**1.**

Привет! Эта задача будет на паттерн **MVC** - **Model-View-Controller**.  
Мы вместе построим архитектуру используя **MVC**. Разберись подробно, что и почему нужно реализовывать так, как я тебе покажу.  
Прочти дополнительную литературу, которую дает профессор в конце уровня.  
Тебя, скорее всего, на собеседовании спросят об этом паттерне либо дадут задание, в котором нужно будет его реализовать.

Итак...  
У тебя есть два пакета: bean, содержащий единственный класс **User**, и **dao**, в котором хранится эмуляция базы данных в пакете **mock** и **UserDao**. **UserDao** - это уровень **ДАО**, т.е. уровень доступа к базе.  
В нем размещают различные методы по **сохранению** и **получению** объектов из базы данных.  
В реальном приложении строку private DataSource dataSource = DataSource.getInstance() не встретить.  
Я реализовал **DataSource** в виде синглтона. В действительности, у тебя будет что-то такое:  
@Autowired  
private DataSource dataSource;

Фреймворк, которым ты будешь пользоваться, сам создаст объект базы данных и инициализирует поле dataSource.

Запомни, с **ДАО** уровнем работают сервисы. Никакие другие классы в **ДАО** не лезут. В сервисах описана бизнес логика.  
Сервисы забирают данные из базы используя **ДАО**, обрабатывают их и отдают тому, кто данные запросил.  
Однако не все данные хранятся в базе. Например, залогиненый пользователь будет храниться в специальном объекте - Модели.  
Объект, который содержит в себе данные, необходимые для отображения информации на клиенте, называется Моделью.  
Также этот объект Модель содержит ссылки на все необходимые сервисы.  
Если данных для отображения очень много, то их выделяют в отдельный объект.

Напишем приложение, которое будет показывать список пользователей и что-то делать с ними, например, обновлять их данные и удалять.

1. Создай пакет **model**, в котором создай класс **ModelData**.  
**ModelData** - это объект, который будет хранить необходимые данные для отображения на клиенте.  
Создай поле с геттером и сеттером **List**<User> users - это будет список пользователей для отображения.

2. Используя любую модель должна быть возможность получить все необходимые данные для отображения. Поэтому в пакете **model** создай интерфейс **Model**, который должен содержать метод ModelData getModelData().

3. В пакете **model** создай класс **FakeModel**, реализующий **Model**. Он нам понадобится на начальном этапе.  
В нем создай поле **ModelData** modelData, которое инициализируй объектом.

4. В интерфейсе **Model** создай метод void loadUsers().  
Реализуй его в FakeModel: инициализируй список пользователей **modelData** любыми данными. Они не влияют на тестирование.

**У меня такие данные:**  
User{name='A', id=1, level=1}  
User{name='B', id=2, level=1}

Думаю, ты помнишь, что все методы интерфейса являются **public**-ами, поэтому модификатор указывать не нужно.  
Программисты часто мОкают данные на начальном этапе. Получение реальных данных реализуется на последних этапах.  
Мокать - это подменять реальные объекты на хардкоженные, тестовые данные.

**Требования:**

* •

Класс ModelData должен быть создан в пакете model.

* •

В классе ModelData должно быть создано приватное поле List<User> users, геттер и сеттер для него.

* •

Интерфейс Model должен быть создан в пакете model. В интерфейсе Model должен быть объявлен метод ModelData getModelData().

* •

Класс FakeModel, реализующий интерфейс Model, должен быть создан в пакете model.

* •

В классе FakeModel должно быть создано и инициализировано приватное поле ModelData modelData.

* •

В интерфейсе Model должен быть объявлен метод void loadUsers().

* •

Метод void loadUsers() в классе FakeModel должен инициализировать список пользователей в объекте modelData любыми данными.

**2.**

1. Создай пакет **controller**, в котором создай класс **Controller**.  
Этот класс будет получать запрос от клиента, оповещать Модель об этом, а Модель, в свою очередь, будет обновлять **ModelData**.

2. Добавь в контроллер поле Model model вместе с сеттером.

3. В контроллере создай публичный метод void onShowAllUsers(), который должен обратиться к модели и инициировать загрузку пользователей.

4. Создай пакет **view**. В нем создай интерфейс **View**.

5. В интерфейс **View** добавь два метода: void refresh(ModelData modelData) и void setController(Controller controller)

**Требования:**

* •

Класс Controller должен быть создан в пакете controller.

* •

В классе Controller должно быть создано приватное поле Model model и сеттер для этого поля.

* •

В классе Controller должен быть создан публичный метод void onShowAllUsers(), в котором у модели должен вызываться метод loadUsers().

* •

Интерфейс View должен быть создан в пакете view.

* •

В интерфейсе View должны быть объявлены два метода: void refresh(ModelData modelData) и void setController(Controller controller).

**3.**

Чтобы понимать, в правильном ли направлении ты движешься, тебе надо видеть данные. Поэтому:

1. В пакете **view** создай класс **UsersView**, реализующий **View**. Он будет отображать список пользователей в консоль.

2. В **UsersView** создай поле-контроллер, также создай ему сеттер.

3. Реализуй логику метода refresh:  
3.1. Выведи в консоль фразу "***All users:***".  
3.2. Выведи в консоль всех пользователей, которые есть в **modelData**.  
Перед каждым пользователем сделай отступ в виде табуляции.  
3.3. В конце выведи визуальный разделитель данных  
===================================================

4. Уже интересно посмотреть, что же получилось.  
Добавь в **UsersView** публичный метод void fireEventShowAllUsers(), который будет эмулировать событие клиента.  
Обратись к контроллеру и вызови его нужный метод для отображения всех пользователей.

5. Класс **Solution** будет эмулятором пользователя. Открой класс **Solution**, стань на красный метод.  
C помощью горячих клавиш Идеи создай проперти(**поле**) для **usersView**.  
Нужен только сеттер. Если у тебя создался геттер, то удали его.

6. Запусти main. Упс, ничего не вывело : (  
Это получилось потому, что данные пришли с сервера, обновились в **ModelData**, но Вью ничего о них не знает.  
Вью сама не умеет себя обновлять. Это делает Контроллер.  
Пойди в контроллер и добавь обновление данных во Вью.  
Напомню, данные хранятся в Модели.

7. Запусти main. У меня теперь такой вывод:  
All users:  
    User{name='A', id=1, level=1}  
    User{name='B', id=2, level=1}  
===================================================  
Ура, идем дальше.

**Требования:**

* •

Класс UsersView, реализующий интерфейс View, должен быть создан в пакете view.

* •

В классе UsersView должно быть создано приватное поле Controller controller и сеттер для этого поля.

* •

Метод refresh класса UsersView должен быть реализован согласно условию.

* •

В классе UsersView должен быть создан публичный метод void fireEventShowAllUsers(), в котором у контроллера должен вызываться метод onShowAllUsers().

* •

В классе Controller должно быть создано приватное поле UsersView usersView и сеттер для этого поля. Геттера для этого поля не должно быть создано.

* •

В методе onShowAllUsers() класса Controller должен вызываться метод refresh у объекта usersView с параметром model.getModelData() после вызова метода loadUsers().

* •

Вызов метода main должен выводить на экран информацию о всех пользователях, которых ты добавляешь в методе loadUsers() класса FakeModel.

**4.**

Пора заменять нашу фейковую Модель на реальную, которая будет получать данные из **DataSource**.  
В пакет **model.service** я добавил сервис для работы с пользователями.  
Также в корне этой задачи ты найдешь утильный класс **Util**.

1. Аналогично **FakeModel** создай модель **MainModel**.

2. Т.к. Модель обращается к сервисам, то в **MainModel** создай поле UserService userService, инициализируй объектом.

3. Реализуй логику метода loadUsers:  
3.1. Достань всех пользователей между **1** и **100** уровнями. (Метод getUsersBetweenLevels(1, 100)).  
3.2. Положи всех пользователей в **modelData**.

4. Обнови **Solution.main**: замени **FakeModel** на **MainModel**.  
Преимущества **MVC** в том, что в любой момент времени легко можно заменить любую часть паттерна.

**Требования:**

* •

Класс MainModel, реализующий интерфейс Model, должен быть создан в пакете model.

* •

В классе MainModel должно быть создано и инициализировано приватное поле ModelData modelData.

* •

В классе MainModel должно быть создано и инициализировано приватное поле UserService userService.

* •

В классе MainModel необходимо реализовать метод getModelData.

* •

Метод loadUsers класса MainModel должен быть реализован согласно условию.

* •

В методе main класса Solution должен создаваться объект класса MainModel вместо объекта класса FakeModel.

**5.**

В сервисе есть метод, который возвращает всех удаленных пользователей. Давай их отобразим.

1. Распредели методы по классам **MVC**:

public void fireEventShowDeletedUsers() {  
...onShowAllDeletedUsers();  
}

public void onShowAllDeletedUsers() {  
...loadDeletedUsers();  
}

public void loadDeletedUsers() {  
List<User> users = userService.getAllDeletedUsers();  
}

Не забудь, что данные, полученные с сервера, необходимо положить в **ModelData**. А потом обновить view.  
Добавь это самостоятельно в нужные методы.

2. Добавь в **Solution.main** вызов нового метода, который ты поместил в view.

3. Добавь в интерфейс **Model** метод, который ты поместил в **Модель**, реализуй его в **FakeModel**: выброси UnsupportedOperationException.

**Требования:**

* •

Необходимо определить правильное расположение метода fireEventShowDeletedUsers() и реализовать этот метод.

* •

Необходимо определить правильное расположение метода onShowAllDeletedUsers() и реализовать этот метод.

* •

Необходимо определить правильное расположение метода loadDeletedUsers() и реализовать этот метод.

* •

В методе main класса Solution необходимо вызвать метод, который ты ранее реализовал в классе UsersView.

* •

Интерфейс Model должен содержать объявление метода, который ты ранее реализовал в классе MainModel.

* •

В классе FakeModel в теле метода, помещенного в интерфейс Model, необходимо бросить UnsupportedOperationException.

**6.**

Функционал отображения удаленных пользователей есть, а самих таких пользователей нет. Давай это исправим.  
Давай сделаем новую Вью, которая будет отвечать за редактирование одного конкретного пользователя.  
**UsersView** отображает список пользователей.  
**EditUserView** будет отображать данные о редактировании конкретного пользователя.  
Для этого нам сначала нужен этот выбранный пользователь.  
Как и любые данные его поместим в **ModelData**.

1. Создай в **ModelData** поле User activeUser с геттером и сеттером (Alt+Insert -> Getter and Setter).

2. Аналогично **UsersView** создай **EditUserView**.  
Логика метода refresh:  
2.1. Вывести в консоль "***User to be edited:***".  
2.2. С новой строки вывести табуляцию и активного пользователя.  
2.3. С новой строки вывести разделитель "===================================================".

3. Создай в контроллере поле EditUserView editUserView с сеттером.

Когда наши данные выводятся в консоль, то совсем не понятно, список каких пользователей - удаленных или нет - выводится.  
Давай сделаем так, чтобы Вью отображала эту информацию. Все данные для отображения хранятся в Моделе.

**Поэтому:**  
4. создай в **ModelData** поле boolean displayDeletedUserList с **геттером** и **сеттером**.

5. Измени метод refresh в **UsersView** так, чтобы он отображал "***All users:***" либо "***All deleted users:***"  
в зависимости от того, какие пользователи находятся в списке.  
Добавь в необходимые методы модели изменение **displayDeletedUserList**.

**Требования:**

* •

В классе ModelData должно быть создано приватное поле User activeUser, геттер и сеттер для этого поля.

* •

Класс EditUserView должен быть создан аналогично классу UsersView: он должен поддерживать интерфейс View, содержать приватное поле Controller controller и сеттер этого поля.

* •

Метод refresh класса EditUserView должен быть реализован согласно условию.

* •

В классе Controller должно быть создано приватное поле EditUserView editUserView и сеттер этого поля.

* •

В классе ModelData должно быть создано приватное поле boolean displayDeletedUserList, геттер и сеттер для этого поля.

* •

Метод refresh в классе UsersView должен быть изменен согласно условию.

* •

Необходимо добавить в некоторые методы класса MainModel вызов метода setDisplayDeletedUserList(boolean) с правильным флагом.

**7.**

1. Распредели методы по классам **MVC**:

public void onOpenUserEditForm(long userId) {  
...loadUserById(userId);  
...refresh(...getModelData());  
}

public void fireEventOpenUserEditForm(long id) {  
...onOpenUserEditForm(id);  
}

public void loadUserById(long userId) {  
User user = userService.getUsersById(userId);  
...setActiveUser(user);  
}

!!!! Пользователь видит Вью со списком пользователей, нажимает на одного из них, запрос идет на сервер, выгребаем новые данные и отображаем другую Вью, которая относится к одному выбранному пользователю.  
Учти это при реализации этого задания.

2. Добавь в метод main открытие формы редактирования для пользователя с id**=126** перед вызовом метода fireEventShowDeletedUsers().

3. Добавь в интерфейс Model метод, который ты поместил в Модель, реализуй его в **FakeModel**: выброси **UnsupportedOperationException**.

**Требования:**

* •

Необходимо определить правильное расположение метода onOpenUserEditForm(long) и реализовать этот метод.

* •

Необходимо определить правильное расположение метода fireEventOpenUserEditForm(long) и реализовать этот метод.

* •

Необходимо определить правильное расположение метода loadUserById(long) и реализовать этот метод.

* •

В методе main класса Solution необходимо вызвать метод открытия формы редактирования для пользователя с id=126 у объекта класса UsersView перед вызовом метода fireEventShowDeletedUsers().

* •

Интерфейс Model должен содержать объявление метода, который ты ранее реализовал в классе MainModel.

* •

В классе FakeModel в теле метода, помещенного в интерфейс Model, необходимо бросить UnsupportedOperationException.

**8.**

1. Следуя принципу **MVC** аналогично реализации предыдущих методов сделай следующее:  
напиши логику удаления пользователя. После удаления должен отображаться список пользователей.

Распредели следующие методы по классам **MVC**:  
public void fireEventUserDeleted(long id)  
public void onUserDelete(long id)  
public void deleteUserById(long id)

**Примечание:** *метод, который ты собираешься добавить в Вью нужно добавить в***EditUserView***.*

2. Добавь в main вызов fireEventUserDeleted(**124L**) перед вызовом метода fireEventShowDeletedUsers().

3. Выполни рефакторинг класса **MainModel**. Теперь, когда есть удаленные пользователи, часть методов стала работать неправильно.  
Почти во всех методах, где требуется список пользователей, нужно работать только с активными(живыми) пользователями.  
Вынеси в отдельный приватный метод**List**<User> getAllUsers() получение списка всех активных пользователей.  
Фильтрация активных пользователей у тебя уже есть - метод **List**<User> filterOnlyActiveUsers(List<User> **allUsers**).  
Отрефактори все методы, которые используют список пользователей. Они должны использовать список живых пользователей.

4. Добавь в интерфейс **Model** метод, который ты поместил в Модель, реализуй его в **FakeModel**: выброси **UnsupportedOperationException**.

**Требования:**

* •

Необходимо определить правильное расположение метода fireEventUserDeleted(long) и реализовать этот метод.

* •

Необходимо определить правильное расположение метода onUserDelete(long) и реализовать этот метод.

* •

Необходимо определить правильное расположение метода deleteUserById(long) и реализовать этот метод.

* •

В методе main класса Solution необходимо вызвать метод fireEventUserDeleted(124L) у объекта класса EditUserView перед вызовом метода fireEventShowDeletedUsers().

* •

В методе main вызов методов должен происходить в такой последовательности: fireEventShowAllUsers(), fireEventOpenUserEditForm(126L), fireEventUserDeleted(124L), fireEventShowDeletedUsers().

* •

Необходимо реализовать приватный метод List<User> getAllUsers() в классе MainModel.

* •

Необходимо выполнить рефакторинг: все методы класса MainModel, которые используют список пользователей, должны использовать список живых пользователей.

* •

Интерфейс Model должен содержать объявление метода, который ты ранее реализовал в классе MainModel.

* •

В классе FakeModel в теле метода, помещенного в интерфейс Model, необходимо бросить UnsupportedOperationException.

**9.**

Это последнее задание по **MVC**.

1. Следуя принципу **MVC** аналогично реализации предыдущих методов сделай следующее:  
напиши логику обновления пользователя. После обновления должен отображаться список пользователей.

Распредели методы по классам **MVC**, используя следующие сигнатуры публичных методов:  
void fireEventUserChanged(String name, long id, int level)  
void onUserChange(String name, long id, int level)  
void changeUserData(String name, long id, int level)

где **name** и **level** - это новые значения для пользователя с выбранным **id**.

**Примечание:** *метод, который ты собираешься добавить в Вью нужно добавить в* **EditUserView**.

2. Добавь в main вызов fireEventUserChanged перед вызовом метода fireEventShowDeletedUsers().

3. Добавь в интерфейс **Model** метод, который ты поместил в Модель, реализуй его в **FakeModel**: выброси **UnsupportedOperationException**.

**Требования:**

* •

Необходимо определить правильное расположение метода fireEventUserChanged(String name, long id, int level) и реализовать этот метод.

* •

Необходимо определить правильное расположение метода onUserChange(String name, long id, int level) и реализовать этот метод.

* •

Необходимо определить правильное расположение метода void changeUserData(String name, long id, int level) и реализовать этот метод.

* •

В методе main необходимо вызвать метод fireEventUserChanged(String, long, int) перед вызовом метода fireEventShowDeletedUsers().

* •

Интерфейс Model должен содержать объявление метода, который ты ранее реализовал в классе MainModel.

* •

В классе FakeModel в теле метода, помещенного в интерфейс Model, необходимо бросить UnsupportedOperationException.