



МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Практическая работа № 2/4 ч.

Разработка мобильных приложений

| | |
|---|---|
| Уровень | (наименование дисциплины (модуля) в соответствии с учебным планом) бакалавриат |
| Форма обучения | (бакалавриат, магистратура, специалитет) очная |
| Направление(-я) подготовки | (очная, очно-заочная, заочная) 09.03.02 «Информационные системы и технологии» |
| Институт | (код(-ы) и наименование(-я)) кибербезопасности и цифровых технологий |
| Кафедра | (полное и краткое наименование) КБ-14 «Цифровые технологии обработки данных» |
| Используются в данной редакции с учебного года | (полное и краткое наименование кафедры, реализующей дисциплину (модуль)) 2024/25 |
| Проверено и согласовано « ____ » _____ 20 ____ г. | (учебный год цифрами) (подпись директора Института/Филиала с расшифровкой) |

Москва 2024 г.

ОГЛАВЛЕНИЕ

| | |
|--------------------------------------|----|
| ВВЕДЕНИЕ..... | 3 |
| 1 ПОСТРОЕНИЕ МОДУЛЬНОГО ПРОЕКТА..... | 5 |
| 1.1 Раздельные модели данных..... | 8 |
| 1.2 Раздельные модули | 12 |
| 2 КОНТРОЛЬНОЕ ЗАДАНИЕ..... | 15 |

ИЗДАНИЕ

ВВЕДЕНИЕ

Модели данных на разных слоях программного обеспечения выполняют различные задачи и используются для разных целей. В зависимости от архитектуры приложения, данные могут проходить через несколько слоев, каждый из которых работает со своей моделью данных, адаптированной для конкретного уровня системы. Далее представлено описание моделей на каждом из уровней:

1. Модель данных на уровне пользовательского интерфейса (UI Layer) – это представление данных, с которыми взаимодействует пользователь. Модель обычно отражает информацию в удобной для отображения в интерфейсе в виде форм, таблиц, списков и т.д. В приложении модель данных может быть представлена в виде DTO (Data Transfer Object), который содержит только те поля, которые нужны для отображения на экране.

Основные задачи:

- подготовка данных для отображения пользователю;
- валидация пользовательского ввода.

Пример модели: если имеется сущность «Пользователь», в UI-модели могут содержаться только *имя*, *аватар* и *адрес электронной почты*, так как это основная информация, которая отображается на экране.

2. Модель данных на уровне бизнес-логики (Business Logic Layer) – на этом уровне происходят основные операции с данными: валидация, обработка, преобразование и проверка логики приложения. **Пример:** для бизнес-логики интернет-магазина может существовать модель «Заказ», которая содержит список товаров, цену, статус оплаты и другие атрибуты, нужные для обработки заказа.

Основные задачи:

- инкапсуляция бизнес-правил и логики;
- обработка данных перед их отправкой на уровень хранения или обратно на уровень пользовательского интерфейса;
- обеспечение консистентности данных в приложении.

Пример модели: модель «Заказ» может включать методы для проверки статуса оплаты, расчета итоговой стоимости, обработки скидок и т.д.

3. Модель данных на уровне доступа к данным (Data Access Layer). На уровне доступа к данным модели часто используются для работы с базой данных или сетевым взаимодействием. Модели могут представлять собой прямое отображение сущностей базы данных (например, таблиц) или более сложные структуры для работы с различными источниками данных.

Пример: ORM-модели представляют собой таблицы базы данных в виде классов, где каждая запись базы данных – это объект, а столбцы таблицы – это атрибуты объекта.

Основные задачи:

- отображение данных между приложением и базой данных (ORM).
- выполнение операций CRUD (создание, чтение, обновление, удаление).
- управление связями между сущностями (например, «один к одному», «один ко многим»).

Пример модели: В ORM-системе модель «Пользователь» может иметь поля, соответствующие колонкам базы данных (ID, имя, email, дата создания).

4. Модели для взаимодействия между слоями (DTO и ViewModel). На разных уровнях системы могут использоваться специальные промежуточные модели для разделения ответственности и упрощения взаимодействия между слоями.

DTO (Data Transfer Object): используется для передачи данных между слоями или через API. Это облегченные модели, содержащие только нужные данные без бизнес-логики. Их часто применяют для уменьшения количества передаваемых данных.

Пример: DTO для пользователя может содержать только поля ID, Имя, и Email для отображения в списке пользователей.

ViewModel – это модель, которая часто используется в архитектуре MVVM (Model-View-ViewModel). ViewModel агрегирует данные для пользовательского интерфейса, предоставляя только те данные и функции, которые нужны для конкретного экрана или представления.

Пример: ViewModel для экрана профиля пользователя может содержать не только поля модели пользователя, но и дополнительные вычисляемые свойства (например, возраст на основе даты рождения).

В качестве примера использования моделей данных на разных уровнях, возможно рассмотреть пример сущности «Продукт» в интернет-магазине:

1. **UI (Presentation Layer):** модель «Продукт» содержит только поля для отображения – название, цену и изображение.
2. **Business Logic Layer (domain):** модель «Продукт» может содержать логику расчета скидки или обработки различных состояний (например, товар в наличии, товар на складе и т.д.).
3. **Data Access Layer:** модель «Продукт» представляет собой ORM-объект, который напрямую соответствует записи в базе данных и содержит все поля таблицы, такие как ID, название, цена, описание, количество на складе.

1 ПОСТРОЕНИЕ МОДУЛЬНОГО ПРОЕКТА

Задание. Требуется открыть приложение из практического занятия №1. Ваша реализация и смысл приложения отличаются от представленного в практическом задании примера. Необходимо изменить приложение по аналогии с примером.

Возвращаемся к практической работе №1 «Чистая архитектура». На рисунке 1 представлена структура проекта. Слой презентейшн содержит Activity. Activity общается с двумя use-case. Use-case в свою очередь имеют доступ к Movie-репозиторию (рисунок 2). В слое domain содержится одна моделька. В слое *data* представлена реализация *MovieRepositoryImplementation*.

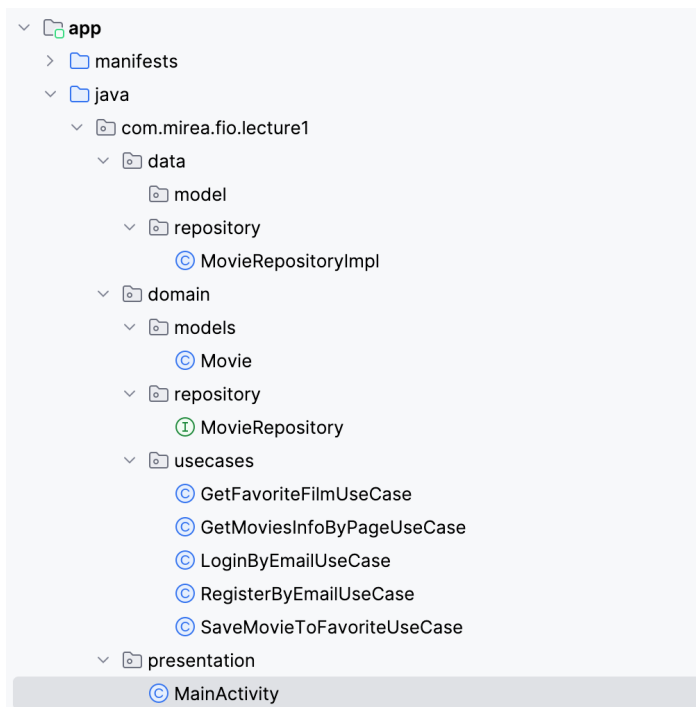


Рисунок 1 - Структура проекта

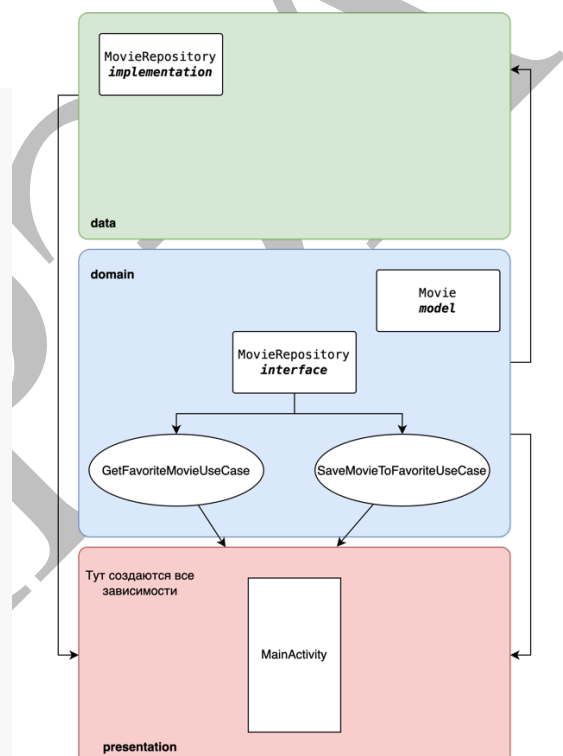


Рисунок 2 - UML-диаграмма проекта

Главный экран приложения содержит поле Edit Text, в которое можно внести какие-то данные и нажать сохранить. Эти данные через use-case передаются в репозиторий и в обратную сторону.

Рассмотрим *use cases*, которые у нас имеются – *GetFavoriteMovieUseCase* и *SaveMovieToFavoriteUseCase*, то есть мы сохраняем любимый фильм и получаем его название. **Вспомним, что имеется в репозитории – у нас имеется интерфейс репозитория, который представляет собой два метода, это *saveMovie* и *getMovie*.** В слое *data* находится реализация и вот здесь в реализации у нас соответственно находится *saveMovie* и *getMovie*, использующие механизм *SharedPreferences*.

В слое domain имеется пакет *Models* и там размещен *Movie*. В классе нет никакой логики, и он содержит только набор полей, в которые передается информация. Почему это делается именно так? Если бы мы не использовали такие модели то, например, в методе *execute* классов *GetFavoriteMovieUseCase* или *SaveMovieToFavoriteUseCase* пришлось бы заменять объект класса, набором примитивов.

Далее рассмотрен класс *MovieRepositoryImpl*. Данный класс у нас уже реализован и не вызывает вопросов в реализации, но на самом деле здесь имеется серьезная ошибка при условии возможного расширения функциональных возможностей класса.

```
public class MovieRepositoryImpl implements MovieRepository {

    private static final String SHARED_PREFS_NAME = "shared_prefs_name";
    private static final String KEY = "movie_name";
    private SharedPreferences sharedPreferences;
    private Context context;
    public MovieRepositoryImpl(Context context) {
        this.context = context;
        sharedPreferences = context.getSharedPreferences(SHARED_PREFS_NAME,
Context.MODE_PRIVATE);
    }

    @SuppressWarnings("CommitPrefEdits")
    @Override
    public boolean saveMovie(Movie movie){
        sharedPreferences.edit().putString(KEY, movie.getName()).commit();
        return true;
    }

    @Override
    public Movie getMovie(){
        String movie_name = sharedPreferences.getString(KEY, "unknown");
        return new Movie(1, movie_name);
    }
}
```

Проблема заключается в том, что прямо в репозитории происходит сохранение данных. Это не очень хорошо, то есть репозиторий по сути своей это такое связующее звено между доменом и датой. Например, кроме того, чтобы сохранять в *SharedPreferences* данные, появится требование отправлять запрос в интернет, что в итоге получится? То есть придется вносить дополнительную логику отправки в интернет в методы *saveMovie* и *getMovie*. И в итоге сформируется класс, в котором одновременно находятся два механизма работы с данными. Первый – это *SharedPreferences*, сохранение в локальное хранилище, а второй механизм – это отправка в интернет.

Получается, что нарушается *solid*-принципы, то есть у нас две зоны ответственности. Поэтому логику именно хранения данных хорошо бы вынести еще в дополнительный какой-то класс отдельный. Как мы это сделаем? Давайте я вам покажу.

Создадим в директории *data* дополнительный пакет, который назовем *Storage*. Теперь внутри директории *Storage* создадим интерфейс и назовем его, например, например *MovieStorage* и здесь создадим два метода, один для получения «*get*» (возвращающий модель «*Movie*»), а другой для сохранения данных «*save*» (возвращается булево значение). Назовем метод *save*, потому что у нас уже есть название *movie* в названии класса, то есть мы понимаем, что сохранить хотим фильм и на вход подадим соответствующую модель «*Movie*».

```
public interface MovieStorage {  
    public Movie get();  
    public boolean save(Movie movie);  
}
```

Далее создадим, например, класс *SharedPrefMovieStorage*, то есть конкретную реализацию и имплементируем интерфейс *MovieStorage*. Затем переносим логику из *MovieRepositoryImplementation* по сохранению данных в *shared preferences*. Также необходимо получить контекст в классе. В *MainActivity* необходимо создать экземпляр класса *MovieStorage* и передать его в репозиторий.

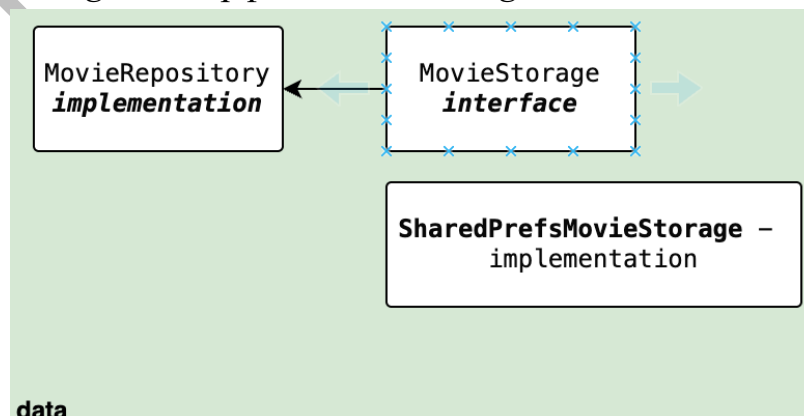
```
MovieStorage sharedPrefMovieStorage = new SharedPrefMovieStorage(this);  
MovieRepository movieRepository = new MovieRepositoryImpl(sharedPrefMovieStorage);
```

В итоге теперь вся логика размещена в отдельном классе, то есть данный класс занимается исключительно сохранением данных. Репозиторий является роутером, который может и с сетевой частью поработать, и с локальным хранилищем.

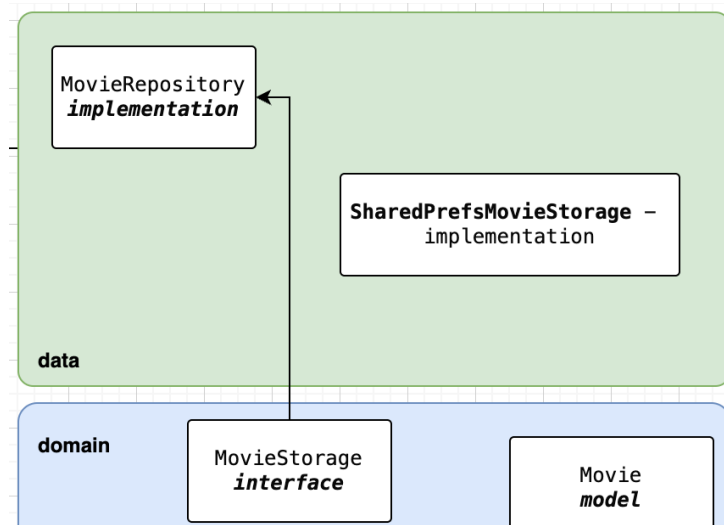
```
public interface MovieStorage {  
    public Movie get();  
    public boolean save(Movie movie);  
}
```

В будущем кроме сохранения в локальное хранилище возможно появление функциональности отправки данных в интернет! Скорее всего, появится в конструкторе еще один класс, который будет называться, например *NetworkAPI*, например. И возможно будет просто совершать вызов этих методов класса. Для наглядности перенесем на диаграмму состояние приложения и посмотрим, как это все выглядит.

Необходимо добавить блок *MovieStorage* — это интерфейс и его реализация *SharedPrefMovieStorage*. Интерфейс *MovieStorage* входит в *MovieRepositoryImpl*.



На первый взгляд выглядит неплохо, но давайте подумаем о то, возможно ли размещение интерфейса в слое *domain* по примеру *MovieRepository*.



Данная реализация неверна, потому что реализация *MovieRepository* размещена только в слое *data* и он не имеет никакого отношения к слою *domain* кроме наследования от интерфейса из этого слоя. Т.е. внутренняя реализация класса *MovieRepositoryImpl* – это ответственность слоя *data* и размещать *MovieStorage* в слое *domain* не имеет смысла. Такая логика возможна, если бы *MovieStorage* использовалась где-нибудь в слое *data* (напр. use-case). На данном этапе мы определили верное размещение интерфейса и его реализацию.

1.1 Раздельные модели данных

Далее рассмотрим вопрос раздельных моделей.

```
package com.mirea.fio.lecture2.data.storage;

import android.content.Context;
import android.content.SharedPreferences;

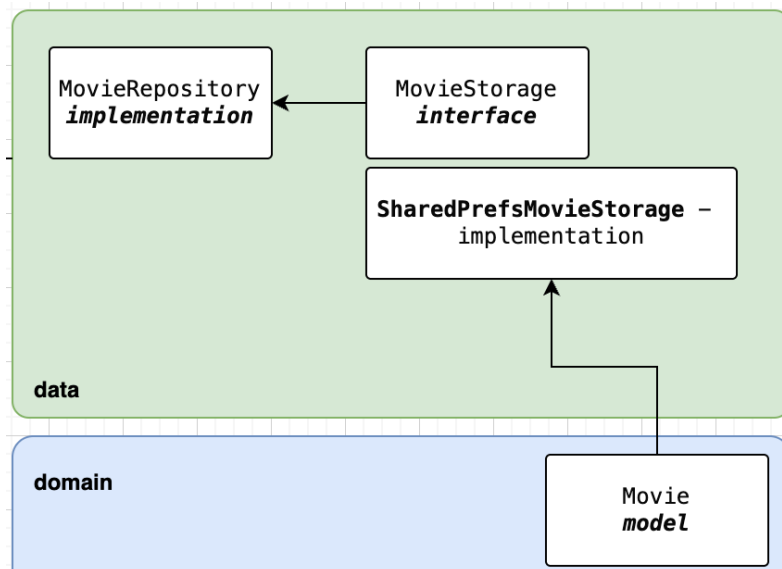
import com.mirea.fio.lecture2.domain.models.Movie;

public class SharedPrefMovieStorage implements MovieStorage {
    ...
    public SharedPrefMovieStorage(Context context) {
        ...
    }

    @Override
    public Movie get() {
        ...
    }

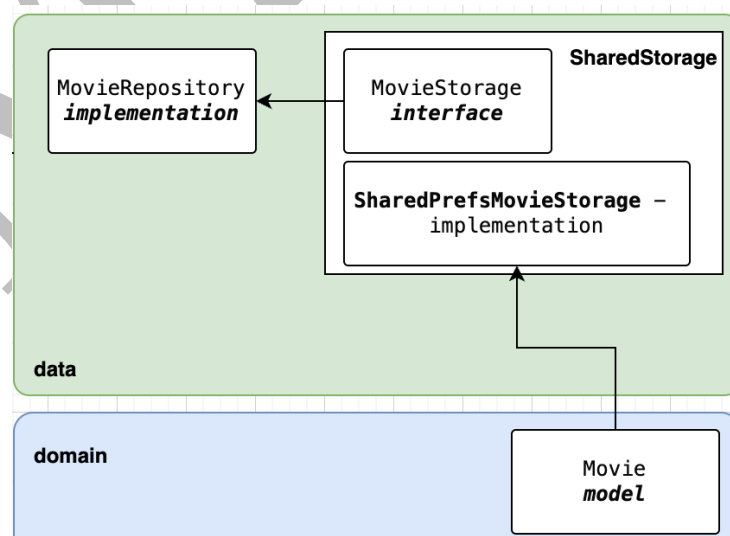
    @Override
    public boolean save(Movie movie) {
        ...
    }
}
```


Класс *SharedPrefsMovieStorage* используют Entity из слоя *domain* – путь до модели указан в импорте класса. Для небольших приложений такое переиспользование допустимо, но в больших проектах – это может быть проблемой. Рассмотрим на диаграмме данную реализацию.



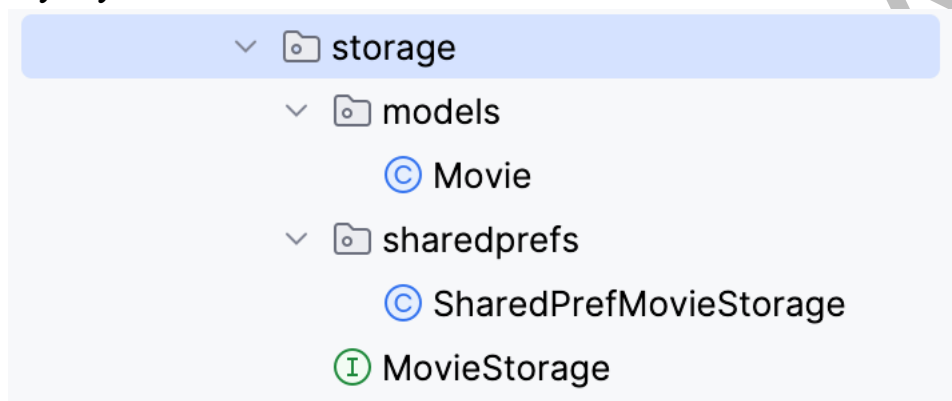
Стоит отметить, что логика сохранения в локальное хранилище абсолютно не независимая, то есть реализация зависит исключительно от каких-то внешних фреймворков или библиотек. Когда вы строите какую-то общую логику нужно стараться делать ее максимально независимой от внешнего мира. Т.е. реализация слоя *domain* независимо от внешнего мира и внутренняя реализация каких-либо процессов также должна не зависеть от внешних слоев.

Выделим локальной хранилище отдельным блоком. Оно является независимым, что говорит о том, что никаких стрелочек к нему не должно быть. Задача стоит в создании обособленной логики блока. Данный блок подключен к *MovieRepositoryImpl*, но он независим.



В представленной схеме имеется зависимость от модели из другого слоя. Для исключения зависимости блока от внешней зависимости достаточно просто создать модель внутри блока и теперь появляется возможность переноса модуля в любой другой проект. Таким образом построена отдельная независимая часть (пускай и внутри слоя *data*), которую возможно использовать с помощью какого-то интерфейса взаимодействия в различных проектах.

В первую очередь необходимо создать отдельные модели в коде. Исключим из слоя *data* модели другого слоя. Создадим в пакете *Storage* модель *Movie* с тремя полями *id*, *name* и *LocalDate* (будем сохранять время записи информации в память) и теперь в интерфейсе *MovieStorage* и *SharedPrefMovieStorage* заменяем используемую модель данных.



```
package com.mirea.fio.lecture2.data.storage.models;

public class Movie {
    private int id;
    private String name;
    private String localDate;

    public Movie(int id, String name, String localDate) {
        this.id = id;
        this.name = name;
        this.localDate = localDate;
    }

    public String getName() {
        return name;
    }

    public int getId() {
        return id;
    }

    public String getLocalDate() {
        return localDate;
    }
}
```

Стоит обратить внимание, что возможно использовать разные модели данных в методах сохранения и получения данных. В данном случае у нас получается независимый блок, который ничего не знает о внешних компонентах приложения. Таким образом формируется независимый слой *data* с множеством таких

независимых блоков. Данные свойства позволят легко поддерживать приложение, легко изменять его и тестировать.

В *SharedPrefMovieStorage* необходимо реализовать логику работы с моделью *Movie*. Стоит обратить внимание, что в данном слое не должно быть скрытой логики работы, т.е. здесь требуется обходиться без различного типа условных конструкций и логики.

```
public class SharedPrefMovieStorage implements MovieStorage {
    private static final String SHARED_PREFS_NAME = "shared_prefs_name";
    private static final String KEY = "movie_name";
    private static final String DATE_KEY = "movie_date";
    private static final String ID_KEY = "movie_id";
    private SharedPreferences sharedPreferences;
    private Context context;

    public SharedPrefMovieStorage(Context context) {
        sharedPreferences = context.getSharedPreferences(SHARED_PREFS_NAME,
Context.MODE_PRIVATE);
    }

    @Override
    public Movie get() {
        String movieName = sharedPreferences.getString(KEY, "unknown");
        String movieDate = sharedPreferences.getString(DATE_KEY,
String.valueOf(LocalDate.now()));
        int movieId = sharedPreferences.getInt(ID_KEY, -1);
        return new Movie(movieId, movieName, movieDate);
    }

    @Override
    public boolean save(Movie movie) {
        sharedPreferences.edit().putString(KEY, movie.getName());
        sharedPreferences.edit().putString(DATE_KEY,
String.valueOf(LocalDate.now()));
        sharedPreferences.edit().putInt(ID_KEY, 1);
        sharedPreferences.edit().commit();
        return true;
    }
}
```

Теперь перейдем в репозиторий, в котором необходимо преобразовать модели одного слоя в другой. Возможно задать вопрос: «создание объектов в данном слое — это разве не плохо». С одной стороны, данные действия накладывают дополнительную логику в репозиторий, но появляется полная независимость блоков от внешних ресурсов. Давайте представим, что происходит при изменении внешней модели данных с *SharedStorage*. В данном случае, наш блок необходимо было бы трансформировать, что вызвало цепочку целого ряда изменений объектов.

Перенос данных из одной модели в другую для реализации потребностей конкретной логики является нормальной ситуацией. Рассмотрим пакет *storage* в котором находятся модель *Movie*, реализация *SharedPrefMovieStorage* и его интерфейс. Внутри *storage* возможно дополнительно создать пакет *models* и *sharedprefs* и перенести туда соответствующие классы.

Преимущество данного подхода заключается в том, что если в будущем потребуется хранение данных в БД, то необходимо будет создать дополнительную реализацию для *MovieStorage*. В результате формируется правильная структура проекта, когда, меняя какую-то часть кода, затрагивается только конкретная часть, не вызывая цепочки изменений.

Дополнительно, для повышения качества кода, возможно создать так называемые методы маппер, которые будут формировать нужные модели для слоев.

```
@SuppressWarnings("CommitPrefEdits")
@Override
public boolean saveMovie(com.mirea.fio.lecture2.domain.models.Movie movie) {
    sharedPrefMovieStorage.save(mapToStorage(movie));
    return true;
}

@Override
public com.mirea.fio.lecture2.domain.models.Movie getMovie() {
    Movie movie = sharedPrefMovieStorage.get();
    return mapToDomain(movie);
}

private Movie mapToStorage(com.mirea.fio.lecture2.domain.models.Movie movie) {
    String name = movie.getName();
    return new Movie(2, name, LocalDate.now().toString());
}

private com.mirea.fio.lecture2.domain.models.Movie mapToDomain(Movie movie) {
    String name = movie.getName();
    return new com.mirea.fio.lecture2.domain.models.Movie(movie.getId(),
        movie.getName());
}
```

1.2 Раздельные модули

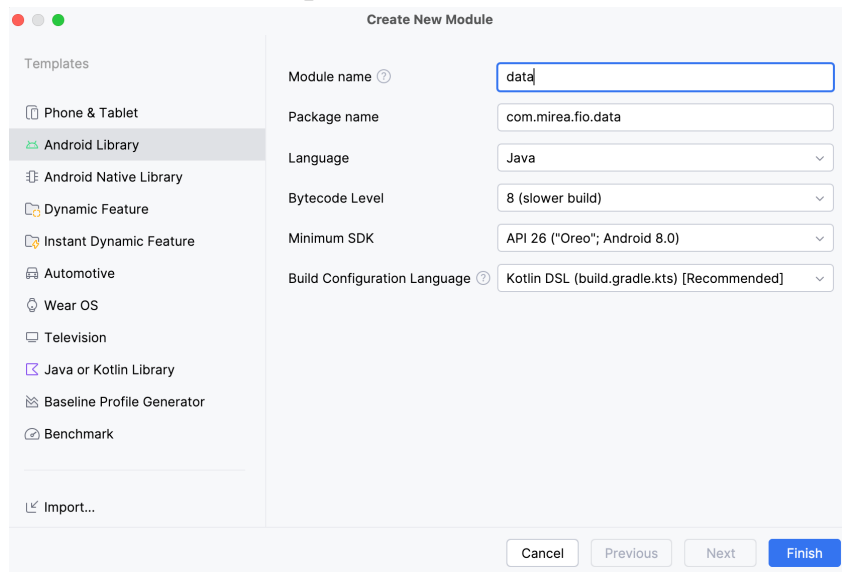
На текущий момент имеется три пакета: *data*, *domain*, пакет *Presentation*. Слой *domain* ничего не знает о слое *data*, а *data*, соответственно, ничего не знает о *Presentation*. И тут имеется важный момент – возможно спокойно обратиться к слою *data* из любого места.

Наш проект имеет правильную архитектуру, но приходит новый сотрудник и он не знаком с правилами чистой архитектуры. В разных слоях он будет использовать разные модели из различных слоев, потому что ему так будет удобней. В итоге, конечно, это могут заметить и поправить, но в больших командах за этим тяжело уследить. И эта проблема...

Второй проблемой является то, что пакеты не обеспечивают сильного разделения, то есть пакеты всё-таки это часть какой-то одной логики.

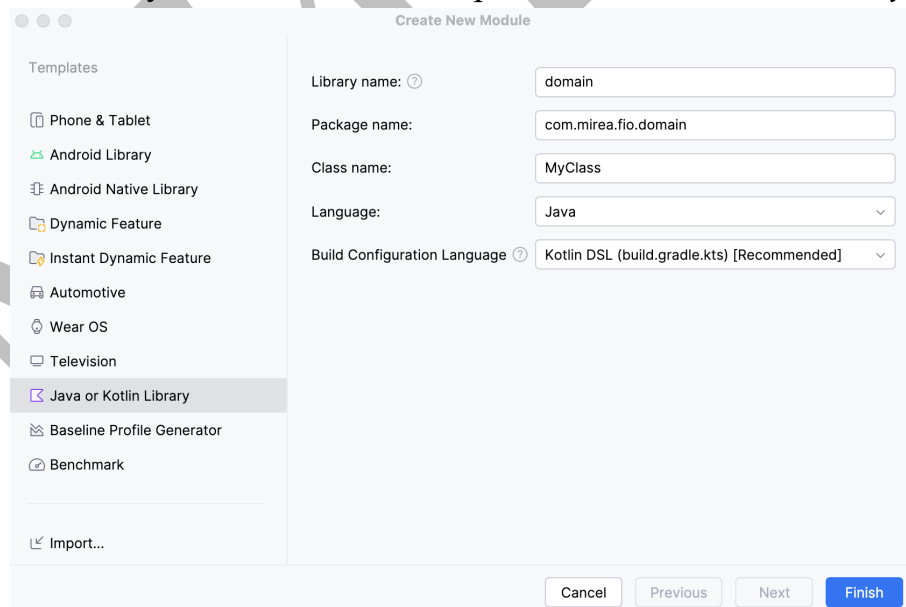
Далее необходимо исправить данные проблемы. Возможно использовать модули, которые находятся внутри вашего проекта. То есть, при создании проекта формируется директория *app* – это модуль. Таких модулей возможно много создать, и они будут между собой связаны.

Для создания нового модуля необходимо вызвать меню *File->New->New Module*. Необходимо указывать тип *Android Library*, потому что слой *data* имеет контекст, и собственно, он может работать с базой данных, *SharedPreferences* и т.д.



Далее рассмотрен файл *build.gradle*. Имеется пометка вверху файла, что это библиотека «*libs.plugins.android.library*». В блоке зависимости возможно удалить *implementation(libs.material)* и *androidTestImplementation(libs.espresso.core)* потому что, слой *data* не работает с пользовательским интерфейсом. Необходимо перенести директорию *data* из модуля *app* в модуль *data*.

Аналогично необходимо создать модуль *domain*, только учитывая, что в качестве шаблона модуля необходимо выбрать «*Java or Kotlin Library*».









После создания и компиляции модуля необходимо перенести директорию *domain* из модуля *app* в модуль *domain*.

Далее необходимо изучить, что получилось. Имеется модуль *app* и в нем размещается *gradle*-файл. На следующем занятии будет рассмотрен паттерн *MVVM* и именно в модуле *app* должны размещаться наследники класса *ViewModel*. Также в

данном модуле могут находиться директории *di* и другие папки, которые могут быть не связаны со слоем *presentation*. Поэтому данный модуль называется именно *app*, а уже внутри него существует директория *presentation*.

Далее необходимо открыть файл `settings.gradle` и убедиться в том, что в нем прописаны все модули проекта.

| | | |
|--|----|---|
|  <code>build.gradle.kts</code> | 22 | <code>rootProject.name = "Lecture2"</code> |
|  <code>gradle.properties</code> | 23 | <code>include(...projectPaths: ":app")</code> |
|  <code>gradlew</code> | 24 | <code>include(...projectPaths: ":data")</code> |
|  <code>gradlew.bat</code> | 25 | <code>include(...projectPaths: ":domain")</code> |
|  <code>local.properties</code> | 26 | |
|  <code>settings.gradle.kts</code> | | |

2 КОНТРОЛЬНОЕ ЗАДАНИЕ

В ходе выполнения практического задания № 1 были сформулированы use-case возможности проекта. Приложение должно удовлетворять следующим функциональным требованиям:

1. Авторизация в приложении.
2. Взаимодействия с каким-либо внешним сервисом, возвращающим данные о стоимости валюты, погоды и т.д. в виде json-формата. Возможно использовать сервисы типа <https://www.wiremock.io/>, <https://mockapi.io/>, <https://github.com/typicode/json-server>, <https://thecatapi.com/> и т.д.
3. Сохранение данных в БД.
4. Отображение списка каких-либо сущностей с изображениями.
5. Отображение страницы сущности.
6. Различные возможности у авторизованного и гостевого пользователя.
7. Использование обученной модели для распознавания изображений, аудио, маски и т.д. (нужно произвести анализ существующих обученных моделей TensorFlow Lite, которые желаете использовать в приложении).

Задание.

1. Требуется нарисовать прототип приложения в *figma* или аналогичных сервисах (рассматривали в прошлом семестре).
2. Требуется создать новые модули *data* и *domain*. Перенести соответствующий код приложения в данные модули.
3. Создать новую активити и реализовать в ней страницу авторизации с использованием firebase auth. Логiku работы с FB распределить между тремя модулями. (работа с firebase рассматривалась в прошлом семестре).
4. В репозитории реализовать три способа обработки данных:
 - SharedPreferences, информация о клиенте
 - Room и класс *NetworkApi* для работы с сетью с замканными данными.