

EPAM Systems, RD Dep.
Конспект и раздаточный материал

Layered Architecture

The basic level of knowledge.

REVISION HISTORY					
Ver.	Description of Change	Author	Date	Approved	
				Name	Effective Date
<1.0>	Первая версия	Ольга Смолякова	<11.01.2017>		

“Расслоение” системы. Базовый уровень знаний.

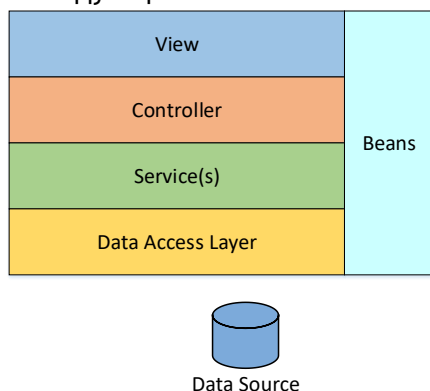
“Расслоение” системы - концепция слоев (layers) - одна из общеупотребительных моделей, используемых разработчиками программного обеспечения для разделения сложных систем на более простые части. Описывая систему в терминах архитектурных слоев, удобно воспринимать составляющие ее подсистемы в виде “слоеного пирога”.

Разделение системы на слои представляет целый ряд преимуществ:

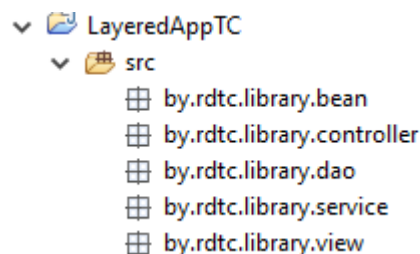
- отдельный слой можно воспринимать как единое самостоятельное целое, не особенно заботясь о наличии других слоев;
- можно выбирать альтернативную реализацию базовых слоев;
- зависимость между слоями можно свести к минимуму;
- каждый слой является удачным кандидатом на стандартизацию;
- созданный слой может служить основой для нескольких различных слоев более высокого уровня;
- слои способны удачно инкапсулировать многое (но не все);
- наличие избыточных слоев нередко снижает производительность системы.

[Мартин Фаулер, Шаблоны корпоративных приложений]

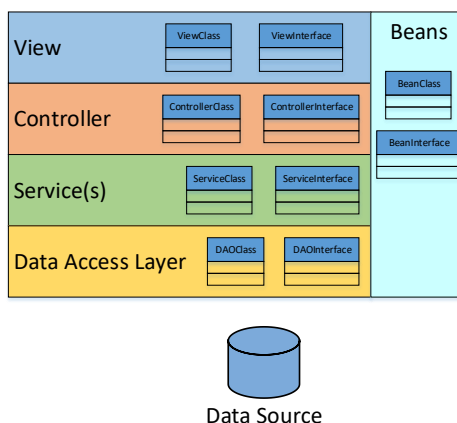
Рассмотрим программную систему, содержащую следующие слои



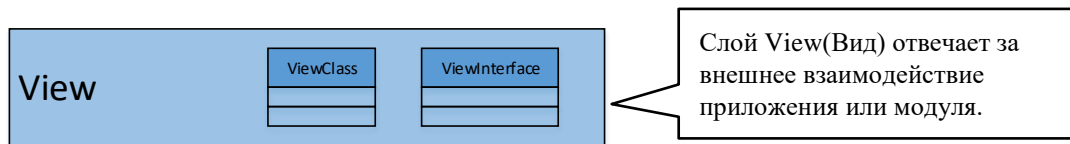
При кодировании концепция слоев может быть реализована в виде пакетов программы.



Внутри каждого слоя располагаются артефакты (классы, интерфейсы и др), обеспечивающие выполнение слоем своих непосредственных функций.

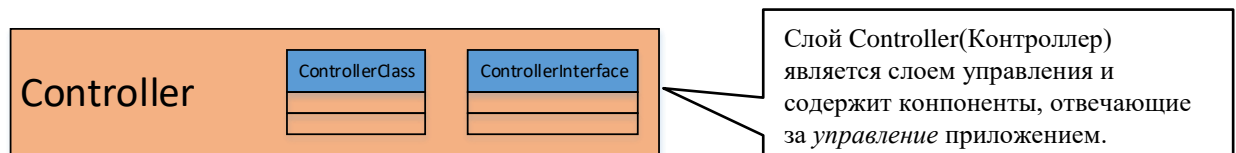


View Layer



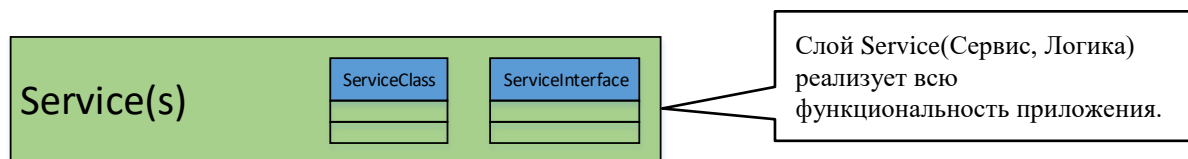
Например, Вид может отвечать за взаимодействие (ввод данных и отображение информации) с пользователем.

Controller Layer



Компоненты этого слоя не имеют право выполнять какую-либо логику. Если проводить аналогию, то слой Контроллер можно ассоциировать с директором (или целым отделом директоров ☺) фирмы, выпускающей, например, окна и двери. Контроллер решает, может ли приложение выполнить пришедший запрос, разрешено ли тому, кто прислал запрос его выполнять, и **кто** будет этот запрос выполнять. Так директор фирмы решает, будет ли фирма браться за этот заказ и какому мастеру получить его выполнение. В такой ассоциации слой Вид можно представлять *заказчиком*, который запрашивает у Контроллера какие-либо действия и ожидает от Контроллера какого-либо результата. На основании полученного от контроллера ответа Вид(он же *заказчик*) может сгенерировать следующий запрос.

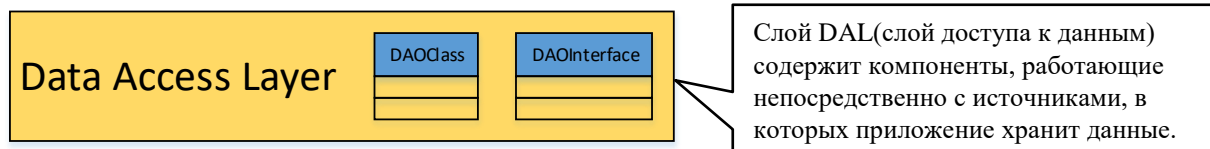
Service Layer



Именно ради действий, реализованных на этом слое, мы и используем приложение. Если приложение должно прогнозировать котировки акций на следующей неделе, то алгоритмы прогноза будут реализованы здесь. Если необходимо в системе зарегистрировать пользователя, то код регистрации будет находиться непосредственно на этом слое.

Если говорить про фирму 'окна-двери', то на слое сервисов у этой фирмы будут находиться конкретные бригады, выполняющие конкретные заказы. Необходимо окно - пожалуйста, это делает бригада номер один, дверь входная - бригада номер два и т.д.

Data Access Layer

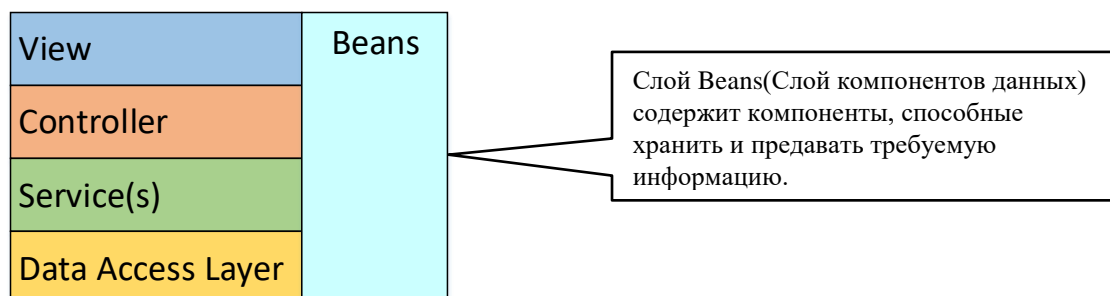


99.9% приложений используют различные системы хранения данных. Самой распространенной на текущий момент является база данных. Код, 'общающийся' с базой данных (да, в принципе, и с любыми другими источниками), зачастую очень специфичен, а не редко и громоздок. Слой DAL решает две задачи. С одной стороны он закрывает от слоя Сервисов знания о том, как непосредственно надо работать с источником; если при реализации сервисов надо получить информацию из источника или сохранить ее, код слоя сервисов будет обращаться к слою DAL с 'просьбой' это сделать. С другой стороны введение слоя доступа к данным позволяет модифицировать и менять источник, даже если это требует изменения программного кода для доступа к нему; слой Сервисов в таком случае будет в безопасности и не потребует модификаций (а это уже половина счастья ☺).

Чаще вместо аббревиатуры DAL вы будете слышать аббревиатуру DAO (Data Access Object). DAO - паттерн проектирования, многочисленные вариации которого очень часто используются при реализации слоя доступа к данным.

А вот для фирмы окна-двери слой доступа к данным будет реализован служащими, которые должны доставлять для бригад сервисов рабочие материалы и инструменты и убирать ненужное на склад, ведь работающей бригаде все равно в какой партии доставили нужное им по типу дерево или стекло.

Beans Layer

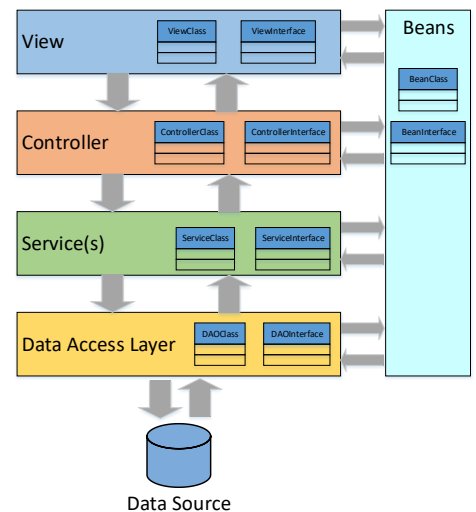


Чтобы сказать одному компоненту программы, что ему необходимо сделать, или чтобы получить результат работы компонента, приложение должно уметь обмениваться (передавать) данными. Однако таким компонентам запрещено обрабатывать данные, они могут только *хранить и предавать*.

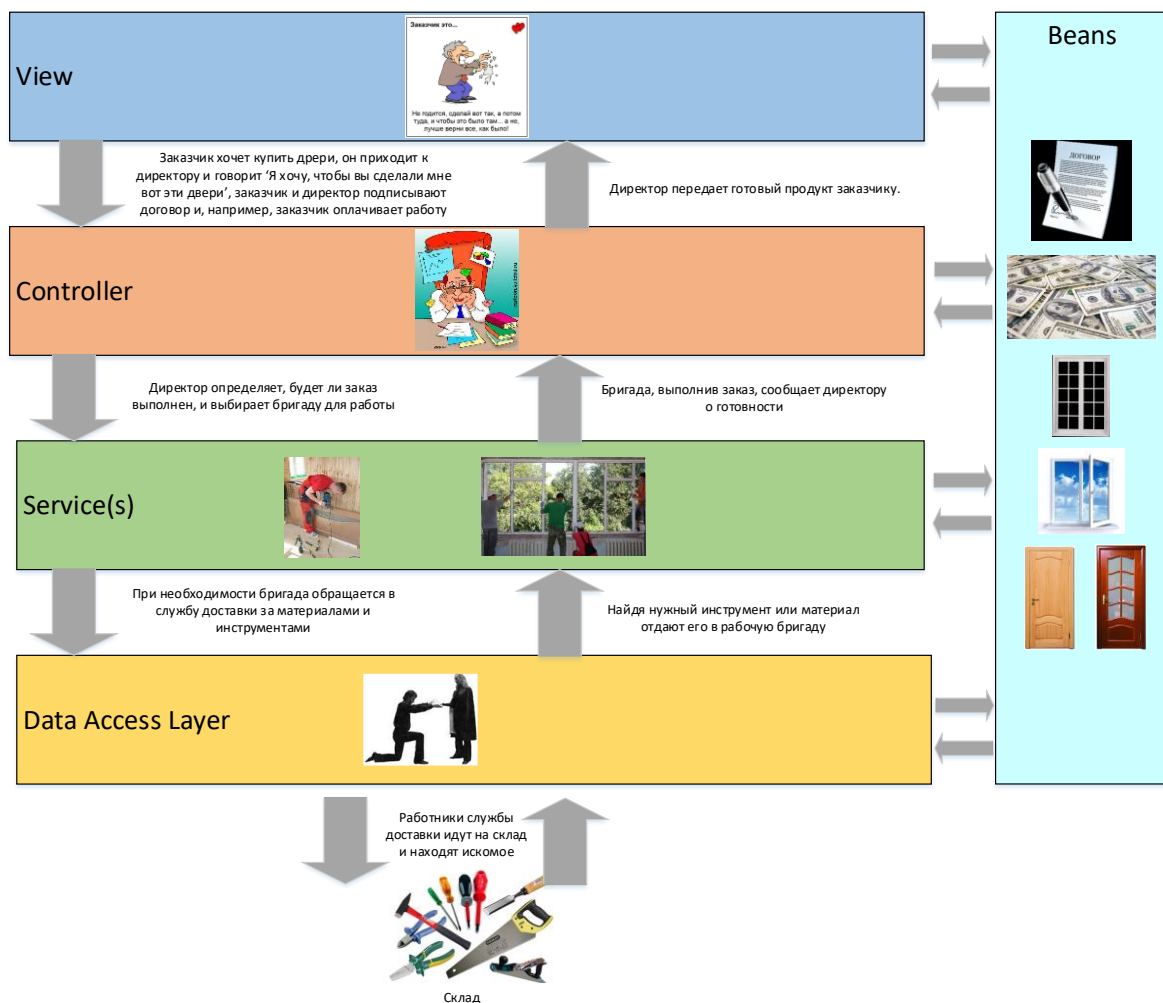
Взаимодействие слоев

Взаимодействие слоев в приложении строго регламентировано. Слой может общаться (т.е. передавать и получать данные) только от слоев, расположенных в архитектуре непосредственно выше и ниже искомого. Если перевести это на язык

программирования, то код, расположенный в слое контроллера не имеет права обращаться к коду, расположенному на слое DAL, и, соответственно, получать от этого кода какие-то данные напрямую; контроллер даже не должен знать, что внизу под сервисами что-то существует ☺.



Попытка представить картину расслоения всех выполняемых задач в реальной жизни может привести, например, к такому:



DAO (code)

Теперь переведем наши архитектурные рассуждения в код и разработаем шаблон для предметной темы “Библиотека”.

Сначала устроим мозговой штурм и определим запросы, которые в приложении будут использоваться для обращения к источнику данных. Причем нам не важно какой конкретно источник данных будет использоваться, запросы следует определять как - ‘Потребуется проверить, есть ли логин и пароль пользователя в системе’ и т.д.

Множество запросов можно разбить на смысловые группы и оформить их в коде в виде интерфейсов.

Например,

```
package by.rdtc.library.dao;

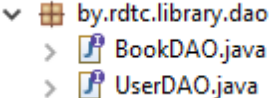
import by.rdtc.library.bean.User;

public interface UserDao {
    void signIn(String login, String password);
    void registration(User user);
}

package by.rdtc.library.dao;

import by.rdtc.library.bean.Book;

public interface BookDAO {
    void addBook(Book book);
    void deleteBook(long idBook);
    void delete(Book book);
}
```



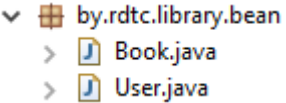
Так же при создании интерфейсов нам понадобятся бейн-классы, объекты которых будут содержать требуемую информацию о пользователе или о книге.

```
package by.rdtc.library.bean;

public class User {
    //stub
    //надеюсь, вы в курсе, какой код тут должен быть написан
}

package by.rdtc.library.bean;

public class Book {
    //stub
}
```



Следующим шагом напомним реализацию интерфейсов для слоя data access, причем источник данных в это случае уже необходимо определить. В нашем случае это будет реляционная база данных.

```
package by.rdtc.library.dao.impl;

import by.rdtc.library.bean.User;
import by.rdtc.library.dao.UserDAO;

public class SQLUserDAO implements UserDAO {

    @Override
    public void signIn(String login, String password) {
```

```
        // именно в этом методы мы связываемся с базой данных и проверяем
        корректность логина и пароля
    }
}
```

```
    @Override
    public void registration(User user) {
    }
}
```

```
package by.rdtc.library.dao.impl;
```

```
import by.rdtc.library.bean.Book;
import by.rdtc.library.dao.BookDAO;
```

```
public class SQLBookDAO implements BookDAO{
```

```
    @Override
    public void addBook(Book book) {
    }
```

```
    @Override
    public void deleteBook(long idBook) {
    }
```

```
    @Override
    public void delete(Book book) {
    }
}
```

```
by.rdtc.library.dao.impl
  > SQLBookDAO.java
  > SQLUserDAO.java
```

Теперь создадим фабрику, которая будет предлагать получить ссылки на объект, класс которого реализует требуемый DAO-интерфейс.

```
import by.rdtc.library.dao.BookDAO;
import by.rdtc.library.dao.UserDAO;
import by.rdtc.library.dao.impl.SQLBookDAO;
import by.rdtc.library.dao.impl.SQLUserDAO;
```

```
public final class DAOFactory {
    private static final DAOFactory instance = new DAOFactory();
```

```
    private final BookDAO sqlBookImpl = new SQLBookDAO();
    private final UserDAO sqlUserImpl = new SQLUserDAO();
```

```
    private DAOFactory() {}
```

```
    public static DAOFactory getInstance() {
        return instance;
    }
```

```
    public BookDAO getBookDAO() {
        return sqlBookImpl;
    }
```

```
    public UserDAO getUserDAO() {
        return sqlUserImpl;
    }
}
```

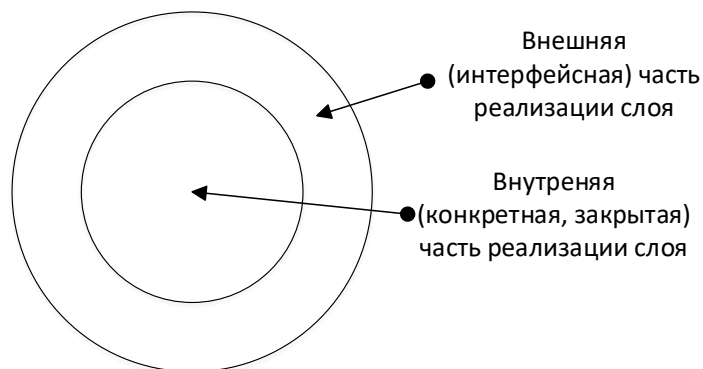
Класс DAOFactory представляет собой singleton, у которого две задачи. Первая - это закрыть от пользователя слоя конкретную реализацию. Вторая - не создавать каждый раз новые объекты типа SQLBookDAO и SQLUserDAO, т.к. многократное создания этих объектов является грубой ошибкой.

Теперь, если понадобится добавить книгу в библиотеку, то нужно будет использовать следующий код:

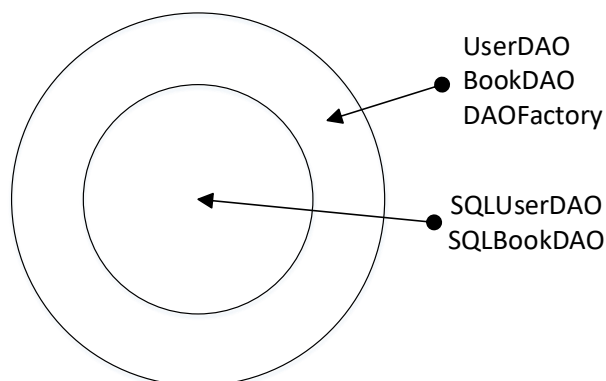
```
DAOFactory daoObjectFactory = DAOFactory.getInstance();
BookDAO bookDAO = daoObjectFactory.getBookDAO();
bookDAO.addBook(book);
```

Зачем же мы все это написали? Ради сокрытия реализации, чтобы при внесении изменения в одно место кода не наблюдать 'ядерной реакции' необходимости менять все остальное.

Реализацию слоя можно представить в виде круговых секций.



Обращаясь к слою, правильно обращаться только к его интерфейсной части. Для приведенного слоя DAO размещения компонентов в кругах выглядит следующим образом.



! Представленная реализация - это, конечно, не единственная возможность написать слой DAO. Существуют и другие решения; главное, чтобы в этих решениях не нарушались основные концепты расслоения системы.

Service (code)

Каждый слой может взаимодействовать только со слоем, лежащим непосредственно сверху и снизу. Так слой сервисов может взаимодействовать со слоем доступа к данным и контроллером.

Напишем слои сервисов аналогично структуре слоя DAO.

Сначала создадим интерфейсы.

```
package by.rdtc.library.service;

import by.rdtc.library.bean.User;

public interface ClientService {
    void singIn(String login, String password);
    void singOut(String login);
    void registration(User user);
}
```

```
package by.rdtc.library.service;

import by.rdtc.library.bean.Book;

public interface LibraryService {
    void addNewBook(Book book);
    void addEditedBook(Book book);
}
```

Потом реализуем интерфейсы.

```
package by.rdtc.library.service.impl;

import by.rdtc.library.bean.User;
import by.rdtc.library.dao.UserDAO;
import by.rdtc.library.dao.factory.DAOFactory;
import by.rdtc.library.service.ClientService;

public class ClientServiceImpl implements ClientService {

    @Override
    public void singIn(String login, String password) {
        // проверяем параметры

        // реализуем функционал логинации пользователя в системе
        DAOFactory daoObjectFactory = DAOFactory.getInstance();
        UserDAO userDAO = daoObjectFactory.getUserDAO();
        userDAO.signIn(login, password);
        //....
    }

    @Override
    public void singOut(String login) {
    }
}
```

```
@Override
public void registration(User user) {
}

}

package by.rdtc.library.service.impl;

import by.rdtc.library.bean.Book;
import by.rdtc.library.service.LibraryService;

public class LibraryServiceImpl implements LibraryService{

    @Override
    public void addNewBook(Book book) {
    }

    @Override
    public void addEditedBook(Book book) {
    }

}
```

!!! Каждый открытый метод реализации слоя сервисов имеет обязанность проверять входящие параметры (кто бы и где бы до него это не делал)!

Далее предоставим возможность получения доступа к реализации, не открывая имена конкретных классов.

```
package by.rdtc.library.service.factory;

import by.rdtc.library.service.ClientService;
import by.rdtc.library.service.LibraryService;
import by.rdtc.library.service.impl.ClientServiceImpl;
import by.rdtc.library.service.impl.LibraryServiceImpl;

public final class ServiceFactory {
    private static final ServiceFactory instance = new ServiceFactory();

    private final ClientService clientService = new ClientServiceImpl();
    private final LibraryService libraryService = new LibraryServiceImpl();

    private ServiceFactory() {}

    public static ServiceFactory getInstance() {
        return instance;
    }

    public ClientService getClientService() {
        return clientService;
    }

    public LibraryService getLibraryService() {
        return libraryService;
    }

}
```

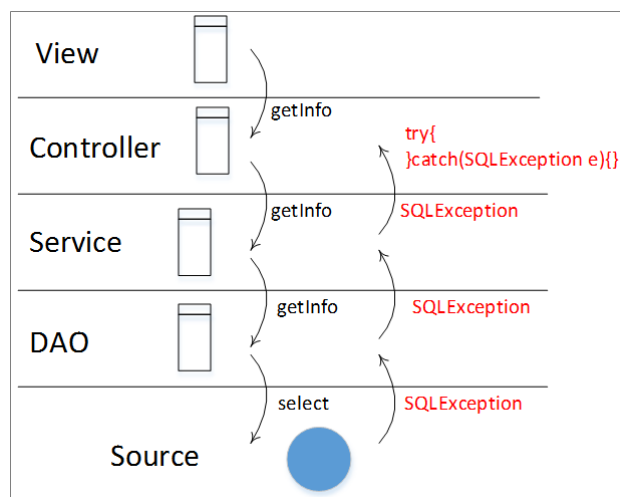
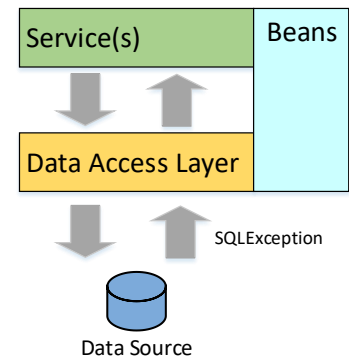
Итак, слой DAO и слой Service у нас есть. Слой Service обращаясь к DAO использует только интерфейсную его часть, и не перегружен знанием конкретной реализации, используемой при доступе к данным.

Обработка исключений

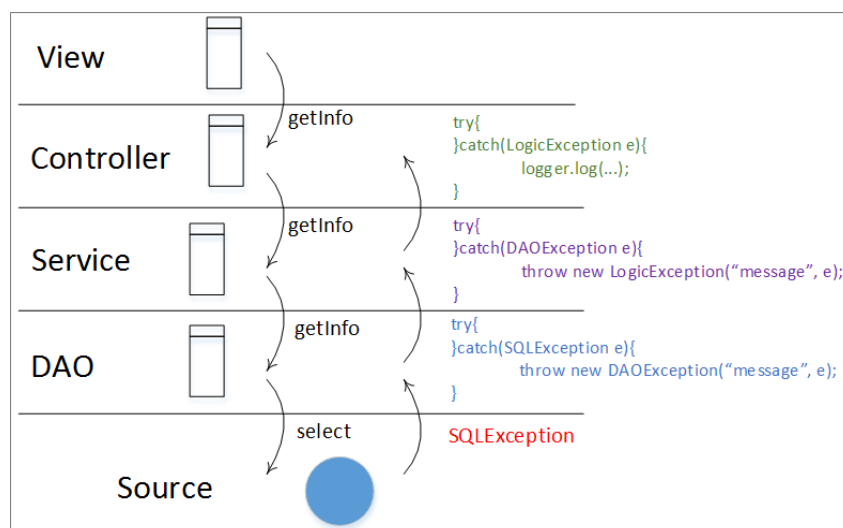
Далее разберемся с тем, как обрабатываются исключения, при необходимости их передачи другому слою.

Итак, слой сервисов обратился к слою DAO, слой DAO в свою очередь к БД. Однако при выполнении запроса к БД выбросилось исключение `SQLException`, которое слой DAO не может исправить. Вопрос - что с ним делать?

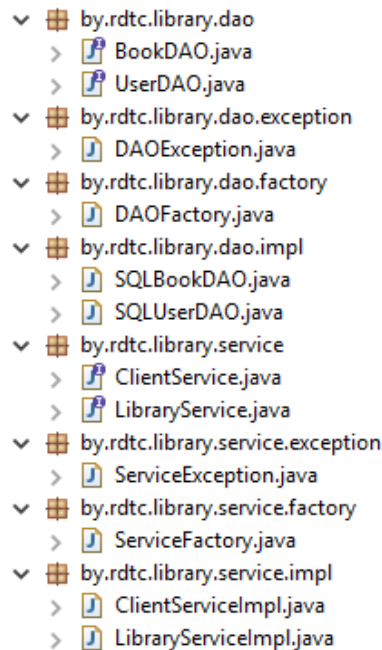
Можно отказаться от обработки исключения и отдать его сервисам. Однако это нарушит концепцию слоев приложения, сервисы узнают о чем-то, что характерно для слоя, лежащего ниже, чем DAO.



Поэтому, для каждого слоя разрабатывается как минимум одно собственное исключение, и при необходимости пробросить исключение на слой выше, генерируется исключение именно того слоя, который его и выбрасывает.



Изменим код нашего приложения так, чтобы предоставить возможность корректной обработки исключений.



DAO

```
package by.rdtc.library.dao;

import by.rdtc.library.bean.Book;
import by.rdtc.library.dao.exception.DAOException;

public interface BookDAO {
    void addBook(Book book) throws DAOException;
    void deleteBook(long idBook) throws DAOException;
    void delete(Book book) throws DAOException;
}

package by.rdtc.library.dao;

import by.rdtc.library.bean.User;
import by.rdtc.library.dao.exception.DAOException;

public interface UserDAO {
    void signIn(String login, String password) throws DAOException;
    void registration(User user) throws DAOException;
}

package by.rdtc.library.dao.exception;

public class DAOException extends Exception{
    private static final long serialVersionUID = 1L;

    public DAOException() {
```

```
        super();
    }

    public DAOException(String message){
        super(message);
    }

    public DAOException(Exception e){
        super(e);
    }

    public DAOException(String message, Exception e){
        super(message, e);
    }
}

package by.rdtc.library.dao.factory;

import by.rdtc.library.dao.BookDAO;
import by.rdtc.library.dao.UserDAO;
import by.rdtc.library.dao.impl.SQLBookDAO;
import by.rdtc.library.dao.impl.SQLUserDAO;

public final class DAOFactory {
    private static final DAOFactory instance = new DAOFactory();

    private final BookDAO sqlBookImpl = new SQLBookDAO();
    private final UserDAO sqlUserImpl = new SQLUserDAO();

    private DAOFactory() {}

    public static DAOFactory getInstance(){
        return instance;
    }

    public BookDAO getBookDAO(){
        return sqlBookImpl;
    }

    public UserDAO getUserDAO(){
        return sqlUserImpl;
    }
}

package by.rdtc.library.dao.impl;

import by.rdtc.library.bean.Book;
import by.rdtc.library.dao.BookDAO;
import by.rdtc.library.dao.exception.DAOException;

public class SQLBookDAO implements BookDAO{
    @Override
    public void addBook(Book book) throws DAOException{

    }

    @Override
    public void deleteBook(long idBook) throws DAOException{
    }
}
```

```
@Override
public void delete(Book book) throws DAOException{
}

}

package by.rdtc.library.dao.impl;

import by.rdtc.library.bean.User;
import by.rdtc.library.dao.UserDAO;
import by.rdtc.library.dao.exception.DAOException;

public class SQLUserDAO implements UserDAO{

    @Override
    public void signIn(String login, String password) throws DAOException{
        // именно в этом методы мы связываемся с базой данных и проверяем
        // корректность логина и пароля
    }

    @Override
    public void registration(User user) throws DAOException{

    }

}

package by.rdtc.library.service;

import by.rdtc.library.bean.User;
import by.rdtc.library.service.exception.ServiceException;

public interface ClientService {
    void signIn(String login, char[] password) throws ServiceException;
    void signIn(String login) throws ServiceException;
    void registration(User user) throws ServiceException;
}

package by.rdtc.library.service;

import by.rdtc.library.bean.Book;
import by.rdtc.library.service.exception.ServiceException;

public interface LibraryService {
    void addNewBook(Book book) throws ServiceException;
    void addEditedBook(Book book) throws ServiceException;
}

package by.rdtc.library.service.exception;

public class ServiceException extends Exception {
    private static final long serialVersionUID = 1L;

    public ServiceException(){
        super();
    }
}
```

```
public ServiceException(String message) {
    super(message);
}

public ServiceException(Exception e) {
    super(e);
}

public ServiceException(String message, Exception e) {
    super(message, e);
}
}

package by.rdtc.library.service.factory;

import by.rdtc.library.service.ClientService;
import by.rdtc.library.service.LibraryService;
import by.rdtc.library.service.impl.ClientServiceImpl;
import by.rdtc.library.service.impl.LibraryServiceImpl;

public final class ServiceFactory {
    private static final ServiceFactory instance = new ServiceFactory();

    private final ClientService clientService = new ClientServiceImpl();
    private final LibraryService libraryService = new LibraryServiceImpl();

    private ServiceFactory() {}

    public static ServiceFactory getInstance() {
        return instance;
    }

    public ClientService getClientService() {
        return clientService;
    }

    public LibraryService getLibraryService() {
        return libraryService;
    }
}

package by.rdtc.library.service.impl;

import by.rdtc.library.bean.User;
import by.rdtc.library.dao.UserDAO;
import by.rdtc.library.dao.exception.DAOException;
import by.rdtc.library.dao.factory.DAOFactory;
import by.rdtc.library.service.ClientService;
import by.rdtc.library.service.exception.ServiceException;

public class ClientServiceImpl implements ClientService {

    @Override
    public void singIn(String login, String password) throws ServiceException {
        // проверяем параметры
        if (login == null || login.isEmpty()) {
            throw new ServiceException("Incorrect login");
        }
    }
}
```

```
        // реализуем функционал логинации пользователя в системе
        try{
            DAOFactory daoObjectFactory = DAOFactory.getInstance();
            UserDAO userDAO = daoObjectFactory.getUserDAO();
            userDAO.signIn(login, password);
        } catch (DAOException e) {
            throw new ServiceException(e);
        }
        //....
    }

    @Override
    public void singOut(String login) throws ServiceException{
    }

    @Override
    public void registration(User user) throws ServiceException{
    }
}

package by.rdtc.library.service.impl;

import by.rdtc.library.bean.Book;
import by.rdtc.library.service.LibraryService;
import by.rdtc.library.service.exception.ServiceException;

public class LibraryServiceImpl implements LibraryService{

    @Override
    public void addNewBook(Book book) throws ServiceException{
    }

    @Override
    public void addEditedBook(Book book) throws ServiceException{
    }
}
```

Controller (Code)

Теперь нужно реализовать слой управления. Определим, что данные на слое Controller будут приходить в виде форматированной строки. Ответ клиенту Контроллер также будет осуществлять в виде форматированной строки. Например, вот так:

```
package by.rdtc.library.controller;

public class Controller {

    public String executeTask(String request){
        return null; //stub
    }
}
```

Предположим, что название команды будет приходить в начале строки до первого пробела.

Тогда обработка различных команд слоем управления может выглядеть следующим образом.


```
package by.rdtc.library.controller;

public class Controller {
    private final char paramDelimiter = ' ';

    public String executeTask(String request){

        String command;
        command = request.substring(0, request.indexOf(paramDelimiter));
        command = command.toUpperCase();

        String response = null;
        switch(command){
            case "SIGN_IN":
                // do action (call services and others) and create a response
                break;
            case "ADD_BOOK":
                // do action and create a response
                break;
            default:
                response = "We can't execute this command";
        }

        return response;
    }
}
```

Однако использовать оператор switch не является хорошей идеей в принципе. Он очень громоздок и тяжело модифицируем. Заменим эту конструкцию, применим паттерн Command.

Сначала создадим интерфейс Command

```
package by.rdtc.library.controller.command;

public interface Command {
    public String execute(String request);
}
```

Затем создадим реализации этого интерфейса - конкретные команды.

```
package by.rdtc.library.controller.command.impl;

import by.rdtc.library.controller.command.Command;
import by.rdtc.library.service.ClientService;
import by.rdtc.library.service.exception.ServiceException;
import by.rdtc.library.service.factory.ServiceFactory;

public class SingIn implements Command{
    @Override
    public String execute(String request) {
        String login = null;
        String password = null;

        String response = null;

        // get parameters from request and initialize the variables login and
        password

        ServiceFactory serviceFactory = ServiceFactory.getInstance();
        ClientService clientService = serviceFactory.getClientService();
    }
}
```

```
        try {
            clientService.signIn(login, password);
            response = "Welcome";
        } catch (ServiceException e) {
            // write log
            response = "Error duiring login procedure";
        }
        return response;
    }
}
```

```
package by.rdtc.library.controller.command.impl;

import by.rdtc.library.controller.command.Command;

public class Register implements Command{
    @Override
    public String execute(String request) {
        // stub
        return null;
    }
}
```

```
package by.rdtc.library.controller.command.impl;

import by.rdtc.library.controller.command.Command;

public class AddBook implements Command {
    @Override
    public String execute(String request) {
        //stub
        return null;
    }
}
```

Реализуем удобный механизм доступа к экземплярам команд (опять же, чтобы не раскрывать детали реализации и не плодить постоянно одни и те же объекты).

```
package by.rdtc.library.controller;

import java.util.HashMap;
import java.util.Map;

import by.rdtc.library.controller.command.Command;
import by.rdtc.library.controller.command.CommandName;
import by.rdtc.library.controller.command.impl.AddBook;
import by.rdtc.library.controller.command.impl.Register;
import by.rdtc.library.controller.command.impl.SignIn;
import by.rdtc.library.controller.command.impl.WrongRequest;

final class CommandProvider {
    private final Map<CommandName, Command> repository = new HashMap<>();

    CommandProvider() {
        repository.put(CommandName.SIGN_IN, new SignIn());
        repository.put(CommandName.REGISTRATION, new Register());
        repository.put(CommandName.ADD_BOOK, new AddBook());
        repository.put(CommandName.WRONG_REQUEST, new WrongRequest());
        //...
    }
}
```

```
}

Command getCommand(String name){
    CommandName commandName =null;
    Command command = null;

    try{
        commandName = CommandName.valueOf(name.toUpperCase());
        command = repository.get(commandName);
    }catch(IllegalArgumentException | NullPointerException e){
        //write log
        command = repository.get(CommandName.WRONG_REQUEST);
    }
    return command;
}
}
```

Класс Controller теперь сможет диспетчеризировать множество команд.

```
package by.rdtc.library.controller;

import by.rdtc.library.controller.command.Command;

public final class Controller {
    private final CommandProvider provider = new CommandProvider();

    private final char paramDelimiter = ' ';

    public String executeTask(String request){
        String commandName;
        Command executionCommand;

        commandName = request.substring(0, request.indexOf(paramDelimiter));
        executionCommand = provider.getCommand(commandName);

        String response;
        response = executionCommand.execute(request);

        return response;
    }
}
```

В такой реализации класс Controller выступает в роли front controller-а (в него поступают все запросы к системе), а объекты - команды - уже как конкретные команды. Задачей контроллера становится найти команду, которая отвечает на управление выполнением конкретного запроса и передать запрос ей. Задачей же команды является извлечь параметры, привести их в вид, который запрашивают методы сервисов, обратиться к слою сервисов для выполнения запроса, получить ответ от сервиса и сформировать свой ответ - тот, который подходит для передачи клиенту (в вид).

View (without code)

Реализацию слоя View (Вид) рассматривать нет необходимости, т.к. она может быть предназначена и для консольных и для приложения с графическим интерфейсом. Основное правило - все общение Вида с другими частями приложения идет через экземпляры класса Controller (и его так же не надо создавать во множественном количестве). Вид формирует запрос - строку определенно вида, передает ее контроллеру и ожидает ответ. Получив ответ - обрабатывает его в соответствии со своими задачами.