**<http://www.spring-source.ru/>**

**Лабораторная работа №2**

# Теоретическая часть

Рассмотрим простое приложение по сложению двух чисел и выводу результата. Все приложение можно написать при помощи лишь одного класса Calculate. Что, если нам понадобилось изменить сложение в методе operate на умножение или выводить результат в методе showResult не на экран, а в файл. В этих случаях код программы нужно постоянно менять.

**public** **class** Calculate {

**public** Calculate() {

}

**public** **static** **void** main(String[] args) {

Calculate calc = **new** Calculate();

calc.execute(10L, 15L);

}

**private** **long** operate(**long** op1, **long** op2) {

**return** op1 + op2;

}

**private** String getOpsName() {

**return** " plus ";

}

**public** **void** execute(**long** op1, **long** op2) {

System.***out***.println ("The result of " + op1 + getOpsName()

+ op2 + " is " + operate(op1, op2) + "!");

}

}

Нам нужно разъединить логику приложения от математических операций при помощи интерфейсов. Первый интерфейс будет называться Operation. В нем мы объявим два метода operate(long op1, long op2) и getOpsName().

**public** **interface** Operation {

**long** operate(**long** op1, **long** op2);

String getOpsName();

}

После этого, если мы захотим сделать сложение, то мы просто будем реализовывать этот интерфейс:

**public** **class** OpAdd **implements** Operation {

**public** OpAdd() {

}

**public** String getOpsName() {

**return** " plus ";

}

**public** **long** operate(**long** op1, **long** op2) {

**return** op1 + op2;

}

}

Для других математических действий все делаем также:

**public** **class** OpMultiply **implements** Operation {

**public** OpMultiply() {

}

**public** String getOpsName() {

**return** " times ";

}

**public** **long** operate(**long** op1, **long** op2) {

**return** op1 \* op2;

}

}

Тогда наш класс Calculate будет выглядеть следующим образом:

**public** **class** Calculate {

**private** Operation ops = **new** OpAdd();

**public** **static** **void** main(String[] args) {

Calculate calc = **new** Calculate();

calc.execute(10L, 15L);

}

**public** **void** execute(**long** op1, **long** op2) {

System.***out***.println("The result of " + op1 + ops.getOpsName()

+ op2 + " is " + ops.operate(op1, op2) + "!");

}

}

Теперь мы имеем OpAdd, OpMultiply и основной класс Calculate. В главном классе мы определяем один из Op\* классов и используем его.

Если мы хотим изменить действия, то нужно просто поменять названия классов. Здесь есть одна небольшая проблема: код Calculate должен быть изменен или перекомпилирован если вам нужно изменить математическую операцию или по другому вывести результат. Код, который создает экземпляр OppAdd жестко закодирован в Calculate.

Способность соединять и рассоединять повторно используемые Java бины в приложении – это важная сторона Spring framework.

Суть в том, что тогда Calculate не будет работать с экземплярами Operation, а делегирует эту задачу Spring контейнеру. Spring контейнер, в свою очередь, читает конфигурационный файл и сам конфигурирует Calculate.

Однако в этом случае класс Calculate должен соответствовать иметь getter’ы и setter’ы своих полей (в более новых версиях Spring не обязательно) и иметь конструктор по умолчанию:

**public** **class** Calculate {

**private** Operation ops;

**public** **void** setOps(Operation ops) {

**this**.ops = ops;

}

**public** **static** **void** main(String[] args) {

ConfigurableApplicationContext context =

**new** ClassPathXmlApplicationContext("beans.xml");

Calculate calc = (Calculate) context.getBean("opsbean");

calc.execute(10L, 15L);

context.close();

}

**public** **void** execute(**long** op1, **long** op2) {

System.***out***.println("The result of " + op1 + ops.getOpsName()

+ op2 + " is " + ops.operate(op1, op2) + "!");

}

}

Конструктор ClassPathXmlApplicationContext берет в качестве аргумента файл описания контекста. Этот файл называется beans.xml только лишь в наших примерах, вы же можете называть его как хотите, но следите, чтобы расширение файла было XML. Файл beans.xml – это конфигурационный файл, описывающий, как связывать объекты вместе. Здесь показан beans.xml файл:

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-2.0.xsd"*>

<bean id=*"add"* class=*"by.vstu.zamok.lab2.calc.OpAdd"* />

<bean id=*"multiply"* class=*"by.vstu.zamok.lab2.calc.OpMultiply"* />

<bean id=*"opsbean"* class=*"by.vstu.zamok.lab2.calc.Calculate"*>

<property name=*"ops"* ref=*"add"* />

</bean>

</beans>

Для конфигурации приложения при помощи аннотаций требуется создать класс в пакете **config:**

@Configuration

@ComponentScan("by.vstu.zamok.lab2")

**public** **class** AppConfig {

}

И Calculate будет выглядеть следующим образом:

@Component

**public** **class** Calculate {

@Autowired

@Qualifier("add")

**private** Operation ops;

**public** **void** setOps(Operation ops) {

**this**.ops = ops;

}

**public** **static** **void** main(String[] args) {

ConfigurableApplicationContext context =

**new** AnnotationConfigApplicationContext(AppConfig.**class**);

Calculate calc = context.getBean(Calculate.**class**);

calc.execute(10L, 15L);

context.close();

}

**public** **void** execute(**long** op1, **long** op2) {

System.***out***.println("The result of " + op1 + ops.getOpsName()

+ op2 + " is " + ops.operate(op1, op2) + "!");

}

}

Аннотация @Component указывает на то что объект данного класса должен быть создан и помещен в контейнер, а @Autowired указывает на то, что данное поле должно быть внедрено контейнером. Компонентам, предназначенным для внедрения по одинаковому интерфейсу Operation, дается имя:

@Component("add")

**public** **class** OpAdd **implements** Operation {

**public** OpAdd() {

}

**public** String getOpsName() {

**return** " plus ";

}

**public** **long** operate(**long** op1, **long** op2) {

**return** op1 + op2;

}

}

@Component("mul")

**public** **class** OpMultiply **implements** Operation {

**public** OpMultiply() {

}

**public** String getOpsName() {

**return** " times ";

}

**public** **long** operate(**long** op1, **long** op2) {

**return** op1 \* op2;

}

}

Если для внедрения возможен только один кандидат (например, если мы захотим внедрить Calculate), аннотация @Qualifier не требуется.

# Практическая часть

## Настройка контейнера при помощи xml конфигурации

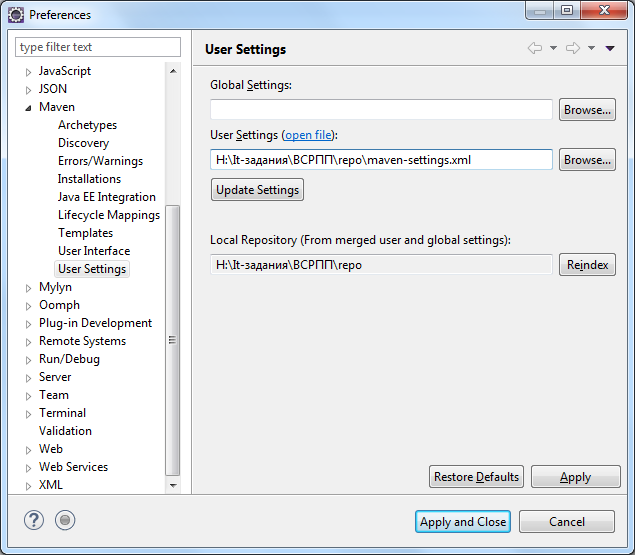
**Скопировать папку repo к себе на компьютер. Отредактировать файл maven-settings.xml:** указать местоположение репозитория в теле тега **<localRepository>**

****

**Настроить местоположение локального репозитория Maven.**

Window🡪Preferences🡪Maven🡪User Settings.

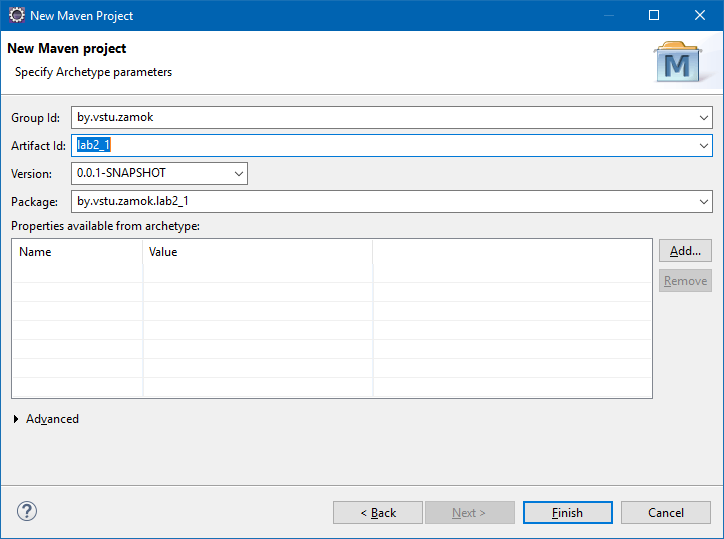
Указываем пользовательский файл конфигурации при помощи кнопки Browse:



После этого местоположение локального репозитория изменится на требуемое.

**Создать Maven проект**

File 🡪 New 🡪 Other 🡪 Maven 🡪 Maven Project 🡪 Next 🡪 ввести в поле **Filter** quickstart 🡪 выбрать **maven-archetype-quickstart** 🡪 Next  
Указать Group ID (разработчик) и Artifact Id (имя проекта) и нажать Finish



**Указать springframework в pom.xml**

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>by.vstu.zamok</groupId>

<artifactId>lab2</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>lab2</name>

<url>http://maven.apache.org</url>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>4.3.10.RELEASE</version>

</dependency>

</dependencies>

<build>

<pluginManagement>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.2</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

</plugins>

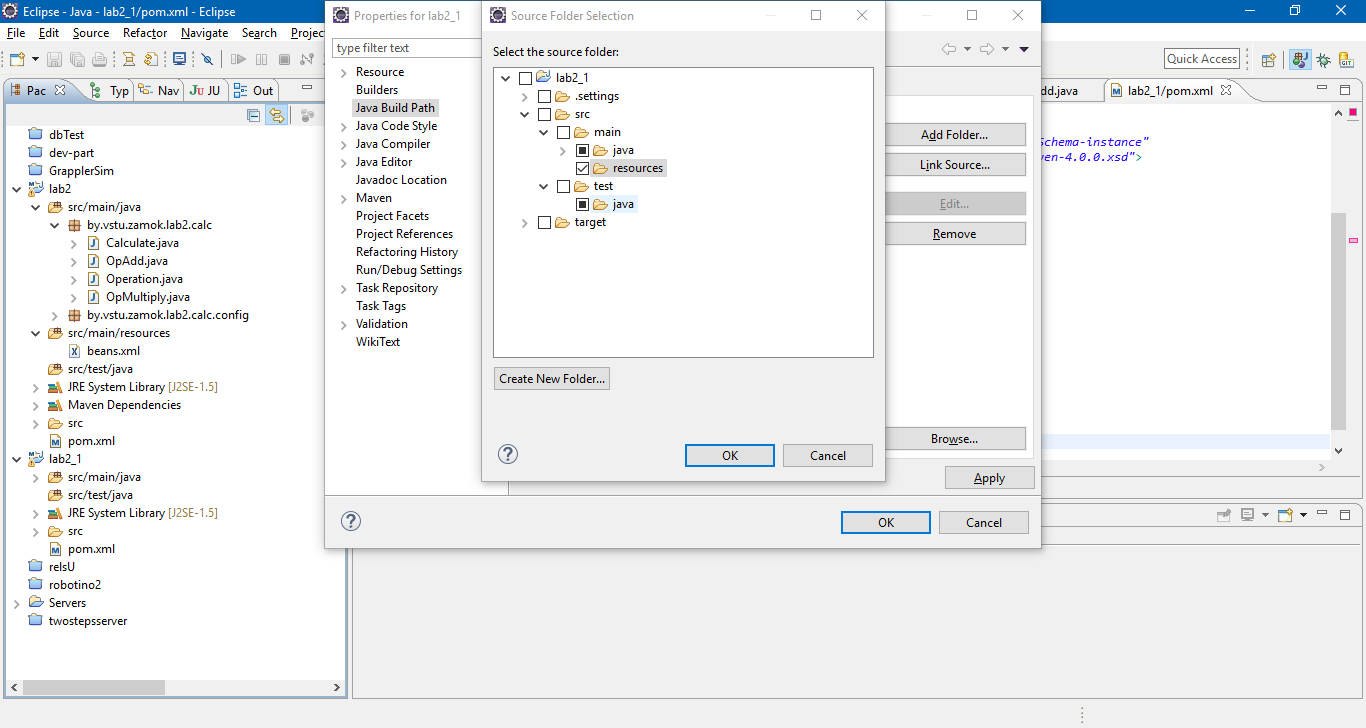
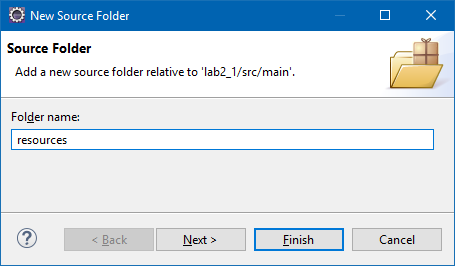
</pluginManagement>

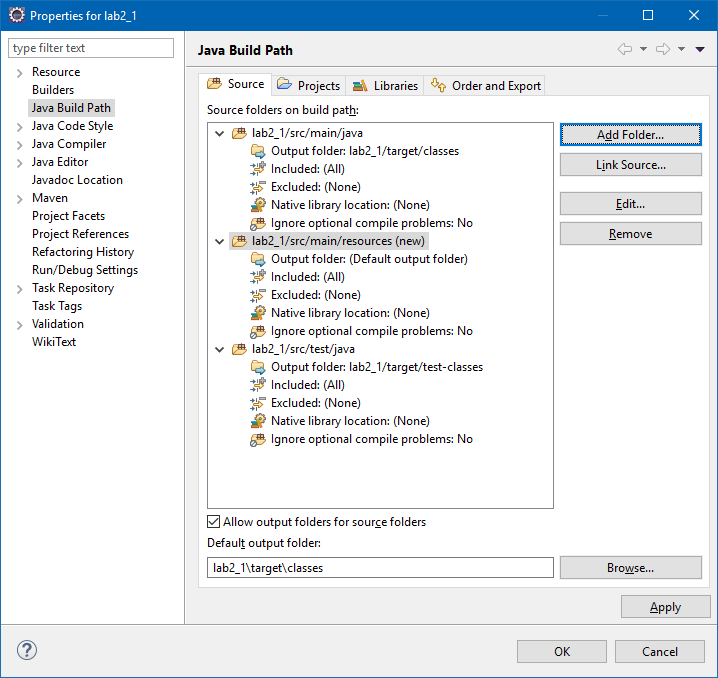
</build>

</project>

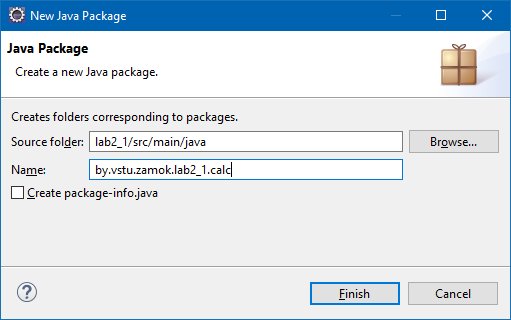
**Удалить все автосгенерированные классы в src/main/java и src/test/java**

**Зайти в Project🡪Properties🡪Java Build Path🡪Source и добавить папку src/main/resources/**





**Создать интерфейс Operation и классы Calculate (версия с ClassPathXmlApplicationContext), OpAdd и OppMultiply из примера в пакете calc (by.vstu.isap.<ваш\_псевдоним>.lab2\_1.calc).**



**public** **class** Calculate {

**private** Operation ops;

**public** **void** setOps(Operation ops) {

**this**.ops = ops;

}

**public** **static** **void** main(String[] args) {

ConfigurableApplicationContext context =

**new** ClassPathXmlApplicationContext("beans.xml");

Calculate calc = (Calculate) context.getBean("opsbean");

calc.execute(10L, 15L);

context.close();

}

**public** **void** execute(**long** op1, **long** op2) {

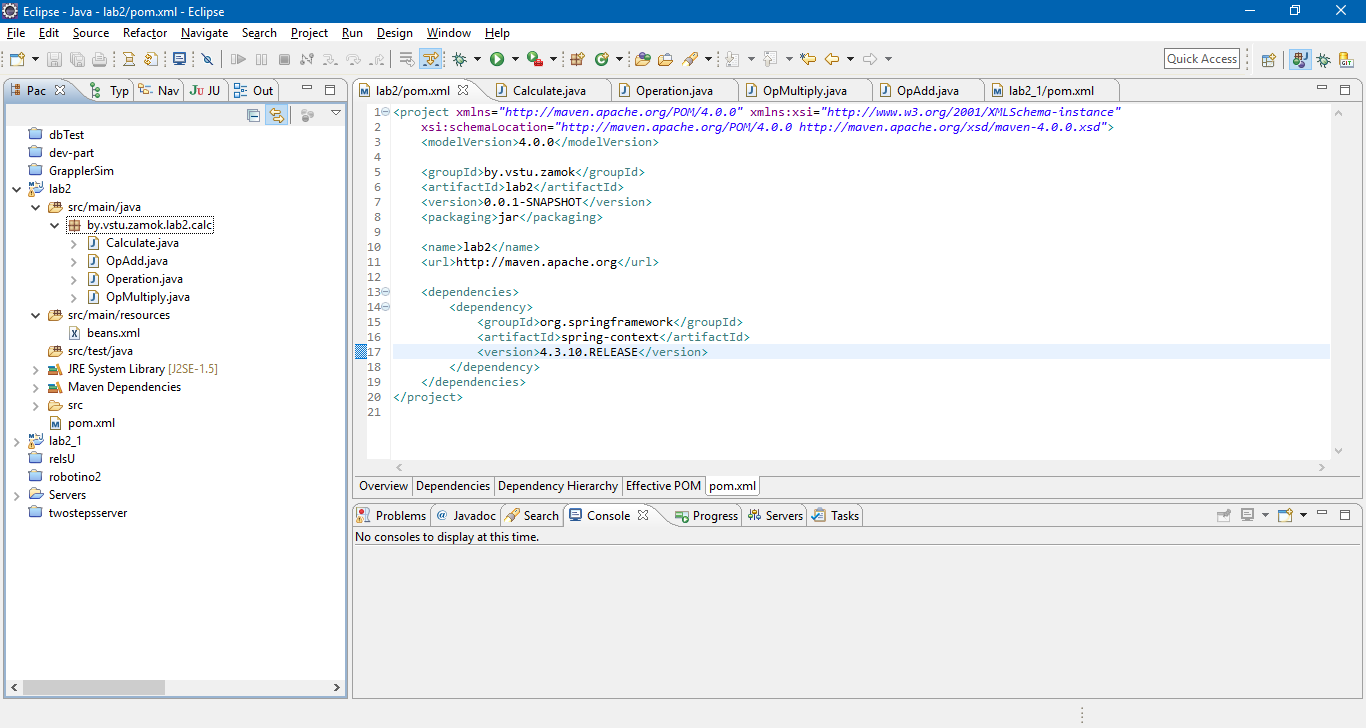
System.***out***.println("The result of " + op1 + ops.getOpsName()

+ op2 + " is " + ops.operate(op1, op2) + "!");

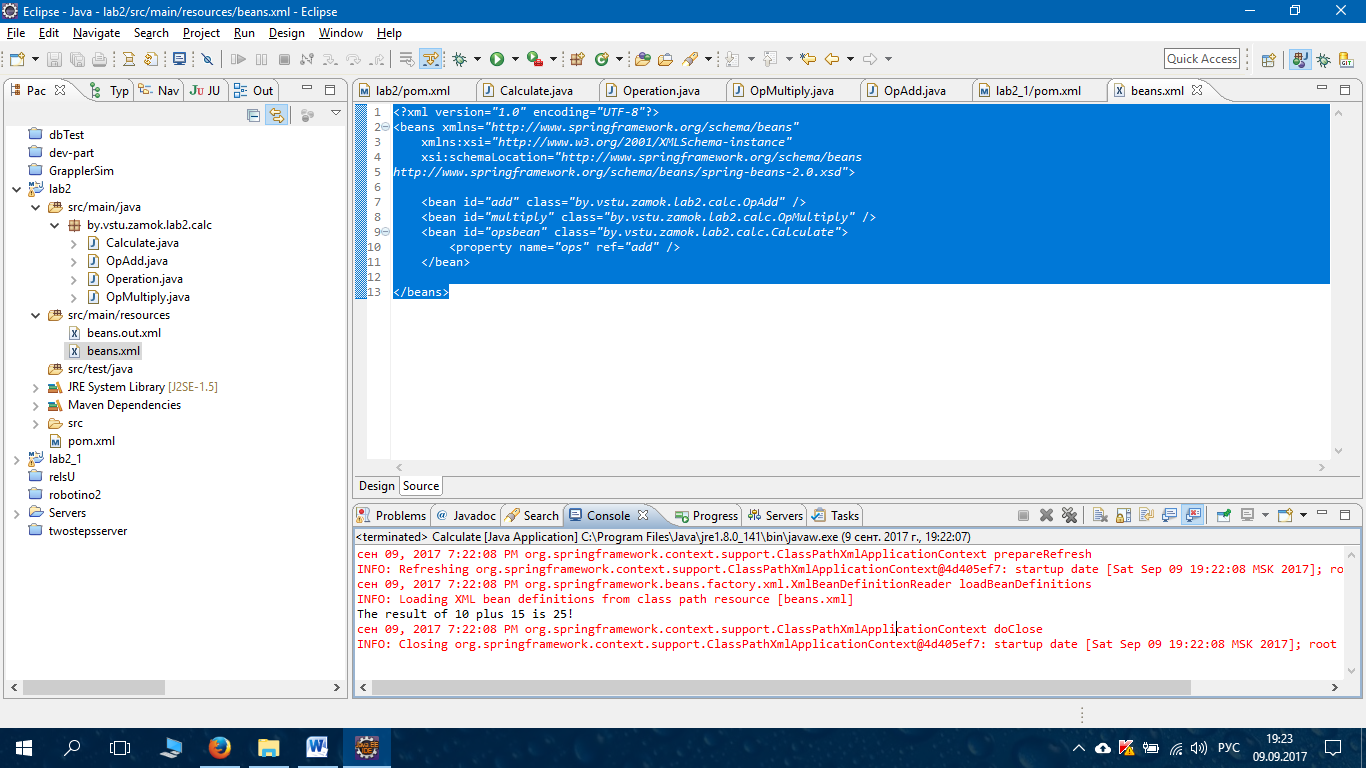
}

}

**Создать файл beans.xml из примера в директории resources, прописать пути к вашим бинам.**



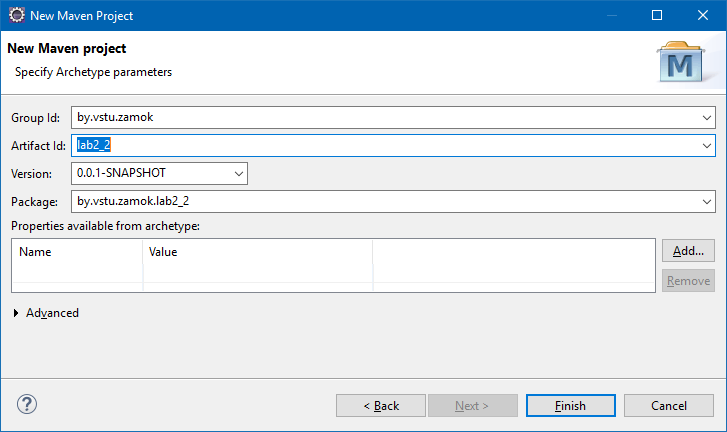
**Запустить приложение:**



## Java Configuration

**Создать Maven проект**

File 🡪 New 🡪 Other 🡪 Maven 🡪 Maven Project 🡪 Next 🡪 ввести в поле **Filter** quickstart 🡪 выбрать **maven-archetype-quickstart** 🡪 Next  
Указать Group ID (разработчик) и Artifact Id (имя проекта) и нажать Finish



**Указать springframework в pom.xml**

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>by.vstu.zamok</groupId>

<artifactId>lab2</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>lab2</name>

<url>http://maven.apache.org</url>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>4.3.10.RELEASE</version>

</dependency>

</dependencies>

<build>

<pluginManagement>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.2</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

</plugins>

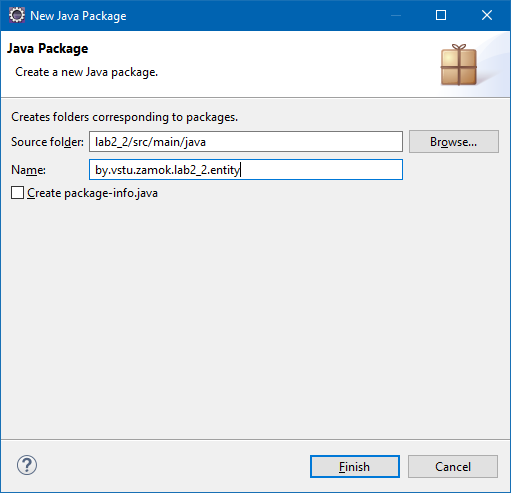
</pluginManagement>

</build>

</project>

**Удалить все автосгенерированные классы в src/main/java и src/test/java**

**Создать пакет entity:**



В этом пакете будут храниться классы-сущности.

**Создать абстрактную сущность:**

**public** **class** AbstractEntity {

**private** Long id;

**public** Long getId() {

**return** id;

}

**public** **void** setId(Long id) {

**this**.id = id;

}

}

И сущности, отражающие предметную область:

Группу

**public** **class** Group **extends** AbstractEntity {

**private** String name;

**private** List<Student> students;

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** List<Student> getStudents() {

**return** students;

}

**public** **void** setStudents(List<Student> students) {

**this**.students = students;

}

}

И студента:

**public** **class** Student **extends** AbstractEntity {

**private** String name;

**private** String surname;

**private** Group group;

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getSurname() {

**return** surname;

}

**public** **void** setSurname(String surname) {

**this**.surname = surname;

}

**public** Group getGroup() {

**return** group;

}

**public** **void** setGroup(Group group) {

**this**.group = group;

}

}

**Создать пакет dao.**

Здесь будут храниться объекты доступа к данным. В нашем случае вместо базы данных будет заглушка.

**Создать интерфейс Dao:**

**public** **interface** Dao<T **extends** AbstractEntity> {

T read(Long id);

List<T> read();

**void** save(T entity);

**void** delete(Long id);

}

И интерфейсы для каждой из сущностей:

Для группы:

**public** **interface** GroupDao **extends** Dao<Group> {

}

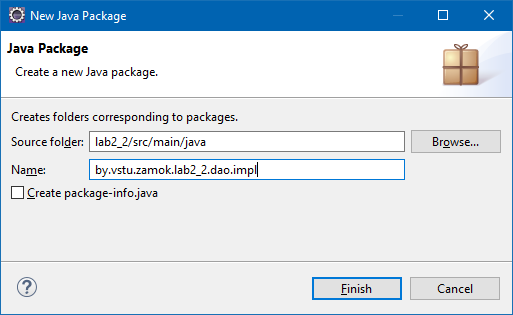
И для студента:

**public** **interface** StudentDao **extends** Dao<Student> {

List<Student> read(Group group);

}

**Создать пакет для реализации DAO dao.impl внутри пакета dao:**



Абстрактная реализация заглушки:

**public** **abstract** **class** AbstractMemDao<T **extends** AbstractEntity> **implements** Dao<T> {

**private** Map<Long, T> memory = **new** ConcurrentHashMap<>();

@Override

**public** T read(Long id) {

**return** memory.get(id);

}

@Override

**public** List<T> read() {

**return** **new** ArrayList<>(memory.values());

}

@Override

**public** **void** save(T entity) {

memory.put(entity.getId(), entity);

}

@Override

**public** **void** delete(Long id) {

memory.remove(id);

}

}

Заглушка группы:

@Component

**public** **class** GroupMemDao **extends** AbstractMemDao<Group> **implements** GroupDao {

}

Заглушка студента:

@Component

**public** **class** StudentMemDao **extends** AbstractMemDao<Student> **implements** StudentDao {

@Override

**public** List<Student> read(Group group) {

List<Student> students = **new** ArrayList<>();

**for** (Student student : read()) {

**if** (student.getGroup().equals(group)) {

students.add(student);

}

}

**return** students;

}

}

**Создать пакет для сервисов:**

DAO должен управлять только одним видом сущностей, в сервисах же можно внедрить несколько DAO для реализации бизнес-логики, кеширования и т.д. В нашем случае сервисы просто делегирют методы DAO, однако этим слоем никогда не следует пренебрегать.

**public** **interface** Service<T **extends** AbstractEntity> {

T read(Long id);

List<T> read();

**void** save(T entity);

**void** delete(T entity);

}

**public** **interface** GroupService **extends** Service<Group> {

}

**public** **interface** StudentService **extends** Service<Student> {

List<Student> read(Group group);

}

**Создать пакет для реализации сервисов service.impl и создать сервисы:**

@Service

**public** **class** GroupServiceImpl **implements** GroupService {

@Autowired

**private** GroupDao dao;

@Override

**public** Group read(Long id) {

**return** dao.read(id);

}

@Override

**public** List<Group> read() {

**return** dao.read();

}

@Override

**public** **void** save(Group entity) {

dao.save(entity);

}

@Override

**public** **void** delete(Group entity) {

dao.delete(entity.getId());

}

}

@Service

**public** **class** StudentServiceImpl **implements** StudentService {

@Autowired

**private** StudentDao dao;

@Override

**public** Student read(Long id) {

**return** dao.read(id);

}

@Override

**public** List<Student> read() {

**return** dao.read();

}

@Override

**public** List<Student> read(Group group) {

**return** dao.read(group);

}

@Override

**public** **void** save(Student entity) {

dao.save(entity);

}

@Override

**public** **void** delete(Student entity) {

dao.delete(entity.getId());

}

}

**Создать пакет config и поместить туда класс конфигурации:**

@Configuration

@ComponentScan("by.vstu.zamok.lab2\_2")

**public** **class** AppConfig {

}

В качестве параметра аннотации @ComponentScan указать корневой пакет приложения.

**Создать пакет runner и класс Main, предназначенный для запуска приложения.**

**public** **class** Main {

**public** **static** **void** main(String[] args) {

ConfigurableApplicationContext context = **new** AnnotationConfigApplicationContext(AppConfig.**class**);

StudentService studentService = context.getBean(StudentService.**class**);

studentService.save(*createStudent*());

System.***out***.println(studentService.read());

context.close();

}

**private** **static** Student createStudent() {

Student student = **new** Student();

student.setId(1L);

student.setName("name");

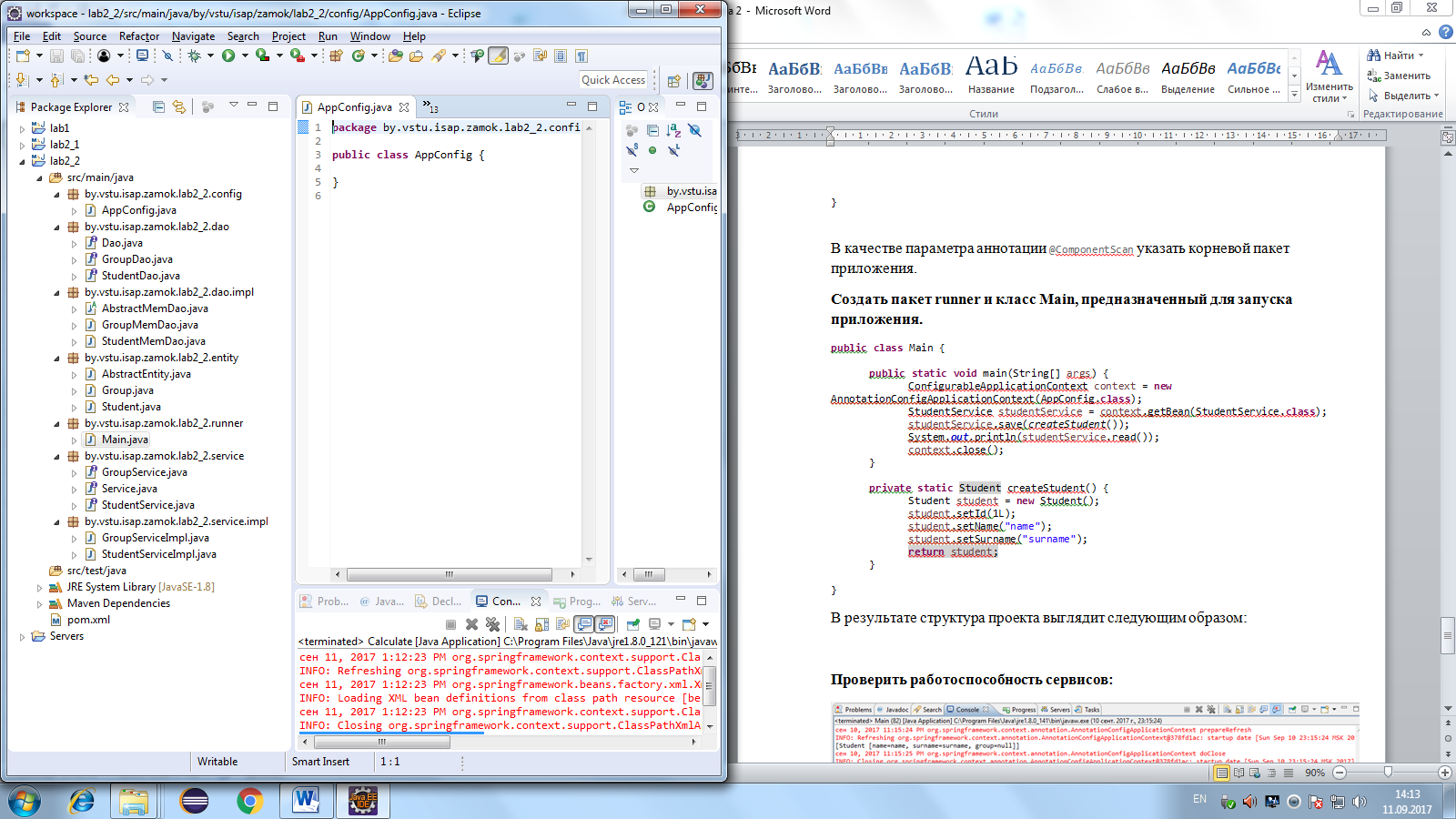
student.setSurname("surname");

**return** student;

}

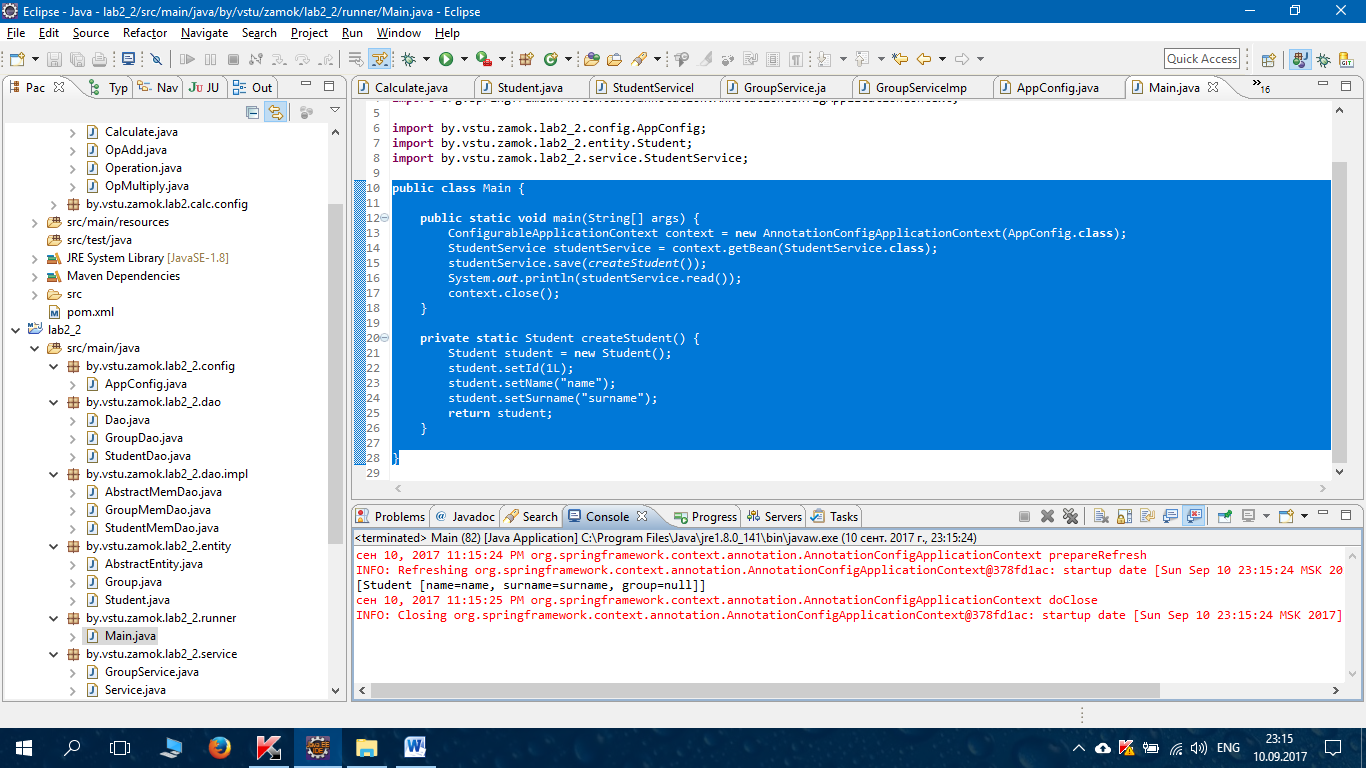
}

В результате структура проекта выглядит следующим образом:



Такая архитектура является трехслойной архитектурой сервера (dao, service, view), в нашем случае отсутствует слой view, отвечающий за отображение и обработку действий пользователя. Каждый слой является независимым и для работы требует только внедрения компонентов низлежащего слоя..

**Проверить работоспособность сервисов:**



# Индивидуальные задания:

**XML конфигурируемое приложение:**

Добавить дополнительную математическую операцию к приложению, показанному в примере.

|  |  |
| --- | --- |
| **Вариант** | **Задание** |
| 1,6,11 | Вычитание |
| 2,7,12 | Деление |
| 3,8,13 | Остаток от деления |
| 4,9,14 | Среднее арифметическое |
| 5,10,15 | Среднее геометрическое |

**Приложение на основе JavaConfiguration**

Создать приложение по примеру соответственно предметной области с предыдущего семестра.