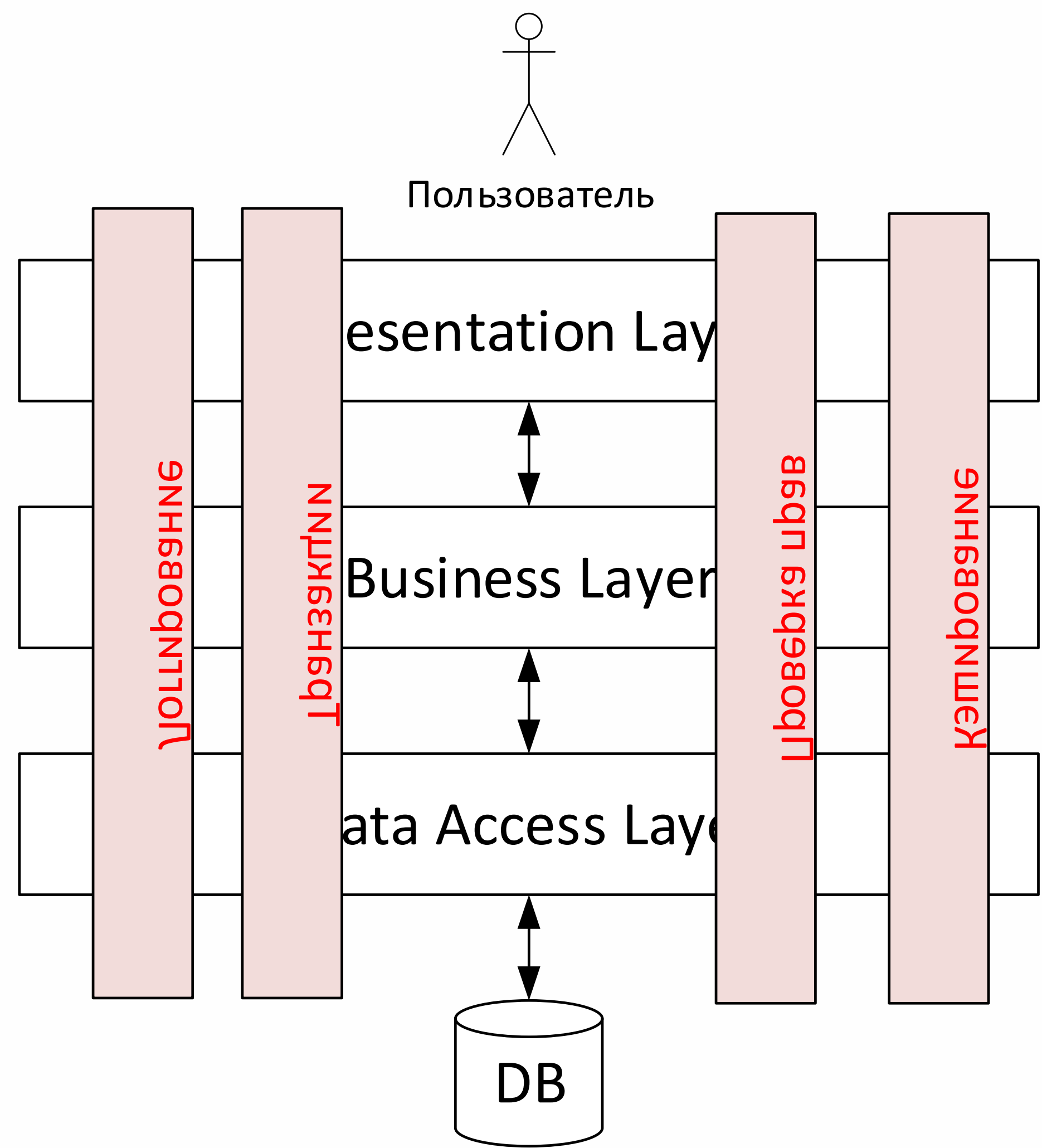
- Сколько строчек бизнес логики?
- Сколько строчек вспомогательной логики?
- Море сервисного кода
- И это только начало
- И это только один метод
- В только одном сервисе
- В одном архитектурном слое



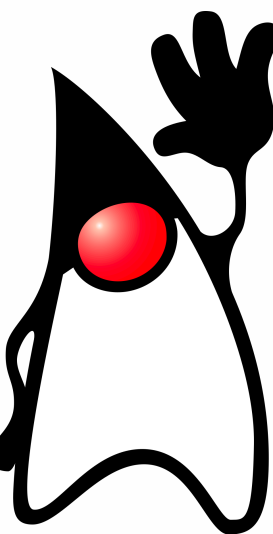
- Такая функциональность называется ортогональной (cross-cutting)
- Логгирование – отличный пример cross-cutting функциональности
- Так много кода получилось, потому что ООП не очень силен в реализации cross-cutting функциональности



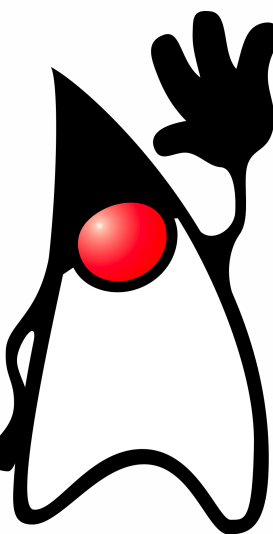




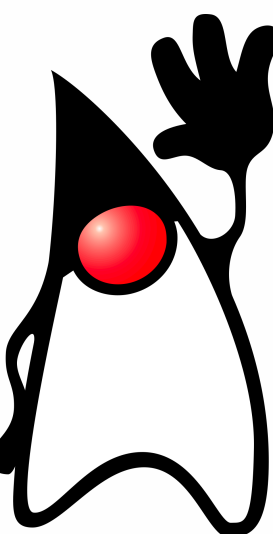
- Какие Вы знаете cross-cutting функциональности?



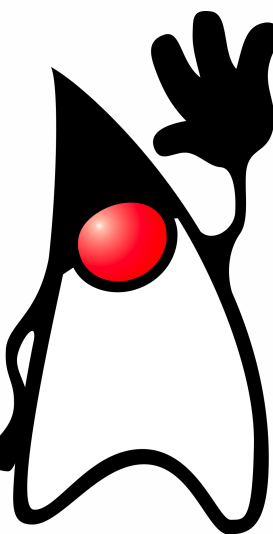
- Логгирование
  - Авторизация
  - Exception Handling
  - Транзакции
  - Кэширование
  - Benchmarking
  - И ПОЧТИ ВСЁ !!!!!
- БИЗНЕС-ЛОГИКИ ЗДЕСЬ НЕТ !!!



- Cross-cutting – слабое место ООП.
- Поэтому придумали некое расширение ООП – АОП (АОР)
- Аспектно-ориентированное программирование
- Это расширение, а не новая парадигма.
- Вот функциональное программирование – это новая парадигма.



- Аспект – это какой-то cross-cutting аспект Вашей программы
- Аспекты пишутся на разных технологиях
- Например, AspectJ
- Spring AOP – это базовая функциональность Spring для АОП (использующая AspectJ, кстати, но не только)
- На Spring AOP написана половина фреймворков
- Хотя сами Вы редко будете писать аспекты

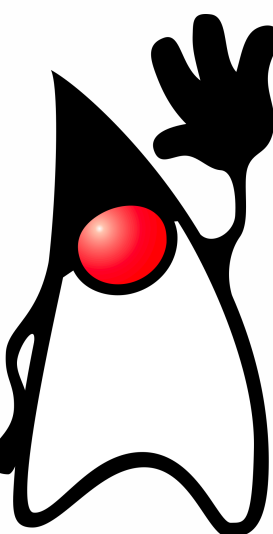




```
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class LoggingAspect {

    @Before("execution(* ru.otus.spring03.dao.BookDaoSimple.*(..))")
    public void logBefore(JoinPoint joinPoint) {
        System.out.println(
            "Вызов метода : " + joinPoint.getSignature().getName()
        );
    }
}
```



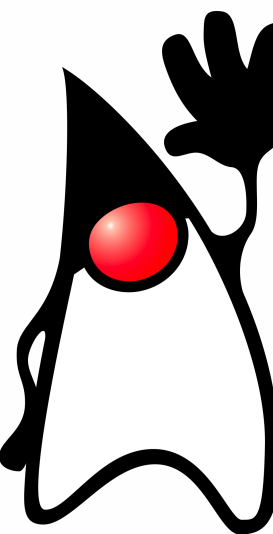
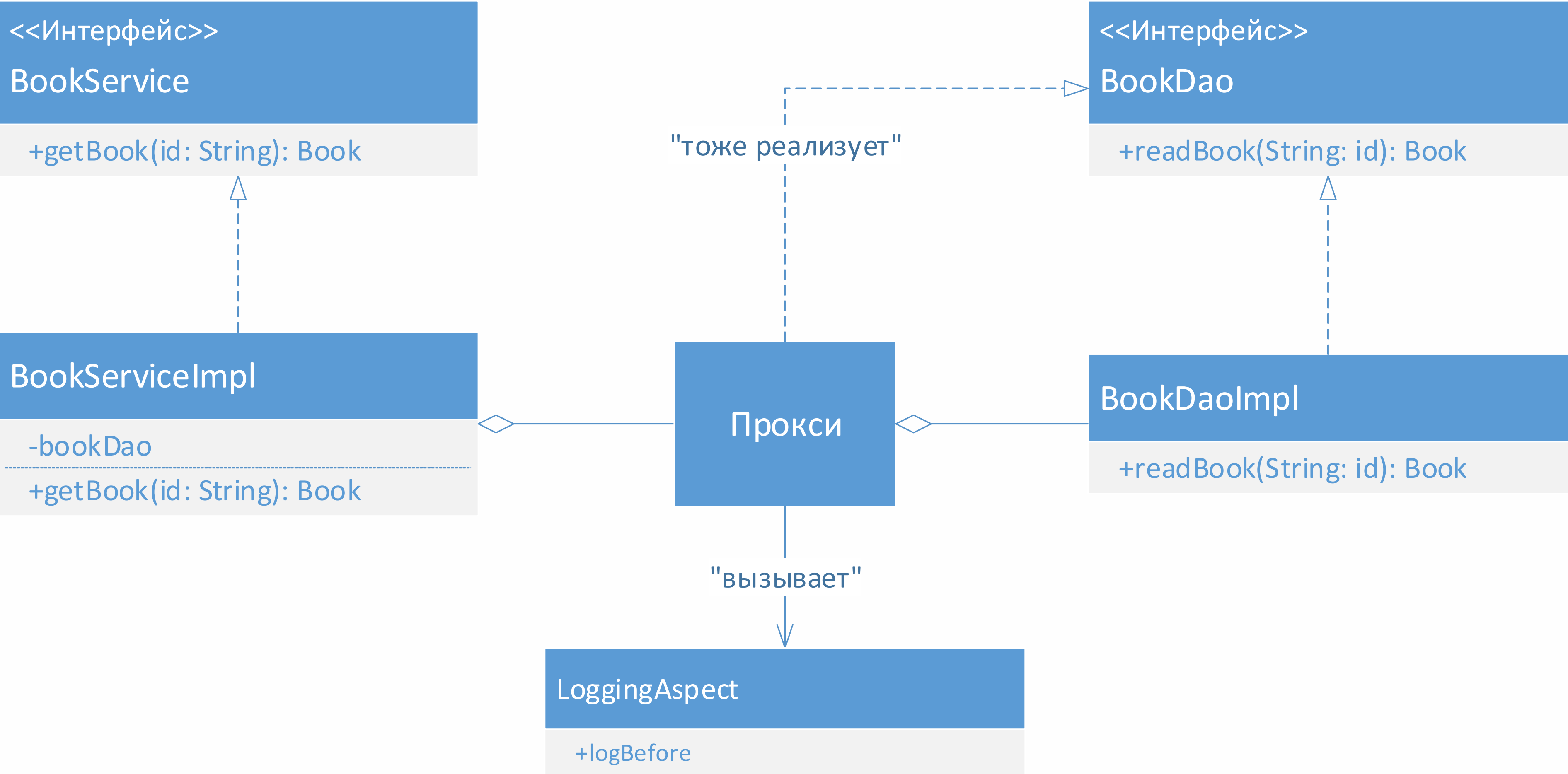
```
@Service
public class BookServiceImpl implements BookService {

    private final BookDao bookDao;

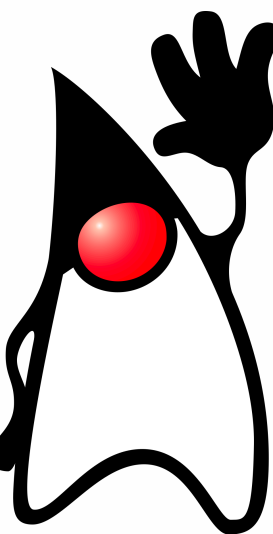
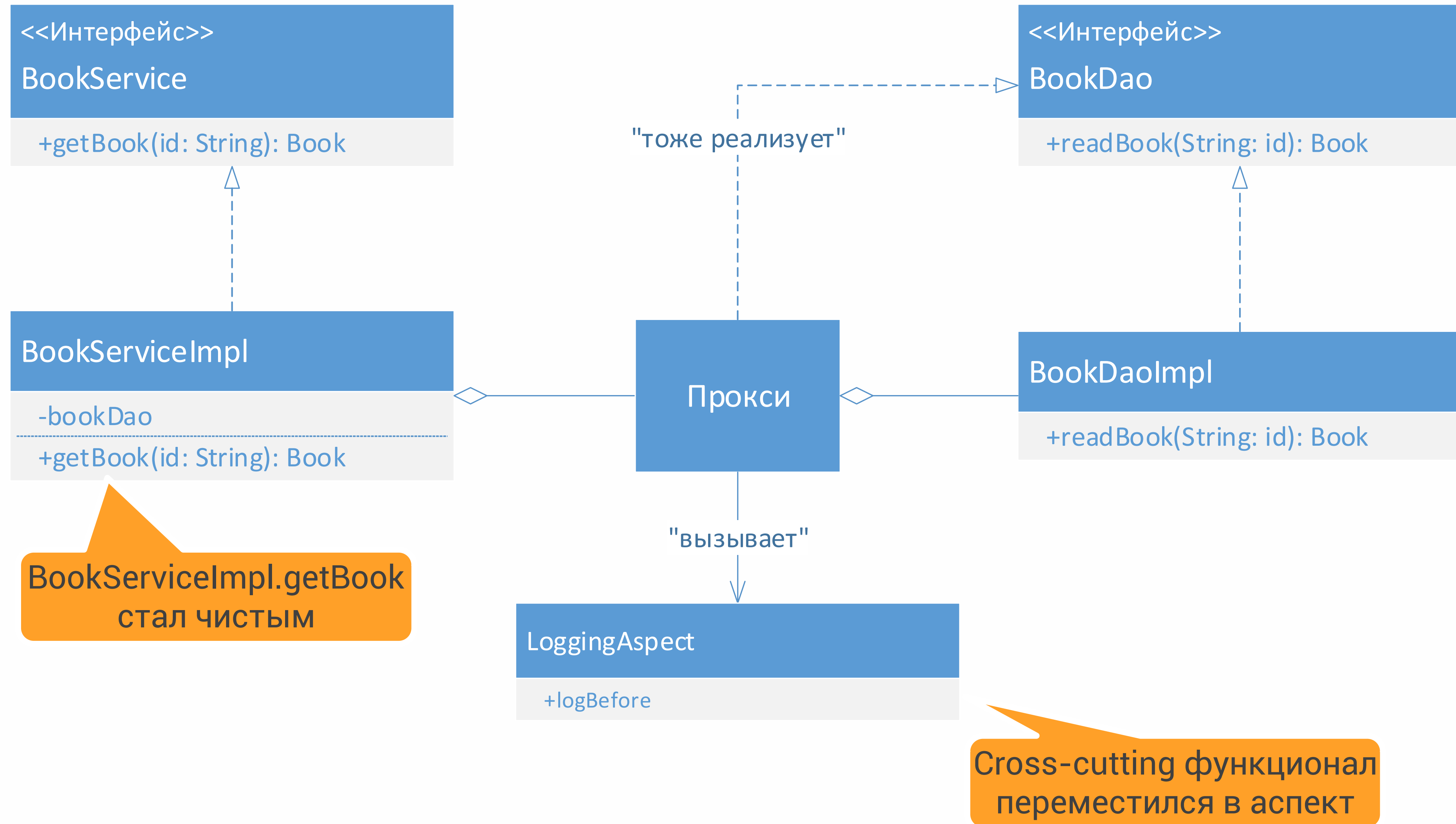
    public BookService(BookDao bookDao) {
        this.bookDao = bookDao;
    }

    @Override
    public Book getBook(Integer bookId) {
        Book book = bookDao.readBook(bookId);
        return book;
    }
}
```



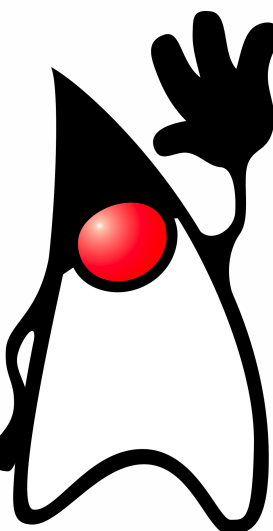


# Диаграмма классов



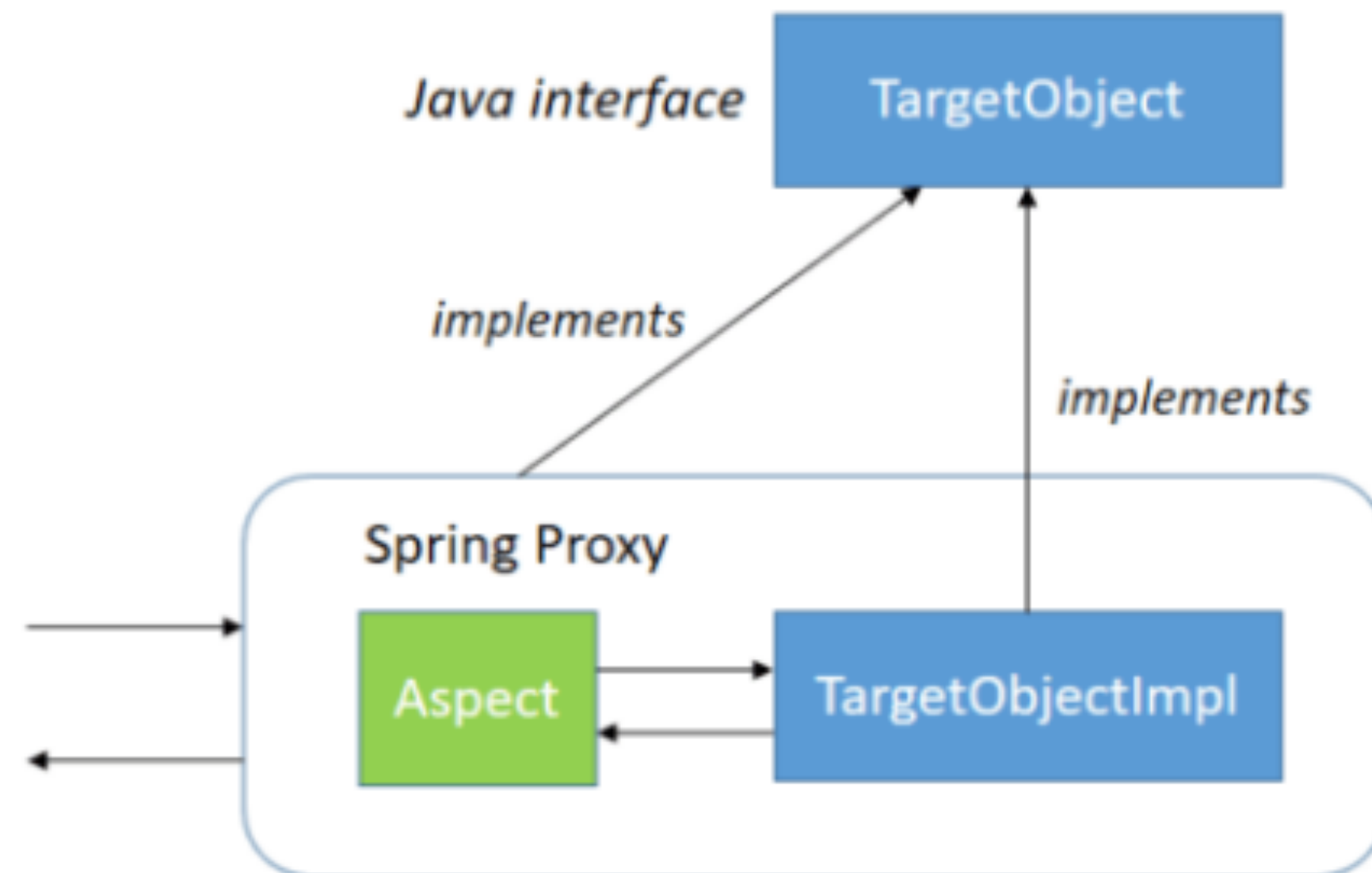
## Роемся во внутренностях:

- BookDao, BookDaoImpl
- В контексте лежит бин bookDaoImpl
- BookServiceImpl использует интерфейс BookDao
- Spring обманывает бин bookServiceImpl и даёт ему обёртку над bookDaoImpl, которая вызывает все аспекты, а потом вызывает метод bookDaoImpl
- Интерфейс этой обёртки – тоже BookDao
- Такая обёртка называется «прокси» (proxy).
- Обёрток вокруг метода может несколько

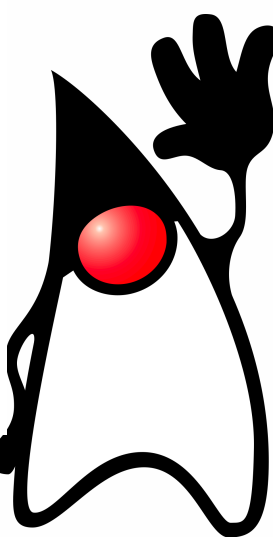
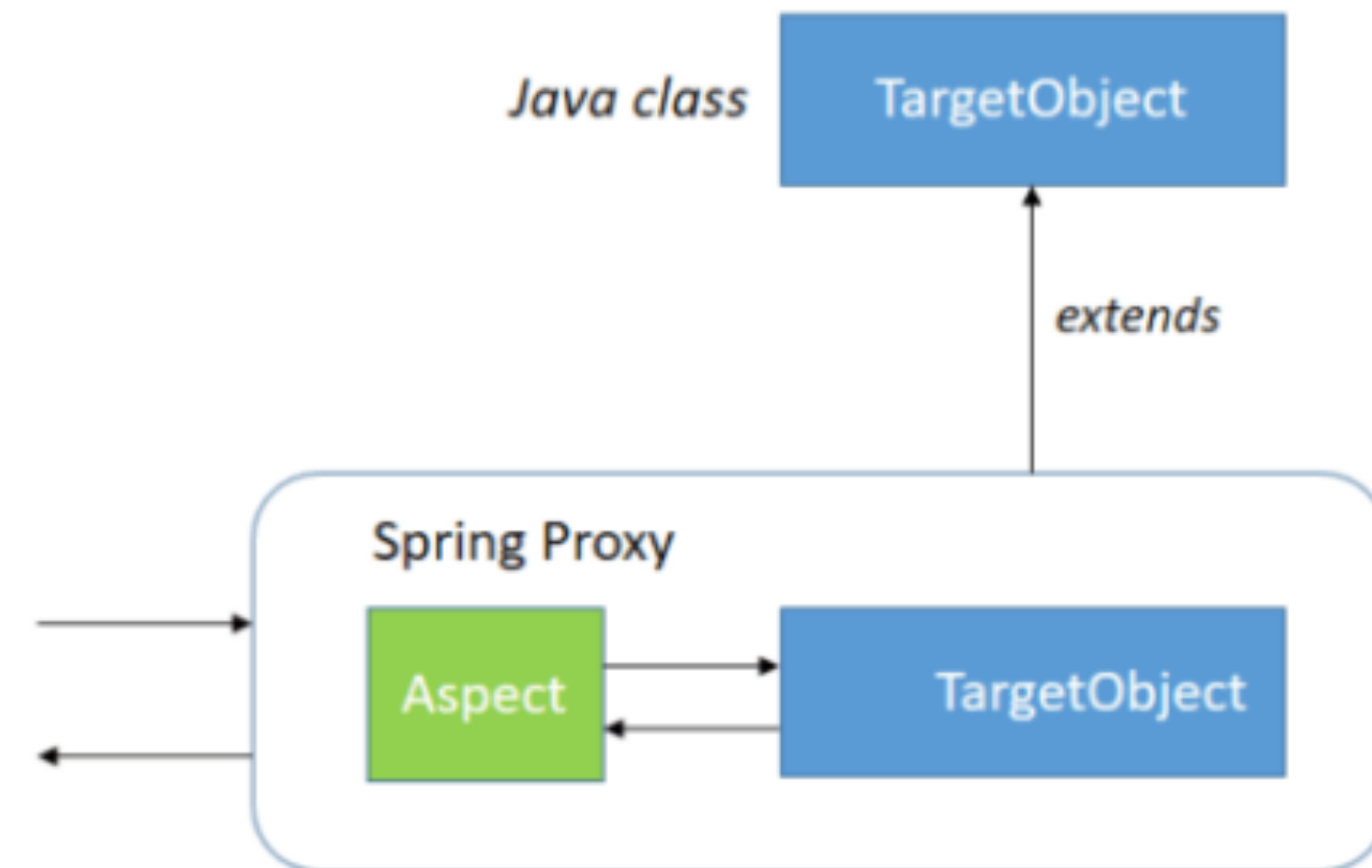


## Spring AOP Process

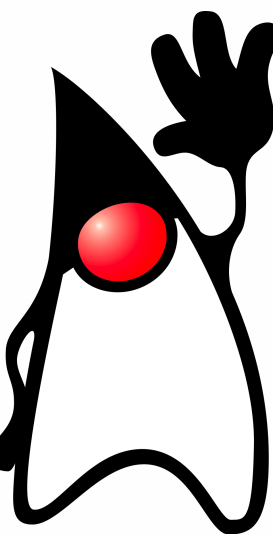
### JDK Proxy (interface based)



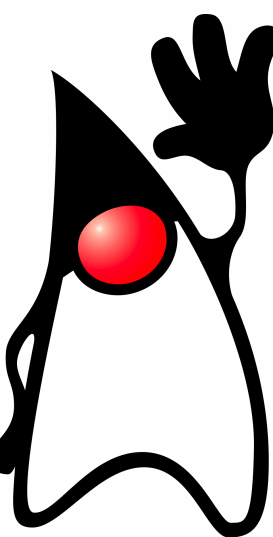
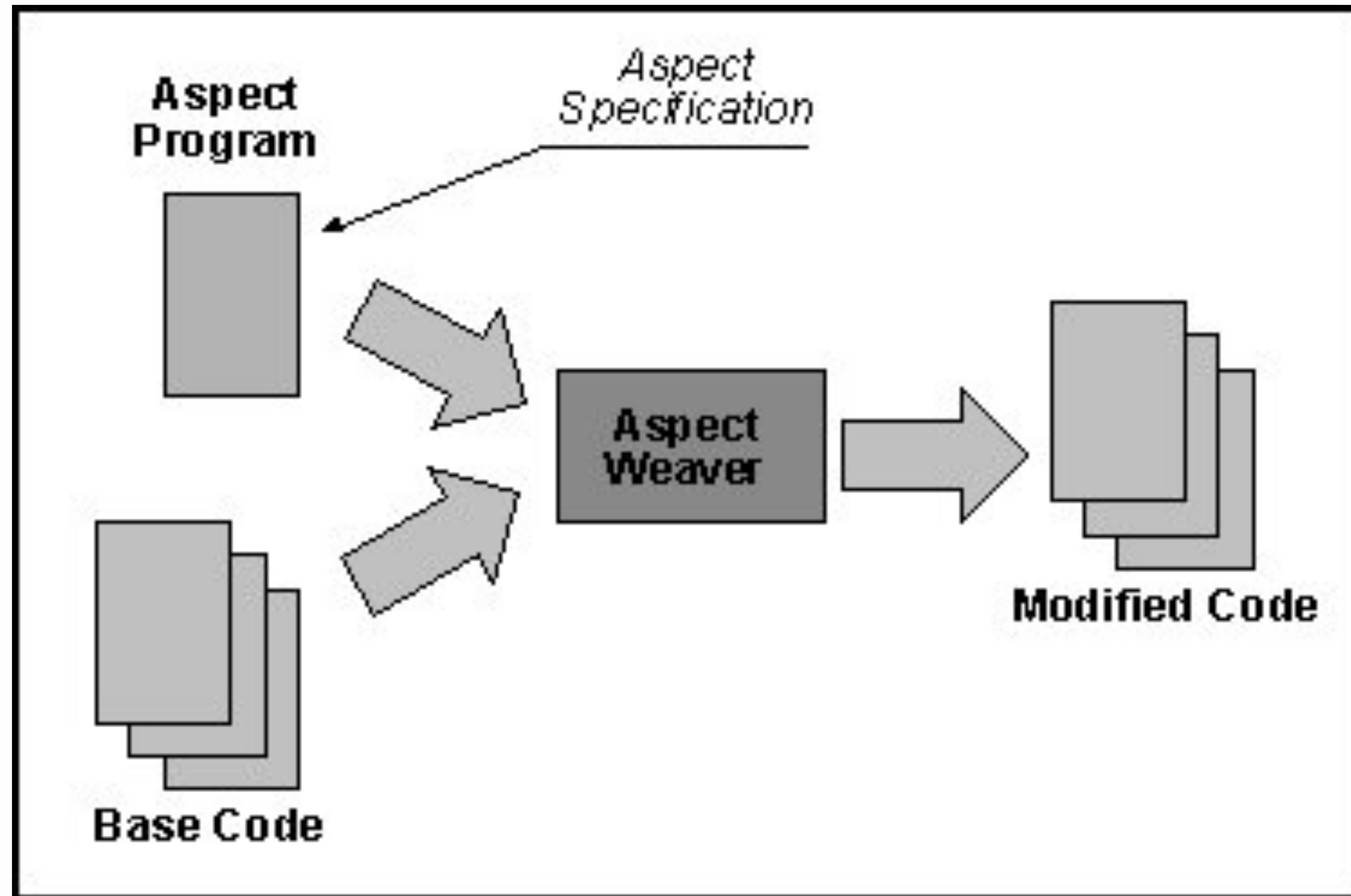
### CGLib Proxy (class based)



- JDK proxy умеет только публичные
- Cglib умеет оборачивать и protected
- Но Spring AOP даже с Cglib не делает всего, а имитирует JDK proxy
- Таким образом вне зависимости от типа прокси – у Вас одно и то же поведение.
- Т.е. можно не париться и знать что только публичные
- Но JPA будет работать и с protected 😊



# Weaving (aspectj-weaver)





# Ваши вопросы?

Если что — их можно задать  
ПОТОМ

```
package ru.otus.spring03.logging;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class LoggingAspect {

    @Before("execution(* ru.otus.spring03.dao.PersonDaoSimple.*(..))")
    public void logBefore(JoinPoint joinPoint) {
        System.out.println(
            "Вызов метода : " + joinPoint.getSignature().getName()
        );
    }
}
```




```
package ru.otus.spring03.logging;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class LoggingAspect {

    @Before("execution(* ru.otus.spring03.dao.PersonDaoSimple.*(..))")
    public void logBefore(JoinPoint joinPoint) {
        System.out.println(
            "Вызов метода : " + joinPoint.getSignature().getName()
        );
    }
}
```



```
package ru.otus.spring03.logging;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class LoggingAspect {

    @Before("execution(* ru.otus.spring03.dao.PersonDaoSimple.*(..))")
    public void logBefore(JoinPoint joinPoint) {
        System.out.println(
            "Вызов метода : " + joinPoint.getSignature().getName()
        );
    }
}
```

Advice



```
@Service
public class BookService implements BookService {

    private final BookDao bookDao;

    public BookService(BookDao bookDao) {
        this.bookDao = bookDao;
    }

    @Override
    public Book getBook(Integer bookId) {
        Book book = bookDao.readBook(bookId);
        return book;
    }
}
```

Join point

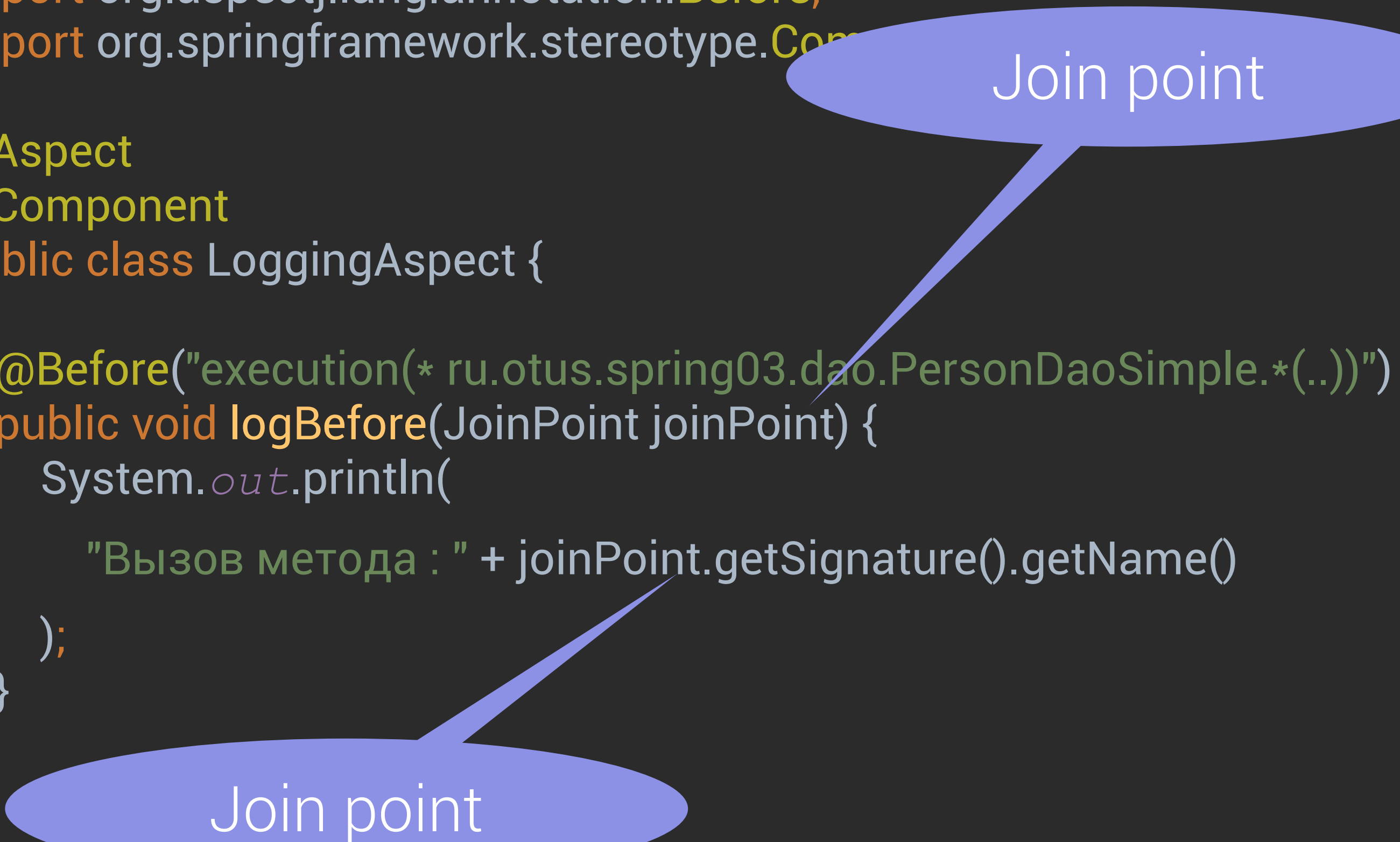


```
package ru.otus.spring03.logging;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class LoggingAspect {

    @Before("execution(* ru.otus.spring03.dao.PersonDaoSimple.*(..))")
    public void logBefore(JoinPoint joinPoint) {
        System.out.println(
            "Вызов метода : " + joinPoint.getSignature().getName()
        );
    }
}
```



```
package ru.otus.spring03.logging;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

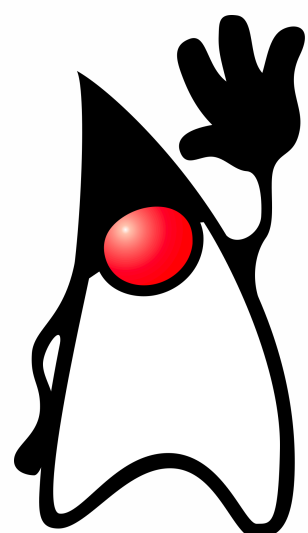
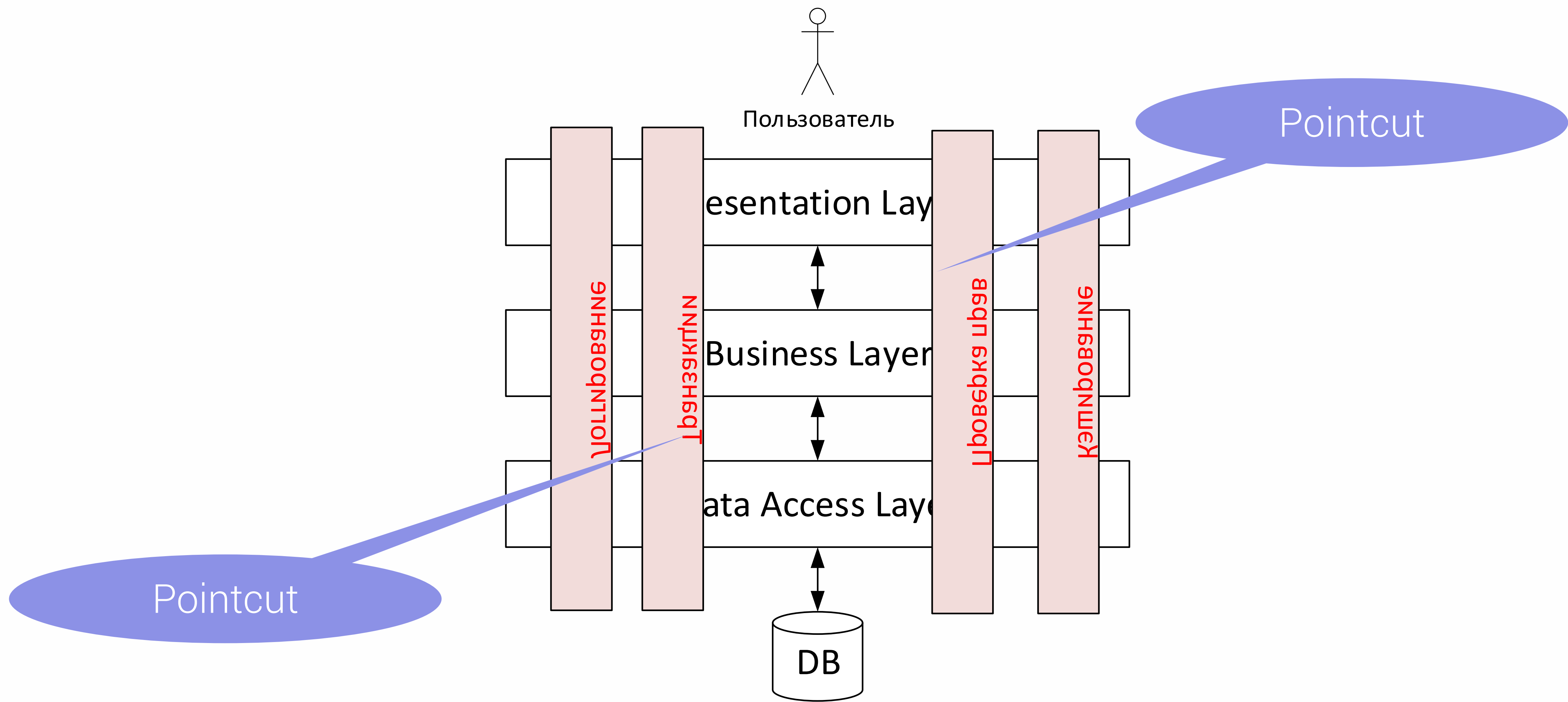
@Aspect
@Component
public class LoggingAspect {

    @Before("execution(* ru.otus.spring03.dao.PersonDaoSimple.*(..))")
    public void logBefore(JoinPoint joinPoint) {
        System.out.println(
            "Вызов метода : " + joinPoint.getSignature().getName()
        );
    }
}
```



Pointcut







```
package ru.otus.spring03.logging;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;

@Order(10)
@Aspect
@Component
public class LoggingAspect {

    @Before("execution(* ru.otus.spring03.dao.PersonDaoSimple.*(..))")
    public void logBefore(JoinPoint joinPoint) {
        System.out.println(
            "Вызов метода : " + joinPoint.getSignature().getName()
        );
    }
}
```

```
package ru.otus.spring03;

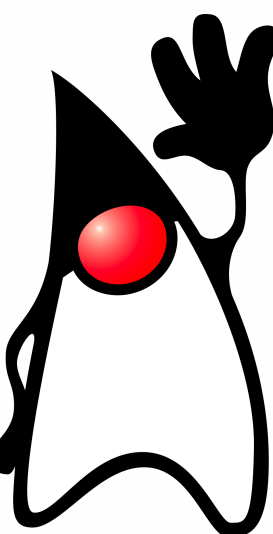
import org.springframework.context.annotation.*;
import ru.otus.spring03.domain.Person;
import ru.otus.spring03.service.PersonService;

@EnableAspectJAutoProxy
@Configuration
@ComponentScan
public class Main {

    public static void main(String[] args) {
        // ...
    }
}
```

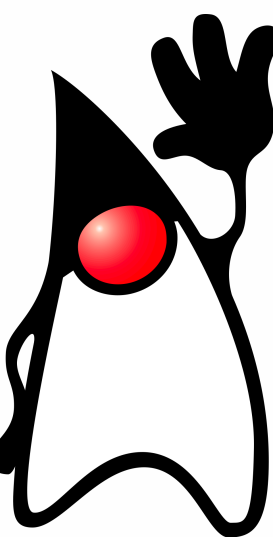


- Maven-dependency aspectj (spring-aop транзитивна spring-context).
- Написать аспект
- @Aspect @Component на аспекте!
- @Before – аннотации advice-ов
- С pointcut-ом
- @EnableAspectJAutoProxy на классе конфигурации (со Spring Boot 2.1.0 вроде не нужно ставить)

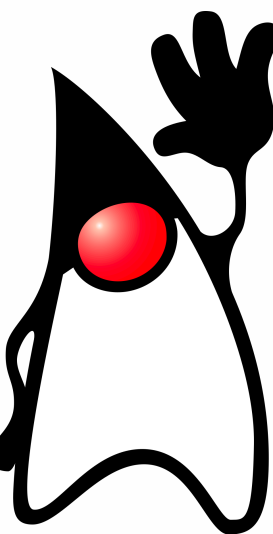


В каком @Order нужно «оборачивать» следующие аспекты:

- Логгирование
- Авторизация
- Exception Handling
- Транзакции
- Кэширование
- Benchmarking



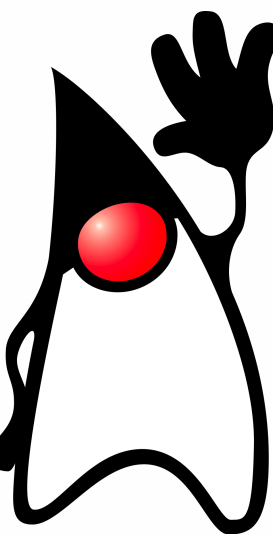
- Cross-cutting – вспомогательная пересекающая функциональность
- AOP – расширение OOP, только для ортогональной функциональности
- Можно писать на разных технологиях, есть Spring AOP, использующий AspectJ



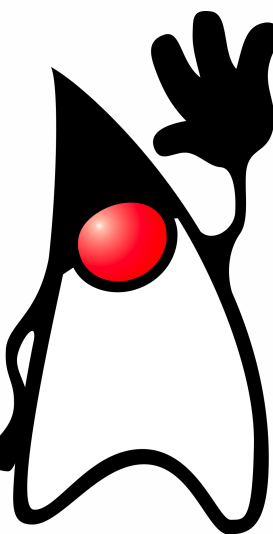
# Ваши вопросы?

Если что — их можно задать  
ПОТОМ

- Advice – это те действия, которые выполняются помимо Jointpoint
- В аспекте может быть несколько Advice-ов
- Напомню, что на метод может быть навешено несколько Advice-ов и, возможно из разных аспектов
- Аспекты можно упорядочивать, advice-ы не всегда
- Advice бывают разные

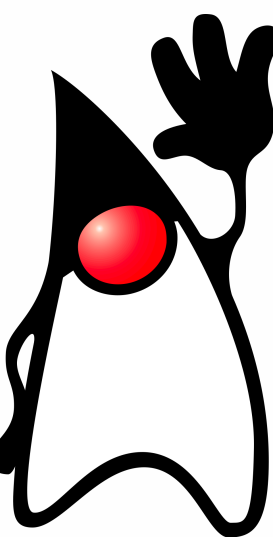


- @Before – выполняется перед точкой входа (есть возможность и не заходить в метод);
- @After – выполняется после точки входа (даже после исключения);
- @Around – до и после (вокруг);
- @AfterReturning – после того, как точка входа завершилась корректно;
- @AfterThrowing – в случае исключения в точке входа.





- Предупрежу, что @AfterThrowing и @AfterReturning портят картину мира машин Тьюринга.
- Можно придумать такую комбинацию упорядоченных аспектов и @AfterThrowing и @AfterReturning, что AspectJ



```
package com.mkyong.aspect;

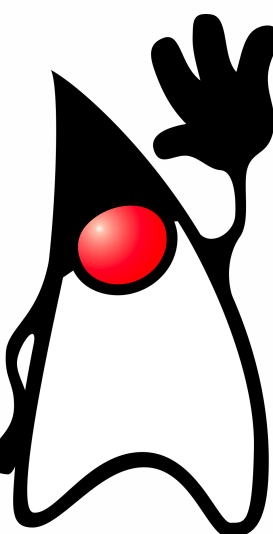
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class LoggingAspect {

    @Before("execution(* com.mkyong.customer.bo.CustomerBo.addCustomer(..))")
    public void logBefore(JoinPoint joinPoint) {

        System.out.println("logBefore() is running!");
        System.out.println("hijacked : " + joinPoint.getSignature().getName());
        System.out.println("*****");
    }

}
```



```
package com.mkyong.aspect;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.After;

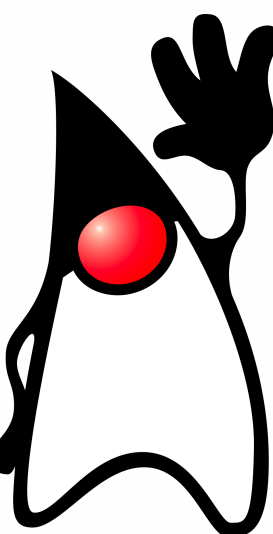
@Aspect
public class LoggingAspect {

    @After("execution(* com.mkyong.customer.bo.CustomerBo.addCustomer(..))")
    public void logAfter(JoinPoint joinPoint) {

        System.out.println("logAfter() is running!");
        System.out.println("hijacked : " + joinPoint.getSignature().getName());
        System.out.println("*****");

    }

}
```



# @AfterReturning advice

```
package com.mkyong.aspect;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterReturning;

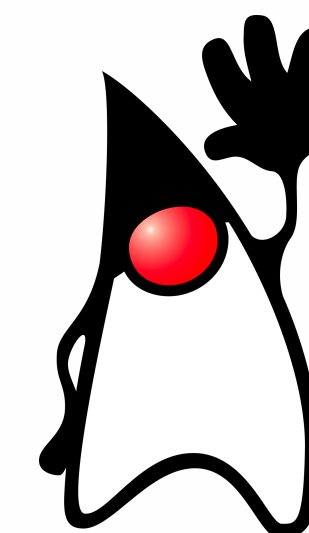
@Aspect
public class LoggingAspect {

    @AfterReturning(
        pointcut = "execution(* com.mkyong.customer.bo.CustomerBo.addCustomerReturnValue(..))",
        returning= "result")
    public void logAfterReturning(JoinPoint joinPoint, Object result) {

        System.out.println("logAfterReturning() is running!");
        System.out.println("hijacked : " + joinPoint.getSignature().getName());
        System.out.println("Method returned value is : " + result);
        System.out.println("*****");

    }

}
```





# @AfterThrowing advice

```
package com.mkyong.aspect;

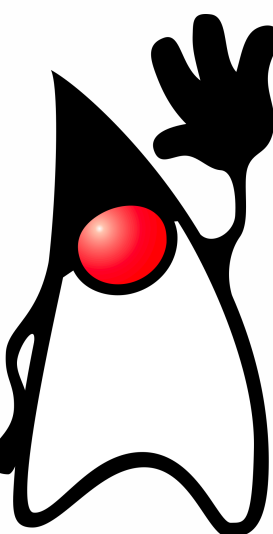
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterThrowing;

@Aspect
public class LoggingAspect {

    @AfterThrowing(
        pointcut = "execution(* com.mkyong.customer.bo.CustomerBo.addCustomerThrowException(..))",
        throwing = "error")
    public void logAfterThrowing(JoinPoint joinPoint, Throwable error) {

        System.out.println("logAfterThrowing() is running!");
        System.out.println("hijacked : " + joinPoint.getSignature().getName());
        System.out.println("Exception : " + error);
        System.out.println("*****");

    }
}
```



# @Around advice

```
package com.mkyong.aspect;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Around;

@Aspect
public class LoggingAspect {

    @Around("execution(* com.mkyong.customer.bo.CustomerBo.addCustomerAround(..))")
    public void logAround(ProceedingJoinPoint joinPoint) throws Throwable {

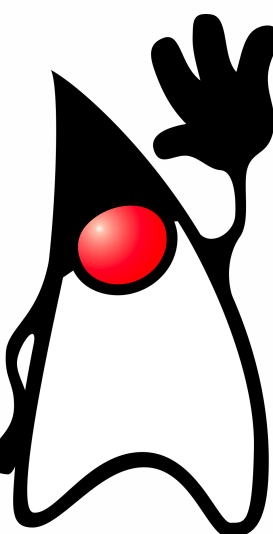
        System.out.println("logAround() is running!");
        System.out.println("hijacked method : " + joinPoint.getSignature().getName());
        System.out.println("hijacked arguments : " + Arrays.toString(joinPoint.getArgs()));

        System.out.println("Around before is running!");
        joinPoint.proceed(); //continue on the intercepted method
        System.out.println("Around after is running!");

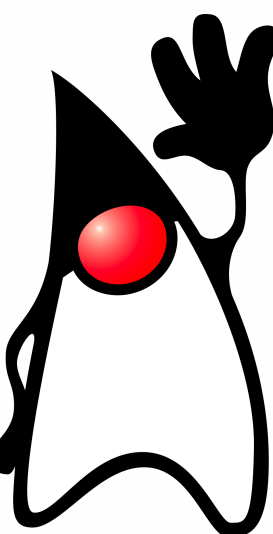
        System.out.println("*****");

    }

}
```

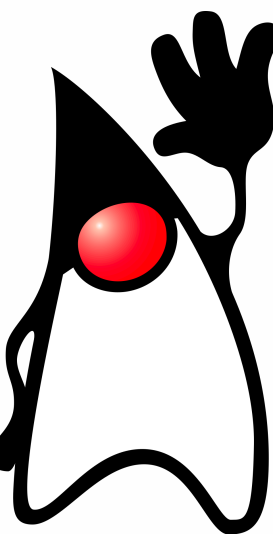


- Вы можете глушить Exception-ы
- Вы можете генерировать Exceptions
- Вы можете вызывать метод по условию
- Вы можете изменять аргументы и возвращаемые значения
- Половина Spring-а написана на Advice



## Какие это advic-ы (@Before... )?

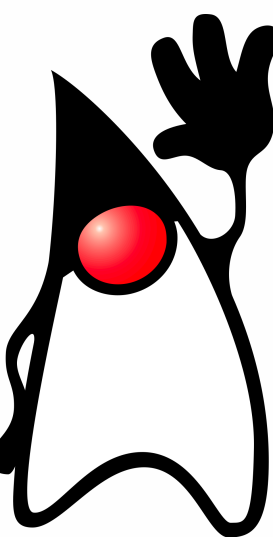
- Логгирование
- Авторизация
- Exception Handling





Какие это advic-ы (@Before... )?

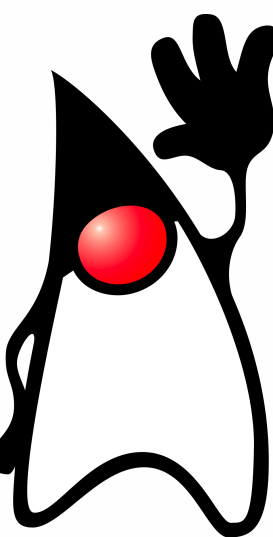
- Транзакции
- Кэширование
- Benchmarking



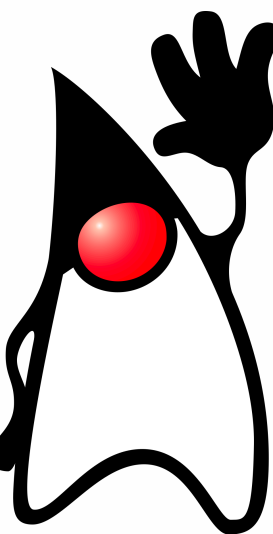
# Ваши вопросы?

Если что — их можно задать  
ПОТОМ

- Pointcuts – язык описания мест Join point
- В Spring AOP принят синтаксис AspectJ
- С pointcut-ами можно вытворять многое



- Предлагаю посмотреть на экран и попробовать поугадывать
- 



Перед каким методом обернётся Advice?

```
@Before("execution(public String ru.otus.Dao.findById(Long))")
```

Что означают следующие Pointcuts?

```
"execution(public String ru.otus.Dao.findById(Long)) "
```

```
"execution(* ru.otus.dao.FooDao.*(..)) "
```

```
"execution(* ru.otus.dao.*.*(..)) "
```

Что означают следующие Pointcuts?

```
"execution(public String ru.otus.Dao.findById(Long)) "
```

– метод прям конкретный публичный

```
"execution(* ru.otus.dao.FooDao.*(..)) "
```

– все публичные методы в классе FooDao

```
"execution(* ru.otus.dao.*.*(..)) "
```

– все публичные методы во всех классах пакета ru.otus.dao

Что означают следующие Pointcuts?

```
"execution(* ru.otus.dao.FooDao.*(..))"
```

```
"within(ru.otus.dao.FooDao)"
```

```
"within(ru.otus.dao..*)"
```



Что означают следующие Pointcuts?

```
"execution(* ru.otus.dao.FooDao.*(..))"
```

- публичные методы в классе

```
"within(ru.otus.dao.FooDao)"
```

- то же самое

```
"within(ru.otus.dao..*)"
```

- все методы внутри классов внутри пакета

Что означают следующие Pointcuts?

```
"target(ru.otus.MyInterface) "
```

```
"@target(org.springframework.stereotype.Repository) "
```

```
"@annotation(org.baeldung.aop.annotations.Loggable) "
```

Что означают следующие Pointcuts?

```
"target(ru.otus.MyInterface) "
```

– методы в классах реализующие интерфейс

```
"@target(org.springframework.stereotype.Repository) "
```

– методы в классах. Помеченных аннотацией

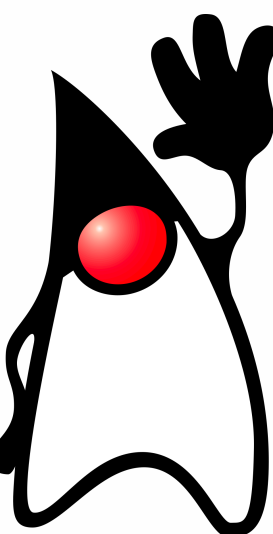
```
"@annotation(org.baeldung.aop.annotations.Loggable) "
```

– методы, помеченные аннотацией

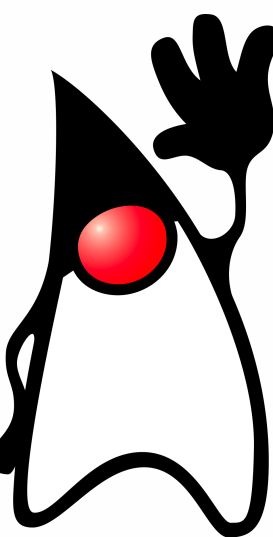
```
"@target (org.springframework.stereotype.Repository) "
```

```
"@annotation (org.baeldung.aop.annotations.Loggable) "
```

- Являются предпочтительными
- @Transactional, @Repository, @Controller так и работают
- Правила приличия –они делают наличие аспекта явным



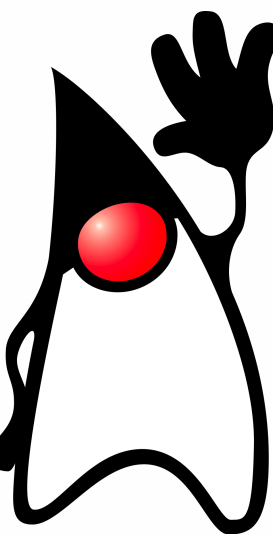
- Pointcuts
- AspectJ Pointcuts
- Пара слов
- Подробнее про язык pointcut-ов: <http://www.baeldung.com/spring-aop-pointcut-tutorial>



# Ваши вопросы?

Если что — их можно задать  
ПОТОМ

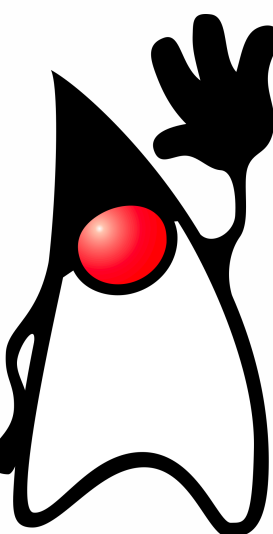
- Залоггировать факт вызова методов PersonDaoSimple.
- (сам класс PersonDaoSimple трогать не нужно)

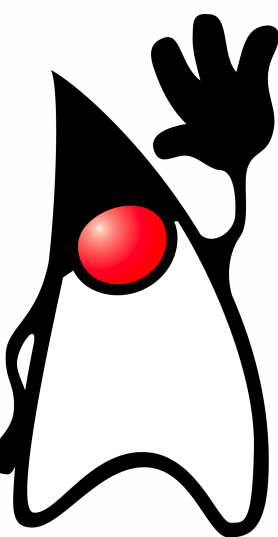
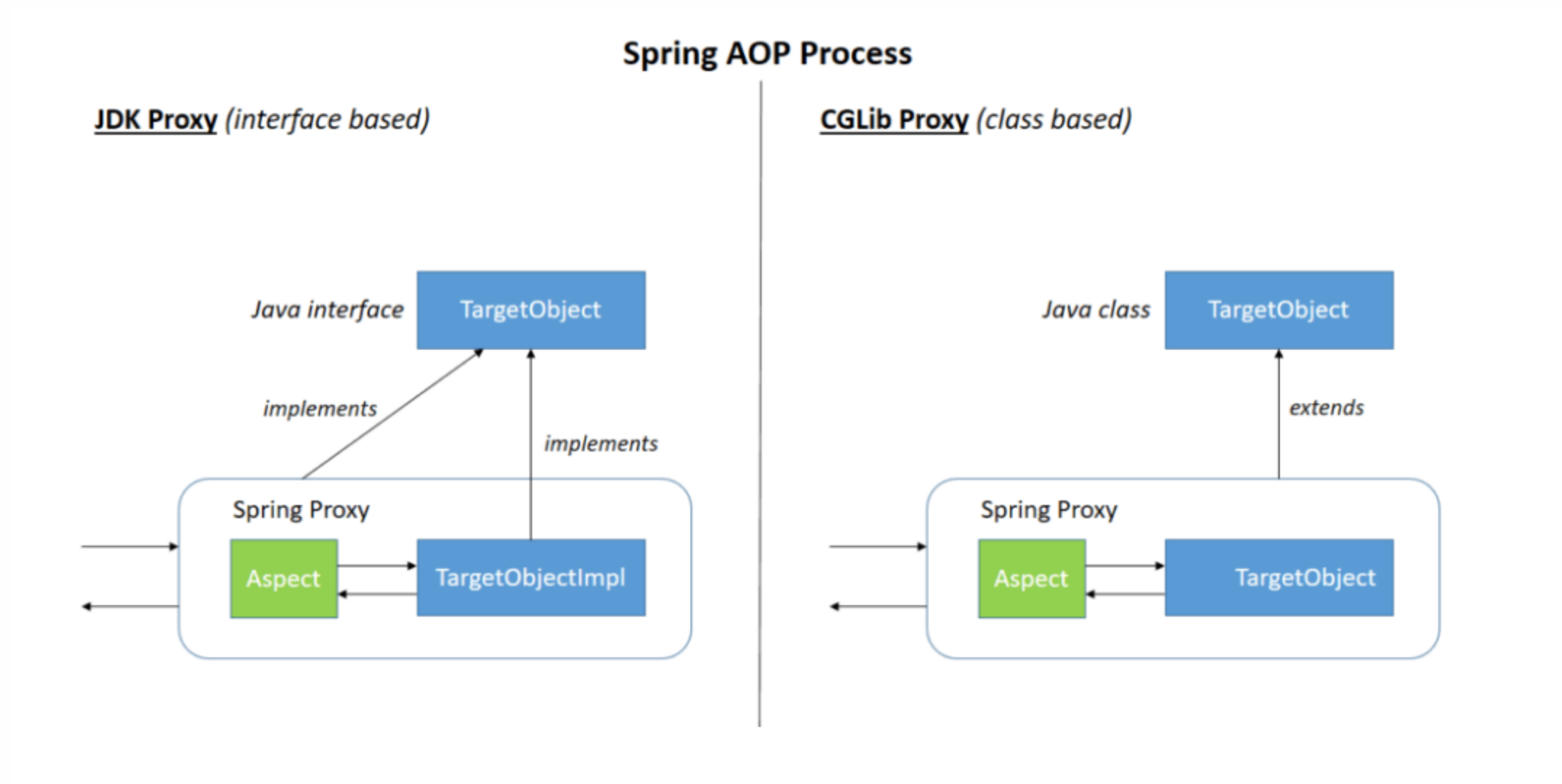


**Конец упражнения!**  
**Ваши вопросы?**

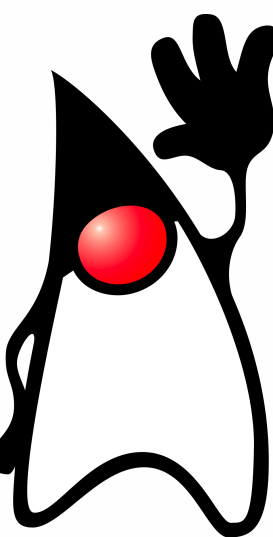


- Только для методов бинов, которые лежат к контексте!
- Только для публичных методов! На private – не заработает!
- Только если Вы вызываете из одного бина другой бин!
- Cglib-проху предоставляет такие возможности, но Spring сделал так, чтобы не было разницы между тем, как проху созданы
- У спринга есть и другие способы создания прокси 😊

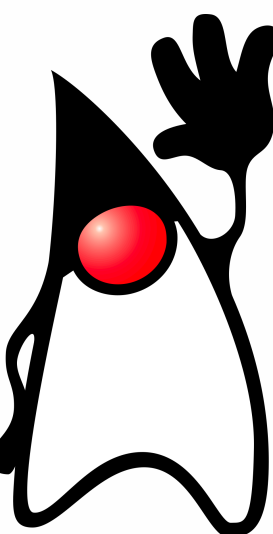




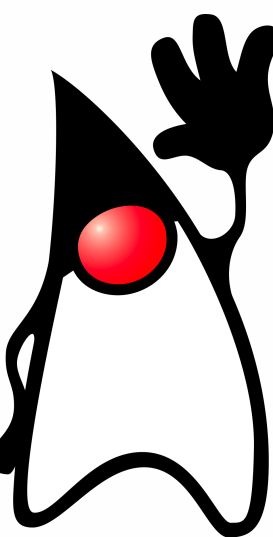
- Аспекты тяжело воспринимаются
- Тяжело пишутся хорошие
- Ещё тяжелее дебажатся
- А тем более покрываются тестами
- Ошибки в аспектах почти невозможно найти



- Делайте аспекты как можно более прозрачными
  - @Annotation pointcut-ы отличный способ
  - Аспекты пишутся опытными программистами.
  - Отлаживаются и покрываются тестами до того как появится в коде
  - Используя спринговую аннотацию помните, что чаще всего там аспект
- 
- Аспект, как паттерн, – средство решения проблемы, а не её создания ☺

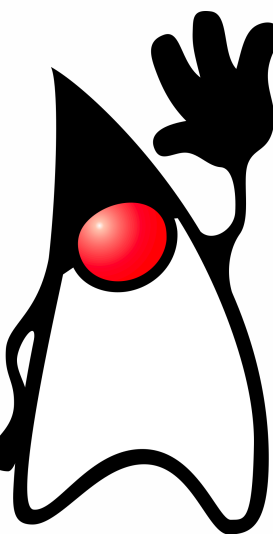


- Не делайте много неявного в аспекте
- Не навешивайте на абсолютно все методы
- Аспект по идее должен быть быстрый
- Ни в коем случае не пишите бизнес-логику на аспектов, они только для cross-cutting



## Примеры использования аспектов:

- Вы хотите залоггировать, какой пользователь вызывает @Deprecated методы
- Вы не хотите пускать сервисы из чёрного списка к метода
- Аудит лог – что изменилось, какие сервисы в итоге вызвались пользователем
- Настройки дополнительной обработки (валидации)



**Спасибо за внимание!**

