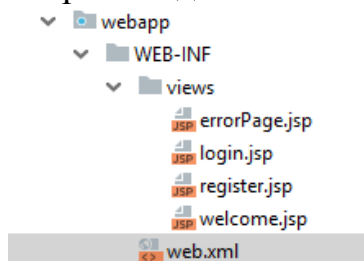


## № 11 -12 JSP, Servlet, Filter, многоуровневая архитектура

### Задание

1. Исследовать технологию JSP.
  - а. Исследуйте и разберите структуру сгенерированного на стороне сервера сервлета jsp.
2. Дополнить проект лабораторной № 10 п.3:  
В проекте должно быть 4 jsp страницы



Страница входа, регистрации, домашняя страница и страница ошибок errorPage.jsp используется для обработки ошибок или исключений с выводом кода ошибки.

Напишите отдельно две JSP страницы типа header (логотип, название и тп.) и footer (фio разработчика, год и т.д.) для включения во все JSP.

Веб- приложение должно работать следующим образом. На welcome страницу могут попасть только авторизированный пользователь. Проверка выполняется на основе данных из бд. Если поля не заполнены или неверно введена информация – выводится сообщение (см. в примере) и пользователь остается на этой странице.

A screenshot of a web browser window. The address bar shows 'localhost:8080/Servlet\_war\_exploded/controller?command=login'. Below the address bar, there is a red error message: 'Неверный логин или пароль, заполните все поля'. Underneath the message, the text 'Вход в систему' is displayed. There are two input fields: 'Имя : ' followed by an empty text box, and 'Пароль : ' followed by an empty text box. Below these fields are two buttons: 'Войти' and 'Регистрация'.

При нажатии на кнопку «Регистрация» переходим на страницу регистрации. При заполнении всех полей, значения для входа сохраняются в бд и выполняется переход на страницу логина. В системе не может быть двух пользователей с одинаковым именем (обрабатывать такого рода ошибки). В случае незаполнения любого из полей – выдавать сообщение об ошибке (см. ниже).

Неверный логин или пароль, заполните все поля

Регистрация нового пользователя

Введите имя :

Введите пароль :

После входа – попадаем на страницу приветствия . Содержит ссылки для перехода (вверху), строку приветствия с выводом имени пользователя и таблицу объектов из бд. Для вывода таблицы используйте стандартные теги JSTL. Нииже расположена форма добавления нового объекта. Категрию объектов сущностей выберите в соответствии с первой буквой вашей фамилии. Например, Купала – компьютеры, карты, конфеты....., Пушкин – принтеры, публикации, продукты и .....

***Добоплнительно: добавьте функции редактирования и удаления. Примените шаблоны и css. Фора представления информации - произвольная.***

[Вход](#)

[Выход](#)

## Добрый день, qaz

Список вашей группы

Имя	Телефон	email
Anna	+37545364754	annd34@gamail.com
Nikita	+37612345667	nik543637@mail.ru
Goga	80173243644	gog@iba.by
anna	+3442392343245	anna23@gmail.com
Olga	+375243539394	olga@tut.by

Новый :

Введите имя

Введите телефон

Введите email

Настройте фильтр таким образом, чтобы в случае если пользователь не авторизовался он не могу попасть на страницу welcom набрав URL и ему возвращалась страница с ошибкой. Станицу с ошибкой можете использовать везде для обработки ошибок.

[http://localhost:8080/Servlet\\_war\\_exploded/controller?command=welcome](http://localhost:8080/Servlet_war_exploded/controller?command=welcome)

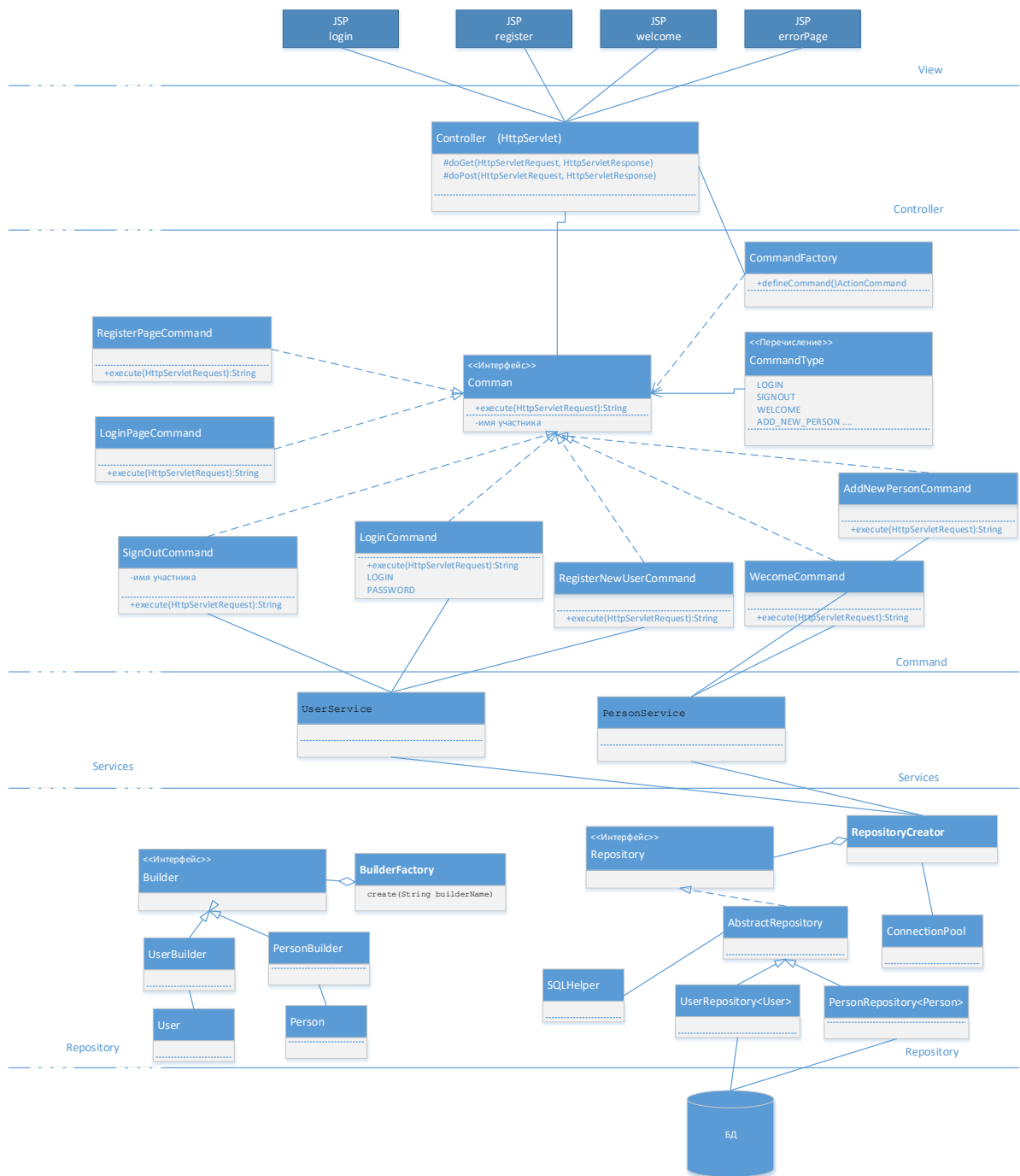
## ERROR...

Status code: Нет авторизации для выполнения данной команды

**Добавьте логгирование.**

**Напишите два unit теста, используйте mock для зависимых объектов.**

Архитектурно приложение должно быть многоуровневое со слоем репозитория, сервисов (команд), представлений (jsp) и одним сервлетом (Controller). Реализовать паттерн команда, абстрактная фабрика, строитель



## Вопросы

1. Что такое jsp? Каков ее состав и назначение?
2. Расскажите о жизненном цикле jsp? В чем отличие jsp и servlet?
3. Перечислите теги jsp и поясните их назначение.
4. Перечислите неявные объекты jsp.
5. Какие области видимости для переменных jsp существуют?
6. Что такое PageContext?
7. Что такое EL, как он используется ?
8. Как задать и настроить error page?
9. Расскажите о взаимодействии jsp-servlet-jsp.

## Методические указания к выполнению работы.

### ЛОГГИРОВАНИЕ

В программировании принято логировать практически все. Java-программы – это очень часто большие серверные приложения без UI, консоли и т.д. Они обрабатывают одновременно запросы тысяч пользователей, и нередко при этом возникают различные ошибки. Особенно, когда разные нити начинают друг другу мешать.

Фактически, единственным способом поиска редко воспроизводимых ошибок и сбоев в такой ситуации есть запись в лог/файл всего, что происходит в каждой нити.

Чаще всего в лог пишется информация о параметрах метода, с которыми он был вызван, все перехваченные ошибки, и еще много промежуточной информации.

Чем полнее лог, тем легче восстановить последовательность событий и отследить причины возникновения сбоя или ошибки.

Изначально в Java не было своего логгера, что привело к написанию нескольких независимых логгеров. Самым распространенным из них стал log4j.

Создается логгер следующей строкой в классе.

```
private static final Logger LOGGER = Logger.getLogger(ConnectionPool.class);
```

getLogger – это его статический метод. В него обычно передают текущий класс. Такой статический объект создают практически в каждом классе.

Обычно, у каждого лог-сообщения есть своя степень важности, и, используя ее можно часть этих сообщений отбрасывать. Степени важности:

Степень важности	Описание
ALL	Все сообщения
TRACE	Мелкое сообщение при отладке
DEBUG	Сообщения важные при отладке
INFO	Просто сообщение
WARN	Предупреждение
ERROR	Ошибка

FATAL	Фатальная ошибка
OFF	Нет сообщения

Если выставить уровень логирования в WARN, то все сообщения, менее важные, чем WARN будут отброшены: TRACE, DEBUG, INFO.

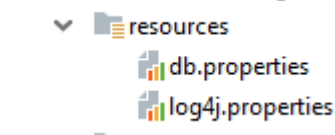
Если выставить уровень фильтрации в FATAL, то будут отброшены даже ERROR'ы.

Есть еще два уровня важности, которые используются при фильтрации – это OFF – отбросить все сообщения и ALL – показать все сообщения (не отбрасывать ничего).

Пример записи в лог

```
LOGGER.error("Can not get Instance", e);
```

Настройки логгера log4j задаются в файле log4j.properties.



```
# Root logger option
log4j.rootLogger=DEBUG, stdout, file

# Redirect log messages to console
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p
%c{1}:%L - %m%n

# Redirect log messages to a log file
log4j.appender.file=org.apache.log4j.RollingFileAppender
#outputs to home
log4j.appender.file.File=log/applicationLog
log4j.appender.file.MaxFileSize=5MB
log4j.appender.file.MaxBackupIndex=10
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p
%c{1}:%L - %m%n
```

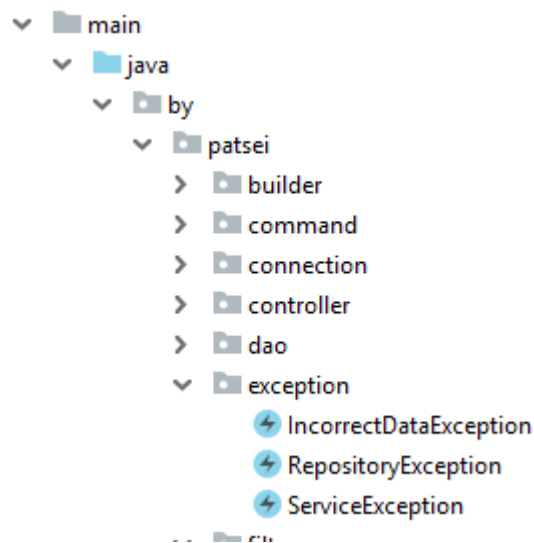
В этом файле можно задать несколько appender'ов – объектов, в которые будут писаться данные. Есть источники данных, а есть – аппендеры – объекты, куда как бы «стекают» данные.

Мы указываем уровень сообщений, которые оставляем. Все менее важные уровни будут отброшены (DEBUG). Там же, через запятую, мы указываем имя объекта, куда будет писаться лог. Указываем тип аппендера – консоль (ConsoleAppender). Указываем, куда именно будем писать – System.out. Задаем класс, который будет управлять шаблонами записей – PatternLayout. Задаем шаблон для записи, который будет использоваться. В примере выше это дата и время.

Согласно настройкам – логи можно найти здесь

## ИСКЛЮЧЕНИЯ

Добавим в проект классы исключений для каждого уровня.



Создайте пакет `exception` и три класса исключений: *RepositoryException*, *ServiceException* и *IncorrectDataException*.

Хотя встроенные исключения Java обрабатывают большинство частых ошибок, создание собственных типов исключений нужно для обработки ситуаций, специфичных для приложения. Все три класса - подклассы `Exception` (который, разумеется, является подклассом `Throwable`). Ваши подклассы не обязаны реализовывать что-либо — важно само их присутствие в системе типов, которое позволит использовать их как исключения.

Идея заключается в том, что бы заворачивать исключение источник в новое исключение. Таким образом можно будет проще проследить место и причину возникновения ошибки. В каждом исключении будет ссылка на предыдущее исключение - напоминает связной список.

```
package by.patsei.exception;
```

```
public class IncorrectDataException extends Exception {
    public IncorrectDataException(String message) {
        super(message);
    }

    public IncorrectDataException(String message, Throwable cause) {
        super(message, cause);
    }

    public IncorrectDataException(Throwable cause) {
        super(cause);
    }
}
```

```
package by.patsei.exception;
```

```
public class RepositoryException extends Exception {
    public RepositoryException(String message) {
        super(message);
    }

    public RepositoryException(String message, Throwable cause) {
```

```

        super(message, cause);
    }

    public RepositoryException(Throwable cause) {
        super(cause);
    }
}

package by.patsei.exception;

public class ServiceException extends Exception {
    public ServiceException(String message) {
        super(message);
    }

    public ServiceException(String message, Throwable cause) {
        super(message, cause);
    }

    public ServiceException(Throwable cause) {
        super(cause);
    }
}

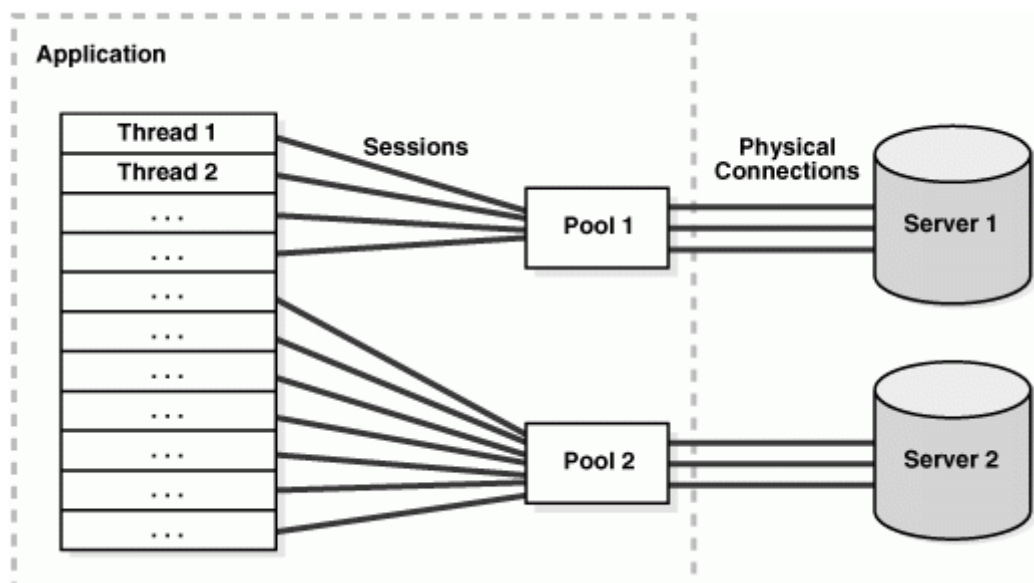
```

## ПУЛ СОЕДИНЕНИЙ (CONNECTION POOL)

Коннект к базе данных требует затрат определенного времени. Особенно, если база данных находится удаленно. Если под каждым запрос делать подключение к базе, то отклик нашего приложения будет невероятно низким по скорости. Не говоря уже о ресурсах, которые оно потребит.

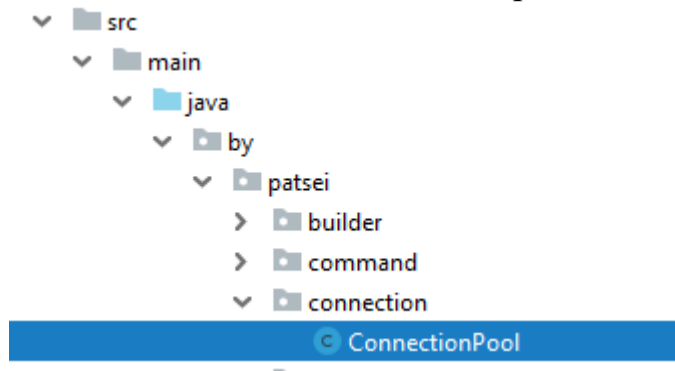
Во избежании таких проблем есть пул соединений, который открывает соединение только первый раз, а при закрытии соединения он не закрывает его, а отправляет в пул в случае если кто-то будет делать очередной запрос. По сути, приложение работает через специальный драйвер, который является оберткой для обычного jdbc драйвера и который имеет дополнительный функционал по работе с пулом.

Настройки для пула соединений программист может прописать вручную: количество активных соединений, время ожидания и т.д.





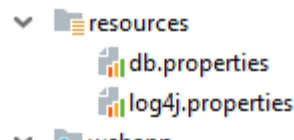
Пул соединений это будет класс *ConnectionPool* в пакете **connection**. Он заменит ConnectorDB ( который впоследствии удалим).



Класс сделаем singleton - это когда нельзя создать больше одного экземпляра данного класса. Для этого делается приватный конструктор и для доступа к экземпляру класса создается специальный метод, который возвращает только один экземпляр класса getInstance()

Далее нужно создать метод getConnection(), через который мы будем получать соединение, но не напрямую, а через **пул соединений**. И вернуть соединение после использования. Для этого в классе есть две блокирующие очереди **freeConnections** и **releaseConnections**.

Инициализируется пул из файла параметров db в ресурсах:



```
db.driver    = com.mysql.jdbc.Driver
db.user      = root
db.password  = root
db.poolsize  = 5
db.url       = jdbc:mysql://localhost:3306/infodb?useSSL=false
db.useUnicode = true
db.encoding  = UTF-8
```

```
package by.patsei.connection;
```

```
import com.mysql.jdbc.Driver;
import org.apache.log4j.Logger;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Enumeration;
import java.util.Locale;
import java.util.Properties;
import java.util.ResourceBundle;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.locks.ReentrantLock;
```

```
public class ConnectionPool {

    private static final Logger LOGGER =
        Logger.getLogger(ConnectionPool.class);
    private static final String PROPERTY_PATH = "db";
    private static final int INITIAL_CAPACITY = 5;
```

```

    private ArrayBlockingQueue<Connection> freeConnections = new
ArrayBlockingQueue<>(INITIAL_CAPACITY);
    private ArrayBlockingQueue<Connection> releaseConnections = new
ArrayBlockingQueue<>(INITIAL_CAPACITY);
    private static ReentrantLock lock = new ReentrantLock();
    private volatile static ConnectionPool connectionPool;

    public static ConnectionPool getInstance() {
        try {
            lock.lock();
            if (connectionPool == null) {
                connectionPool = new ConnectionPool();
            }
        } catch (Exception e) {
            LOGGER.error("Can not get Instance", e);
            throw new RuntimeException("Can not get Instance", e);
        } finally {
            lock.unlock();
        }
        return connectionPool;
    }

    private ConnectionPool() throws SQLException {
        try {
            lock.lock();
            if (connectionPool != null) {
                throw new UnsupportedOperationException();
            } else {
                DriverManager.registerDriver(new Driver());
                init();
            }
        } finally {
            lock.unlock();
        }
    }

    private void init() {
        Properties properties = new Properties();
        ResourceBundle resource = ResourceBundle.getBundle(PROPERTY_PATH,
Locale.getDefault());
        if (resource == null) {
            LOGGER.error("Error while reading properties");
        } else {
            String connectionURL = resource.getString("db.url");
            String initialCapacityString = resource.getString("db.poolsize");
            String user = resource.getString("db.user");
            String pass = resource.getString("db.password");
            Integer initialCapacity = Integer.valueOf(initialCapacityString);

            for (int i = 0; i < initialCapacity; i++) {
                try {
                    Connection connection =
DriverManager.getConnection(connectionURL, user, pass);
                    freeConnections.add(connection);
                } catch (SQLException e) {
                    LOGGER.error("Pool can not initialize", e);
                    throw new RuntimeException("Pool can not initialize", e);
                }
            }
        }
    }

    public Connection getConnection() {
        try {
            Connection connection = freeConnections.take();
            releaseConnections.offer(connection);
        }
    }

```

```

        LOGGER.info("Connection was taken, the are free connection " +
freeConnections.size());
        return connection;
    } catch (InterruptedException e) {
        throw new RuntimeException("Can not get database", e);
    }
}

public void releaseConnection(Connection connection) {
    releaseConnections.remove(connection);
    freeConnections.offer(connection);
    LOGGER.info("Connection was released, the are free connection " +
freeConnections.size());
}

public void destroy() {
    for (int i = 0; i < freeConnections.size(); i++) {
        try {
            Connection connection = (Connection) freeConnections.take();
            connection.close();
        } catch (InterruptedException e) {
            LOGGER.error("Connection close exception", e);
        } catch (SQLException e) {
            LOGGER.error("database is not closed", e);
            throw new RuntimeException("database is not closed", e);
        }
    }
    try {
        Enumeration<java.sql.Driver> drivers = DriverManager.getDrivers();
        while (drivers.hasMoreElements()) {
            java.sql.Driver driver = drivers.nextElement();
            DriverManager.deregisterDriver(driver);
        }
    } catch (SQLException e) {
        LOGGER.error("Drivers were not deregistrated", e);
    }
}
}

```

Как видно, в классе много логирования. Практически перед каждым действием, тогда будет ясно в каком месте проблемы и можно разобраться с работой пула.

## УРОВЕНЬ РЕПОЗИТОРИЯ

Начнем строить уровень организации взаимодействия с базой данных DAO или репозиторий. В чем отличие?

Data Access Object (DAO) — широко распространенный паттерн для сохранения объектов бизнес-области в базе данных. В самом широком смысле, DAO — это класс, содержащий CRUD методы для конкретной сущности.

Паттерн имеет следующие преимущества:

- Отделяет бизнес-логику, использующую данный паттерн, от механизмов сохранения данных и используемых ими API;
- Сигнатуры методов интерфейса независимы от содержимого класса модели.

Здесь проблема заключается в том, что обязанности DAO не описаны четко. Большая часть людей представляет DAO некими вратами к базе данных и добавляет в него методы как только находит новый способ, которым они

хотели бы общаться с базой данных. Поэтому нередко можно увидеть сильно раздутый DAO.

## **Паттерн Repository**

Лучшим решением будет использование паттерна Repository. «Repository представляет собой все объекты определенного типа в виде концептуального множества. Его поведение похоже на поведение коллекции, за исключением более развитых возможностей для построения запросов».

В нем тоже будут методы add и update - выглядят идентично методам DAO. Метод remove отличается от метода удаления, определенного в DAO тем, что принимает объект модели в качестве параметра.

Представление репозитория как коллекции меняет его восприятие. Вы избегаете раскрытия типа репозиторию. Контракты методов add/remove/update это просто абстракция коллекции.

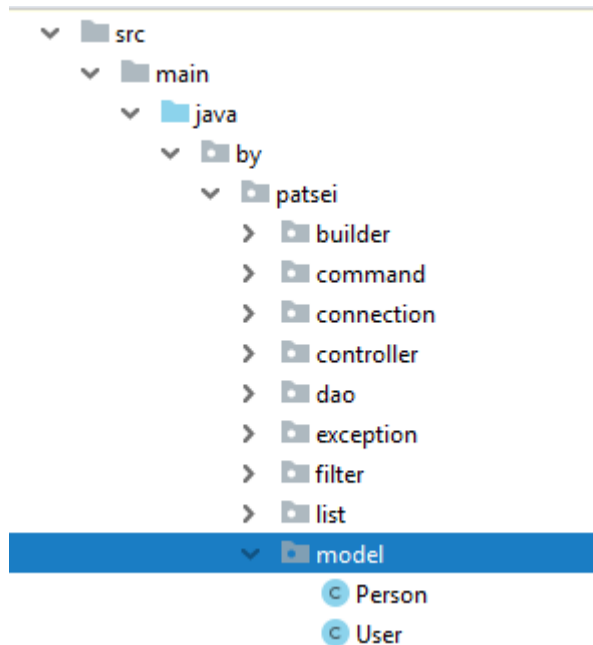
Однако, метод query является особенным. Такого метода в классе коллекции нет. Что он делает?

Репозиторий отличается от коллекции, если рассматривать возможности для построения запросов. Имея коллекцию объектов в памяти, довольно просто перебрать все ее элементы и найти интересующий нас экземпляр. Репозиторий работает с большим набором объектов, чаще всего, находящихся вне оперативной памяти в момент выполнения запроса. Нецелесообразно загружать все объекты в память, если нам необходим один конкретный объект. Вместо этого, мы передаем репозиторию критерий, с помощью которого он сможет найти один или несколько объектов. Репозиторий может сгенерировать SQL запрос в том случае, если он использует базу данных в качестве бекэнда, или он может найти необходимый объект перебором, если используется коллекция в памяти.

Одна из часто используемых реализаций критерия — паттерн Specification (далее спецификация). Спецификация — это простой предикат, который принимает объект бизнес-области и возвращает boolean.

Репозиторий, использующий базу данных в качестве бекэнда, может использовать интерфейс Спецификации для получения параметров SQL запроса

Пакет model



Там у нас две модели Пользователи - User :

```
package by.patsei.model;

import java.io.Serializable;
import java.util.Objects;

public class User implements Serializable {

    private int id;
    private String login;
    private byte[] passw;

    public User(int id, String login, byte[] passw) {
        this.id = id;
        this.login = login;
        this.passw = passw;
    }

    public User(String login, byte[] passw) {
        this.id = 0;
        this.login = login;
        this.passw = passw;
    }

    public User() {
    }

    public void setId(int id) {
        this.id = id;
    }

    public void setLogin(String login) {
        this.login = login;
    }

    public void setPassw(byte[] passw) {
        this.passw = passw;
    }

    public int getId() {
        return id;
    }

    public String getLogin() {
```

```

        return login;
    }

    public byte[] getPassw() {
        return passw;
    }

    @Override
    public String toString() {
        return "User{" +
            "id=" + id +
            ", login='" + login + '\'' +
            ", passw='" + passw + '\'' +
            '}';
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (o == null || getClass() != o.getClass()) {
            return false;
        }
        User user = (User) o;
        return Objects.equals(getId(), user.getId()) &&
            Objects.equals(getLogin(), user.getLogin()) &&
            Objects.equals(getPassw(), user.getPassw());
    }

    @Override
    public int hashCode() {
        return Objects.hash(getLogin(), getPassw(), getId());
    }
}

```

и Персоны – Person:

```

package by.patsei.model;

import java.io.Serializable;
import java.util.Objects;

public class Person implements Serializable {
    private int id;
    private String name;
    private String phone;
    private String email;

    public Person(Person person) {
        name = person.name;
        phone = person.phone;
        email = person.email;
        id = person.id;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setPhone(String phone) {
        this.phone = phone;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}

```

```

    }

    public String getName() {
        return name;
    }

    public String getPhone() {
        return phone;
    }

    public String getEmail() {
        return email;
    }

    public Person(int id, String name, String phone, String email) {
        this.id = id;
        this.name = name;
        this.phone = phone;
        this.email = email;
    }

    public Person() {
    }

    public Person(String name, String phone, String email) {
        this.id = 0;
        this.name = name;
        this.phone = phone;
        this.email = email;
    }

    @Override
    public String toString() {
        return "Person{" +
            "id=" + id +
            ", name='" + name + '\'' +
            ", phone='" + phone + '\'' +
            ", email='" + email + '\'' +
            '}';
    }

    public int getId() {
        return id;
    }

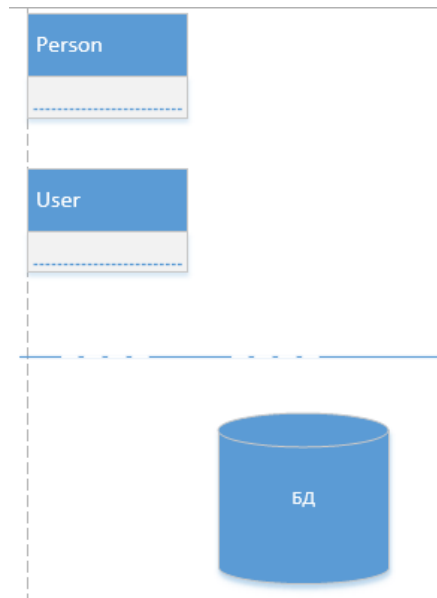
    public void setId(int id) {
        this.id = id;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (o == null || getClass() != o.getClass()) {
            return false;
        }
        Person person = (Person) o;
        return Objects.equals(getEmail(), person.getEmail()) &&
            Objects.equals(getPhone(), person.getPhone()) &&
            Objects.equals(getName(), person.getName()) &&
            Objects.equals(getId(), person.getId());
    }

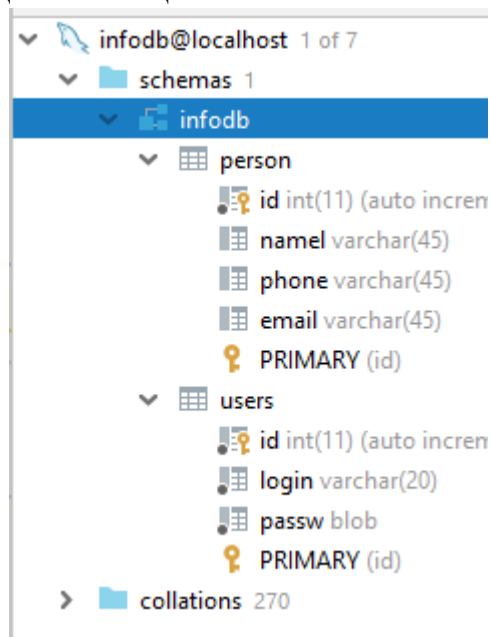
    @Override
    public int hashCode() {
        return Objects.hash(getId(), getEmail(), getPhone(), getName());
    }

```

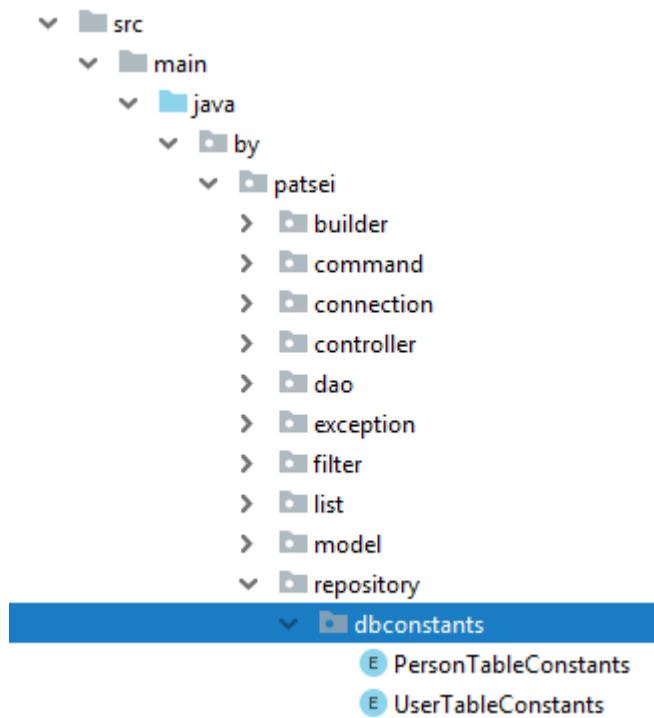
```
}  
}
```



Создайте пакет **repository**. В нем определяете пакет **dbconstants** с двумя перечислениями ( у нас две таблицы бд в примпре) с именами столбцов для каждой таблицы *PersonTableConstants* и *UserTableConstants*.







В случае изменения или переименования столбцов таблиц нужно будет модифицировать эти перечисления.

```
package by.patsei.repository.dbconstants;

public enum PersonTableConstants {
    ID("id"),
    NAME("name1"),
    PHONE("phone"),
    EMAIL("email");

    private String fieldName;

    private PersonTableConstants(String fieldName) {
        this.fieldName = fieldName;
    }

    public String getFieldName() {
        return fieldName;
    }
}
```

```
package by.patsei.repository.dbconstants;

public enum UserTableConstants {

    ID("id"),
    LOGIN("login"),
    PASSWORD("passw");

    private String fieldName;

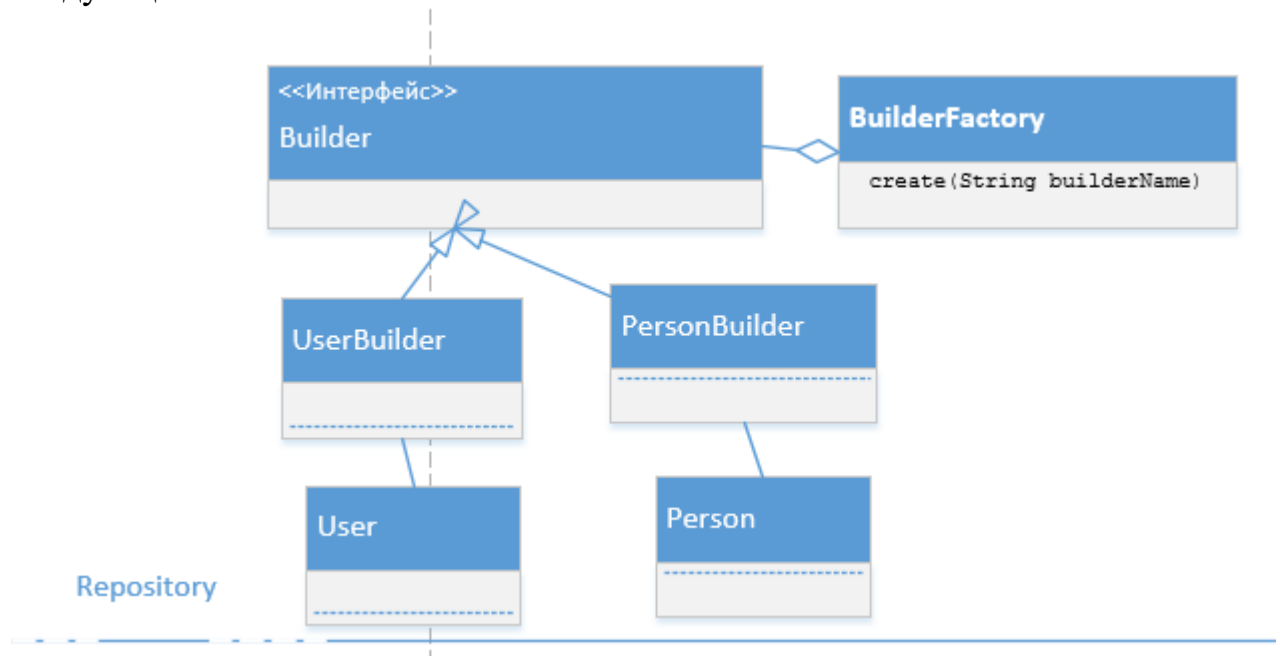
    private UserTableConstants(String fieldName) {
        this.fieldName = fieldName;
    }

    public String getFieldName() {
        return fieldName;
    }
}
```

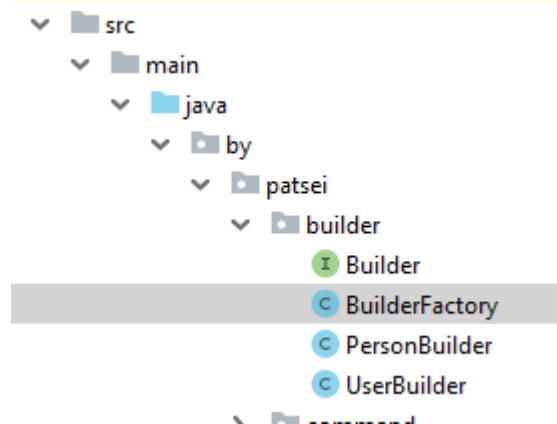
Для того чтобы выполнять отображение `ResultSet`, получаемого из базы данных на объекты классов Java надо реализовать паттерн **Builder**.

**Builder (строитель)** — это порождающий паттерн проектирования, который позволяет создавать сложные объекты пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.

Нам нужны будут объекты `User` и `Person`. Диаграмма классов будет следующая.



Создайте пакет **builder**



Интерфейс *Builder* содержит единственный метод `build`:

```
package by.patsei.builder;

import by.patsei.exception.RepositoryException;
import java.sql.ResultSet;

public interface Builder <T> {
    T build(ResultSet resultSet) throws RepositoryException,
    RepositoryException;
}
```

*BuilderFactory* содержит метод `create`, который в зависимости от строки создаёт или *PersonBuilder*, или *UserBuilder*:

```
package by.patsei.builder;

public class BuilderFactory {

    private static final String USER = "user";
    private static final String PERSON = "person";
    private static final String MESSAGE= "Unknown Builder name!";

    public static Builder create(String builderName) {
        switch (builderName) {
            case USER: {
                return new UserBuilder();
            }
            case PERSON: {
                return new PersonBuilder();
            }
            default:
                throw new IllegalArgumentException(MESSAGE);
        }
    }
}
```

*PersonBuilder* реализует `build` и строит `Person`:

```
package by.patsei.builder;
import by.patsei.exception.RepositoryException;
import by.patsei.model.Person;
import by.patsei.repository.dbconstants.PersonTableConstants;
import java.sql.ResultSet;
import java.sql.SQLException;

public class PersonBuilder implements Builder <Person>{
    @Override
    public Person build(ResultSet resultSet) throws RepositoryException {
        try {
            int id = resultSet.getInt(PersonTableConstants.ID.getFieldName());
            String name =
resultSet.getString(PersonTableConstants.NAME.getFieldName());
            String phone =
resultSet.getString(PersonTableConstants.PHONE.getFieldName());
            String email =
resultSet.getString(PersonTableConstants.EMAIL.getFieldName());
            return new Person(id, name, phone, email);
        } catch (SQLException exception) {
            throw new RepositoryException(exception.getMessage(), exception);
        }
    }
}
```

*UserBuilder* извлекает параметры из `ResultSet` и строит `User`:

```
package by.patsei.builder;

import by.patsei.exception.RepositoryException;
import by.patsei.model.User;
import by.patsei.repository.dbconstants.UserTableConstants;
import java.sql.ResultSet;
import java.sql.SQLException;

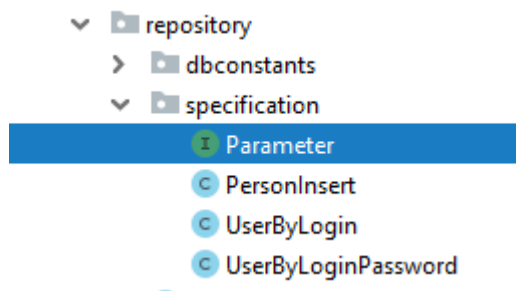
public class UserBuilder implements Builder<User> {
    @Override
```

```

    public User build(ResultSet resultSet) throws RepositoryException {
        try {
            int id = resultSet.getInt(UserTableConstants.ID.getFieldName());
            String login =
resultSet.getString(UserTableConstants.LOGIN.getFieldName());
            byte[] password =
resultSet.getBytes(UserTableConstants.PASSWORD.getFieldName());
            return new User(id, login, password);
        } catch (SQLException exception) {
            throw new RepositoryException(exception.getMessage(), exception);
        }
    }
}

```

Вернемся к репозиторию. Как было описано выше все запросы будут выполняться через метод `query(String sqlString, Parameter parameter)` со строкой SQL и параметрами. Для представления параметров создадим пакет **specification**.



Интерфейс `Parameter` должен возвращать список объектов — параметров (м.б. разных типов).

```

package by.patsei.repository.specification;

import java.util.List;

public interface Parameter {
    List<Object> getParameters();
}

```

Согласно выбранным функциям в приложении понадобится проверять логин пользователя — один параметр:

```

package by.patsei.repository.specification;

import java.util.Arrays;
import java.util.List;

public class UserByLogin implements Parameter {

    private String login;

    public UserByLogin(String login) {
        this.login = login;
    }

    @Override
    public List<Object> getParameters() {
        return Arrays.asList(login);
    }
}

```

Добавлять пользователя - два параметрами логин и пароль:

```

package by.patsei.repository.specification;

import java.util.Arrays;
import java.util.List;

public class UserByLoginPassword implements Parameter {

    private String login;
    private byte [] password;

    public UserByLoginPassword(String login, byte [] password) {
        this.login = login;
        this.password = password;
    }

    @Override
    public List<Object> getParameters() {
        return Arrays.asList(login, password);
    }
}

```

Добавлять новую персону – три параметра:

```

package by.patsei.repository.specification;

import java.util.Arrays;
import java.util.List;

public class UserByLoginPassword implements Parameter {

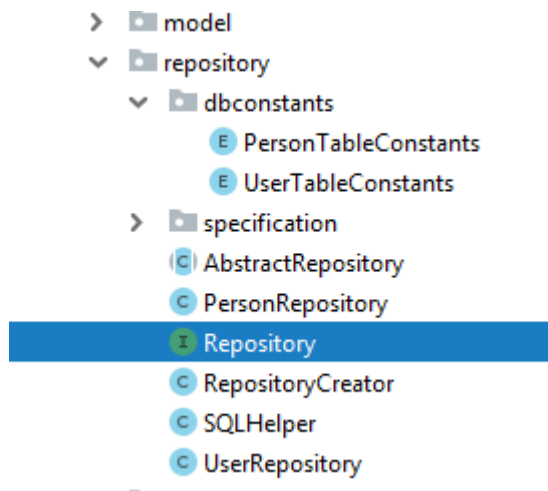
    private String login;
    private byte [] password;

    public UserByLoginPassword(String login, byte [] password) {
        this.login = login;
        this.password = password;
    }

    @Override
    public List<Object> getParameters() {
        return Arrays.asList(login, password);
    }
}

```

Теперь можно строить паттерн репозиторий. Разработку начинаем с интерфейса *Repository*. Он содержит необходимы минимум для определенных операций – найти всех, сохранить и два метода для выполнения запросов `queryForSingleResult` и `query`



```

package by.patsei.repository;

import by.patsei.exception.RepositoryException;
import by.patsei.repository.specification.Parameter;
import java.util.List;
import java.util.Optional;

public interface Repository <T> {
    List<T> query(String sqlString, Parameter parameter) throws
RepositoryException;
    Optional<T> queryForSingleResult(String sqlString, Parameter parameter)
throws RepositoryException;
    List<T> findAll() throws RepositoryException;
    Integer save(T object) throws RepositoryException;
}

```

Второй класс, который нам понадобится *SQLHelper*. Он содержит константы для выполнения запросов (строки с SQL) и метод `makeInsertQuery`. Он позволяет построить произвольный INSERT запрос для разных таблиц и параметров, которые передаются в списке параметров метода.

```

package by.patsei.repository;

import by.patsei.repository.dbconstants.PersonTableConstants;
import by.patsei.repository.dbconstants.UserTableConstants;

import java.util.Map;

public class SQLHelper {

    public static final String ID = "id";
    private static final String INSERT_QUERY = "INSERT INTO ";
    private static final String VALUES = "VALUES";
    private static final String WHERE = "WHERE ";
    private static final String SELECT = "SELECT";
    private static final String USER_TABLE = "users";
    private static final String PERSON_TABLE = "person";

    public final static String SQL_GET_PERSONS = "select * from " + USER_TABLE;
    public final static String SQL_INSERT_PERSON = "INSERT INTO " + PERSON_TABLE + "(" +
PersonTableConstants.NAME +
    "," + PersonTableConstants.PHONE + "," + PersonTableConstants.EMAIL + ") VALUES ( ? ,
?, ?)";
    public final static String SQL_GET_USER = "SELECT " + UserTableConstants.ID.getFieldName() +
", " +
    UserTableConstants.LOGIN.getFieldName() + ", " +
    UserTableConstants.PASSWORD.getFieldName() + " from " + USER_TABLE + " WHERE " +
    UserTableConstants.LOGIN.getFieldName() + " =? and " +
    UserTableConstants.PASSWORD.getFieldName() + " =?";
    public final static String SQL_CHECK_LOGIN = "SELECT " +
UserTableConstants.LOGIN.getFieldName() + " FROM " +
    USER_TABLE + " WHERE " + UserTableConstants.LOGIN.getFieldName() + " = ?";
    public final static String SQL_INSERT_USER = "INSERT INTO " + USER_TABLE + "(" +
    UserTableConstants.LOGIN.getFieldName() + " , " +

```

```

        UserTableConstants.PASSWORD.getFieldName() + ") VALUES (? , ?)";

    public static String makeInsertQuery(Map<String, Object> fields, String table) {
        StringBuilder columns = new StringBuilder("(");
        StringBuilder values = new StringBuilder("(");

        for (Map.Entry<String, Object> entry : fields.entrySet()) {
            String column = entry.getKey();
            if (column.equals(ID)) {
                continue;
            }
            columns.append(column).append(", ");
            values.append("?, ");
        }

        values.deleteCharAt(values.lastIndexOf(", "));
        columns.deleteCharAt(columns.lastIndexOf(", "));
        values.append(")");
        columns.append(")");

        return INSERT_QUERY + table + columns + VALUES + values + ";";
    }
}

```

Следующий класс *AbstractRepository* реализует интерфейс:

```

package by.patsei.repository;

import by.patsei.builder.Builder;
import by.patsei.builder.BuilderFactory;
import by.patsei.exception.RepositoryException;
import org.apache.log4j.Logger;

import java.sql.*;
import java.util.*;

public abstract class AbstractRepository<T> implements Repository<T> {

    private Connection connection;

    private static final Logger LOGGER =
        Logger.getLogger(AbstractRepository.class);
    private static final String GET_ALL_QUERY = "SELECT * FROM ";
    private final String WHERE_ID_CONDITION = " WHERE id_" + getTableName() +
        "=(?)";
    protected final String DELETE_QUERY = "DELETE from " + getTableName() + "
    where id_" + getTableName() + "=(?)";

    protected abstract String getTableName();

    AbstractRepository(Connection connection) {
        this.connection = connection;
    }

    private static Integer getType(Object object) {
        if (object instanceof Integer) {
            return Types.INTEGER;
        }
        if (object instanceof Float) {
            return Types.FLOAT;
        }
        if (object instanceof String) {
            return Types.VARCHAR;
        }
        else {
            return Types.NULL;
        }
    }

    public static void prepare(PreparedStatement preparedStatement,

```

```

List<Object> parameters) throws SQLException {
    int length = parameters.size();
    for (int i = 0; i < length; i++) {
        if (parameters.get(i) == null) {
            preparedStatement.setNull(i + 1, getType(parameters.get(i)));
        } else {
            preparedStatement.setObject(i + 1, parameters.get(i));
        }
    }
}

public static void prepare(PreparedStatement preparedStatement, Map<String,
Object> fields, String tableName) throws SQLException {
    int i = 1;
    for (Map.Entry<String, Object> entry : fields.entrySet()) {
        Object value = entry.getValue();
        String key = entry.getKey();
        if (!key.equals(SQLHelper.ID)) {
            if (value == null) {
                preparedStatement.setNull(i++, getType(value));
            } else {
                preparedStatement.setObject(i++, value);
            }
        }
    }
    Object id = fields.get(SQLHelper.ID);
    if (id != null) {
        preparedStatement.setString(i++, String.valueOf(id));
    }
}

List<T> executeQuery(String sql, Builder<T> builder, List<Object>
parameters) throws RepositoryException {
    List<T> objects = new ArrayList<>();
    try {
        PreparedStatement preparedStatement =
connection.prepareStatement(sql);
        prepare(preparedStatement, parameters);
        ResultSet resultSet = preparedStatement.executeQuery();
        while (resultSet.next()) {
            T item = builder.build(resultSet);
            objects.add(item);
        }
    } catch (SQLException e) {
        throw new RepositoryException(e.getMessage(), e);
    }
    return objects;
}

protected Optional<T> executeQueryForSingleResult(String query, Builder<T>
builder, List<Object> parameters) throws RepositoryException {
    List<T> items = executeQuery(query, builder, parameters);
    return items.size() == 1 ?
        Optional.of(items.get(0)) :
        Optional.empty();
}

protected abstract Map<String, Object> getFields(T obj);

@Override
public Integer save(T object) throws RepositoryException {
    String sql;
    Map<String, Object> fields = getFields(object);
    sql = SQLHelper.makeInsertQuery(fields, getTableName());
    return executeSave(sql, fields);
}

```



```

        private Integer executeSave(String query, Map<String, Object> fields)
        throws RepositoryException {
            try {
                PreparedStatement preparedStatement =
connection.prepareStatement(query, Statement.RETURN_GENERATED_KEYS);
                preparedStatement.prepare(preparedStatement, fields, getTableName());
                LOGGER.info(preparedStatement.toString());
                preparedStatement.executeUpdate();
                ResultSet resultSet = preparedStatement.getGeneratedKeys();
                Integer generatedId = null;
                while (resultSet.next()) {
                    generatedId = resultSet.getInt(1);
                }
                return generatedId;
            } catch (SQLException e) {
                throw new RepositoryException(e.getMessage(), e);
            }
        }

@Override
public List<T> findAll() throws RepositoryException {
    Builder builder = BuilderFactory.create(getTableName());
    String query = GET_ALL_QUERY + getTableName();
    return executeQuery(query, builder, Collections.emptyList());
}
}

```

Класс *RepositoryCreator* – инициализирует *ConnectionPool* и соединение с бд и предоставляет один из репозиториях *PersonRepository* или *UserRepository*

```

package by.patsei.repository;

import by.patsei.connection.ConnectionPool;
import java.sql.Connection;

public class RepositoryCreator implements AutoCloseable {
    private ConnectionPool connectionPool;
    private Connection connection;

    public RepositoryCreator() {
        connectionPool = ConnectionPool.getInstance();
        connection = connectionPool.getConnection();
    }

    public UserRepository getUserRepository() {
        return new UserRepository(connection);
    }

    public PersonRepository getPersonRepository() {
        return new PersonRepository(connection);
    }

@Override
public void close() {
    connectionPool.releaseConnection(connection);
}
}

```

Теперь репозитории. *UserRepository* добавляет необходимые методы для работы с таблицей users.

```

package by.patsei.repository;

```

```

import by.patsei.builder.UserBuilder;
import by.patsei.exception.RepositoryException;
import by.patsei.model.User;
import by.patsei.repository.dbconstants.UserTableConstants;
import by.patsei.repository.specification.Parameter;

import java.sql.Connection;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Optional;

public class UserRepository extends AbstractRepository <User>{

    private static final String TABLE_NAME = "users";

    public UserRepository(Connection connection){
        super(connection);
    }

    @Override
    protected String getTableName() {
        return TABLE_NAME;
    }

    @Override
    public List<User> query(String sqlString, Parameter paramater) throws
RepositoryException {
        String query = sqlString;
        List<User> users = executeQuery(query,new UserBuilder(),
paramater.getParameters());
        return users;
    }

    @Override
    public Optional<User> queryForSingleResult(String sqlString, Parameter
parameter) throws RepositoryException {
        List<User> user = query(sqlString, parameter);
        return user.size() == 1 ?
            Optional.of(user.get(0)) :
            Optional.empty();
    }

    public Map<String,Object> getFields(User user) {
        Map<String,Object> fields = new HashMap<>();
        fields.put(UserTableConstants.LOGIN.getFieldName(), user.getLogin());
        fields.put(UserTableConstants.PASSWORD.getFieldName(),
user.getPassw());
        return fields;
    }
}

```

### *PersonRepository:*

```

package by.patsei.repository;

import by.patsei.builder.PersonBuilder;
import by.patsei.exception.RepositoryException;
import by.patsei.model.Person;
import by.patsei.repository.dbconstants.PersonTableConstants;
import by.patsei.repository.specification.Parameter;

import java.sql.Connection;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

```

```

import java.util.Optional;

public class PersonRepository extends AbstractRepository<Person> {
    private static final String TABLE_NAME = "person";

    public PersonRepository(Connection connection) {
        super(connection);
    }

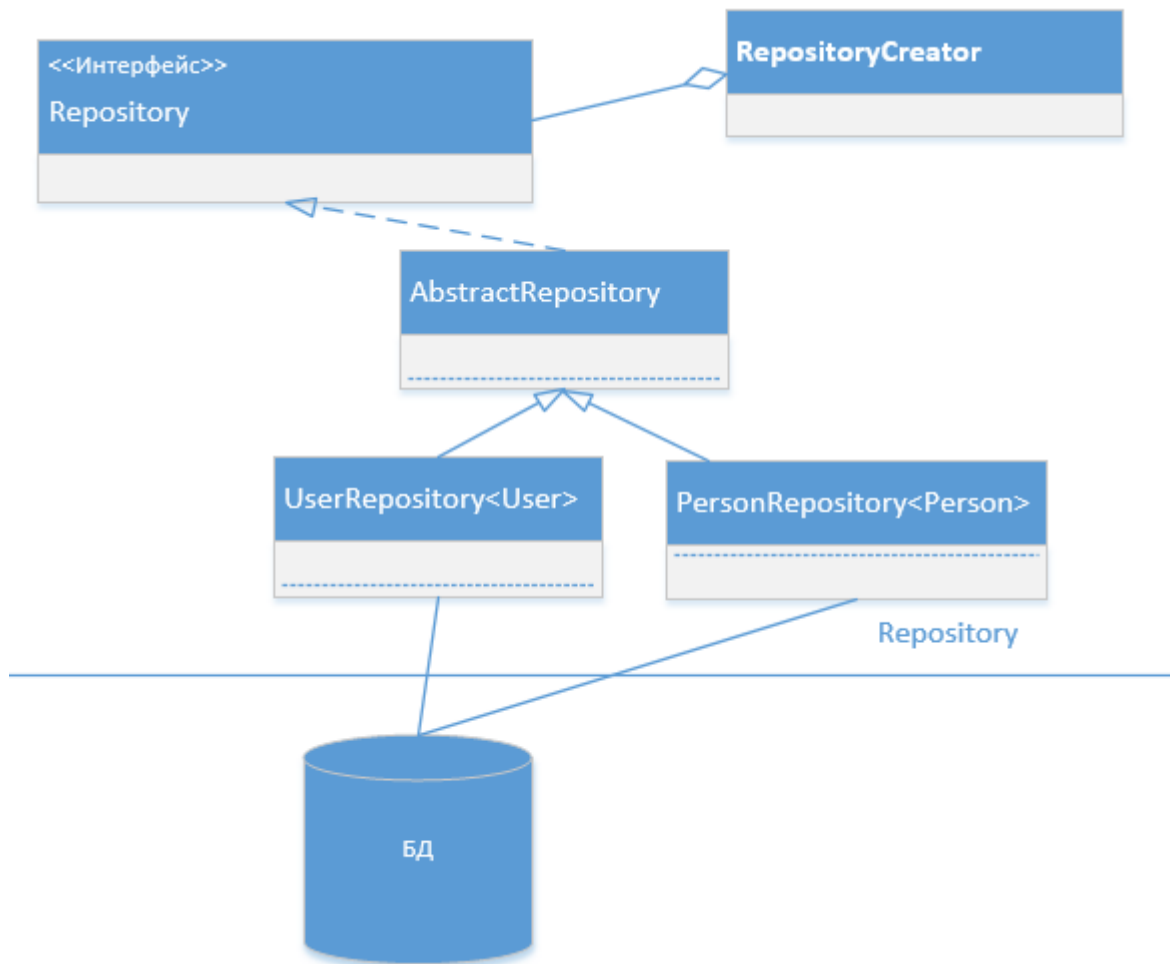
    @Override
    protected String getTableName() {
        return TABLE_NAME;
    }

    @Override
    public List<Person> query(String sqlString, Parameter paramater) throws
RepositoryException {
        String query = sqlString;
        List<Person> persons = executeQuery(query, new PersonBuilder(),
paramater.getParameters());
        return persons;
    }

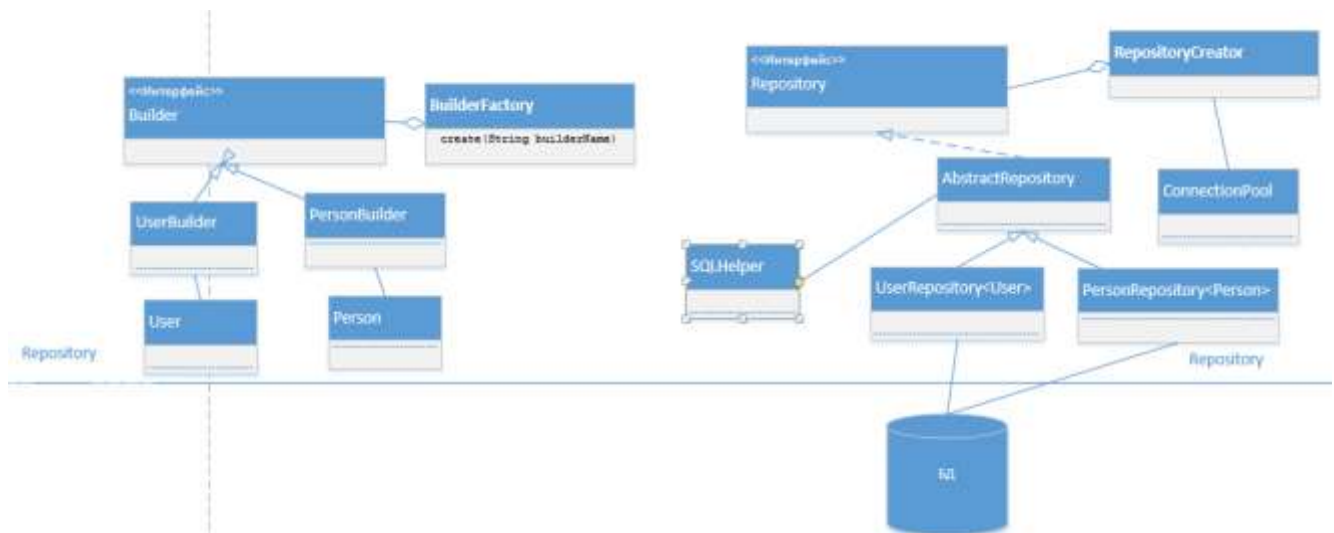
    @Override
    public Optional<Person> queryForSingleResult(String sqlString, Parameter
parameter) throws RepositoryException {
        List<Person> person = query(sqlString, parameter);
        return person.size() == 1 ?
            Optional.of(person.get(0)) :
            Optional.empty();
    }

    public Map<String, Object> getFields(Person person) {
        Map<String, Object> fields = new HashMap<>();
        fields.put(PersonTableConstants.NAME.getFieldName(), person.getName());
        fields.put(PersonTableConstants.PHONE.getFieldName(),
person.getPhone());
        fields.put(PersonTableConstants.EMAIL.getFieldName(),
person.getEmail());
        return fields;
    }
}

```



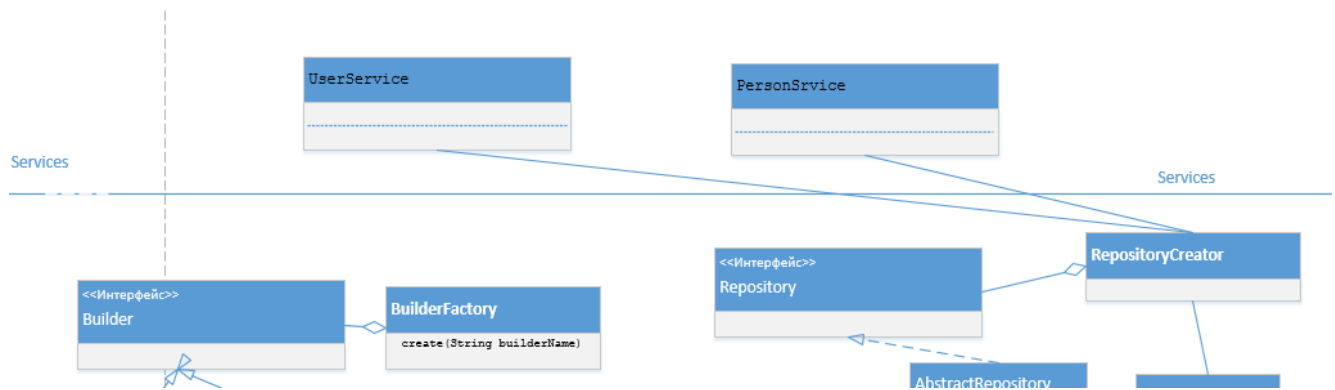
Этот слой готов.



## УРОВНЬ СЕРВИСОВ

Чем может помочь служебный уровень? Можно назвать несколько причин:

- 1) Разделить компоненты
- 2) Вы можете применять определенные бизнес-правила на своем уровне обслуживания.
- 3) Предоставить услуге фасад одного или нескольких репозиториях. Например дать **два отдельных репозитория**, участвующих в одной транзакции.



В сервис слое сидит бизнес логика, для реализации которой он обращается к разным репозиториям. В нашем приложении нужно два сервиса для User – содержит метод для входа и сохранения пользователя при регистрации

```

package by.patsei.service;

import by.patsei.exception.RepositoryException;
import by.patsei.exception.ServiceException;
import by.patsei.model.User;
import by.patsei.repository.RepositoryCreator;
import by.patsei.repository.SQLHelper;
import by.patsei.repository.UserRepository;
import by.patsei.repository.specification.UserByLogin;
import by.patsei.repository.specification.UserByLoginPassword;

import java.util.Optional;

public class UserService {

    public Optional<User> login(String login, byte[] password) throws
    ServiceException {
        try (RepositoryCreator repositoryCreator = new RepositoryCreator()) {
            UserRepository userRepository =
            repositoryCreator.getUserRepository();
            UserByLoginPassword params = new UserByLoginPassword(login,
            password);
            return userRepository.queryForSingleResult(SQLHelper.SQL_GET_USER,
            params);
        } catch (RepositoryException e) {
            throw new ServiceException(e.getMessage(), e);
        }
    }

    public Integer save(User user) throws ServiceException {
        try (RepositoryCreator repositoryCreator = new RepositoryCreator()) {
            UserRepository userRepository =
            repositoryCreator.getUserRepository();
            UserByLogin param = new UserByLogin(user.getLogin());
            if (!userRepository.queryForSingleResult(SQLHelper.SQL_CHECK_LOGIN,
            param).isPresent()) {
                return userRepository.save(user);
            } else {
                return 0;
            }
        } catch (RepositoryException exception) {
            throw new ServiceException(exception.getMessage(), exception);
        }
    }
}

```

и для Person – извлечение списка персон и сохранение новой персоны:

```
package by.patsei.service;

import by.patsei.exception.RepositoryException;
import by.patsei.exception.ServiceException;
import by.patsei.model.Person;
import by.patsei.repository.PersonRepository;
import by.patsei.repository.RepositoryCreator;

import java.util.List;

public class PersonService {

    public List<Person> findAll() throws ServiceException {
        try (RepositoryCreator repositoryCreator = new RepositoryCreator()) {
            PersonRepository personRepository =
repositoryCreator.getPersonRepository();
            return personRepository.findAll();
        } catch (RepositoryException e) {
            throw new ServiceException(e.getMessage(), e);
        }
    }

    public void save(Person person) throws ServiceException {
        try (RepositoryCreator repositoryCreator = new RepositoryCreator()) {
            PersonRepository personRepository =
repositoryCreator.getPersonRepository();
            personRepository.save(person);
        } catch (RepositoryException exception) {
            throw new ServiceException(exception.getMessage(), exception);
        }
    }
}
```

Таким образом, служба содержит информацию и функциональные возможности, связанные с выполнением атомных единиц работы. Он должен концептуально отображаться в задачах.

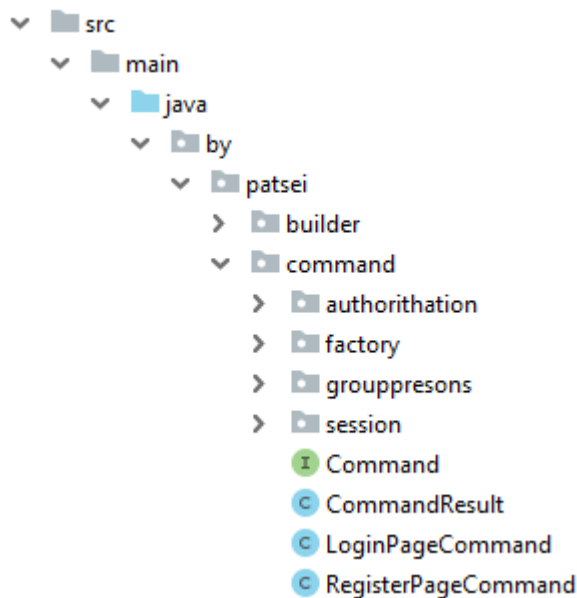
## УРОВЕНЬ БИЗНЕС\_ЛОГИКИ (Command)

. Бизнес-логика — это реализация предметной области в информационной системе. К ней относятся, например, формулы расчёта ежемесячных выплат по ссудам, автоматизированная отправка сообщений электронной почты руководителю проекта по окончании выполнения частей задания, отказ от отеля при отмене рейса авиакомпанией и т. д.

Часто на этом уровне реализуют шаблон *command facade*, который является комбинацией двух шаблонов - *facade* и *command*.

Шаблон команда отличается инкапсулированием отдельных действий в объекты повторного использования, чье поведение может быть параметризовано для каждого запроса.

Для этого в проекте создайте пакет **command** с вложенными пакетами (см. рис).



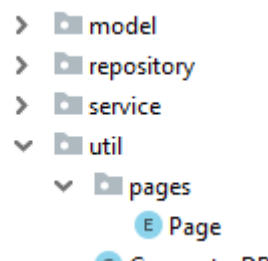
Для удобства управления командами их разделяют на группы (пакеты). Начнем разработку с интерфейса *Command*:

```
package by.patsei.command;

import by.patsei.exception.IncorrectDataException;
import by.patsei.exception.ServiceException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public interface Command {
    CommandResult execute(HttpServletRequest request, HttpServletResponse response) throws ServiceException, IncorrectDataException, ServletException, IOException;
}
```

Команды будут перенаправлять на разные страницы приложения, поэтому в пакете **util** создайте пакет **pages** с перечислением всех страниц



```
package by.patsei.util.pages;

public enum Page {

    LOGIN_PAGE("/WEB-INF/views/login.jsp"),
    REGISTER_PAGE("/WEB-INF/views/register.jsp"),
    WELCOME_PAGE("/WEB-INF/views/welcome.jsp"),
    ERROR_PAGE("/WEB-INF/views/errorPage.jsp");

    private final String value;

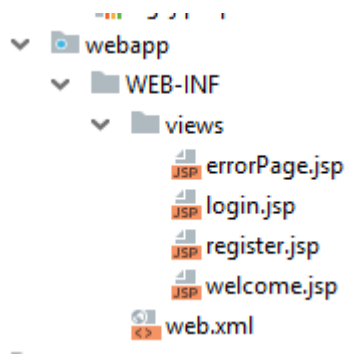
    Page(String value) {
        this.value = value;
    }
}
```

```

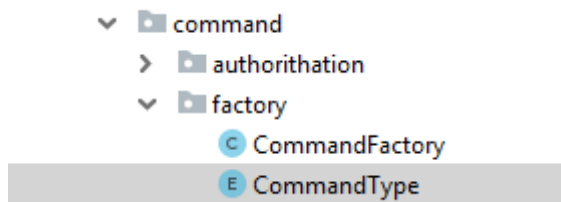
    }

    public String getPage() {
        return value;
    }
}

```



Пакет **factory** реализует фабрику для команд.



В перечислении *CommandType* – содержатся все команды:

```

package by.patsei.command.factory;
/*
Хранилище команд
*/
public enum CommandType {

    LOGIN("login"),
    SIGN_OUT("sign_out"),
    WELCOME("welcome"),
    REGISTER_NEW_USER("register_new_user"),
    ADD_NEW_PERSON ("add_new_person"),
    LOGIN_PAGE("login_page"),
    REGISTRATION_PAGE("registration_page");

    private String command;
    private CommandType(String command) {
        this.command = command;
    }
}

```

Класс *CommandFactory* создает объект команду по запросу:

```

package by.patsei.command.factory;

import by.patsei.command.Command;
import by.patsei.command.LoginPageCommand;
import by.patsei.command.RegisterPageCommand;
import by.patsei.command.authorithation.LoginCommand;
import by.patsei.command.authorithation.RegisterNewUserCommand;
import by.patsei.command.authorithation.SingOutCommand;
import by.patsei.command.grouppresons.AddNewPersonCommand;
import by.patsei.command.grouppresons.WelcomeCommand;

```



//Создает команды

```
public class CommandFactory {
    public static Command create(String command) {
        command = command.toUpperCase();
        System.out.println(command);
        CommandType commandEnum = CommandType.valueOf(command);
        Command resultCommand;
        switch (commandEnum) {
            case LOGIN: {
                resultCommand = new LoginCommand();
                break;
            }
            case REGISTER_NEW_USER: {
                resultCommand = new RegisterNewUserCommand();
                break;
            }
            case SIGN_OUT: {
                resultCommand = new SingOutCommand();
                break;
            }
            case ADD_NEW_PERSON: {
                resultCommand = new AddNewPersonCommand();
                break;
            }
            case LOGIN_PAGE: {
                resultCommand = new LoginPageCommand();
                break;
            }
            case WELCOME: {
                resultCommand = new WelcomeCommand();
                break;
            }
            case REGISTRATION_PAGE: {
                resultCommand = new RegisterPageCommand();
                break;
            }
            default: {
                throw new IllegalArgumentException("Invalid command" +
commandEnum);
            }
        }
        return resultCommand;
    }
}
```

Следующий класс *CommandResult*. Он нужен для установи страницы, если будет нужен переход и установки признака – перенаправление или переадресация.

```
package by.patsei.command;

import java.util.Objects;
public class CommandResult {
    private String page;
    private boolean isRedirect;

    public CommandResult() {
    }
}
```

```

public CommandResult(String page) {
    this.page = page;
}

public CommandResult(String page, boolean isRedirect) {
    this.page = page;
    this.isRedirect = isRedirect;
}

public String getPage() {
    return page;
}

public void setPage(String page) {
    this.page = page;
}

public boolean isRedirect() {
    return isRedirect;
}

@Override
public boolean equals(Object o) {
    if (this == o) {
        return true;
    }
    if (o == null || getClass() != o.getClass()) {
        return false;
    }
    CommandResult that = (CommandResult) o;
    return isRedirect() == that.isRedirect() &&
        Objects.equals(getPage(), that.getPage());
}

@Override
public int hashCode() {
    return Objects.hash(getPage(), isRedirect());
}
}

```

Класс *LoginPageCommand* будет использоваться для перехода на страницу логина:

```

package by.patsei.command;

import by.patsei.exception.IncorrectDataException;
import by.patsei.exception.ServiceException;
import by.patsei.util.pages.Page;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class LoginPageCommand implements Command {
    @Override
    public CommandResult execute(HttpServletRequest request,
        HttpServletResponse response) throws ServiceException, IncorrectDataException {
        System.out.println("LOGIN PAGE");
        return new CommandResult(Page.LOGIN_PAGE.getPage(), false);
    }
}

```

Аналогично класс *RegisterPageCommand* будет использоваться для перехода на страницу регистрации:

```

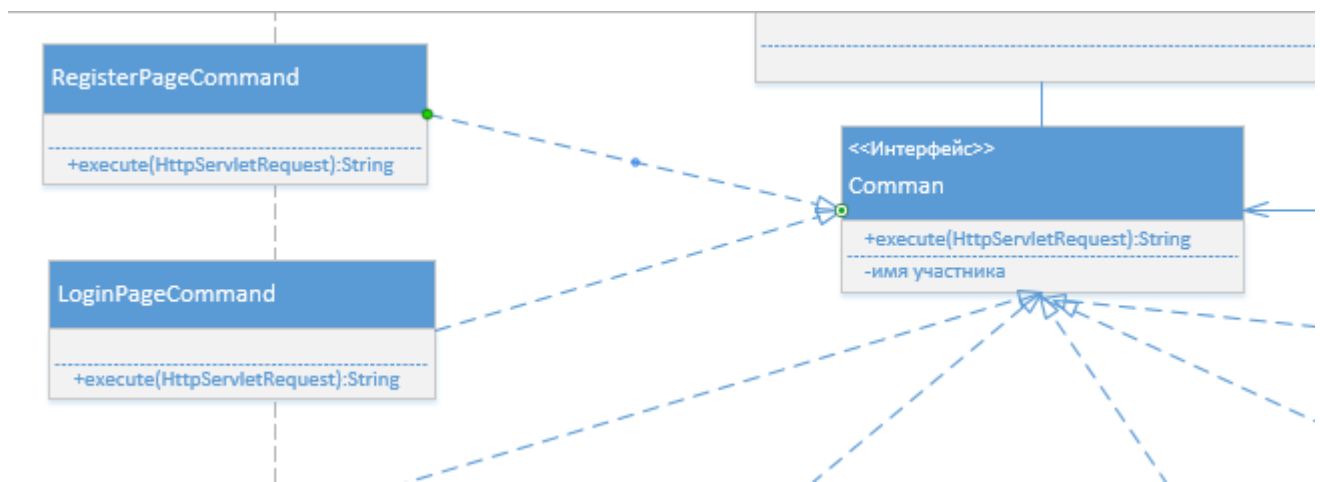
package by.patsei.command;

import by.patsei.exception.IncorrectDataException;
import by.patsei.exception.ServiceException;
import by.patsei.util.pages.Page;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public class RegisterPageCommand implements Command {
    @Override
    public CommandResult execute(HttpServletRequest request,
        HttpServletResponse response) throws ServiceException, IncorrectDataException,
        ServiceException, IOException {
        System.out.println("REGISTER_PAGE");
        return new CommandResult(Page.REGISTER_PAGE.getPage(), false);
    }
}

```



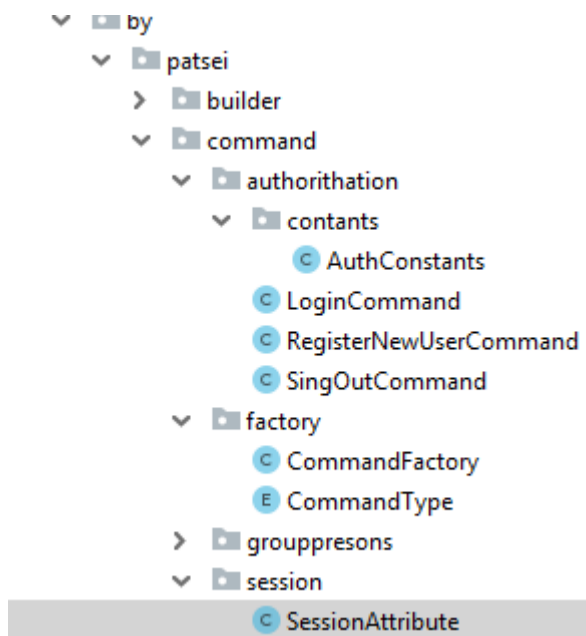
Пакет **session** содержит класс *SessionAttribute* с константами всех ключей атрибутов сессии на этом уровне. Такой атрибут один - имя пользователя. Можно использовать еще роль.

```

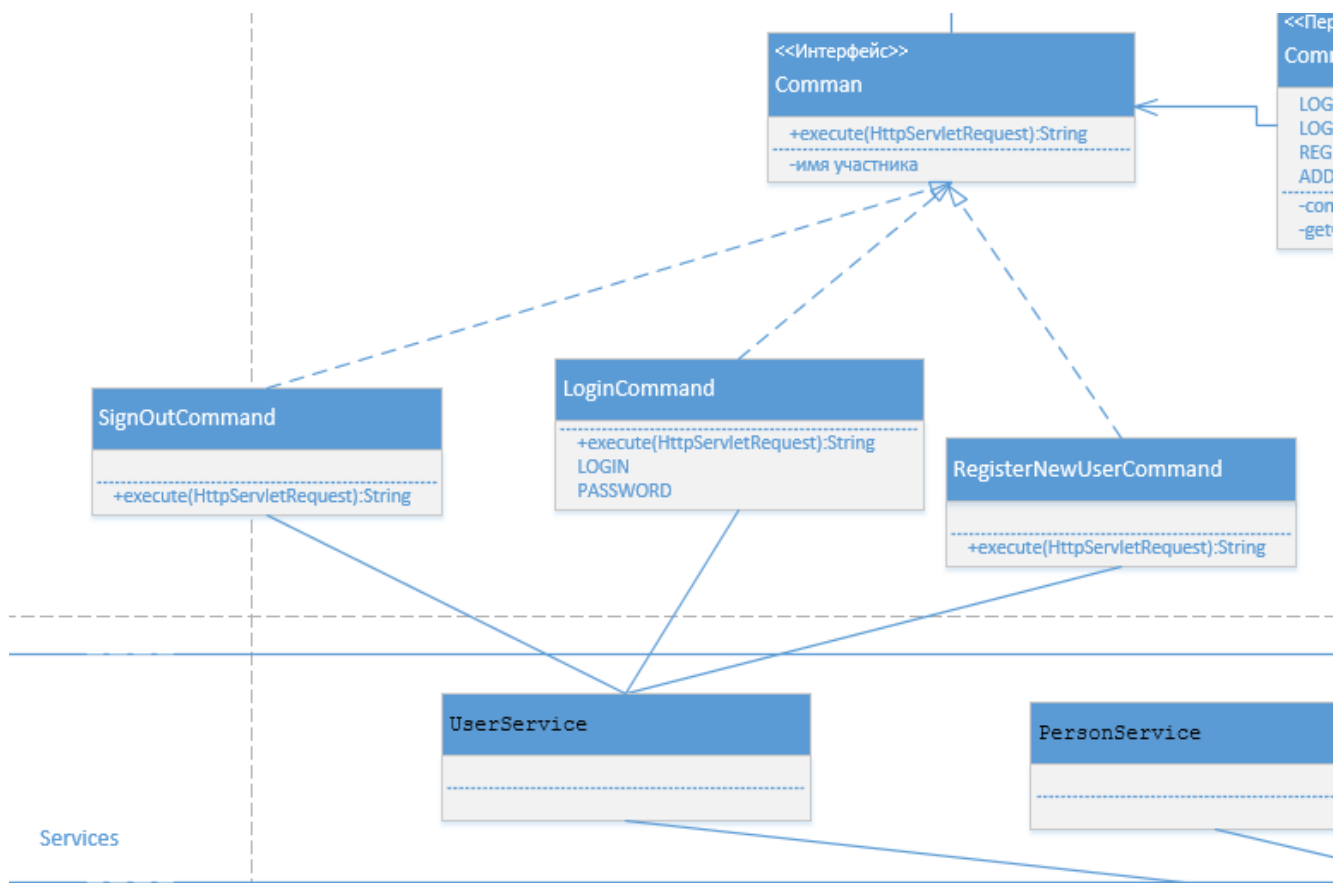
package by.patsei.command.session;

public class SessionAttribute {
    public final static String NAME = "username";
    public final static String ROLE = "role";
}

```



Следующий пакет **authorithation** – инкапсулирует логику аутентификации пользователей.



*AuthConstants* - класс с константами для сообщений, ключей параметров и т.п..

```

package by.patsei.command.authorithation.contants;

public class AuthConstants {

    //Login.jsp
    public final static String LOGIN = "loginName";
    public final static String PASSWORD = "password";
}
    
```

```
//ERROR MESSAGE
    public final static String ERROR_MESSAGE = "errorMessage";
    public final static String ERROR_MESSAGE_TEXT = "Неверный логин или
    пароль, заполните все поля";
    public final static String AUTHENTICATION_ERROR_TEXT = "Неверный логи или
    пароль!!";
    public final static String REGISTER_ERROR_MESSAGE_IF_EXIST = "Выберите
    другое имя, такой пользователь существует";
    public final static String REGISTER_ERROR = "errorRegister";
    public final static String COMMAND_WELCOME = "/controller?command=welcome";

//REGISTER JSP
    public final static String NAME_FOR_REGISTER = "newLoginName";
    public final static String PASSWORD_FOR_REGISTER = "newPassword";
}
```

## LoginCommand:

```
package by.patsei.command.authorithation;

import by.patsei.command.Command;
import by.patsei.command.CommandResult;
import by.patsei.command.session.SessionAttribute;
import by.patsei.exception.IncorrectDataException;
import by.patsei.exception.ServiceException;
import by.patsei.model.User;
import by.patsei.service.UserService;
import by.patsei.util.HashPassword;
import by.patsei.util.pages.Page;
import org.apache.log4j.Logger;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import java.io.IOException;
import java.util.Optional;

import static by.patsei.command.authorithation.contants.AuthConstants.*;
import static java.util.Optional.of;
import static org.apache.commons.lang3.StringUtils.isEmpty;

public class LoginCommand implements Command {

    private static final Logger LOGGER =
        Logger.getLogger(LoginCommand.class.getName());

    private void setAttributesToSession(String name, HttpServletRequest
request) {
        HttpSession session = request.getSession();
        session.setAttribute(SessionAttribute.NAME, name);
    }

    @Override
    public CommandResult execute(HttpServletRequest request,
HttpServletResponse response) throws ServiceException, IncorrectDataException,
ServletException, IOException {
        boolean isUserFind = false;
        Optional<String> login = of(request)
            .map(httpServletRequest ->
                httpServletRequest.getParameter(LOGIN));
        Optional<String> password = of(request)
            .map(httpServletRequest ->
                httpServletRequest.getParameter(PASSWORD));
        if (isEmpty(login.get()) || isEmpty(password.get())) {
```

```

        return forwardLoginWithError(request, ERROR_MESSAGE,
ERROR_MESSAGE_TEXT);
    }
    byte[] pass = HashPassword.getHash(password.get());
    isUserFind = initializeUserIfExist(login.get(), pass, request);
    if (!isUserFind) {
        LOGGER.info("user with such login and password doesn't exist");
        return forwardLoginWithError(request, ERROR_MESSAGE,
AUTHENTICATION_ERROR_TEXT);
    } else {
        LOGGER.info("user has been authorized: login:" + login + "
password:" + password);
        return new CommandResult(COMMAND_WELCOME, false);
    }
}

public boolean initializeUserIfExist(String login, byte[] password,
HttpServletRequest request) throws ServiceException {
    UserService userService = new UserService();
    Optional<User> user = userService.login(login, password);
    boolean userExist = false;
    if (user.isPresent()) {
        setAttributesToSession(user.get().getLogin(), request);
        userExist = true;
    }
    return userExist;
}

private CommandResult forwardLoginWithError(HttpServletRequest request,
final String ERROR, final String ERROR MESSAGE) {
    request.setAttribute(ERROR, ERROR MESSAGE);
    return new CommandResult(Page.LOGIN_PAGE.getPage(), false);
}
}

```

*SingOutCommand* – очищает атрибуты и переходит на страницу логина

```

package by.patsei.command.authorithation;

import by.patsei.command.Command;
import by.patsei.command.CommandResult;
import by.patsei.command.session.SessionAttribute;
import by.patsei.exception.IncorrectDataException;
import by.patsei.exception.ServiceException;
import by.patsei.util.pages.Page;
import org.apache.log4j.Logger;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class SingOutCommand implements Command {

    private static final Logger LOGGER =
    Logger.getLogger(SingOutCommand.class.getName());

    @Override
    public CommandResult execute(HttpServletRequest request,
    HttpServletResponse response) throws ServiceException, IncorrectDataException {
        HttpSession session = request.getSession();
        String username =
        (String)session.getAttribute(SessionAttribute.NAME);
        LOGGER.info("user was above: name:" + username);
        session.removeAttribute(SessionAttribute.NAME);
        return new CommandResult(Page.LOGIN_PAGE.getPage(), false);
    }
}

```

*RegisterNewUserCommand* - регистрирует нового пользователя при несовпадении логина с существующими пользователями

```
package by.patsei.command.authorithation;

import by.patsei.command.Command;
import by.patsei.command.CommandResult;
import by.patsei.exception.IncorrectDataException;
import by.patsei.exception.ServiceException;
import by.patsei.model.User;
import by.patsei.service.UserService;
import by.patsei.util.HashPassword;
import by.patsei.util.pages.Page;
import org.apache.log4j.Logger;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.util.Optional;
import static by.patsei.command.authorithation.contants.AuthConstants.*;
import static java.util.Optional.of;
import static org.apache.commons.lang3.StringUtils.isEmpty;

public class RegisterNewUserCommand implements Command {

    private static final Logger LOGGER =
        Logger.getLogger(RegisterNewUserCommand.class.getName());

    private CommandResult forwardToRegisterWithError(HttpServletRequest request,
        String ERROR, String ERROR_MESSAGE) {
        request.setAttribute(ERROR, ERROR_MESSAGE);
        return new CommandResult(Page.REGISTER_PAGE.getPage(), false);
    }

    private CommandResult forwardToLogin(HttpServletRequest request) {
        return new CommandResult(Page.LOGIN_PAGE.getPage(), false);
    }

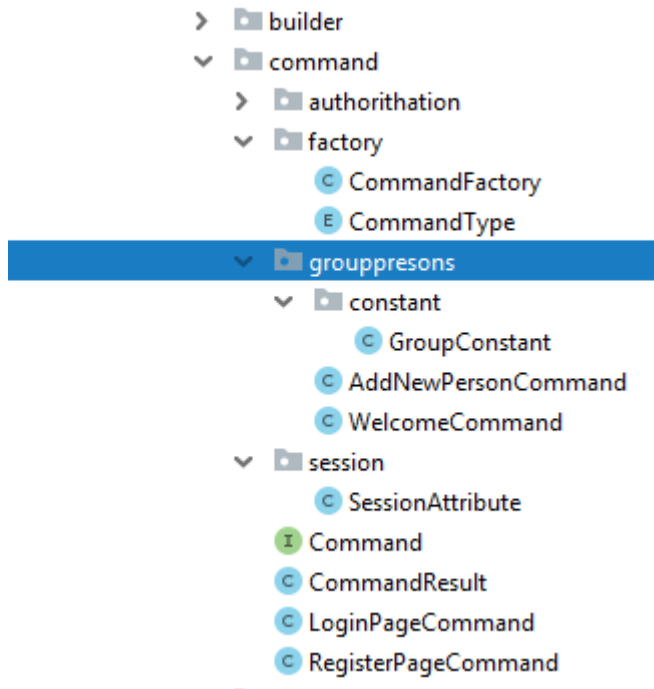
    @Override
    public CommandResult execute(HttpServletRequest request,
        HttpServletResponse response) throws ServiceException, IncorrectDataException {
        Optional<String> login = of(request)
            .map(httpServletRequest ->
                httpServletRequest.getParameter(NAME_FOR_REGISTER));
        Optional<String> password = of(request)
            .map(httpServletRequest ->
                httpServletRequest.getParameter(PASSWORD_FOR_REGISTER));
        if (isEmpty(login.get()) || isEmpty(password.get())) {
            LOGGER.info("invalid login or password format was received:" +
                login + " " + password);
            return forwardToRegisterWithError(request, REGISTER_ERROR,
                ERROR_MESSAGE_TEXT);
        }
        byte[] pass = HashPassword.getHash(password.get());
        User user = new User(login.get(), pass);
        UserService userService = new UserService();
        int userCount = userService.save(user);
        if (userCount != 0) {
            LOGGER.info("user was registered: login:" + login + " password:" +
                password);
            return forwardToLogin(request);
        } else {
            LOGGER.info("invalid login or password format was received:" +
                login + " " + password);
            return forwardToRegisterWithError(request, REGISTER_ERROR,
                REGISTER_ERROR_MESSAGE_IF_EXIST);
        }
    }
}
```

```

    }
}
}

```

В отдельном пакете **grouppresons** разместим команды работы с персонами.



Класс констант *GroupConstant*

```

package by.patsei.command.grouppresons.constant;

public class GroupConstant {

    //welcome.jsp
    public static final String NEWNAME = "nname";
    public static final String NEWPHONE = "nphone";
    public static final String NEWEMAIL = "nemail";

    //ERROR MESSAGE
    public final static String ERROR_MESSAGE = "errorMessage";
    public final static String ERROR_MESSAGE_TEXT = "Заполните все поля";
    public final static String LISTGROUP = "group";
}

```

Класс *WelcomeCommand* – загружает список персон и переходит к странице welcome:

```

package by.patsei.command.grouppresons;

import by.patsei.command.Command;
import by.patsei.command.CommandResult;
import by.patsei.exception.IncorrectDataException;
import by.patsei.exception.ServiceException;
import by.patsei.model.Person;
import by.patsei.service.PersonService;
import by.patsei.util.pages.Page;
import static by.patsei.command.grouppresons.constant.GroupConstant.*;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.util.List;

```



```

public class WelcomeCommand implements Command {
    @Override
    public CommandResult execute(HttpServletRequest request,
        HttpServletResponse response)
        throws ServiceException, IncorrectDataException {

        PersonService personService = new PersonService();
        List<Person> clients = personService.findAll();
        if (!clients.isEmpty()) {
            request.setAttribute(LISTGROUP, clients);
        }
        return new CommandResult(Page.WELCOME_PAGE.getPage(), false);
    }
}

```

Класс *AddNewPersonCommand* – добавляет новую персону и выводит новый СПИСОК

```

package by.patsei.command.grouppresons;

import by.patsei.command.Command;
import by.patsei.command.CommandResult;
import by.patsei.exception.IncorrectDataException;
import by.patsei.exception.ServiceException;
import by.patsei.model.Person;
import by.patsei.service.PersonService;
import by.patsei.util.pages.Page;
import org.apache.log4j.Logger;
import static by.patsei.command.grouppresons.constant.GroupConstant.*;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.util.List;
import java.util.Optional;
import static java.util.Optional.of;
import static org.apache.commons.lang3.StringUtils.isEmpty;

public class AddNewPersonCommand implements Command {
    private static final Logger LOGGER =
        Logger.getLogger(AddNewPersonCommand.class.getName());

    @Override
    public CommandResult execute(HttpServletRequest request,
        HttpServletResponse response)
        throws ServiceException, IncorrectDataException, ServletException {

        PersonService personService = new PersonService();
        Optional<String> newName = of(request)
            .map(httpServletRequest ->
                httpServletRequest.getParameter(NEWNAME));
        Optional<String> newPhone = of(request)
            .map(httpServletRequest ->
                httpServletRequest.getParameter(NEWPHONE));
        Optional<String> newEmail = of(request)
            .map(httpServletRequest ->
                httpServletRequest.getParameter(NEWEMAIL));

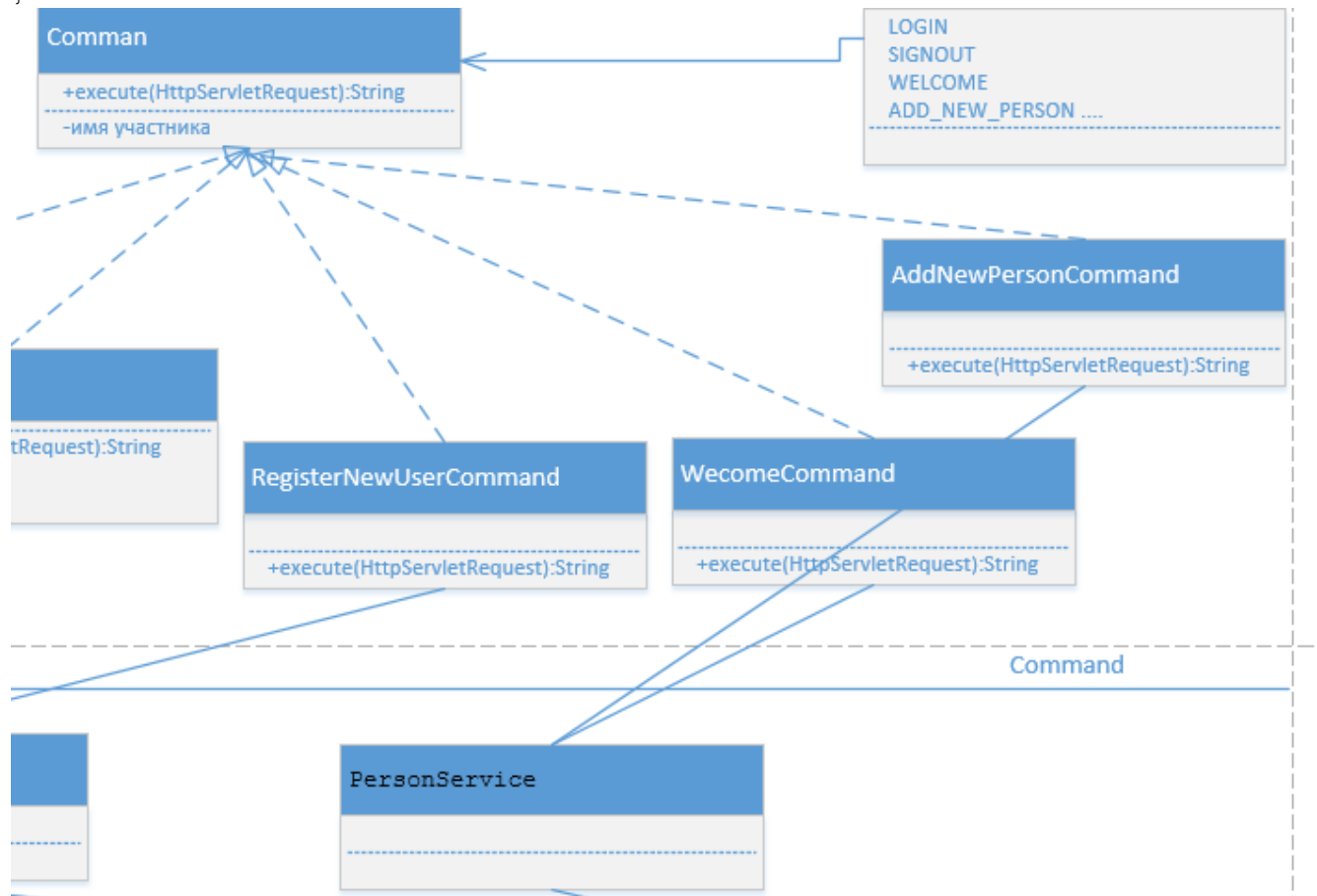
        if (isEmpty(newName.get()) || isEmpty(newPhone.get()) ||
            isEmpty(newEmail.get())) {
            LOGGER.info("missing parameter for new person in addition mode");
            request.setAttribute(ERROR_MESSAGE, ERROR_MESSAGE_TEXT);
        } else {
            Person newperson = new Person(newName.get(), newPhone.get(),

```

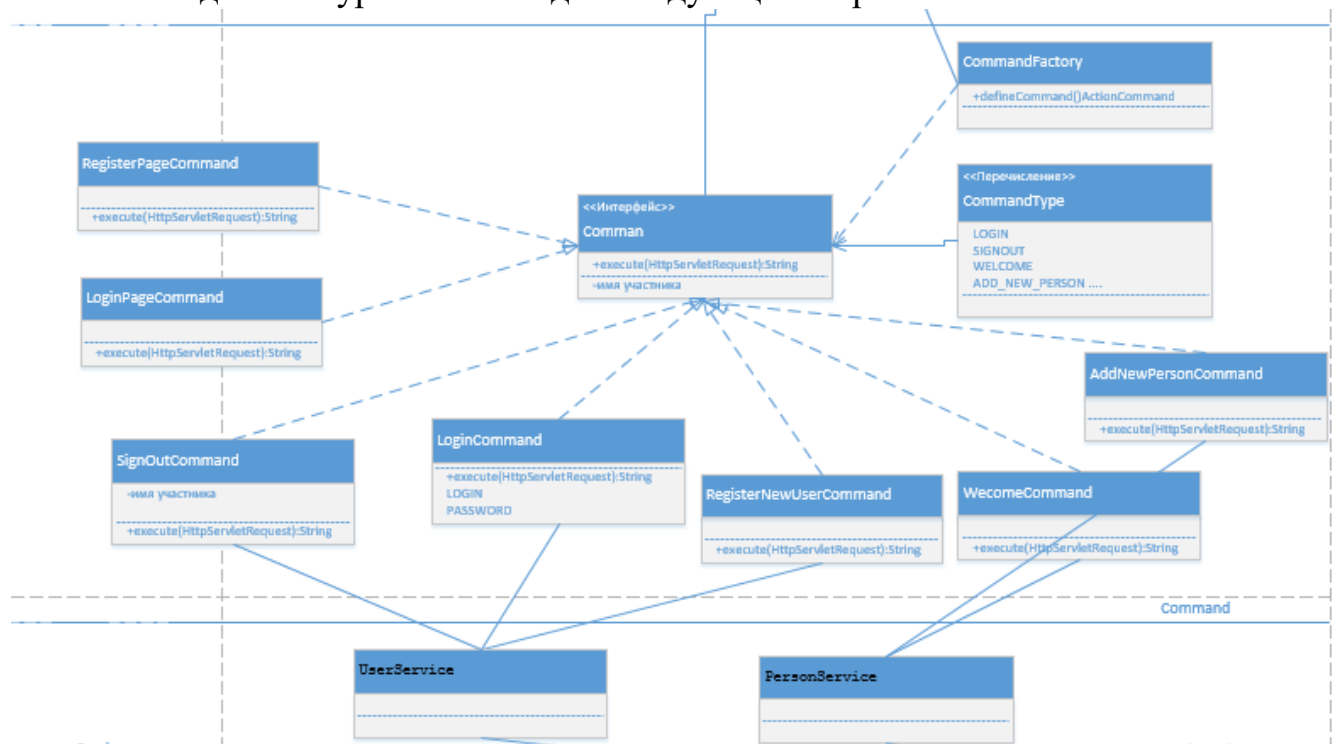
```

newEmail.get());
        personService.save(newperson);
    }
    List<Person> clients = personService.findAll();
    if (!clients.isEmpty()) {
        request.setAttribute(LISTGROUP, clients);
    }
    return new CommandResult(Page.WELCOME_PAGE.getPage(), false);
}
}

```



Итак схема данного уровня выглядит следующим образом.



## CONTROLLER

В сложных системах есть много одинаковых действий, которые надо производить во время обработки запросов. Это, например, контроль безопасности, логин и добавление пользователя. Когда поведение входного контроллера разбросано между несколькими объектами, дублируется большое количество кода. Помимо прочего возникают сложности смены поведения в реальном времени.

Паттерн Front Controller объединяет всю обработку запросов, пропуская запросы через единственный объект-обработчик. Этот объект содержит общую логику поведения, которая может быть изменена в реальном времени при помощи декораторов. После обработки запроса контроллер обращается к конкретному объекту для отработки конкретного поведения.

Эту роль в приложении будет выполнять *Controller*:

```
package by.patsei.controller;

import by.patsei.command.Command;
import by.patsei.command.CommandResult;
import by.patsei.command.factory.CommandFactory;
import by.patsei.connection.ConnectionPool;
import by.patsei.exception.IncorrectDataException;
import by.patsei.exception.ServiceException;
import by.patsei.util.pages.Page;
import org.apache.log4j.Logger;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public class Controller extends HttpServlet {

    private static final String COMMAND = "command";
    private static final String ERROR_MESSAGE = "error_message";
    private static final Logger LOGGER =
        Logger.getLogger(Controller.class.getName());

    @Override
    public void destroy() {
        ConnectionPool.getInstance().destroy();
    }

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException {
        processRequest(request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException {
        processRequest(request, response);
    }
}
```

```

    }

    private void processRequest(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {

        String command = request.getParameter(COMMAND);
        LOGGER.info(COMMAND + " = " + command);
        Command action = CommandFactory.create(command);

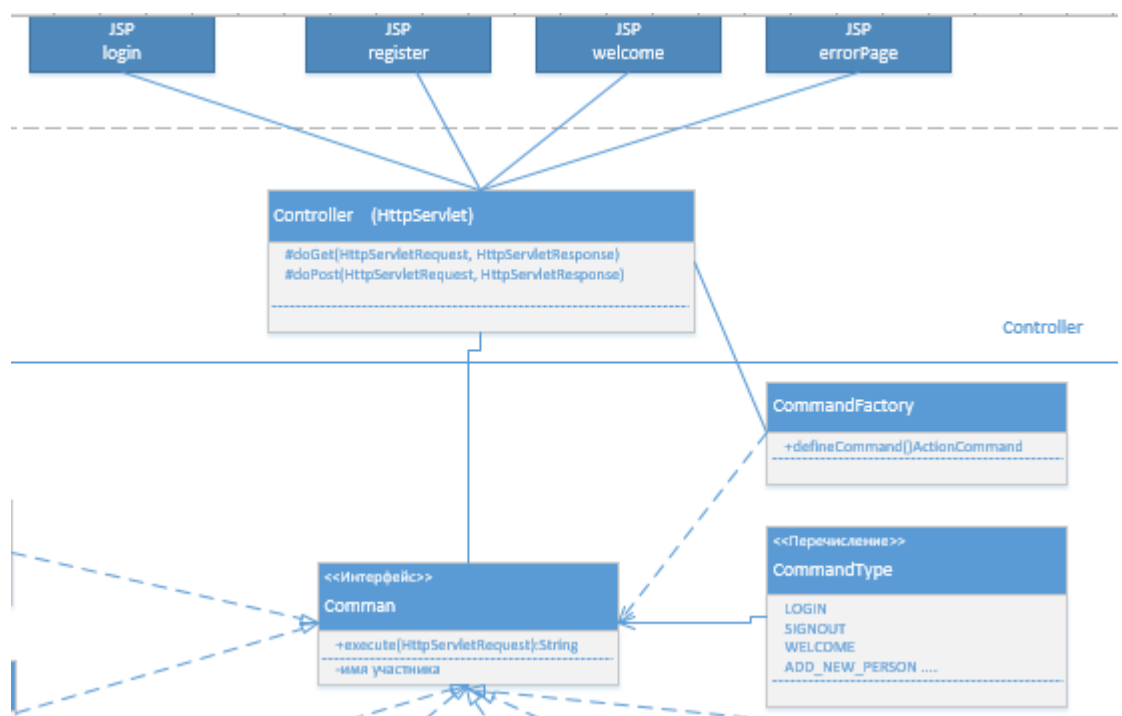
        CommandResult commandResult;
        try {
            commandResult = action.execute(request, response);
        } catch (ServiceException | IncorrectDataException e) {
            LOGGER.error(e.getMessage(), e);
            request.setAttribute(ERROR_MESSAGE, e.getMessage());
            commandResult = new CommandResult(Page.ERROR_PAGE.getPage(),
false);
        }

        String page = commandResult.getPage();
        if (commandResult.isRedirect()) {
            sendRedirect(response, page);
        } else {
            dispatch(request, response, page);
        }
    }

    private void dispatch(HttpServletRequest request, HttpServletResponse
response, String page) throws ServletException, IOException {
        ServletContext servletContext = getServletContext();
        RequestDispatcher requestDispatcher =
servletContext.getRequestDispatcher(page);
        requestDispatcher.forward(request, response);
    }

    private void sendRedirect(HttpServletResponse response, String page) throws
IOException {
        response.sendRedirect(page);
    }
}

```



Т.е. сервлет не генерирует ответ сам, а только выступает в роли контроллера запросов. Такая архитектура построения приложений носит название MVC (Model/View/Controller).

Model — классы бизнес-логики и длительного хранения, View — страницы JSP, Controller — сервлет. Применение шаблона на практике приводит к тому, что трехуровневая базовая модель распадается на многоуровневую. Число и назначение слоев может существенно отличаться в зависимости от разработанного архитектурного решения.

## FILTER

Для того чтобы фильтр работал и не допускал перехода на [http://localhost:8080/Servlet\\_war\\_exploded/controller?command=welcome](http://localhost:8080/Servlet_war_exploded/controller?command=welcome) не авторизированных пользователей, его надо немного поправить

```
package by.patsei.filter;

import by.patsei.command.session.SessionAttribute;
import by.patsei.util.pages.Page;

import javax.servlet.*;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;
import java.io.IOException;
import java.util.logging.Logger;

@WebFilter(urlPatterns = {"/controller"})
public class LoginRequiredFilter implements Filter {
    private static final String COMMAND = "command";
    private static final String WELCOME = "welcome";
    private static final String ERROR_MESSAGE = "error_message";
    private static final String ERROR_TEXT = "Нет авторизации для выполнения данной команды";
    private static final Logger LOGGER =
        Logger.getLogger(LoginRequiredFilter.class.getName());

    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain chain) throws ServletException, IOException {
        HttpServletRequest request = (HttpServletRequest) req;
        String command = request.getParameter(COMMAND);
        LOGGER.info("Filter is working " + COMMAND + " = " + command);
        if (!command.equals(WELCOME)) {
            chain.doFilter(req, resp);
        } else {
            if (request.getSession().getAttribute(SessionAttribute.NAME) != null) {
                chain.doFilter(req, resp);
            } else {
                request.setAttribute(ERROR_MESSAGE, ERROR_TEXT);
                request.getRequestDispatcher(Page.ERROR_PAGE.getPage()).forward(req, resp);
            }
        }
    }
}
```

## JSP

### Дескриптор развертывания веб-приложения web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
  version="4.0">

  <servlet>
    <servlet-name>controller</servlet-name>
    <servlet-class>by.patsei.controller.Controller</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>controller</servlet-name>
    <url-pattern>/controller</url-pattern>
  </servlet-mapping>

  <welcome-file-list>
    <welcome-file>/WEB-INF/views/login.jsp</welcome-file>
  </welcome-file-list>

</web-app>
```

определяет соответствие между путями URL и сервлетами, которые эти URL будут обрабатывать. Веб-сервер использует эту конфигурацию, чтоб определить сервлет для обработки данного запроса и вызвать метод класса.

Для определения действия – пересылаете контроллеру параметры. Например, register.jsp

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
  <title>Title</title>
</head>
<body>

<p><font color="red">${errorRegister}</font></p>

<form
  action="${pageContext.servletContext.contextPath}/controller?command=register_new_user" method="POST">
  <p> Регистрация нового пользователя </p>
  <p> Введите имя : <input name="newLoginName" type="text" />
  </p>
  <p> Введите пароль : <input name="newPassword" type="password" />
  </p>
  <input class ="button-main-page" type="submit" value="Зарегистрировать"/>

</form>
</body>
</html>
```