МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ВОЛГОГРАДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

А.Е. Андреев, М.А. Кузнецов

Перспективные Web-технологии

Учебное пособие

Волгоград
2021

УДК 004.021

Андреев А.Е.

Перспективные Web-технологии: учеб. пособие / А.Е. Андреев, М.А. Кузнецов, ВолгГТУ. – Волгоград, 2021.–74 с.

В учебном пособии рассмотрены методические материалы по курсу «Перспективные Web-технологии». Пособие предназначено для студентов магистратуры по направлению 09.04.01 «Информатика и вычислительная техника».

СОДЕРЖАНИЕ

Создание Android-приложение на базе Architecture Components

Цели работы:

1. Рассмотреть набор компонентов Architecture Components, понять, как они взаимодействуют друг с другом.
2. Научиться создавать Android-приложение на базе Architecture Components.

Общие сведения об Android Architecture Components

Android Architecture Components – это набор библиотек и компонентов, помогающих разрабатывать Android-приложения решая, типичные задачи разработки, такие как: управление жизненным циклом или сохранения данных в БД. Android Architecture Components являются частью Android Jetpack

Android Architecture Components помогают структурировать приложение и делают его более масштабируемым, тестируемым, устойчивым к багам с меньшим количеством кода, и как следствие багов.

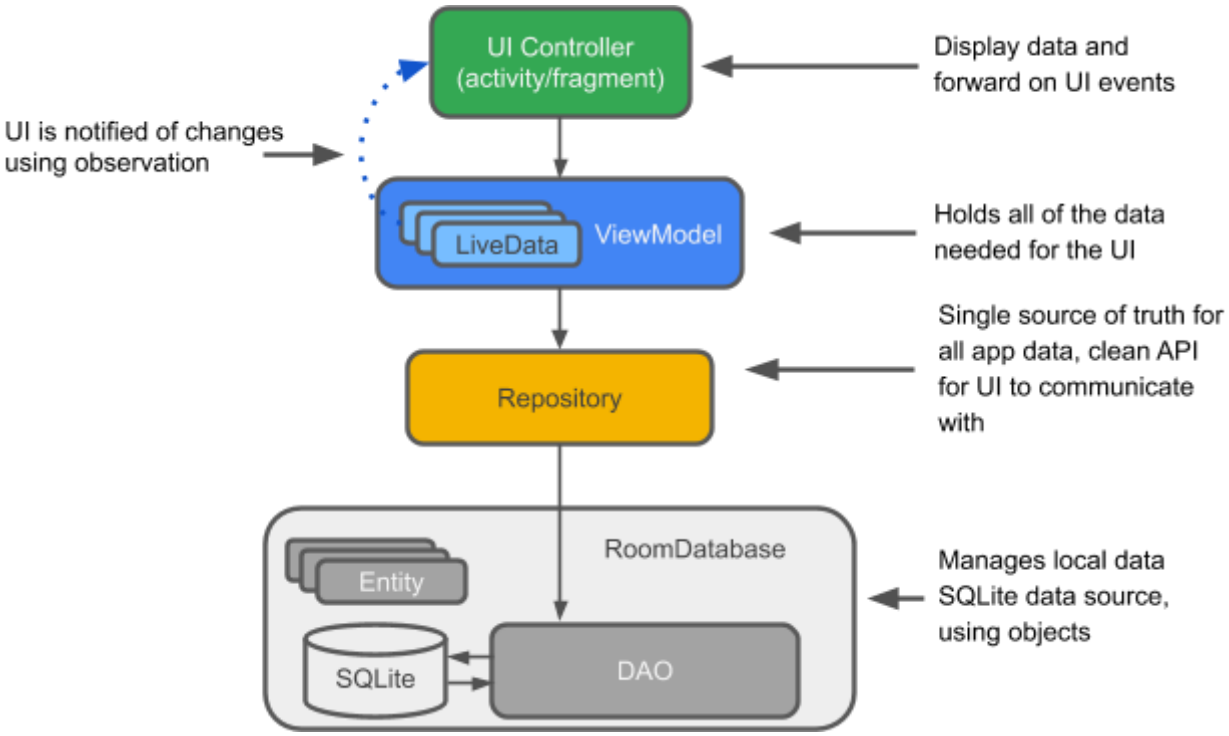


Рисунок 1 – Взаимодействие компонентов

В этом мини-курсе мы рассмотрим как работать с набором таких компонентов как: LiveData, ViewModel и Room. Но для начала, давайте поговорим о каждом из компонентов по отдельности, о его сущностях и как они работают друг с другом. Схематично, взаимодействие рассматриваемых компонентов можно представить как на рис. 1.

Давайте рассмотрим каждый из элементов по отдельности:

- Entity – класс/модель с аннотациями, описывающий таблицу базы данных при работе с Room
- База данных SQLite – способ хранения данных на устройстве. Room является высокоуровневым интерфейсом для SQLite и занимается созданием и поддержкой базы данных.
- DAO (data access object) – это класс, содержащий CRUD (create/read/update/delete) методы для конкретной сущности
- База данных Room – упрощает работу с SQLite и имеет множество методов для работы с SQLite. Использует DAO для запросов в БД SQLite
- Репозиторий (Repository) – класс для скрытия деталей реализации чтения данных из различных источников данных.
- ViewModel – посредник между репозиторием (данными) и графическим интерфейсом. Экземпляры ViewModel умеют переживать пересоздание Activity или Fragment
- LiveData – обёртка над данными, на изменения которых можно подписаться. Кэширует последнюю версию данных и оповещает подписчиков, когда данные изменились. LiveData автоматически следит и реагирует на методы жизненного цикла.

Программное обеспечение для разработки под Android

Android Studio начиная с версии 3.0

Смартфон на Android или эмулятор

Порядок создания проекта и добавление необходимых зависимостей

1. Откройте Android Studio и нажмите Start a new Android Studio project.
2. В окне Create New Project выберите Empty Activity и нажмите Next (см. рис. 2).
3. Назовите приложение RoomWordSample и нажмите Finish

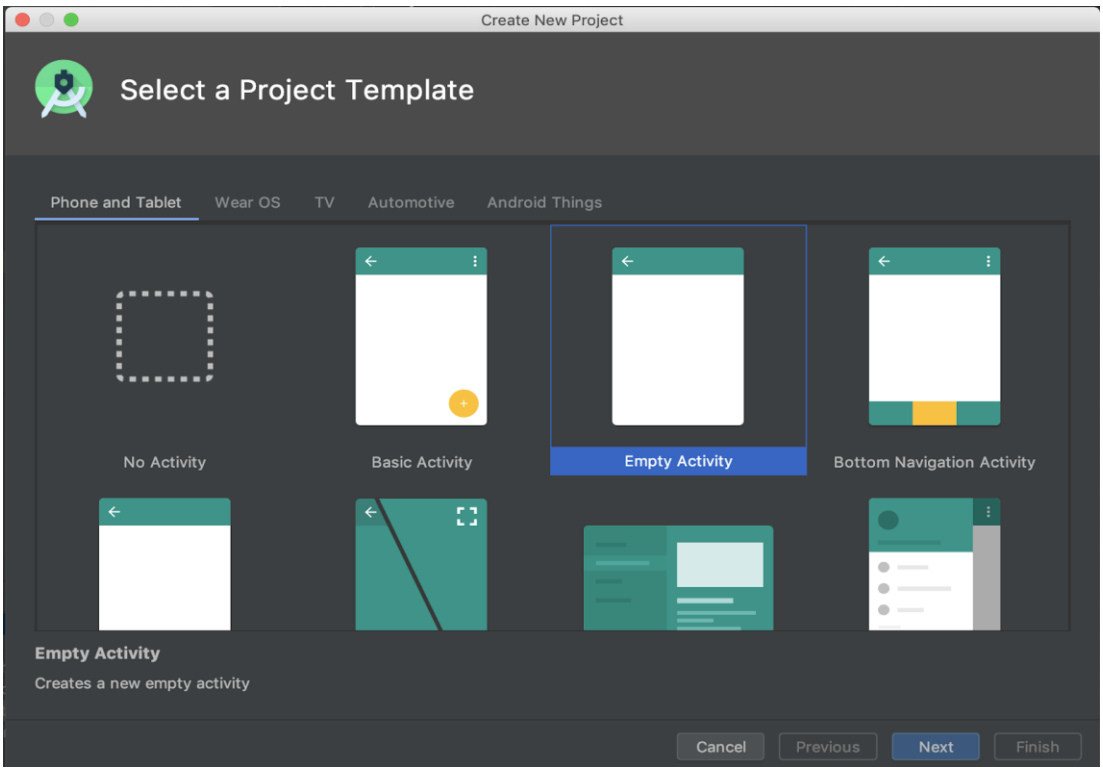


Рисунок 2 – Выбор шаблона приложения

Добавление необходимых библиотек

Для использования Android Architecture Components необходимо сделать следующее:

1. Нажмите на вкладку Projects и раскройте папку Gradle Scripts.
2. Откройте build.gradle (Module: app).
3. Добавьте следующий блок compileOptions вместо блока android чтобы установить target и source compatibility на1.8, что позволит нам использовать JDK 8 lambdas:

```
compileOptions {
    sourceCompatibility = 1.8
    targetCompatibility = 1.8
}
```

4. Замените блок dependencies на:

```
dependencies {
    implementation "androidx.appcompat:appcompat:$rootProject.appCompatVersion"

    // Dependencies for working with Architecture components
    // You'll probably have to update the version numbers in build.gradle (Project)

    // Room components
    implementation "androidx.room:room-runtime:$rootProject.roomVersion"
    annotationProcessor "androidx.room:room-compiler:$rootProject.roomVersion"
    androidTestImplementation "androidx.room:room-testing:$rootProject.roomVersion"

    // Lifecycle components
    implementation "androidx.lifecycle:lifecycle-viewmodel:$rootProject.lifecycleVersion"
    implementation "androidx.lifecycle:lifecycle-livedata:$rootProject.lifecycleVersion"
    implementation "androidx.lifecycle:lifecycle-common-
java8:$rootProject.lifecycleVersion"

    // UI
    implementation
"androidx.constraintlayout:constraintlayout:$rootProject.constraintLayoutVersion"
    implementation "com.google.android.material:material:$rootProject.materialVersion"

    // Testing
    testImplementation "junit:junit:$rootProject.junitVersion"
    androidTestImplementation "androidx.arch.core:core-
testing:$rootProject.coreTestingVersion"
    androidTestImplementation ("androidx.test.espresso:espresso-
```

```
    core:$rootProject.espressoVersion", {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
    androidTestImplementation "androidx.test.ext:junit:$rootProject.androidxJunitVersion"
}
```

5. В конец файла build.gradle (Project: RoomWordsSample) добавьте номера версий библиотек следующим образом:

```
ext {
    appCompatVersion = '1.3.0'
    constraintLayoutVersion = '2.0.4'
    coreTestingVersion = '2.1.0'
    lifecycleVersion = '2.3.1'
    materialVersion = '1.3.0'
    roomVersion = '2.3.0'
    // testing
    junitVersion = '4.13.2'
    espressoVersion = '3.1.0'
    androidxJunitVersion = '1.1.2'
}
```

Создание сущности

Для хранения заметок нам понадобится простая таблица, она выглядит следующим образом (см. рис. 3)

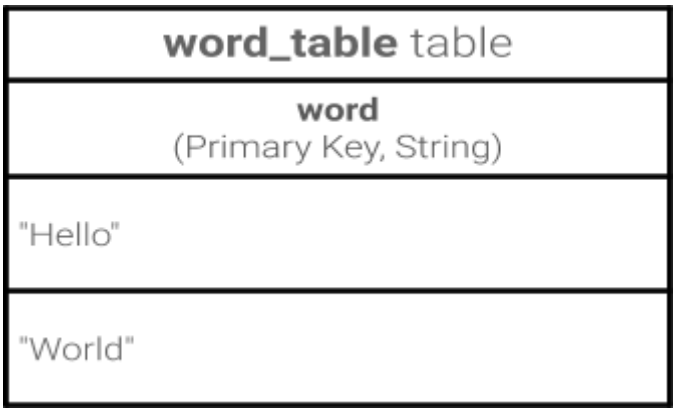


Рисунок 3 – Создание сущности

Room позволяет создавать таблицы, используя Entity. Давайте рассмотрим как.

Создайте новый Kotlin класс и создадите `public class Word` Этот класс описывает сущность – Entity, которая представляет собой таблицу SQLite в которой будут хранится данные, в нашем случае заметки. Каждое публичное поле представляет собой столбец в таблице. Room создаст из такого data класса таблицу и значения полей класса будут соответствовать значениям из таблицы.

```
public class Word {

  private String mWord;

  public Word(@NonNull String word) {this.mWord = word;}

  public String getWord(){return this.mWord;}
}
```

Чтобы Room мог создать из data класса таблицу необходимо добавить аннотации. С помощью аннотаций Room сможет сгенерировать код для построения таблицы. Для генерации таблицы из data класса обновите класс **Word** следующим образом:

```
@Entity(tableName = "word_table")
public class Word {

  @PrimaryKey
  @NonNull
  @ColumnInfo(name = "word")
```

```
    private String mWord;

    public Word(@NonNull String word) {this.mWord = word;}

    public String getWord(){return this.mWord;}
}
```

Давайте разберём каждую аннотацию:

- `@Entity(tableName = "word_table")`
  Каждый `@Entity` класс представляет собой таблицу SQLite. Вам нужно аннотировать data класс аннотацией `@Entity` чтобы Room мог создать таблицу этого data класса. Если необходимо, чтобы имя таблицы отличалось от имени data класса, вы можете использовать параметр `tableName` для указания имени. В данном случае имя таблицы будет `"word_table"`.
- `@PrimaryKey`
  Каждая сущность должна иметь primary key. В данном конкретном примере, для упрощения, каждая заметка будет одновременно являться и первичным ключом.
- `@ColumnInfo(name = "word")`
  Данная аннотация позволяет задать имя столбцу. Название столбца `"word"`.
- Каждое свойство, хранимое в БД должно быть публично или иметь метод “получения”. В нашем примере мы используем `getWord()`
- `@NonNull` Обозначает, что возвращаемое значение параметра, поля или метода никогда не может быть нулевым

Создание DAO

DAO (data access object) позволяет выполнять SQL – запросы для чтения или обновления данных. Используя DAO Room сгенерирует запросы для работы с данными, например `@Insert` Таким образом вам достаточно только описать методы доступа к вашему объекту, а Room сгенерирует запросы.

DAO должен быть интерфейсом или абстрактным классом. По умолчанию,

все запросы в БД должны выполняться в отдельном потоке. Room поддерживает корутины, для этого используется `suspend` модификатор

Давайте создадим DAO который будет

- Возвращать все слова алфавитном порядке
- Вставлять слова
- Удалять все слова

1. Создадим новый файл класса и назовём его WordDao.
2. Скопируйте и вставьте следующий код в WordDao и исправьте его, чтобы он смог скомпилироваться:

```
@Dao
public interface WordDao {

  // allowing the insert of the same word multiple times by passing a
  // conflict resolution strategy
  @Insert(onConflict = OnConflictStrategy.IGNORE)
  void insert(Word word);

  @Query("DELETE FROM word_table")
  void deleteAll();

  @Query("SELECT * FROM word_table ORDER BY word ASC")
  List<Word> getAlphabetizedWords();
}
```

- WordDao является интерфейсом; DAOs могут быть либо интерфейсом, либо абстрактным классом.
- Аннотация `@Dao` обозначает DAO класс для Room.
- `void insert(Word word);` описывает метод вставки слова:
- `@Insert` аннотация относится к DAO методу вставки, где вам не нужно самостоятельно описывать SQL-запрос (К таким же методам относятся `@Delete` или `@Update` для обновления строк)
- `onConflict = OnConflictStrategy.IGNORE`: Выбранная стратегия игнорирует

новое слово, если уже существует такое же.

- **deleteAll**(): объявление метода для удаления всех слов.
- Вспомогательного метода для удаления нет (как например при вставке или обновлении) поэтому для удаления необходимо добавить аннотацию **@Query**
- **@Query**("DELETE FROM word_table"): **@Query** аннотация используется для создания SQL – запроса, сам запрос нужно добавить в виде строки.
- List<Word> **getAlphabetizedWords**(): метод для получения списка слов
- **@Query**("SELECT * FROM word_table ORDER BY word ASC"): возвращает список слов в порядке возрастания

Класс LiveData

Обычно, когда меняются данные, нужно обновить состояние и графического интерфейса, который отображал данные. Это значит, что необходимо подписаться на обновления данных, чтобы реагировать на обновления соответствующе. Это может быть сложной задачей, однако теперь с помощью LiveData это стало проще простого.

LiveData – это компонент, умеющий обрабатывать методы жизненного цикла и входящий в так называемую lifecycle library. Достаточно использовать класс `LiveData` в качестве возвращаемого значения в описании модели и Room сгенерирует весь необходимый код для обновления модели, когда данные в таблице базы данных изменятся. Если необходимо обновлять данные нужно использовать `MutableLiveData` вместо `LiveData`. Класс `MutableLiveData` имеет два публичных метода, позволяющих обновлять значения объекта: `setValue(T)` и `postValue(T)`

В `WordDao` поменяйте код следующим образом:

```
@Query("SELECT * FROM word_table ORDER BY word ASC")
LiveData<List<Word>> getAlphabetizedWords();
```

Позже мы добавим `Observer` в `MainActivity` для отслеживания изменений.

База данных Room

Знакомство с базой данных Room

- Room – высокоуровневый интерфейс для доступа к базе данных SQLite.
- Room позволяет легко работать с данными, до этого использовался `SQLiteOpenHelper`
- Room использует DAO для выполнения запросов к базе данных.
- По умолчанию, для большей производительности и оптимизации UI, Room не позволяет выполнять запросы в главном потоке. Все запросы работают асинхронно в фоновом потоке.
- Во время компиляции Room проверяет SQL выражения и сразу предупреждает об ошибках в запросе.

Чтобы создать класс базы данных, необходимо создать абстрактный класс, который наследуется от `RoomDatabase` Обычно достаточно одного экземпляра базы данных для всего приложения, поэтому используется паттерн Singleton. Создайте класс БД как показано ниже:

```
@Database(entities = {Word.class}, version = 1, exportSchema = false)
public abstract class WordRoomDatabase extends RoomDatabase {

  public abstract WordDao wordDao();

  private static volatile WordRoomDatabase INSTANCE;
  private static final int NUMBER_OF_THREADS = 4;
  static final ExecutorService databaseWriteExecutor =
      Executors.newFixedThreadPool(NUMBER_OF_THREADS);

  static WordRoomDatabase getDatabase(final Context context) {
      if (INSTANCE == null) {
          synchronized (WordRoomDatabase.class) {
              if (INSTANCE == null) {
                  INSTANCE =
Room.databaseBuilder(context.getApplicationContext(),
                          WordRoomDatabase.class, "word_database")
                          .build();
              }
          }
      }
```

```
            return INSTANCE;
        }
    }
```

1. Класс базы данных Room должен быть **abstract** и наследоваться от `RoomDatabase`
2. С помощью аннотации **@Database** данный класс помечен как класс базы данных Room. Кроме этой аннотации нужно добавить список сущностей (таблиц), которые должны быть в этой БД. Также не забудьте указать версию базы данных с помощью version = 1. Для простоты мы не станем рассматривать миграции базы данных, поэтому параметр `exportSchema` выставлен в **false**. В продакшн версии приложения вы должны учесть этот параметр, чтобы иметь возможность экспортировать схему базы данных.
3. Чтобы получить доступ к DAO нужно создать абстрактное свойство с типом возвращаемого объекта, в данном пример `WordDao`
4. **getDatabase** возвращает синглтон. Во время первого обращения будет создана БД Room, используя с названием `"word_database"`.
5. Мы создали службу `ExecutorService` с фиксированным пулом потоков, которую вы будете использовать для асинхронного выполнения операций с базой данных в фоновом потоке.

Обратите внимание, если вы меняете схему базы данных (например добавляете новые поля, меняете тип данных) то необходимо поднимать версию и реализовать стратегию миграции для клиентов со старой версией базы данных. Для этого примера мы выбрали стратегию удаления и пересоздания базы данных. Минус такого подхода в том, что все данные будут потеряны при обновлении версии. Для более подробного описания как реализовать миграцию можно ознакомиться с этой статьей https://medium.com/androiddevelopers/understanding-migrations-with-room-f01e04b07929

Знакомство с патерном Repository (Репозиторий) и создание слоя для доступа к данным

Репозиторий не является частью Android Architecture Components или частью Android. Это абстракция, скрывающая реализацию доступа к источнику данных. Такой способ является полезным и очень популярным, потому что в этом случае вы жестко не завязаны на какую-то конкретную реализацию источника данных. С помощью репозитория ваш клиентский код (например код из Activity или Fragment) не знает к какому источнику данных он обращается. Например, вам нужно получить список слов. Вы можете получить данные из сети, из файла или из локальной базы данных. Но, если вы не используете паттерн репозиторий, то каждый раз, когда у вас меняется источник данных, вам необходимо переписывать и менять код в вашем Activity или Fragment'е. Используя же абстракцию в виде репозитория вы обращаетесь только к классу, который реализует интерфейс репозитория и детали скрыты. Таким образом ваш код устойчив к изменениям, является более гибким и соответствует принципу single responsibility.

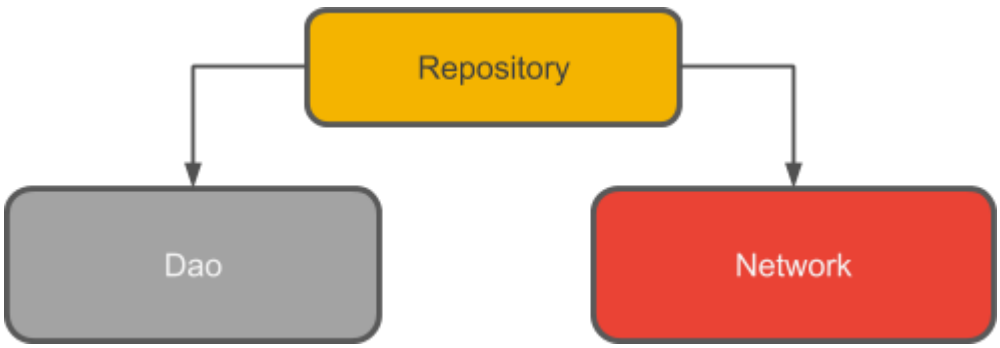


Рисунок 4 – Паттерн репозиторий

В мобильной разработке репозиторий часто используется для смены источника данных: либо для получения данных из сети, или получение уже закешированных данных из локальной базы данных.

Создание репозитория для доступа к данным

Для создания класса репозитория создайте класс `WordRepository` как показано ниже:

```
class WordRepository {

    private WordDao mWordDao;
    private LiveData<List<Word>> mAllWords;

    // Note that in order to unit test the WordRepository, you have to remove the Application
    // dependency. This adds complexity and much more code, and this sample is not about testing.
    // See the BasicSample in the android-architecture-components repository at
    // https://github.com/googlesamples
    WordRepository(Application application) {
        WordRoomDatabase db = WordRoomDatabase.getDatabase(application);
        mWordDao = db.wordDao();
        mAllWords = mWordDao.getAlphabetizedWords();
    }

    // Room executes all queries on a separate thread.
    // Observed LiveData will notify the observer when the data has changed.
    LiveData<List<Word>> getAllWords() {
        return mAllWords;
    }

    // You must call this on a non-UI thread or your app will throw an exception. Room ensures
    // that you're not doing any long running operations on the main thread, blocking the UI.
    void insert(Word word) {
        WordRoomDatabase.databaseWriteExecutor.execute(() -> {
            mWordDao.insert(word);
        });
    }
}
```

Давайте пройдёмся по основным моментам:

- Вместо экземпляра базы данных мы передаём только DAO в репозиторий. Это нужно потому что нам необходим доступ только к данным из таблицы Word, а DAO имеет все методы для получения и обновления данных.
- Метод `getAllWords()` возвращает список слов LiveData из Room; мы можем сделать это благодаря тому, как мы определили метод

`getAlphabetizedWords` для возврата `LiveData` на шаге "Класс LiveData". Room выполняет все запросы в отдельном потоке. Затем наблюдаемые `LiveData` уведомят наблюдателя в основном потоке об изменении данных.

- Нам не нужно запускать вставку в основном потоке, потому что мы используем `ExecutorService`, который создали в `WordRoomDatabase` для работы вставки в фоновом потоке.

Создание ViewModel

`ViewModel` является частью lifecycle library и предоставляет данные для UI и умеет переживать изменения конфигурации. **`ViewModel`** является посредником между репозиторием и UI. Кроме того, вы можете использовать **`ViewModel`** для обмена данными между фрагментами.

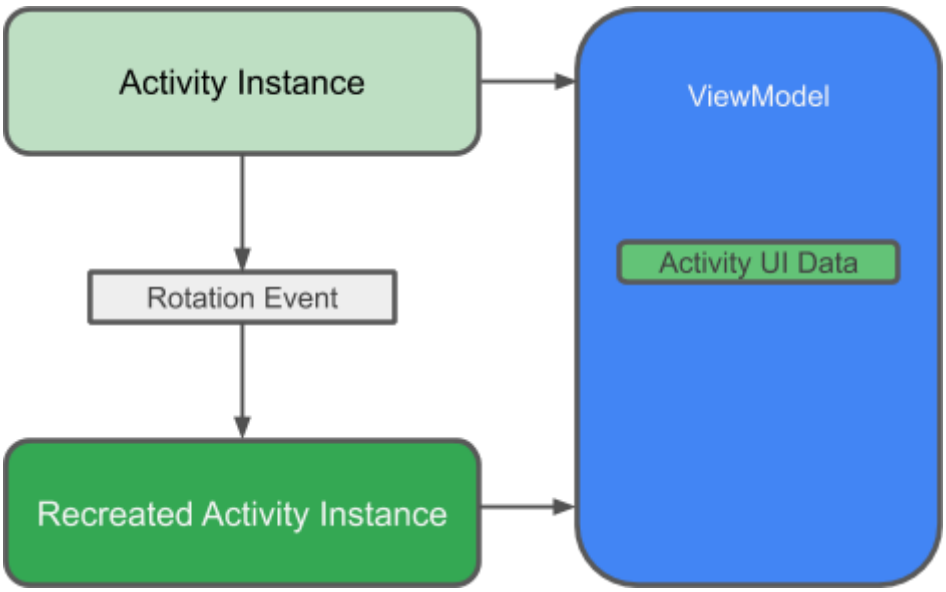


Рисунок 5 - ViewModel

`ViewModel` содержит данные для UI и умеет корректно обрабатывать изменения конфигурации, то есть переживать пересоздание Activity/Fragment.

Отдельное от Activity/Fragment хранение данных, который нужно отображать в UI позволяет следовать принципу единственной ответственности (single responsibility principle) Таким образом, пока Activity или Fragment нужны только для создания экрана, **`ViewModel`** в свою очередь занимается наполнением и обработкой данных, отображаемых в UI – элементах. Лучше всего использовать **`ViewModel`** совместно с `LiveData` для данных, которые могут изменится после отображения на экране. Использование `LiveData` имеет несколько преимуществ:

- Можно подписаться на обновление данных (вместо постоянной проверки на изменения вручную). Тогда UI обновится только в том случае, когда это необходимо, то есть когда изменятся данные.
- Репозиторий (как источник данных) и UI полностью разделены и общаются через **`ViewModel`**.
- Не нужно делать запросы в БД из **`ViewModel`** (эта логика реализуется в репозитории). Это делает код более тестируемым.

Создайте класс **`WordViewModel`** как показано ниже.

```
public class WordViewModel extends AndroidViewModel {
  private WordRepository mRepository;
  private final LiveData<List<Word>> mAllWords;
  public WordViewModel (Application application) {
     super(application);
     mRepository = new WordRepository(application);
     mAllWords = mRepository.getAllWords();
  }
  LiveData<List<Word>> getAllWords() { return mAllWords; }
  public void insert(Word word) { mRepository.insert(word); }
```

}

Как обычно, давайте разберём, что здесь есть:

1. Класс **`WordViewModel`** принимает `application` в качестве аргумента и наследуется от `AndroidViewModel`.
2. Добавлена закрытая переменная для хранения ссылки на репозиторий
3. Добавлен метод **`getAllWords`**`()` для возврата кэшированного списка слов.
4. Реализован конструктор, который создает `WordRepository`.
5. В конструкторе инициализировали allWords LiveData, использую репозиторий.
6. Метод **`insert`**() является методом – обёрткой для вызова метода **`insert`**() в репозитории. Так как работать с БД рекомендуется в отдельном потоке мы используем корутины для вызова метода в отдельном потоке, на случай если вставка элемента затянется, то мы не будем блокировать главный поток, а значит для пользователя работа приложения будет максимально удобной.

Создание ячейки списка для отображения UI

Создание стиля для ячейки списка

Создадим тему для приложения, установив для темы AppTheme следующее значение `Theme.MaterialComponents.Light.DarkActionBar` Добавьте стиль в список элементов values/styles.xml:

```
<resources>

    <!-- Base application theme. -->
    <style name="AppTheme" parent="Theme.MaterialComponents.Light.DarkActionBar">
        <!-- Customize your theme here. -->
```

```
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
    </style>

    <!-- The default font for RecyclerView items is too small.
    The margin is a simple delimiter between the words. -->
    <style name="word_title">
        <item name="android:layout_width">match_parent</item>
        <item name="android:layout_marginBottom">8dp</item>
        <item name="android:paddingLeft">8dp</item>
        <item name="android:background">@android:color/holo_orange_light</item>
        <item
name="android:textAppearance">@android:style/TextAppearance.Large</item>
    </style>
</resources>
```

Создание файла верстки для ячейки

Чтобы создать элемент списка, ячейку, добавьте следующий код в layout/recyclerview_item.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
```

```
<TextView
    android:id="@+id/textView"
    style="@style/word_title"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="@android:color/holo_orange_light" />
</LinearLayout>
```

Создание верстки экрана

В файле `layout/activity_main.xml`, замените `TextView` списком `RecyclerView` и добавьте плавающую кнопку (FAB). Теперь вёрстка файла должна выглядеть так:

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recyclerview"
        android:layout_width="0dp"
        android:layout_height="0dp"
        tools:listitem="@layout/recyclerview_item"
```

```
    android:padding="@dimen/big_padding"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

<com.google.android.material.floatingactionbutton.FloatingActionButton
    android:id="@+id/fab"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_margin="16dp"
    android:contentDescription="@string/add_word"/>

</androidx.constraintlayout.widget.ConstraintLayout>
```

Добавление иконки в FAB

Давайте добавим иконку, которая будет помогать пользователю понять, что она нужна для добавления. Для этого необходимо:

1. Выберите File > New > Vector Asset.
2. Нажмите на иконку Android робот в поле Clip Art:

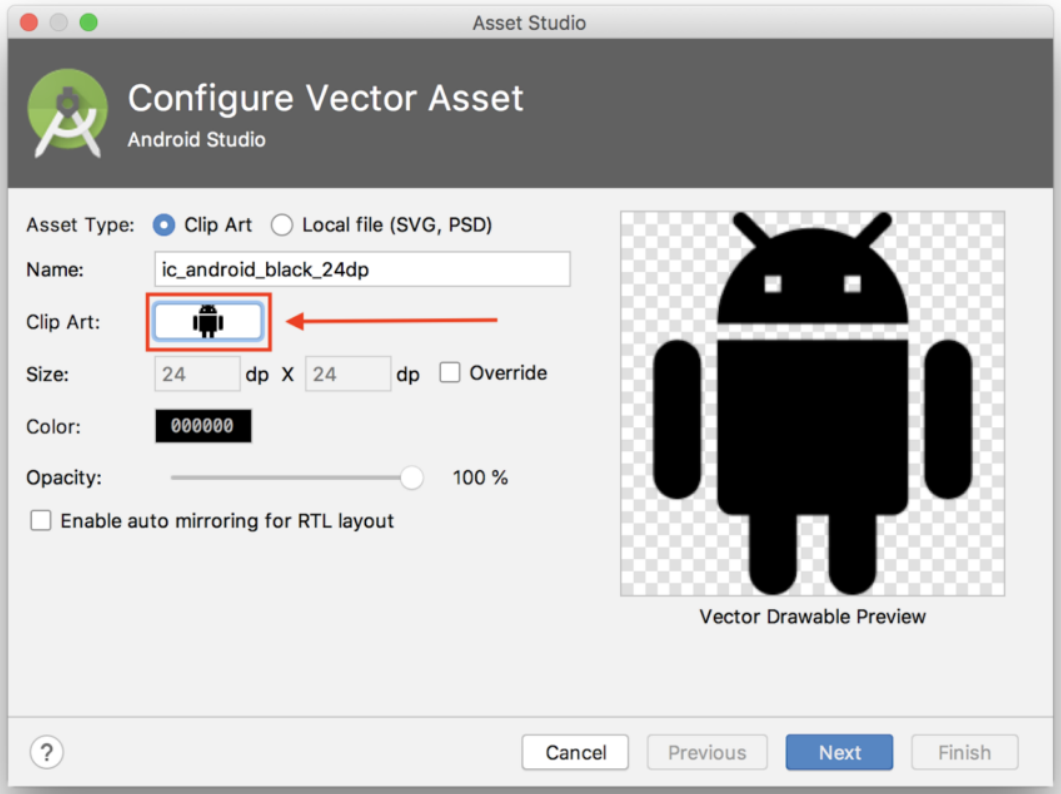


Рисунок 6 – Добавление иконки

3. Найдите категорию “add” и выберите иконку ‘+’ . Нажмите OK.

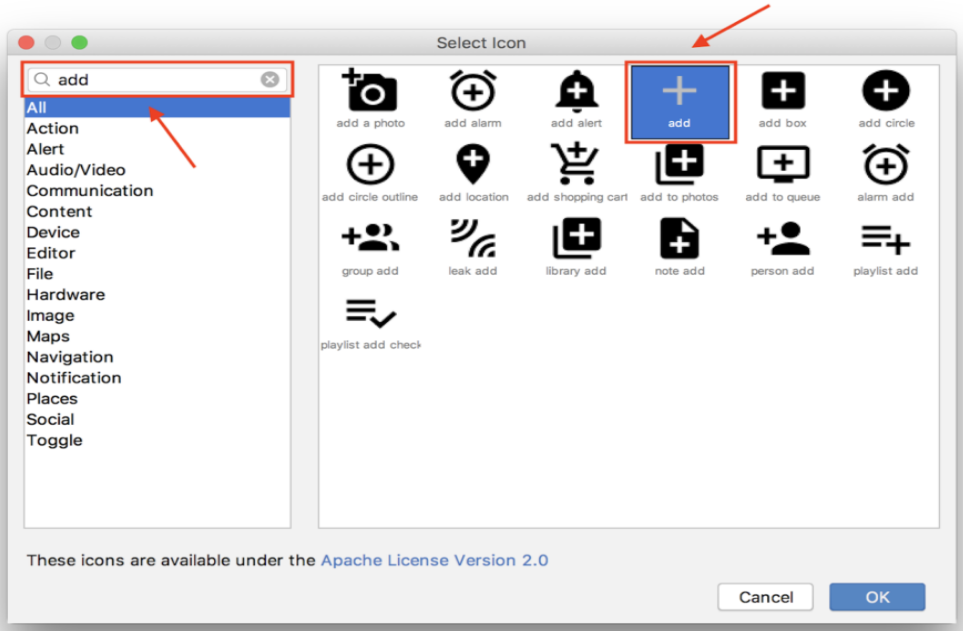


Рисунок 7 – Выбор стандартной иконки

4. После этого нажмите Next

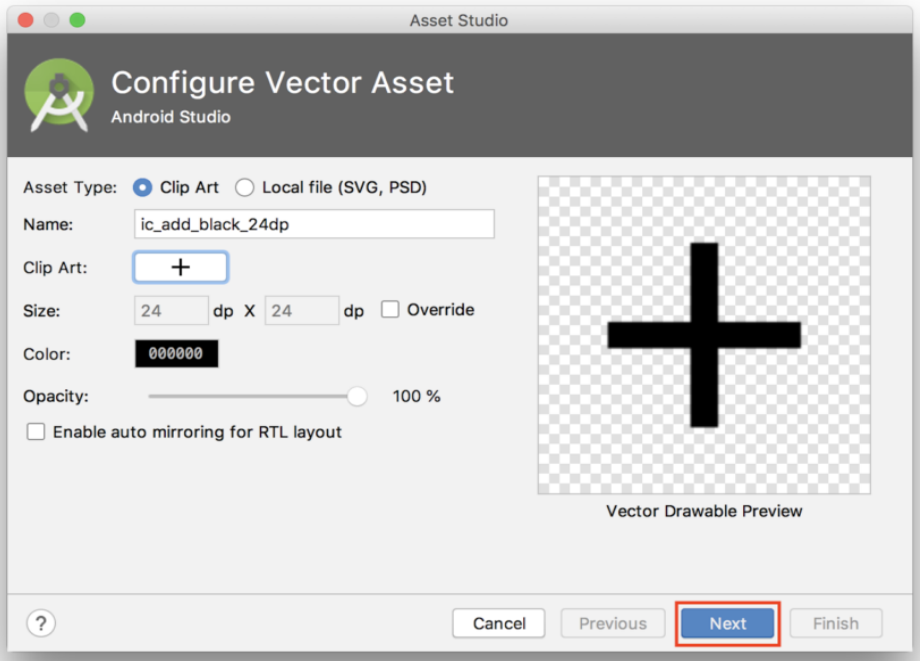


Рисунок 8 – Выбор свойств изображения

5. Выберите путь `main > drawable` и нажмите кнопку Finish чтобы добавить изображение

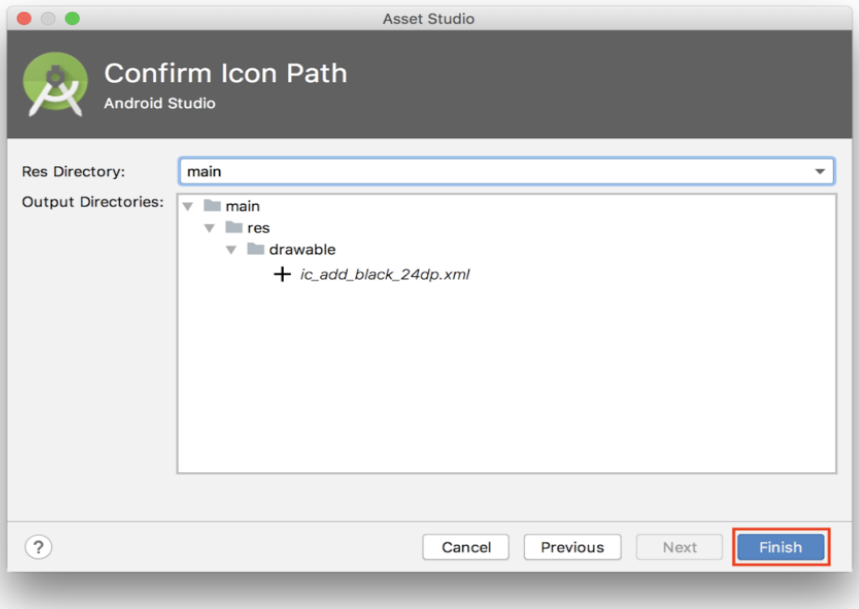


Рисунок 9 – Добавление изображения в ресурсы

6. Обновите файл `layout/activity_main.xml`, добавив в FAB только что добавленный в проект ресурс иконки:

```
<com.google.android.material.floatingactionbutton.FloatingActionButton
    android:id="@+id/fab"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_margin="16dp"
    android:contentDescription="@string/add_word"
    android:src="@drawable/ic_add_black_24dp"/>
```

Создание адаптера и добавление RecyclerView

Теперь мы отобразим данные в `RecyclerView`, что будет лучше чем просто поместить данные в `TextView`.

Для отображения данных в `RecyclerView` создайте адаптер как показано ниже

```
class WordViewHolder extends RecyclerView.ViewHolder {
    private final TextView wordItemView;

    private WordViewHolder(View itemView) {
        super(itemView);
        wordItemView = itemView.findViewById(R.id.textView);
    }

    public void bind(String text) {
        wordItemView.setText(text);
    }

    static WordViewHolder create(ViewGroup parent) {
        View view = LayoutInflater.from(parent.getContext())
                .inflate(R.layout.recyclerview_item, parent, false);
        return new WordViewHolder(view);
    }
}
```

Создайте класс **`WordListAdapter`**, который расширяет ListAdapter. Создайте DiffUtil.ItemCallback как статический класс **`WordListAdapter`**. Вот код:

```
public class WordListAdapter extends ListAdapter<Word, WordViewHolder> {

    public WordListAdapter(@NonNull DiffUtil.ItemCallback<Word> diffCallback) {
        super(diffCallback);
    }

    @Override
    public WordViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
        return WordViewHolder.create(parent);
    }

    @Override
    public void onBindViewHolder(WordViewHolder holder, int position) {
        Word current = getItem(position);
        holder.bind(current.getWord());
    }

    static class WordDiff extends DiffUtil.ItemCallback<Word> {

        @Override
        public boolean areItemsTheSame(@NonNull Word oldItem, @NonNull Word newItem) {
            return oldItem == newItem;
        }

        @Override
        public boolean areContentsTheSame(@NonNull Word oldItem, @NonNull Word newItem)
{
            return oldItem.getWord().equals(newItem.getWord());
        }
```

```
    }
}
```

После этого добавьте `RecyclerView` в метод `onCreate()` в `MainActivity`. В методе `onCreate()` после `setContentView`:

```
RecyclerView recyclerView = findViewById(R.id.recyclerview);

final WordListAdapter adapter = new WordListAdapter(new
WordListAdapter.WordDiff());

recyclerView.setAdapter(adapter);

recyclerView.setLayoutManager(new LinearLayoutManager(this));
```

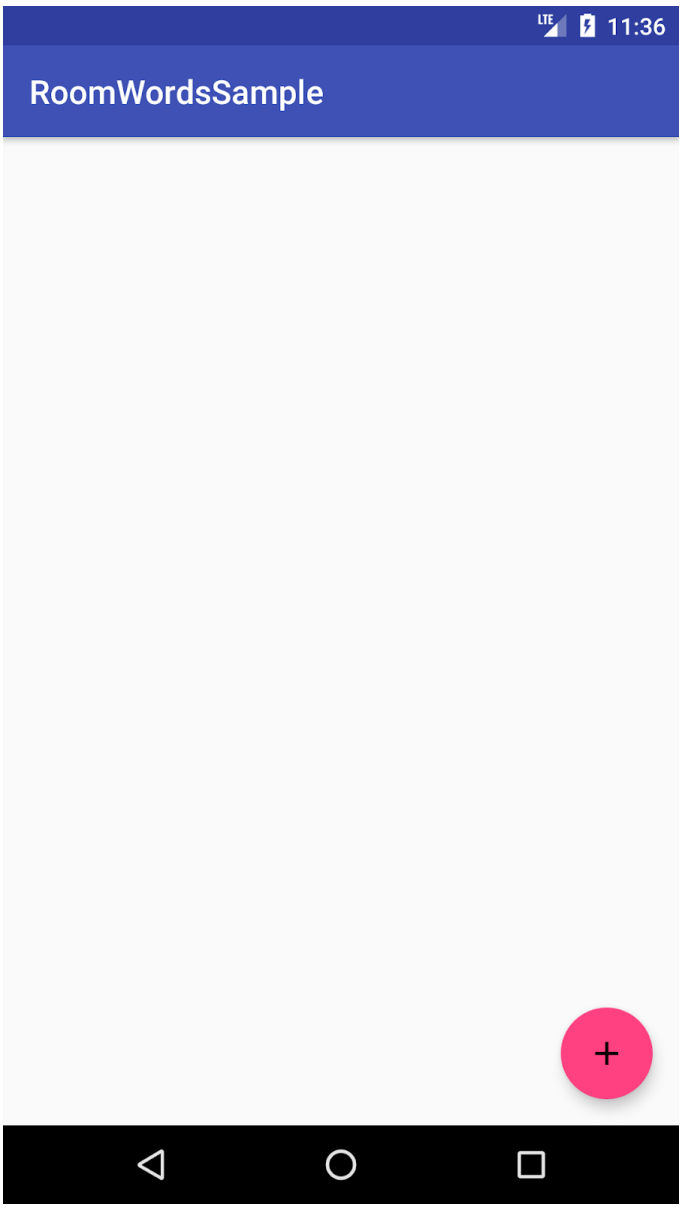


Рисунок 10 – Запуск приложения

Запустите приложение, чтобы проверить, что все работает корректно (см. рис. 10). Пока мы не добавили ни одного элемента, поэтому в данный момент экран приложения будет пустой.

Добавление записи в БД используя Room

В данный момент в нашей БД нет данных. Мы рассмотрим добавление данных двумя способами: добавим часть данных вручную, открыв соединение с базой данных, а второй – создадим `Activity` для добавления новых слов. Для удаления всех данных и заполнения базы данных записями, мы создадим RoomDatabase.Callback и переопределим метод `onCreate()`.

Вот код для создания колбэка в классе `WordRoomDatabase`. Поскольку вы не можете выполнять операции с базой данных в потоке пользовательского интерфейса, `onCreate()` использует ранее определенный `databaseWriteExecutor` для выполнения лямбда-выражения в фоновом потоке. Лямбда удаляет содержимое базы данных, затем заполняет ее двумя словами "Hello" и "World". Не стесняйтесь добавлять больше слов!

```
private static RoomDatabase.Callback sRoomDatabaseCallback = new
RoomDatabase.Callback() {
    @Override
    public void onCreate(@NonNull SupportSQLiteDatabase db) {
        super.onCreate(db);

        // If you want to keep data through app restarts,
        // comment out the following block
        databaseWriteExecutor.execute(() -> {
            // Populate the database in the background.
            // If you want to start with more words, just add them.
            WordDao dao = INSTANCE.wordDao();
            dao.deleteAll();

            Word word = new Word("Hello");
            dao.insert(word);
            word = new Word("World");
            dao.insert(word);
        });
    }
```

```
};
```

После этого, добавьте реализованный колбэк в метод создания экземпляра базы данных

```
.addCallback(sRoomDatabaseCallback)
```

Создание Activity

Теперь мы создадим Activity, которая умеет добавлять слова в БД.

Для этого сначала добавьте следующие строковые ресурсы в `values/strings.xml`:

```
<string name="hint_word">Word...</string>
<string name="button_save">Save</string>
<string name="empty_not_saved">Word not saved because it is empty.</string>
```

И добавьте цвета в `value/colors.xml`:

```
<color name="buttonLabel">#FFFFFF</color>
```

Кроме этого, давайте добавим размеры, для этого

1. Выберите модуль app в окне Project
2. Выберите File > New > Android Resource File
3. Из доступных категорий выберите Dimension
4. Выберите имя: dimens

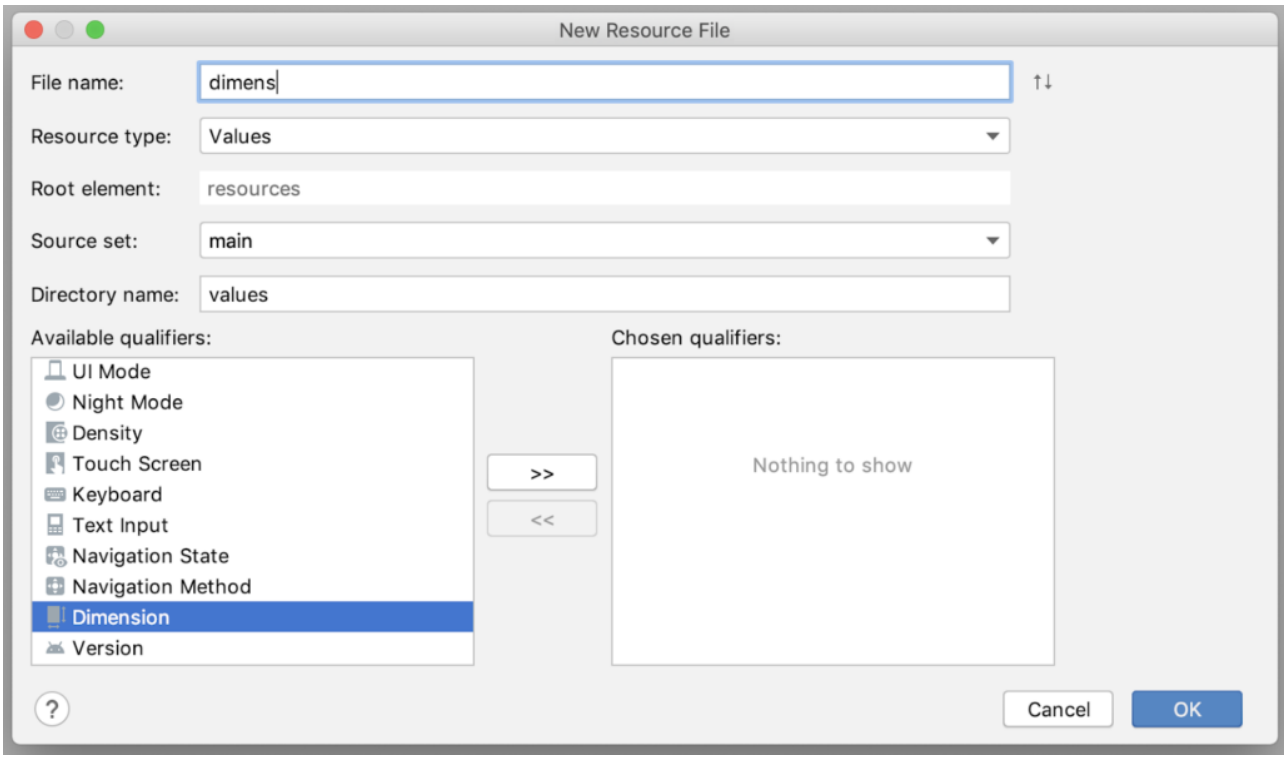


Рисунок 11 - Создание Activity

Вставьте значения размеров в `values/dimens.xml` как показано ниже:

```
<dimen name="small_padding">8dp</dimen>

<dimen name="big_padding">16dp</dimen>
```

После этого создайте новую Activity

1. Выберите File > New > Activity > Empty Activity
2. Введите название `NewWordActivity`.
3. Не забудьте проверить что новая Activity появилась в Android Manifest

```
<activity android:name=".NewWordActivity"></activity>
```

Обновите `activity_new_word.xml` как показано ниже:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
```

```
    <EditText
        android:id="@+id/edit_word"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:minHeight="@dimen/min_height"
        android:fontFamily="sans-serif-light"
        android:hint="@string/hint_word"
        android:inputType="textAutoComplete"
        android:layout_margin="@dimen/big_padding"
        android:textSize="18sp" />

    <Button
        android:id="@+id/button_save"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="@color/colorPrimary"
        android:text="@string/button_save"
        android:layout_margin="@dimen/big_padding"
        android:textColor="@color/buttonLabel" />

</LinearLayout>ty>
```

Далее, обновите код в новой Activity как показано ниже:

```
public class NewWordActivity extends AppCompatActivity {
```

```
    public static final String EXTRA_REPLY =
"com.example.android.wordlistsql.REPLY";


    private  EditText mEditWordView;


    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_new_word);
        mEditWordView = findViewById(R.id.edit_word);


        final Button button = findViewById(R.id.button_save);
        button.setOnClickListener(view -> {
            Intent replyIntent = new Intent();
            if (TextUtils.isEmpty(mEditWordView.getText())) {
                setResult(RESULT_CANCELED, replyIntent);
            } else {
                String word = mEditWordView.getText().toString();
                replyIntent.putExtra(EXTRA_REPLY, word);
                setResult(RESULT_OK, replyIntent);
            }
            finish();
        });
    }
}
```

Подключение к базе данных

Последним шагом является подключение пользовательского интерфейса к базе данных путем сохранения новых слов, введенных пользователем, и отображения текущего содержимого базы данных word в окне `RecyclerView`.

Чтобы отобразить текущее наполнение базы данных добавьте наблюдателя, который наблюдает за `LiveData` в `ViewModel`.

Всякий раз, когда данные изменяются, вызывается колбэк `onChanged()`, который вызывает метод `setWords()` адаптера для обновления кэшированных данных адаптера и обновления отображаемого списка.

В `MainActivity` создайте переменную для `ViewModel`:

```
private WordViewModel mWordViewModel;
```

Используйте ViewModelProvider , чтобы связать вашу `ViewModel` с вашей `Activity`.

Когда ваша `Activity` запустится в первый раз, `ViewModelProviders` создаст `ViewModel`. Когда `Activity` уничтожается, например, в результате изменения конфигурации, `ViewModel` сохраняется. Когда `Activity` создается заново, `ViewModelProviders` возвращает существующую `ViewModel`.

В `onCreate()` под блоком кода `RecyclerView` получите `ViewModel` из `ViewModelProvider`:

```
mWordViewModel = new
ViewModelProvider(this).get(WordViewModel.class);
```

Также в `onCreate()` добавьте наблюдателя для `LiveData` возвращаемых `getAlphabetizedWords()`. Метод `onChanged()` срабатывает, когда наблюдаемые данные изменяются и действие выходит на передний план:

```
mWordViewModel.getAllWords().observe(this, words -> {
    // Update the cached copy of the words in the adapter.
    adapter.submitList(words);
});
```

Определите код запроса в качестве элемента `MainActivity`:

```
public static final int NEW_WORD_ACTIVITY_REQUEST_CODE = 1;
```

В `MainActivity` добавьте код `onActivityResult()` для `NewWordActivity`. Если действие возвращается с `RESULT_OK`, вставьте возвращенное слово в базу данных, вызвав метод `insert()` `WordViewModel`:

```
public void onActivityResult(int requestCode, int
resultCode, Intent data) {

   super.onActivityResult(requestCode, resultCode, data);



   if (requestCode == NEW_WORD_ACTIVITY_REQUEST_CODE &&
resultCode == RESULT_OK) {

       Word word = new
Word(data.getStringExtra(NewWordActivity.EXTRA_REPLY));

       mWordViewModel.insert(word);

   } else {

       Toast.makeText(

               getApplicationContext(),

               R.string.empty_not_saved,

               Toast.LENGTH_LONG).show();

   }

};
```

В `MainActivity` запустите `NewWordActivity`, когда пользователь нажмет

на кнопку. В `MainActivity` `onCreatefab` найдите `FloatingActionButton` и `onClickListener` с помощью этого кода:

```
FloatingActionButton fab = findViewById(R.id.fab);

fab.setOnClickListener( view -> {

   Intent intent = new Intent(MainActivity.this,
NewWordActivity.class);

   startActivityForResult(intent,
NEW_WORD_ACTIVITY_REQUEST_CODE);

});
```

Теперь запустите ваше приложение. Когда будет добавлено новое слово в базу данных `NewWordActivity`, пользовательский интерфейс автоматически обновит это.

Итог

Теперь, когда мы получили рабочее приложение, давайте ещё раз посмотрим на то, что мы сделали. Вот структура нашего приложения:

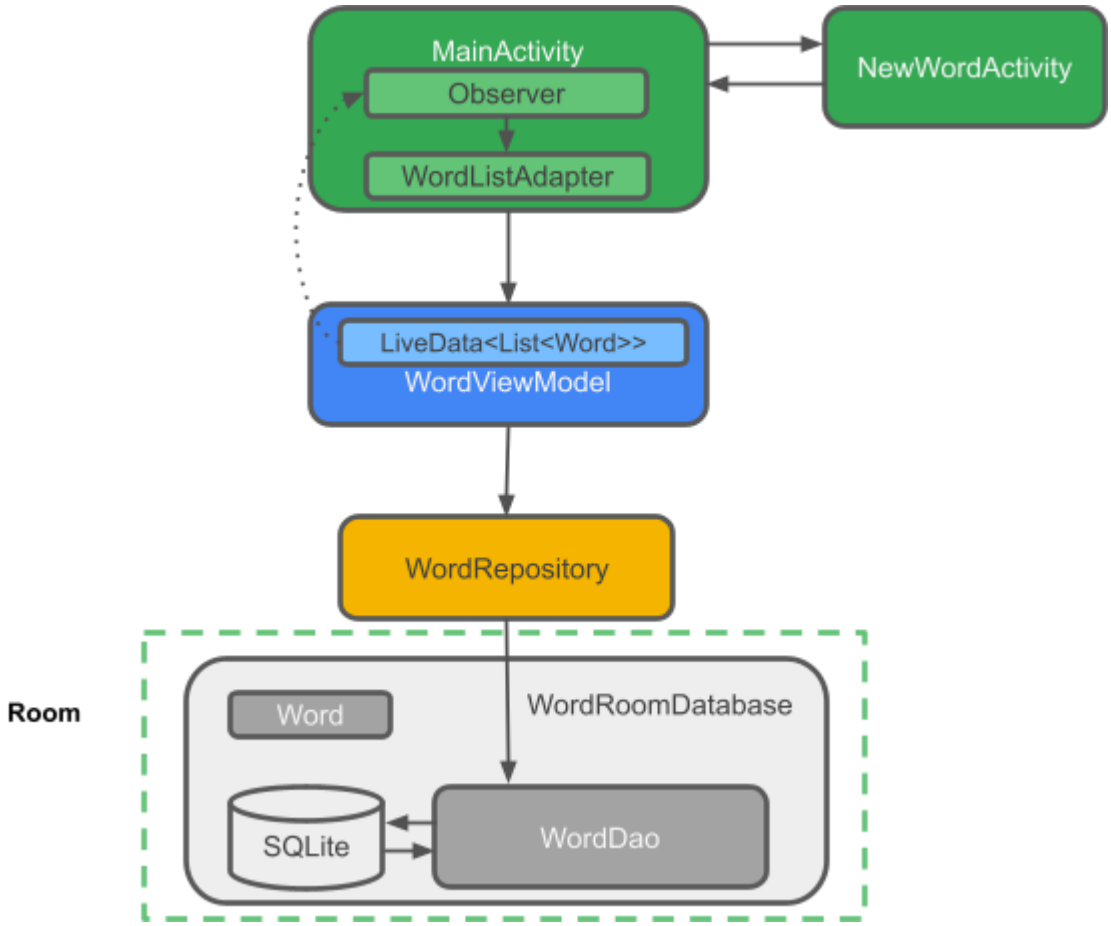


Рисунок 12 – Архитектура приложения

Компоненты приложения:

- `MainActivity`: отображает слова списком, используя `RecyclerView` и `WordListAdapter`. В `MainActivity` `Observer`, который наблюдает за `LiveData` из базы данных и оповещает, когда они изменяются.
- `NewWordActivity`: добавляет новое слово в список.
- `WordViewModel`: предоставляет методы для доступа к уровню данных и возвращает `LiveData`, чтобы `MainActivity` мог настроить связь `Observer`.*
- `LiveData<List<Word>>`: делает возможным автоматическое обновление в компонентах UI. В `MainActivity` `Observer`, который наблюдает за `LiveData` из базы данных и оповещает, когда они изменяются.
- `Repository`: управляет одним или несколькими источниками данных. Хранилище предоставляет методы для `ViewModel` для взаимодействия с

базовым поставщиком данных. В этом приложении этот сервер представляет собой базу данных `Room`

- `Room`: является оболочкой и реализует базу данных SQLite. `Room` делает за вас много работы, которую раньше вам приходилось делать самому.
- `DAO`: сопоставляет вызовы метода с запросами базы данных, так что, когда хранилище вызывает метод, такой как `getAlphabetizedWords()`, `Room` может извлечь `SELECT * FROM word_table ORDER BY word ASC`.
- `Word`: класс сущности, который содержит одно слово.
- `Views` и `Activities` (и `Fragments`) взаимодействуют с данными только через `ViewModel`. Таким образом, не имеет значения, откуда берутся данные.

Контрольные вопросы

1 Что из себя представляет Android Architecture Components? Где это применимо?

2 Перечислите компоненты Android Architecture Components, которые использовались в ходе лабораторной работы.

3 Что такое Live Data?

4 Опишите принцип работы базы данных Room.

5 Что такое репозиторий, и когда он используется?

6 Плюсы использования ViewModel с Live Data?

Создание Web приложения

План

1. Постановка задачи
2. Планирование
3. Разработка класса треугольника через тесты
4. Консольное приложение
5. Присоединяем форму
6. Простейший рефакторинг
7. Добавляем БД/файл и строим диаграмму классов
8. Рефакторинг – применяем принцип DIP
9. Попробуем паттерн MVC и другие …
10. Упаковка

Постановка задачи

1. Написать программу на C# с GUI для вычисления гипотенузы и площади прямоугольного треугольника
2. Предусмотреть также работу с БД, в которой будут храниться объекты-треугольники, задаваемые двумя катетами . Их можно записывать и извлекать.
3. Предусмотреть также сохранение и извлечение данных о треугольниках в текстовом файле.
4. Предусмотреть запуск программы в консольном режиме.
5. Предусмотреть запуск программы в веб-режиме

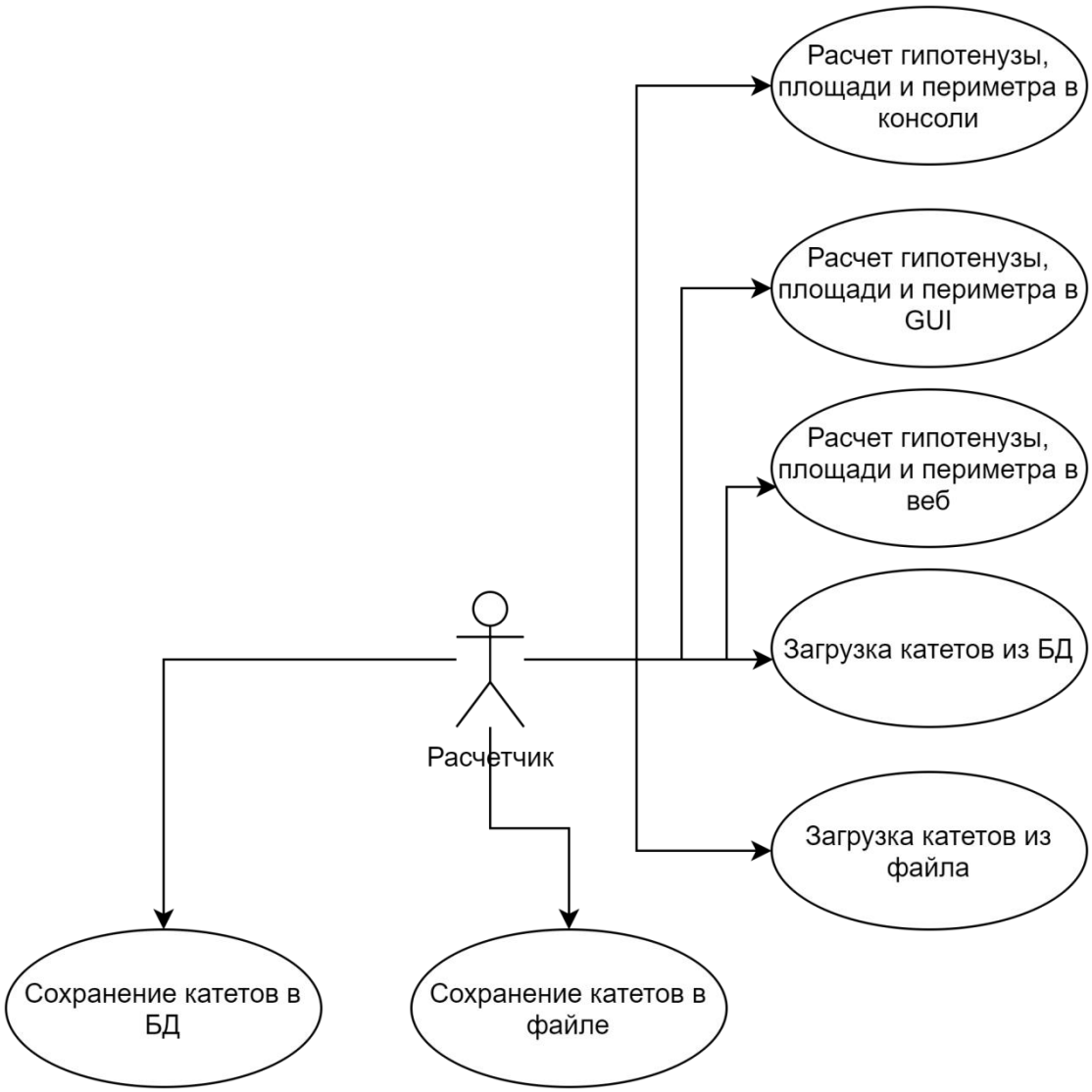


Рисунок 13 - Диаграмма использования

План версий

Пожелание	**Трудоемкость (недель)**	**Итерация №**	**Версия №**
1. Расчет гипотенузы в консольном режиме	0,2	1	1
2. Графический интерфейс	0,2	2	1
3. С БД	0,2	3	2
4. С файлом	0,1	4	2
5. Веб-версия	0,4	…	…

Планирование

План первой итерации

Пожелание	Трудоем-кость (дней)	Ответственный
0. Создание проекта и проч.	0,1	Р
1. Класс прямоуг. Треугольника	0,7	Р
2. Приложение (консольное)	0,2	Р

Разработка класса прямоугольного треугольника через тесты. Создаем решение и проекта (.NET Core 5)

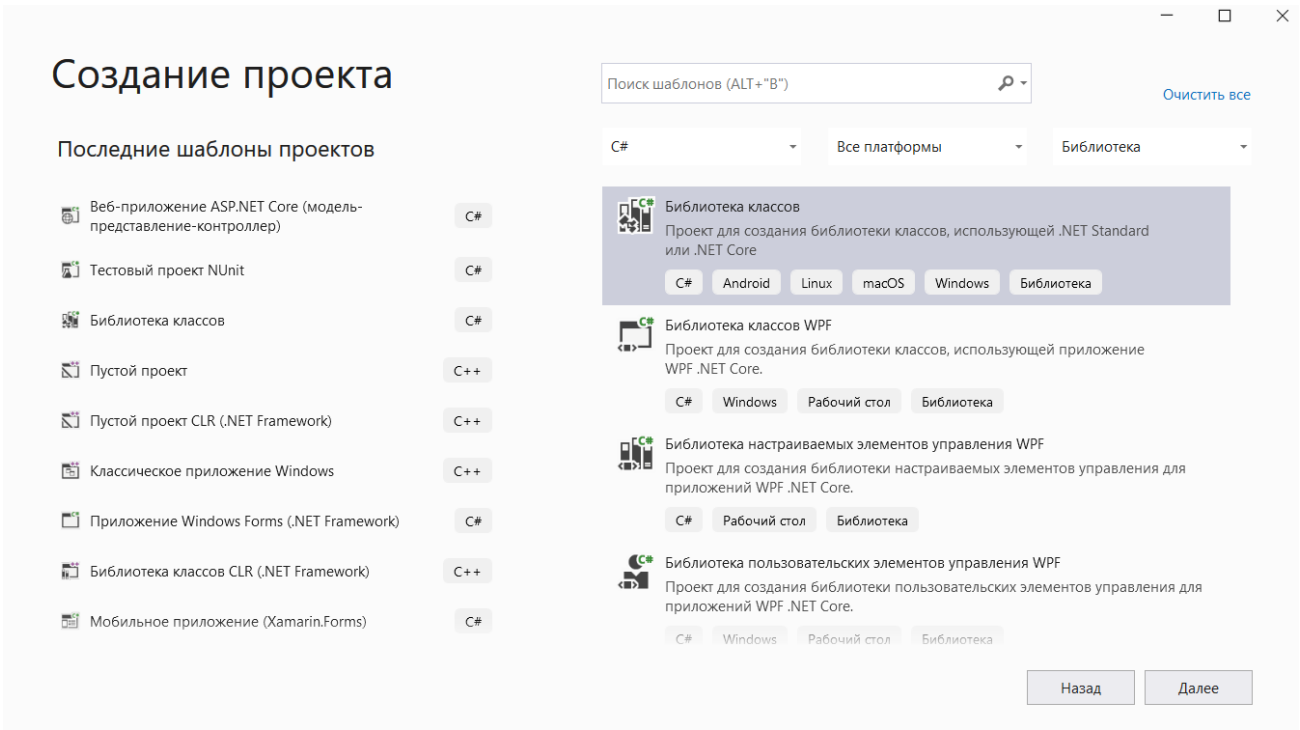


Рисунок 14 – Создание проекта

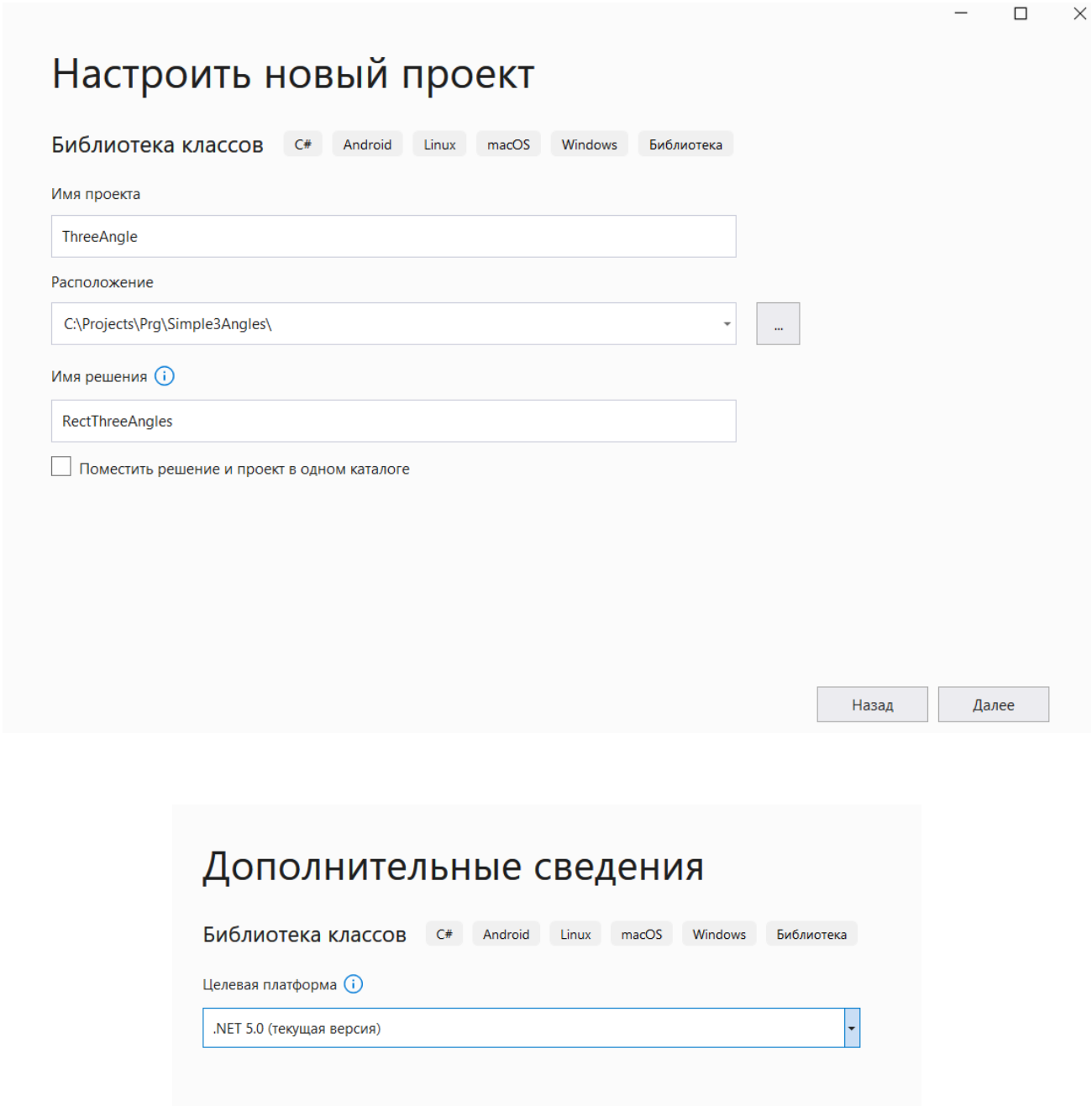
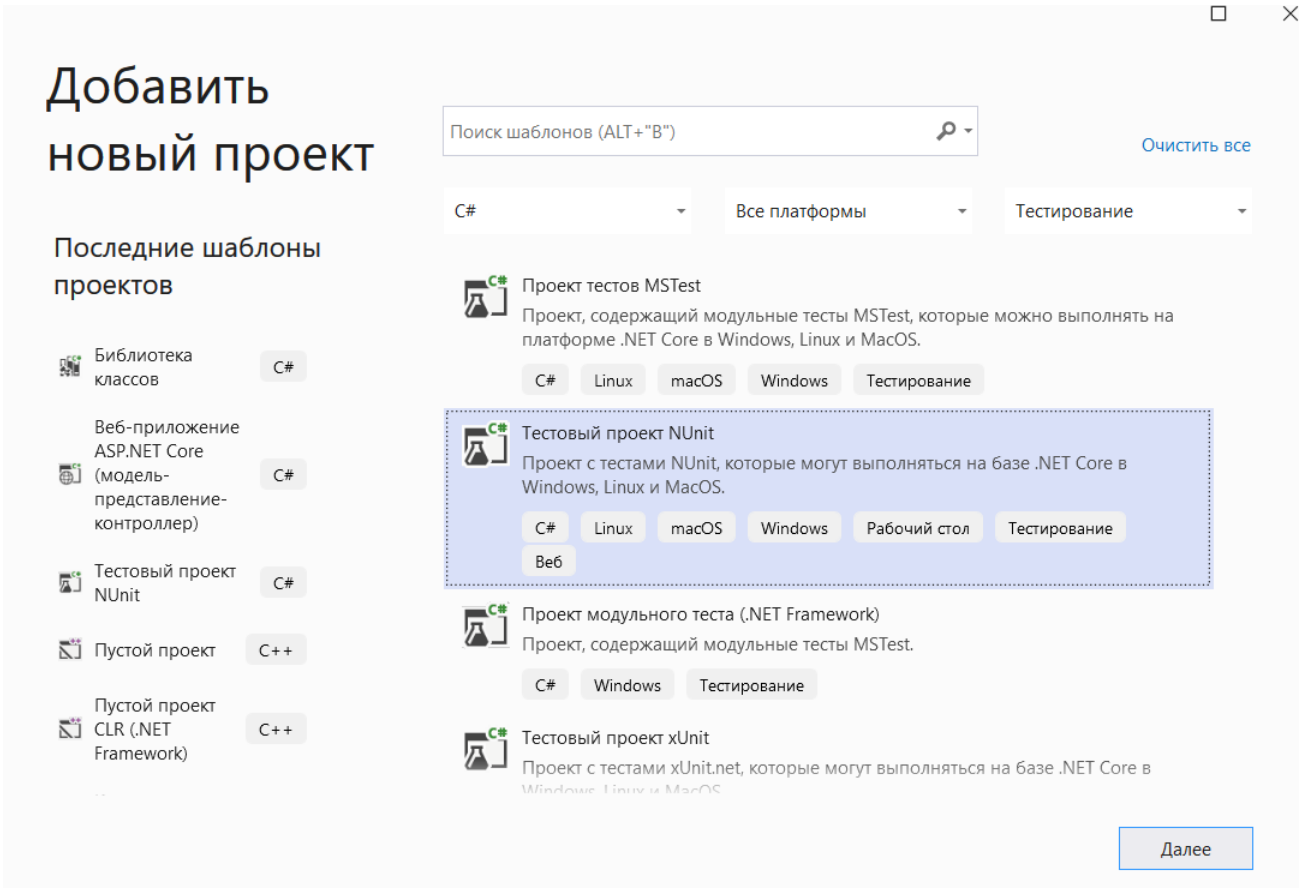


Рисунок 15 – Настройка проекта



Ссылка на библиотеку в тестовом проекте

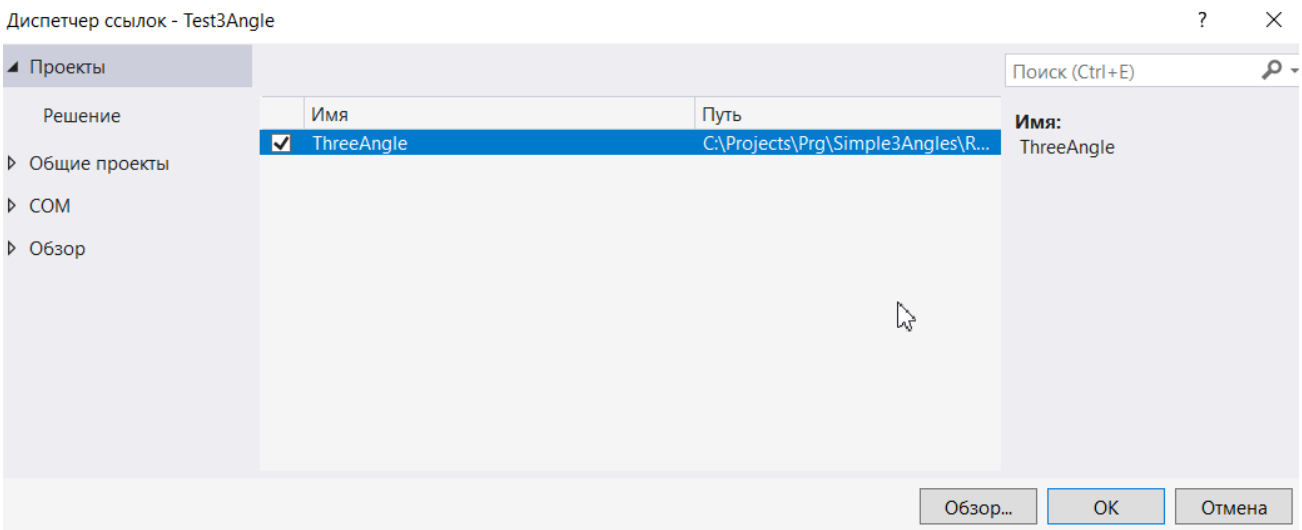


Рисунок 16 – Добавление новых свойств проекта

Список тестов

1. Тест создания объекта и считывания полей

2. Тест вырожденного треугольника (гипотенуза)
3. Тест на ошибку при создании с некорректными параметрами
4. Тест расчета гипотенузы
5. Тест определения площади
6. Тест определения площади прямоугольника, куда вписан
7. Тест вывода в строку

Примеры тестов

```
public class Test3Angle  {
      ThreeAngle s3angle;
      [SetUp]
      public void Init()  {
         s3angle = new ThreeAngle(3, 4);
      }
      [Test]
      public void testCreateAndRead()  {
         Assert.AreEqual(3, s3angle.A);
         Assert.AreEqual(4, s3angle.B);
      }
      [Test]
      public void testCalcC()
      {
         Assert.AreEqual(5, s3angle.C);
      }
```

Описание класса

```
  public class ThreeAngle
     {
        public ThreeAngle(double a, double b) {
           A = a; B = b;
        }
        public double A { get; set; }
        public double B { get; set; }
        public double C {
           get  {
              return Math.Sqrt(A * A + B * B);
           }
        }
        public double Per {
           get  {
```

```
            return A + B + C;
        }
    }

public double getArea(bool threeAngle)
    {
        return threeAngle ? A * B / 2 : A * B;
    }
    public override string ToString()
    {
        return A.ToString() + " " + B.ToString() +
            " " + C.ToString();
    }
}
```

Запуск тестов

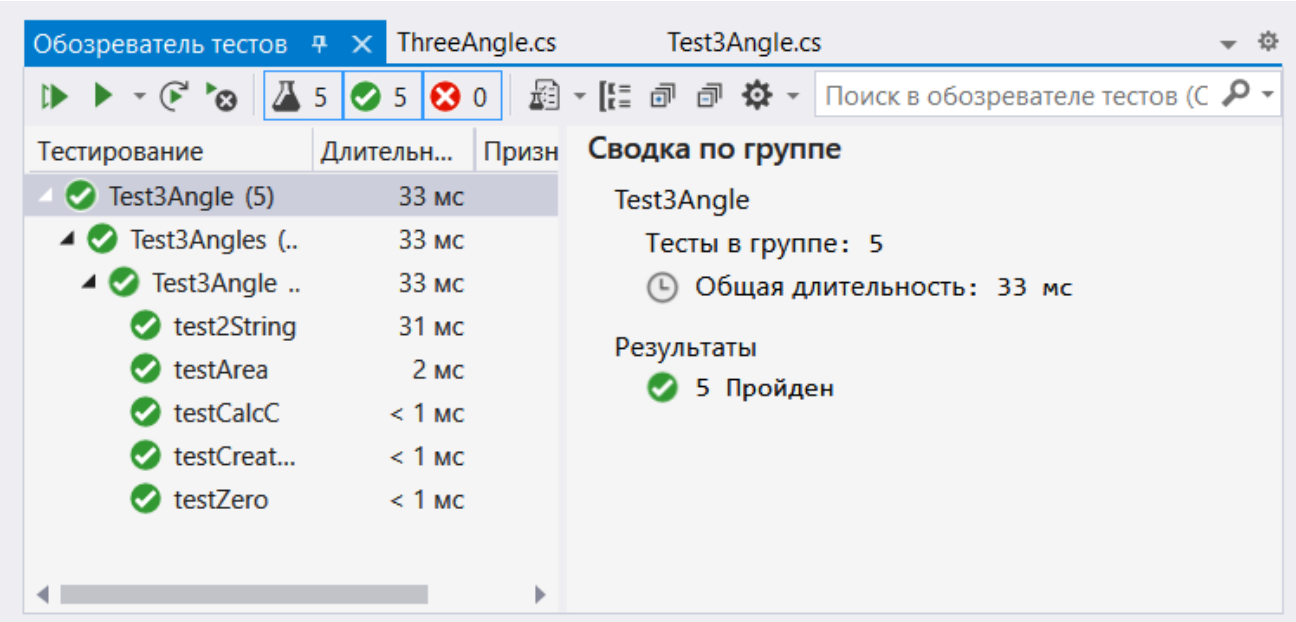


Рисунок 17 – Запуск теста

Консольное приложение

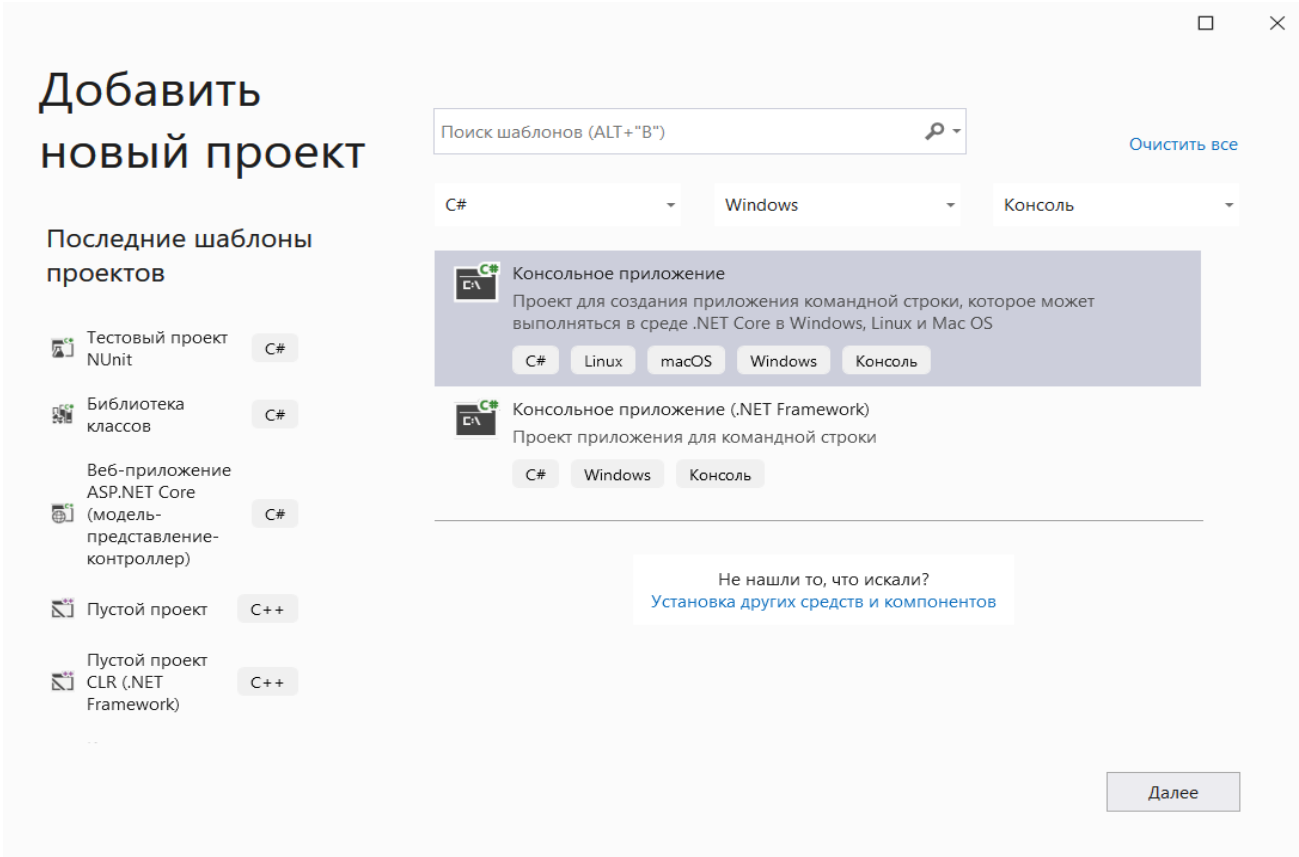


Рисунок 18 – Новый проект консольного приложения

Нужно добавить ссылку на проект с классом

```
using System;
using ThreeAngles;
namespace Rect3AnglesConsole
{
   class Program
   {
      static void Main(string[] args)
      {
         ThreeAngle s3angle = new ThreeAngle(
            Double.Parse(args[0]), Double.Parse(args[1]));
         Console.WriteLine(s3angle.ToString());
         Console.ReadKey();
      }
   }
}
```

После запуска с параметрами 3 4

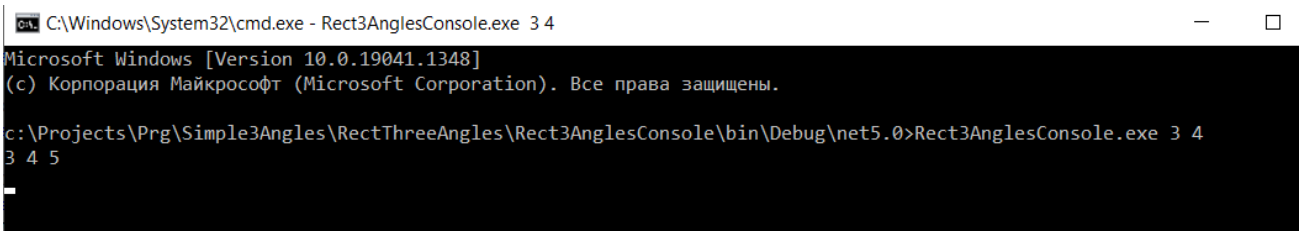
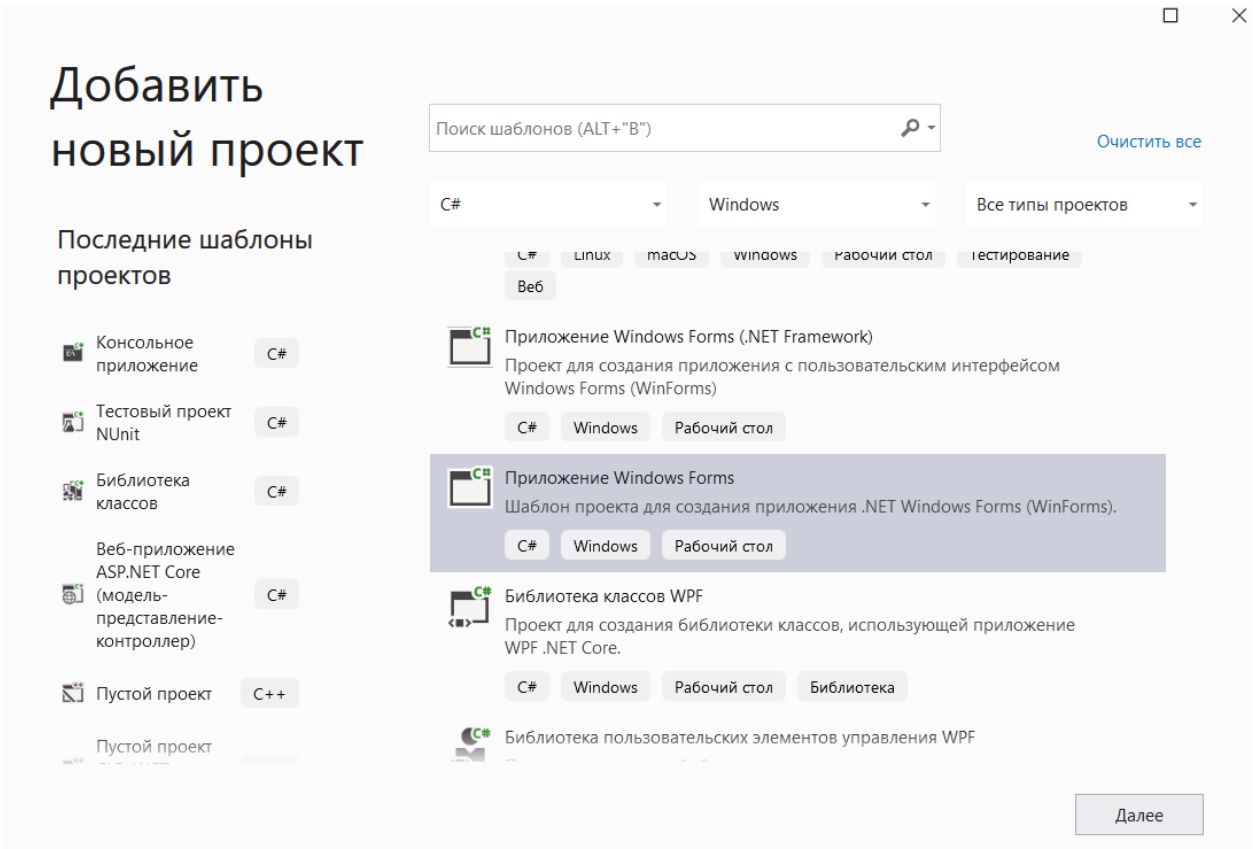


Рисунок 19 – Запуск консольного приложениея

Присоединяем форму



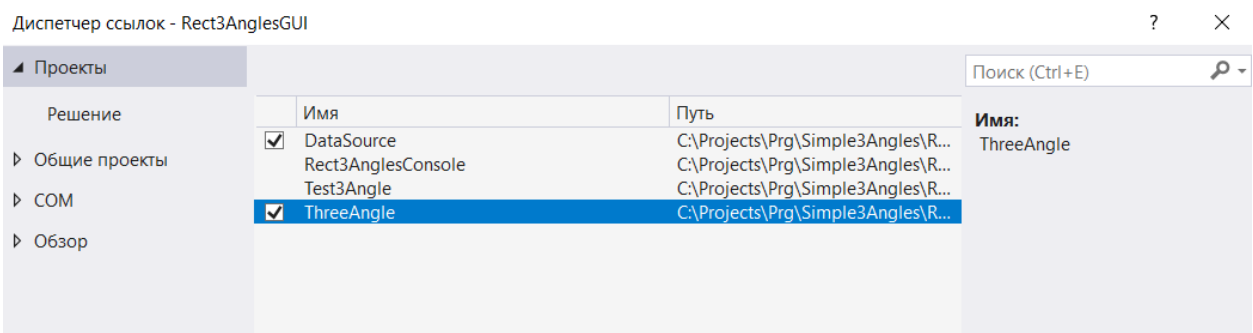


Рисунок 20 – Новый проект графического приложения

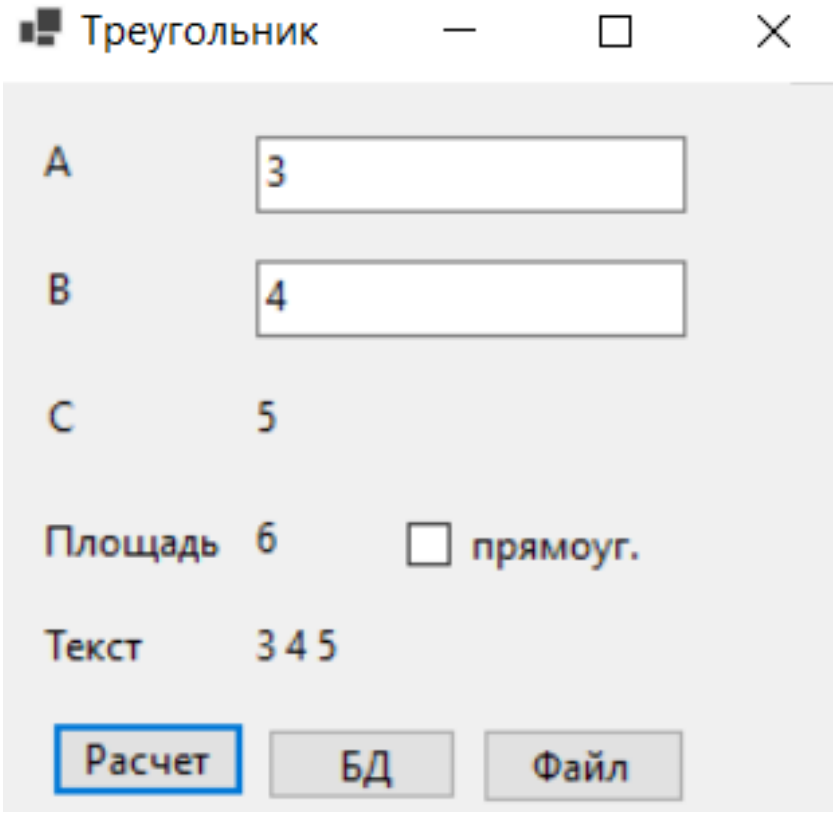


Рисунок 21 – Форма ввода параметров

```
public partial class Form1 : Form {
        ThreeAngle t;
        public Form1()
        {
           InitializeComponent();
           t = new ThreeAngle(3, 4);  // Создание объекта конкретного
класса
        }
        private void button1_Click(object sender, EventArgs e)
        {
           try
           {
              t.A =   Double.Parse(textBox1.Text);
```

```
        t.B =   Double.Parse(textBox2.Text);
        label4.Text = t.C.ToString();
        label5.Text = t.getArea(!checkBox1.Checked).ToString();
        label6.Text = t.ToString();
    }
    catch
    {
        MessageBox.Show("Ошибка при вводе !");
    }
}
```

Простейший рефакторинг

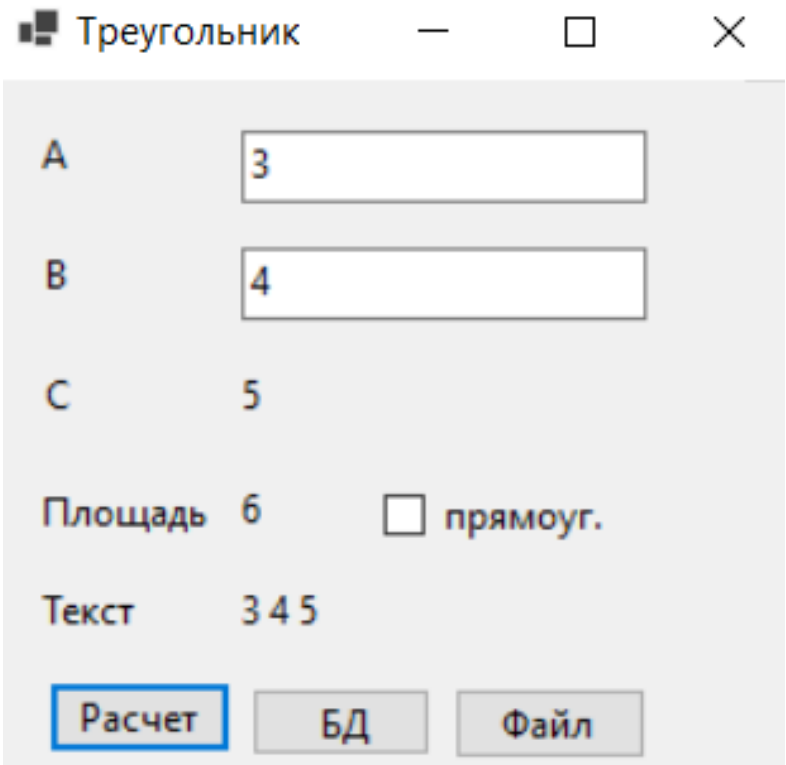
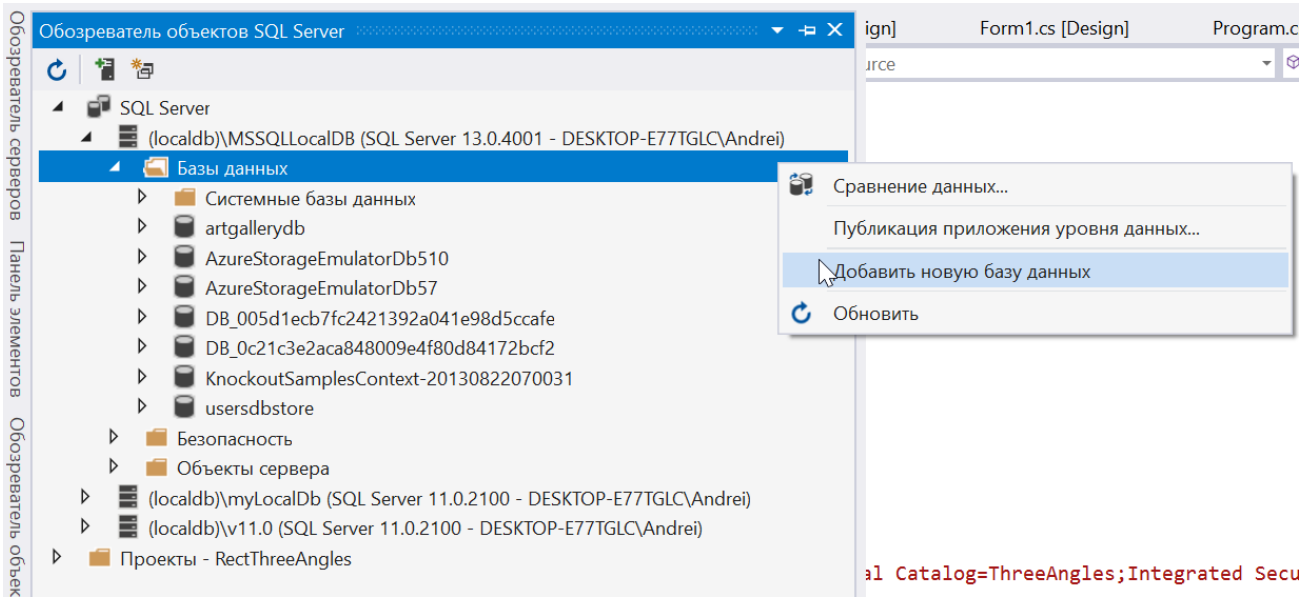
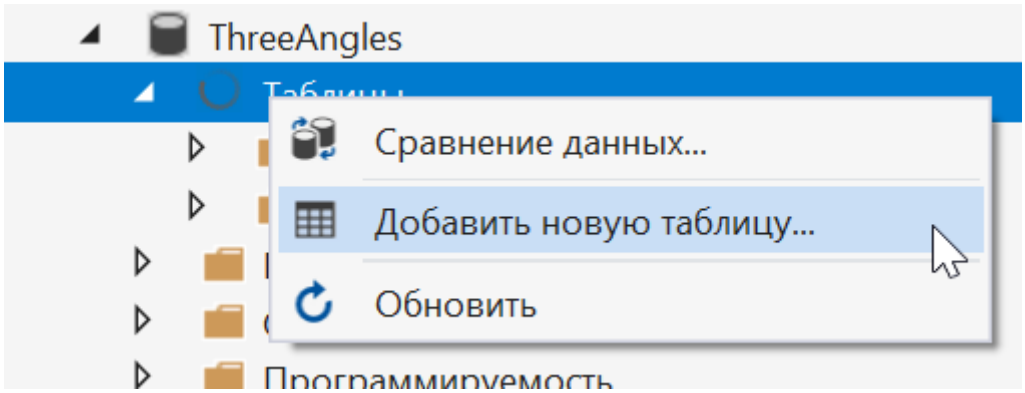


Рисунок 22 - Рефакторинг

Добавляем базу данных (БД LocalDb)





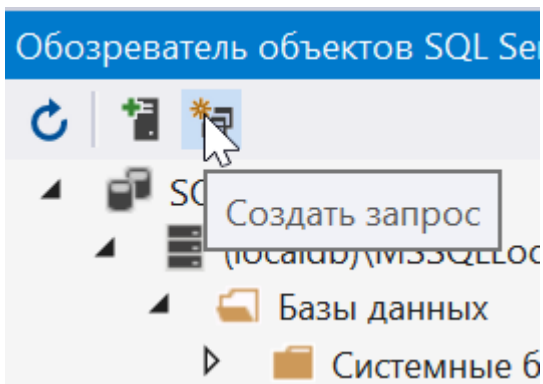
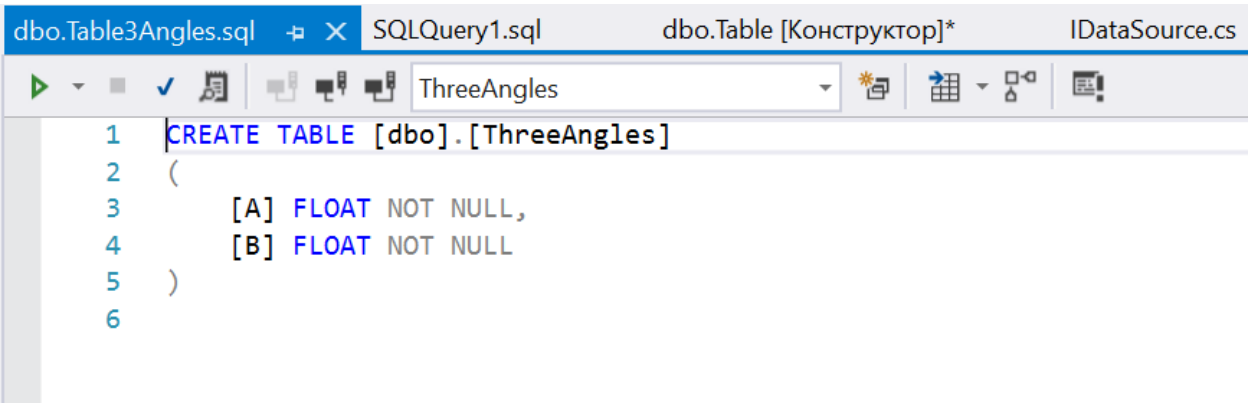


Рисунок 23 - Добавление базы данных



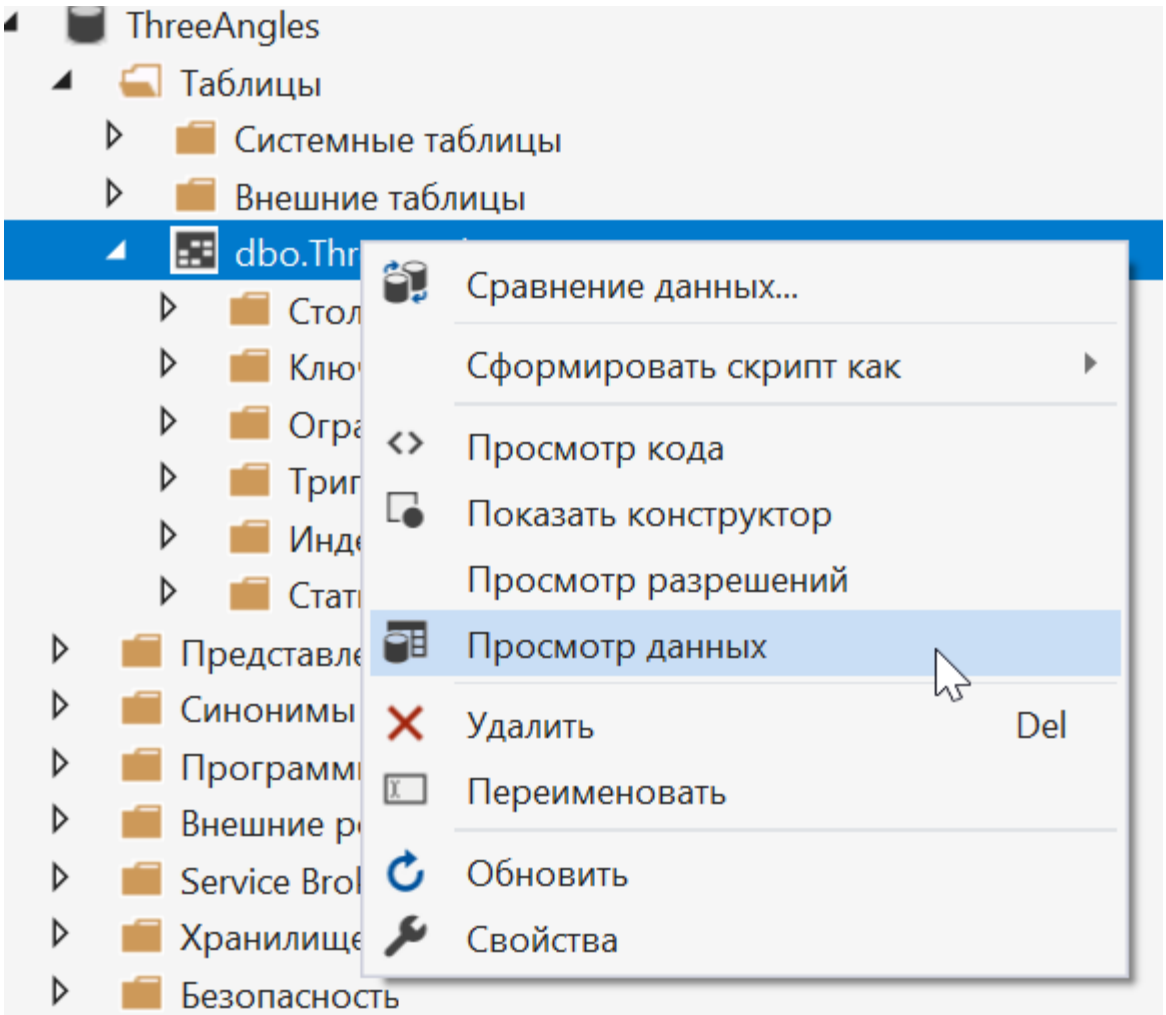


Рисунок 24 – Редактирование таблиц

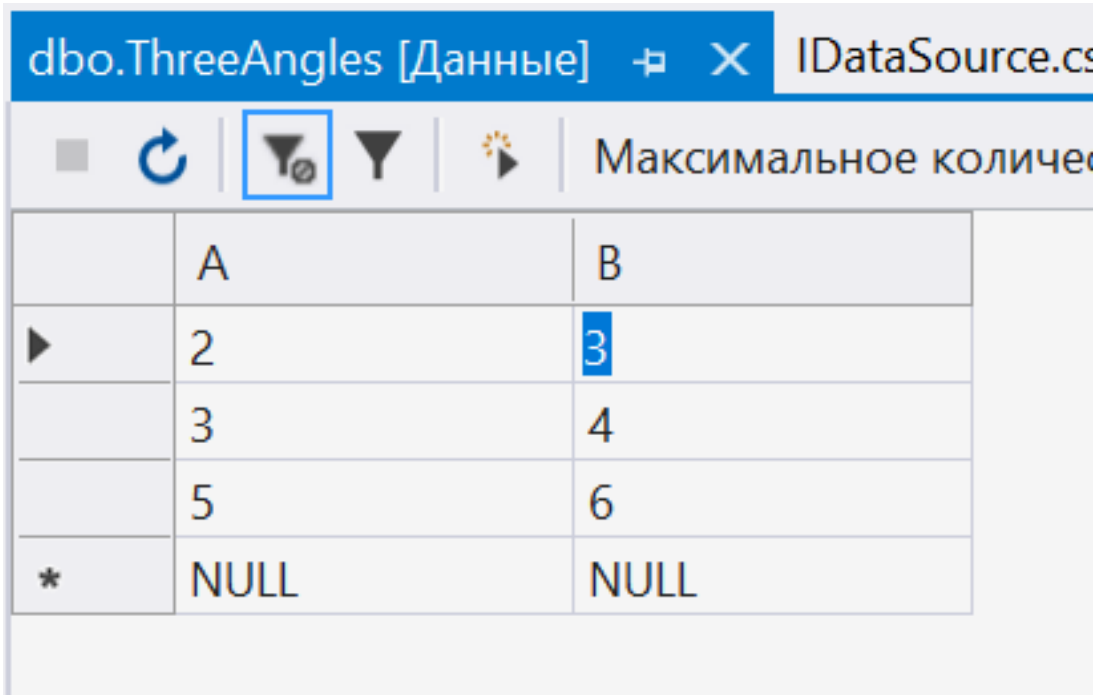


Рисунок 25 – Просмотр базы данных

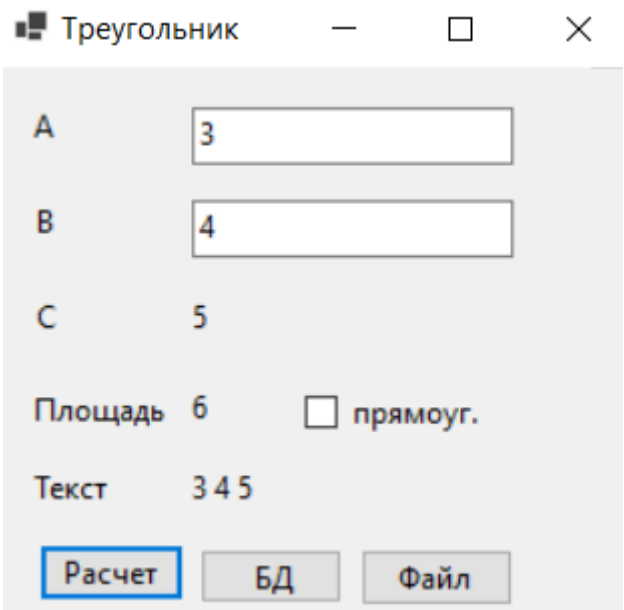


Рисунок 26 – Запуск отображения формы

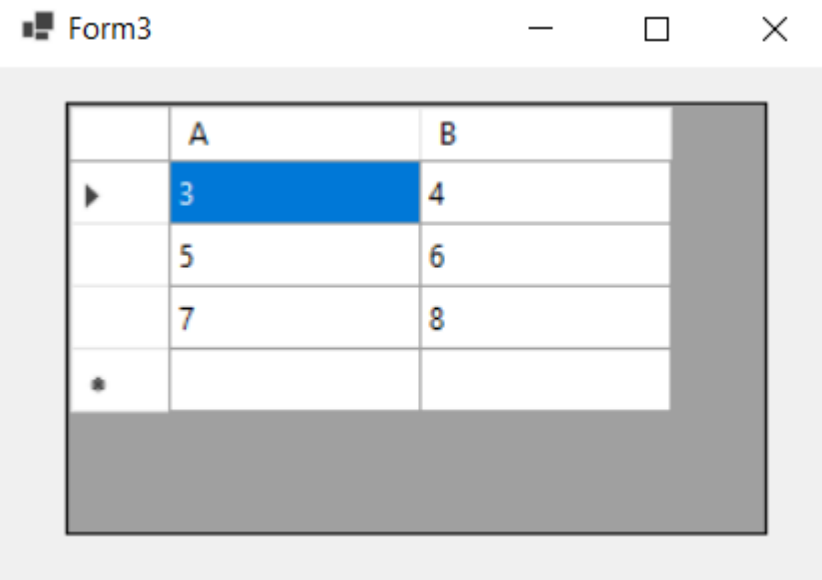


Рисунок 27 – Отображение содержимого базы данных

Код для работы с базой данных представлен ниже.

```
    public List<ThreeAngle> getAB_List()
{
    List<ThreeAngle> lst = new List<ThreeAngle>();
    SqlConnection cnn = new SqlConnection(
"Data Source=(localdb)\\MSSQLLocalDB;Initial Catalog=ThreeAngles;Integrated
Security=True;Connect
Timeout=30;Encrypt=False;TrustServerCertificate=False;ApplicationIntent=ReadWrite;MultiSubne
tFailover=False");
    SqlDataAdapter da = new SqlDataAdapter(
      "select * from ThreeAngles", cnn);
    DataSet ds = new DataSet();
    da.Fill(ds);
    cnn.Close();
    DataTable dt = ds.Tables[0];
    foreach (DataRow dr in dt.Rows)
    {
      lst.Add(new ThreeAngle(Double.Parse(dr[0].ToString()),
        Double.Parse(dr[1].ToString())));
    }
    return lst;
```

```
}
```

Работа с источником данных в формах

Код для работы с формой показан ниже.

```
Form3 frm = new Form3(new DBDS());
frm.ShowDialog();
textBox1.Text = frm.A.ToString();
textBox2.Text = frm.B.ToString();

foreach (ThreeAngle t in ds.getAB_List())
    {
        try
        {
           string[] row = new string[] {
                                              t.A.ToString(), t.B.ToString()};
           dataGridView1.Rows.Add(row);
        }
        catch   {}
      }
    }
```

Рефакторинг – выделяем интерфейс для источника данных (применяем принцип DIP)

```
namespace DataSource
{
   public interface IDataSource
   {
      List<ThreeAngle> getAB_List();
   }
}
```

Добавляем загрузку из файла

Код для загрузки из файла.

```
public class FileDS : IDataSource
```

```
{
    #region IDataSource Members
    public List<ThreeAngle> getAB_List()
    {
        List<ThreeAngle> lst = new List<ThreeAngle>();
        StreamReader rdr = new StreamReader("in.csv");
        String all = rdr.ReadToEnd();
        String[] arr = all.Split(new string[] { "\r\n" },
            StringSplitOptions.RemoveEmptyEntries);
        foreach (string s in arr)
        {
            String[] arr1 = s.Split(new string[] { ";" },
                StringSplitOptions.RemoveEmptyEntries);
            double a = Double.Parse(arr1[0]);
            double b = Double.Parse(arr1[1]);
            lst.Add(new ThreeAngle(a, b));
        }
        return lst;
    }
    #endregion
}
```

Рефакторинг – реализуем интерфейс для источника данных – ORM Entity Framework

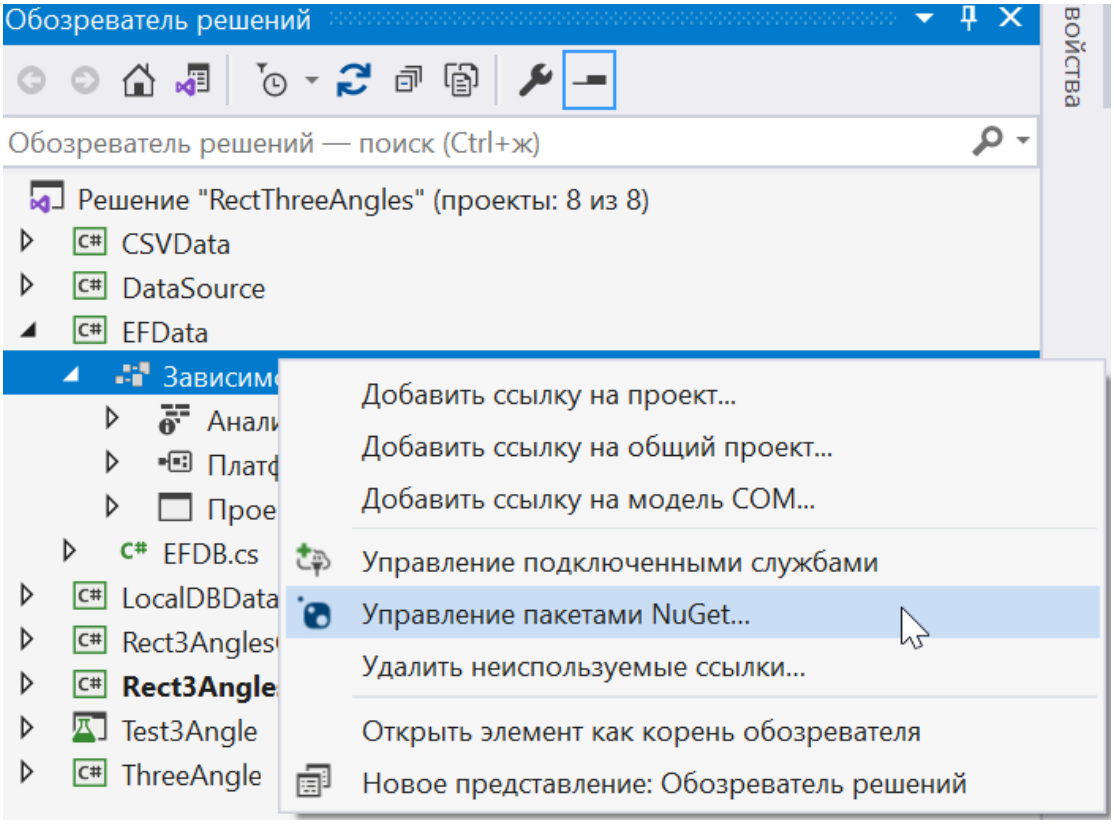


Рисунок 28 – Добавление интерфейса для доступа к абстрактному источнику данных

При установке выберите версию, соответствующую .NET Core SDK проекта (в данном случае 5.0)

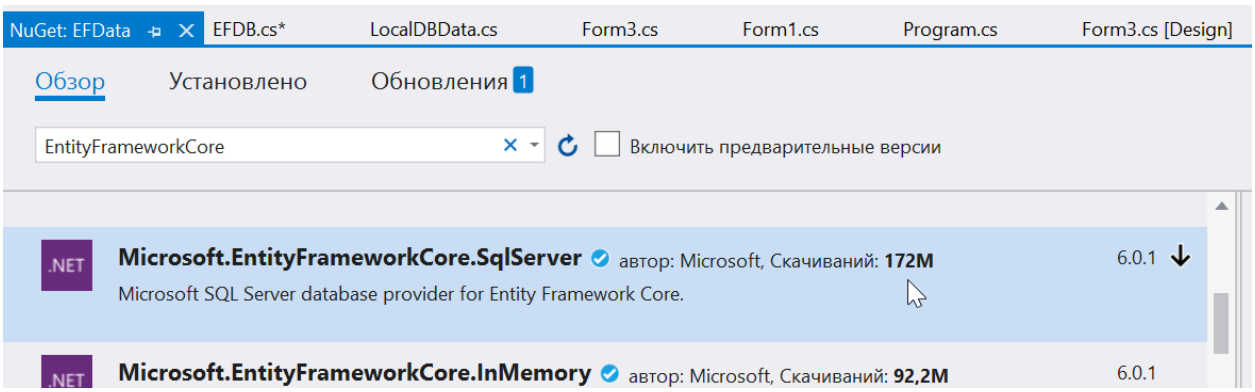


Рисунок 29 – Добавление программного пакета NuGet

Код для работы с ORM представлен ниже.

```
public class EFDB : DbContext, IDataSource {
    DbSet<ThreeAngle> ThreeAngles { get; set; }
    public EFDB()  {
       Database.EnsureCreated();
    }
    protected override void OnConfiguring(DbContextOptionsBuilder
        optionsBuilder)  {

optionsBuilder.UseSqlServer("Server=(localdb)\\MSSQLLocalDB;
      Database=threeangles_db;Trusted_Connection=True;");
    }
    protected override void OnModelCreating(ModelBuilder
modelBuilder) {
       modelBuilder.Entity<ThreeAngle>().HasNoKey();
    }

    public List<ThreeAngle> getAB_List()  {
       return ThreeAngles.ToList<ThreeAngle>();
    }
}
```

ORM EF – Code First БД threeangles_db и таблица ThreeAngles созданы автоматически.

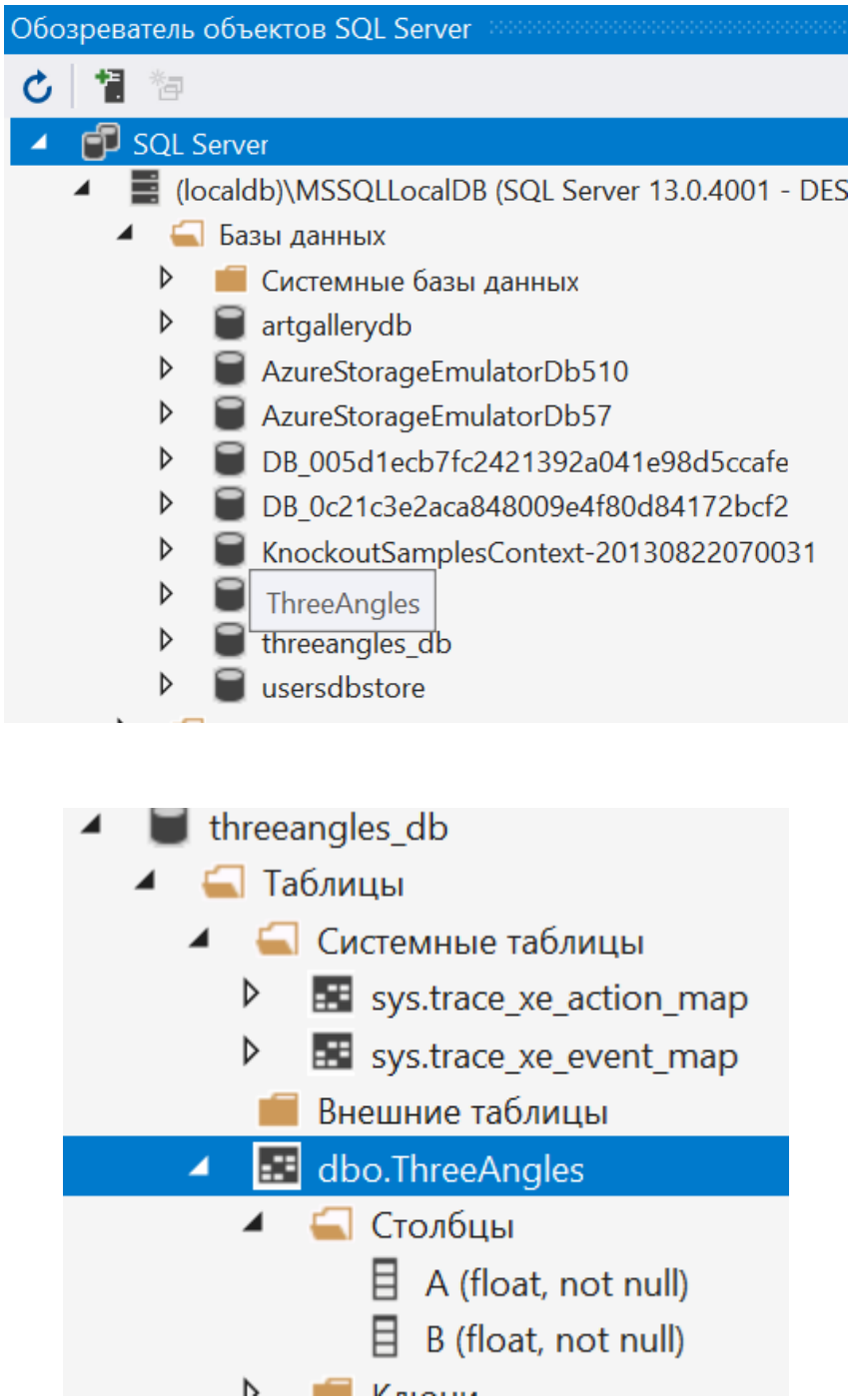


Рисунок 30 – Обзор автоматически созданной базы данных

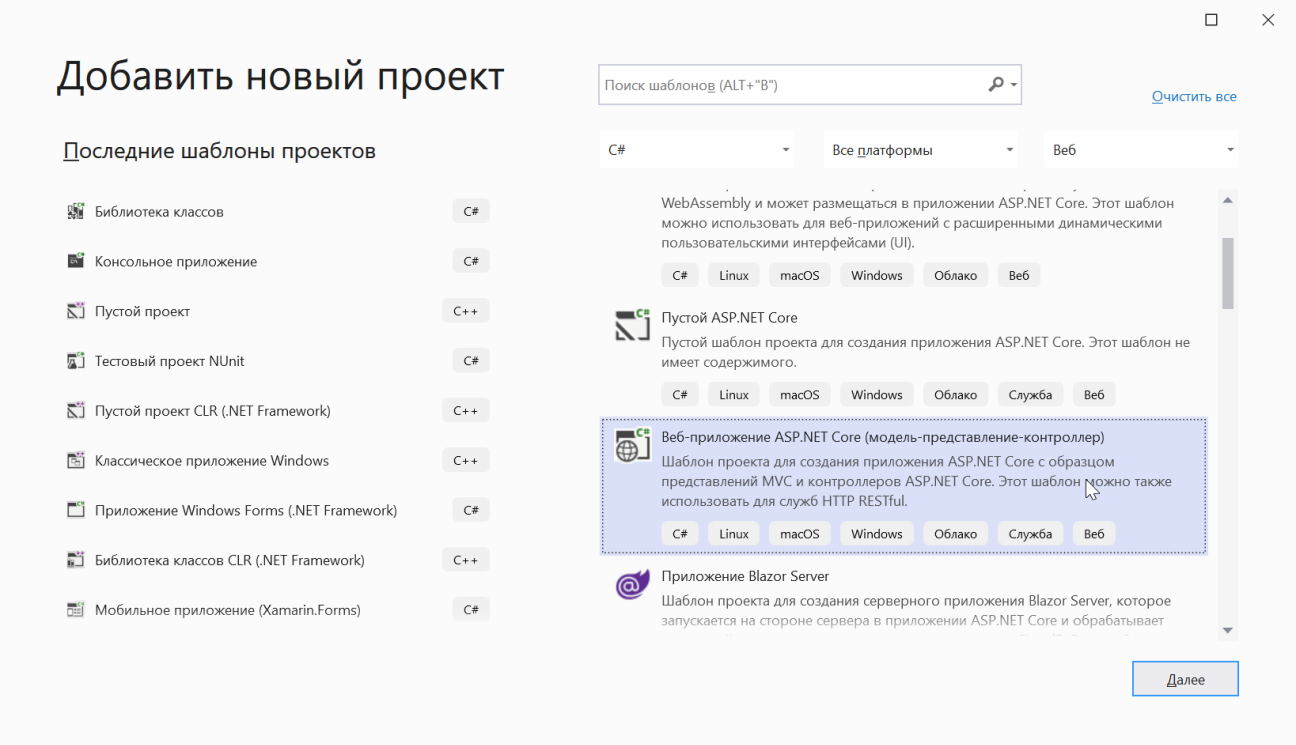


Рисунок 31 – Новый проект Web приложения

Код Web приложения представлен ниже.

Контроллер HomeController.cs

```
public class HomeController : Controller
 {
     private readonly ILogger<HomeController> _logger;
      private IDataSource db;

     public HomeController(ILogger<HomeController> logger) {
        _logger = logger;
        db = new EFDB();
     }

     public IActionResult Index()
     {
        ViewBag.A = "3";
```

```
            ViewBag.B = "4";
            ViewBag.C = "0";
            ViewBag.Area = "0";
            ViewBag.Per = "0";
            return View(db.getAB_List().ToList());
        }

public IActionResult Privacy() {
            return View();
 }
 [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None,
NoStore = true)]
 public IActionResult Error() {
            return View(new ErrorViewModel { RequestId = Activity.Current?.Id
??
              HttpContext.TraceIdentifier });
}

[HttpPost]
public IActionResult Index(string A, string B) {
      ThreeAngle t = new ThreeAngle(Double.Parse(A), Double.Parse(B));

      ViewBag.A = A; ViewBag.B = B; ViewBag.C = t.C.ToString();
      ViewBag.Area = t.getArea(false).ToString();
      ViewBag.Per = t.Per.ToString();
      return View(db.getAB_List().ToList());
}
```

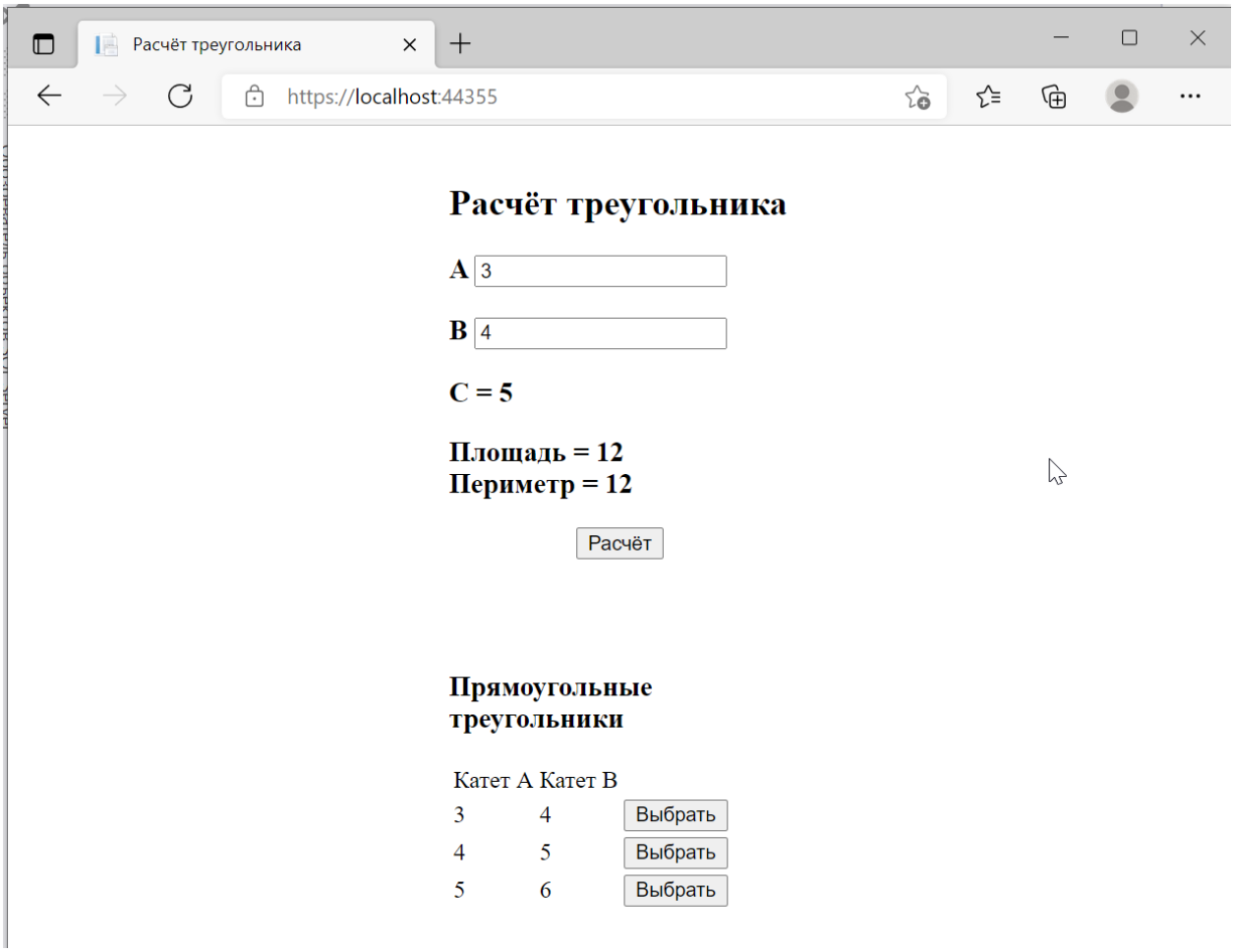


Рисунок 32 - Форма (view Index.cshtml)

Разметка Core MVC – View (Index.cshtml – Razor)

```
@{
    ViewData["Title"] = "Home Page";
    Layout = null;
}
<style>
    .center { width: 230px;  padding: 10px;  margin: auto;  }
</style>
<!doctype html>
<html>
<head>
    <title>Расчёт треугольника</title>
```

```
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width">
    <script type="text/javascript">
        function setAB(a, b) {
            let form = document.forms.frm
            frm.elements.A.value = a
            frm.elements.B.value = b
        }
    </script>
</head>

<body>
    <div class="center">
        <form method="post" name="frm">
            <h2>Расчёт треугольника</h2>
            <h3 />
            A <input type="text" name="A" value=@ViewBag.A />
            <h3 />
            B <input type="text" name="B" value=@ViewBag.B />
            <h3 />
            C = <label name=lable4>@ViewBag.C</label>
            <h3 />
            Площадь =  <label name=lable5>@ViewBag.Area</label><br>
            Периметр =  <label name=lable6>@ViewBag.Per</label>
            <p style="text-align: center"><button type="submit"
                name="action">Расчёт</button></p>
        </form>
    </div>
    <br>
<div class="center">
```

```
        <h3>Прямоугольные треугольники</h3>
        <table>
            <tr>
                <td>Катет A</td>
                <td>Катет B</td>
                <td></td>
            </tr>
            @foreach (var tri in Model)
            {
                <tr>
                    <td>@tri.A</td>
                    <td>@tri.B</td>
                    <td><input type="button" onclick="setAB(@tri.A, @tri.B);"
                        value="Выбрать" /></td>
                </tr>
            }
        </table>
    </div>
</body>
</html>
```

Строим диаграмму классов (ObjectiF 7.1)

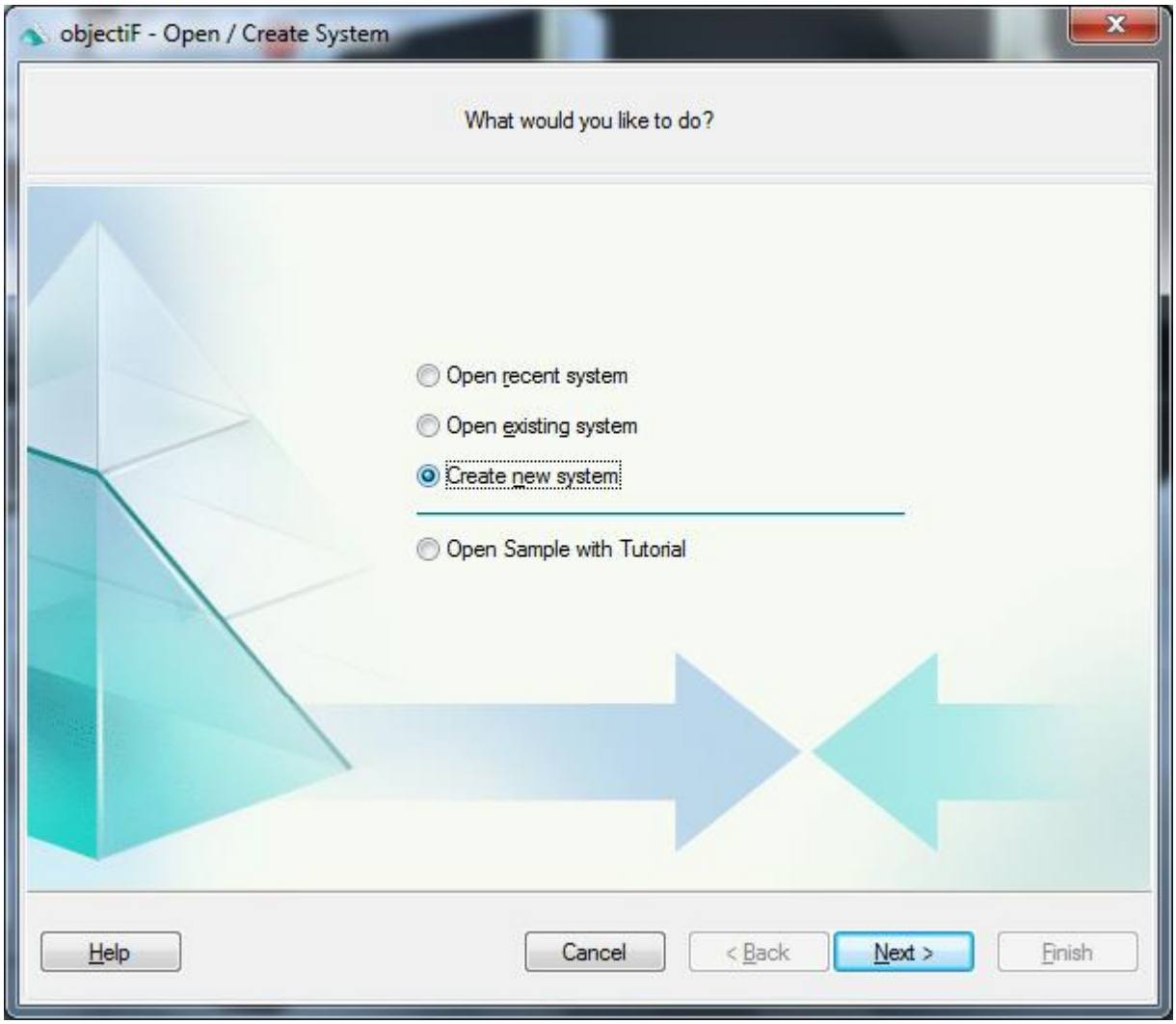


Рисунок 33 – Построение диаграммы классов

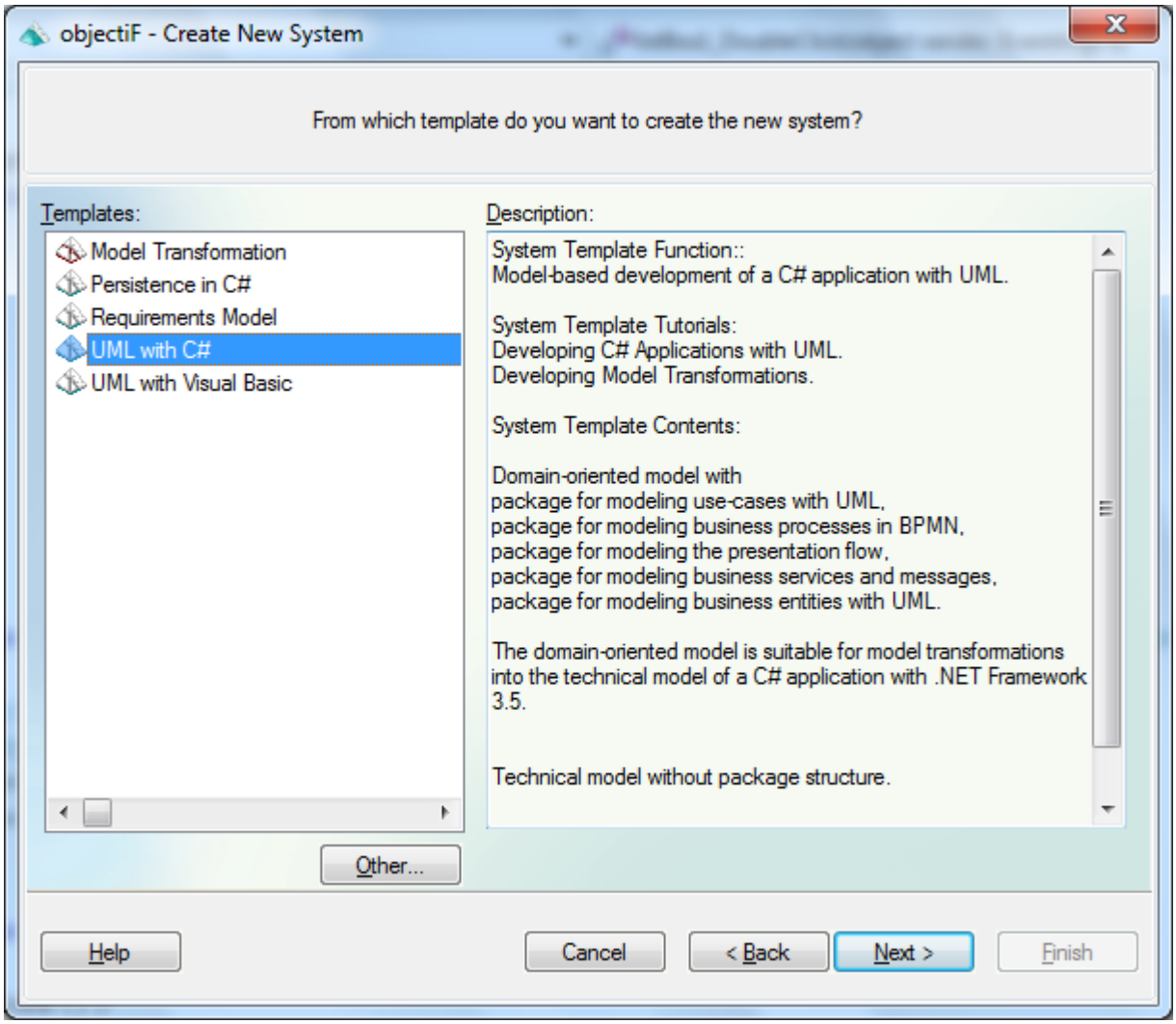


Рисунок 34 – Выбор построениея UML на основе кода C#

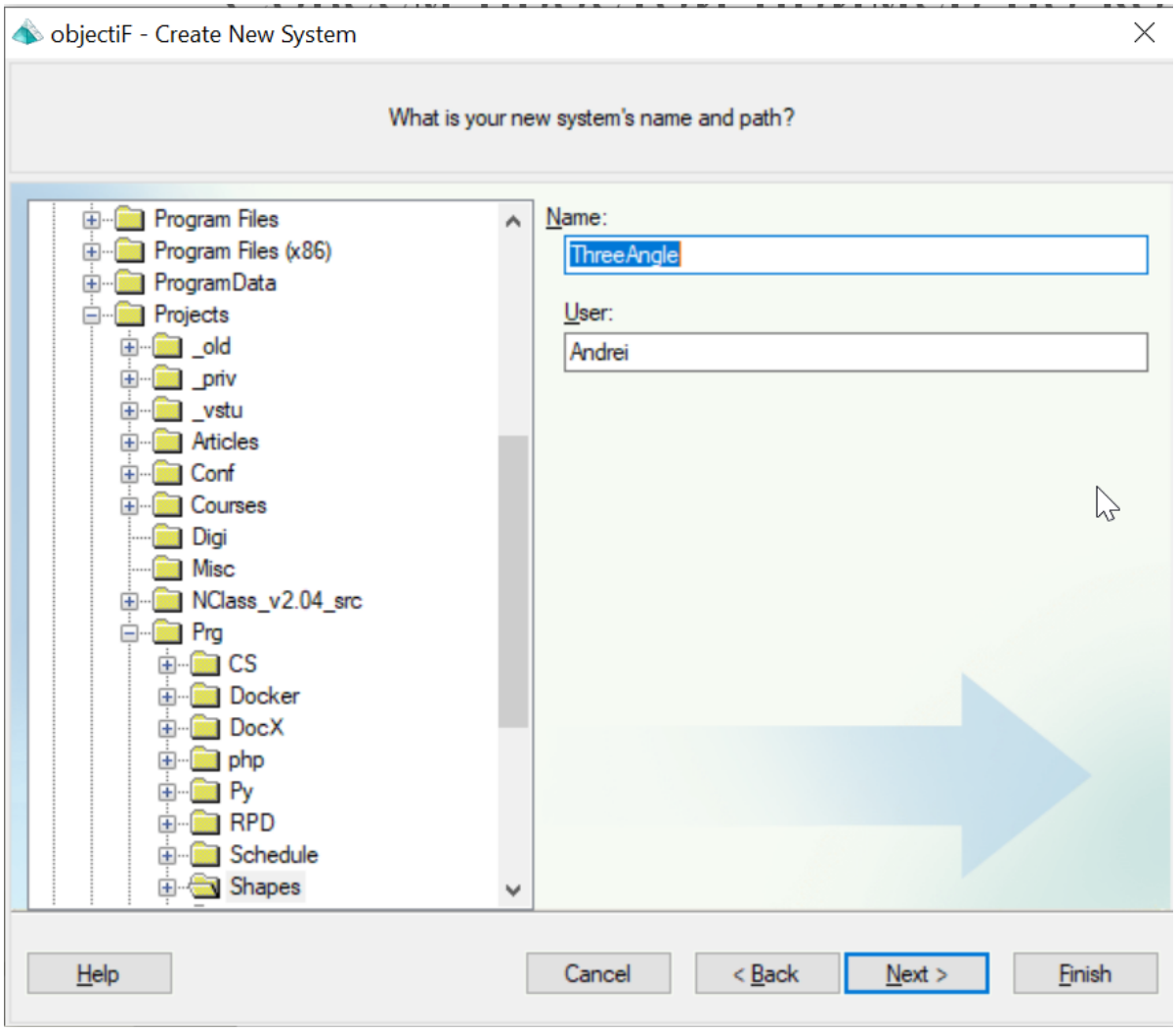


Рисунок 35 – Выбор выходного файла

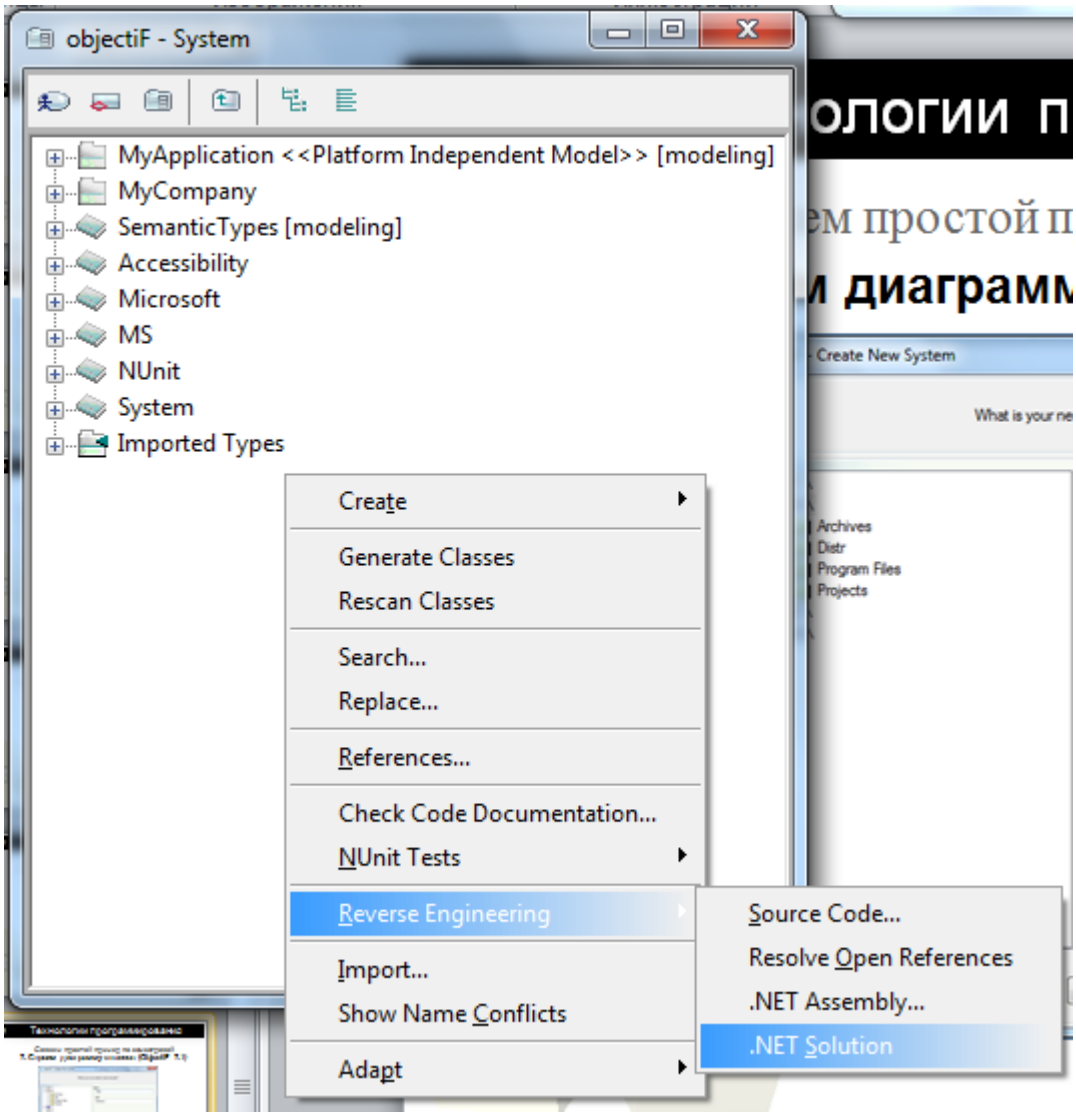


Рисунок 36 – Обратный инжиниринг кода

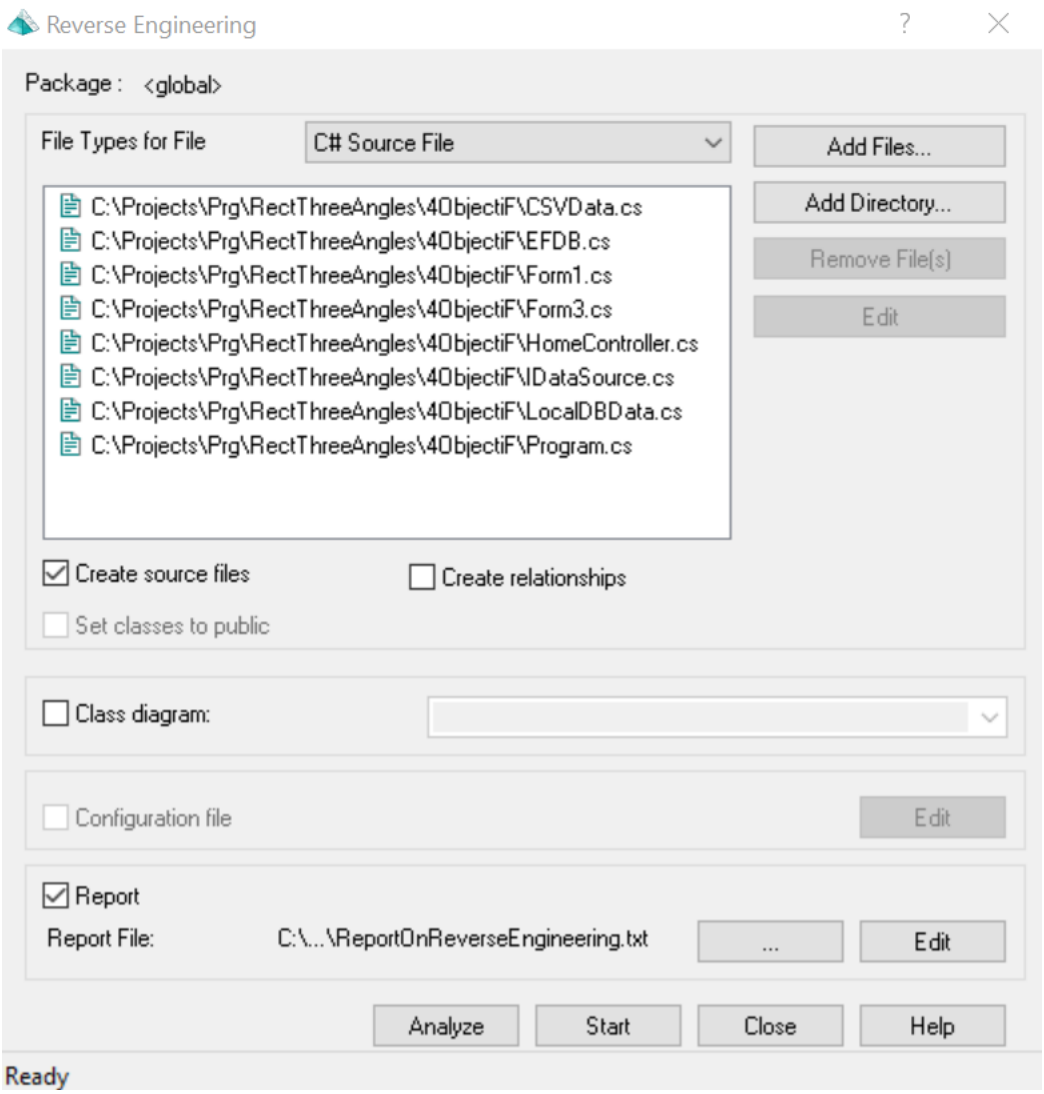


Рисунок 37 – Выбор файла для обратного инжиниринга

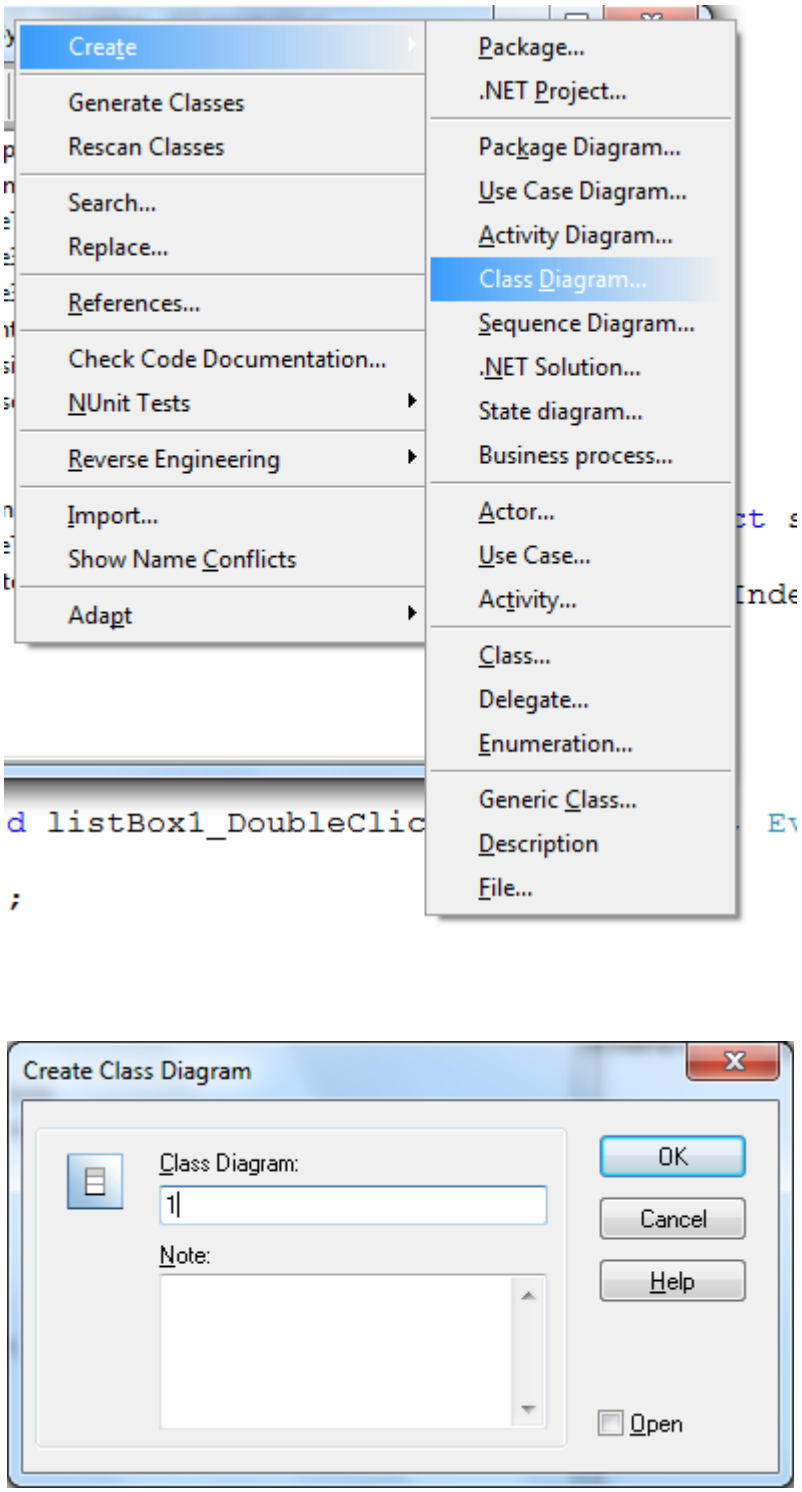


Рисунок 38 – Выбор типа диаграммы «Class Diagram»

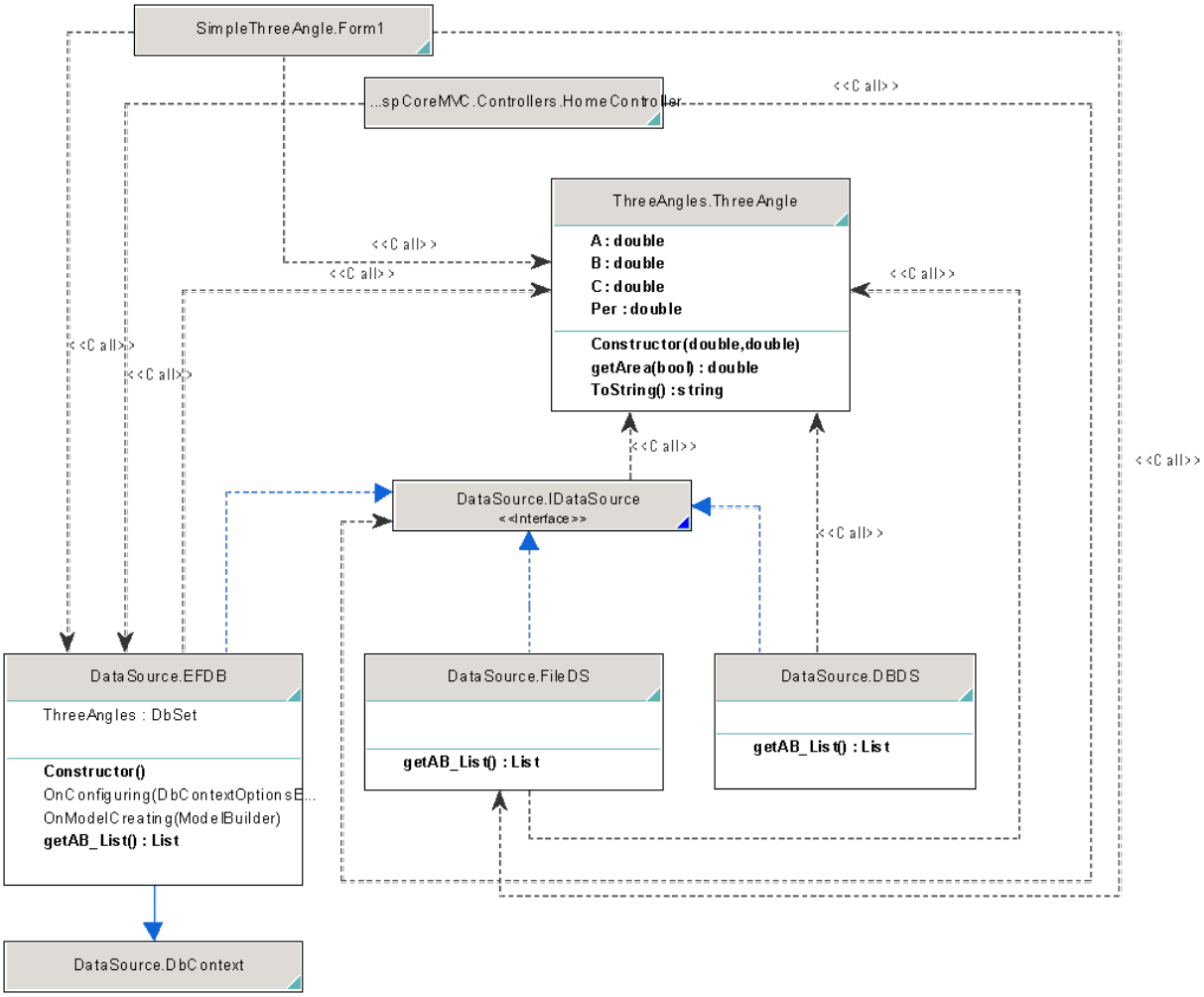


Рисунок 39 – Отображение результата построения диаграммы классов

Строим диаграмму пакетов (ObjectiF 7.1)

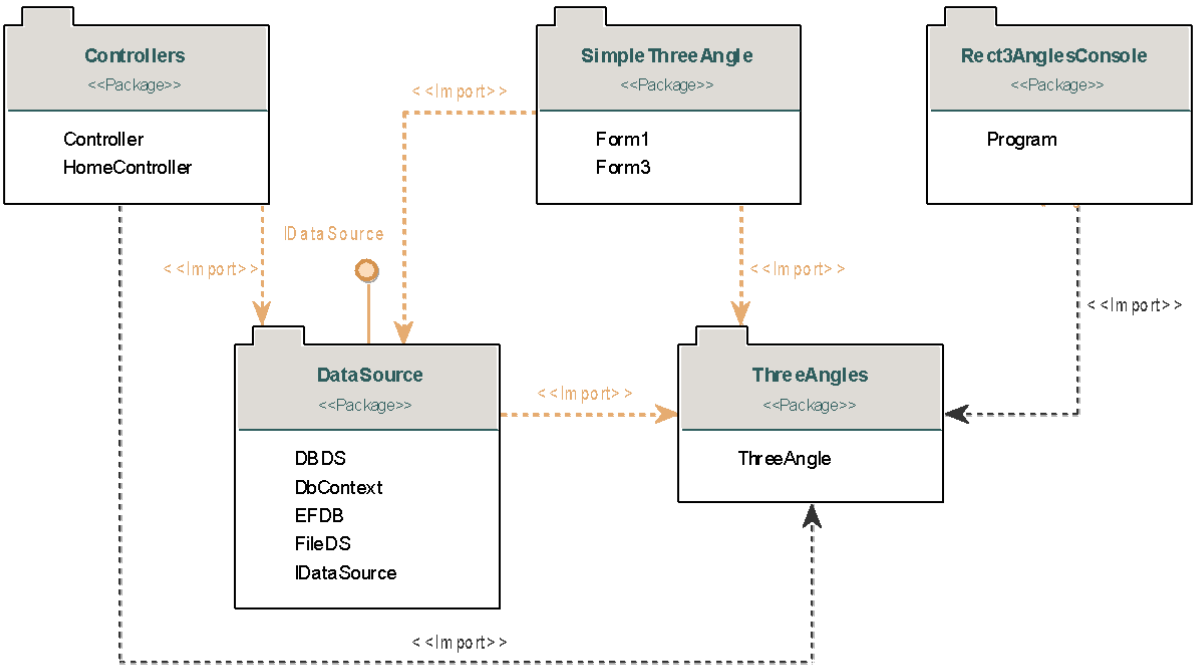


Рисунок 40 – Отображение результата построения диаграммы пакетов

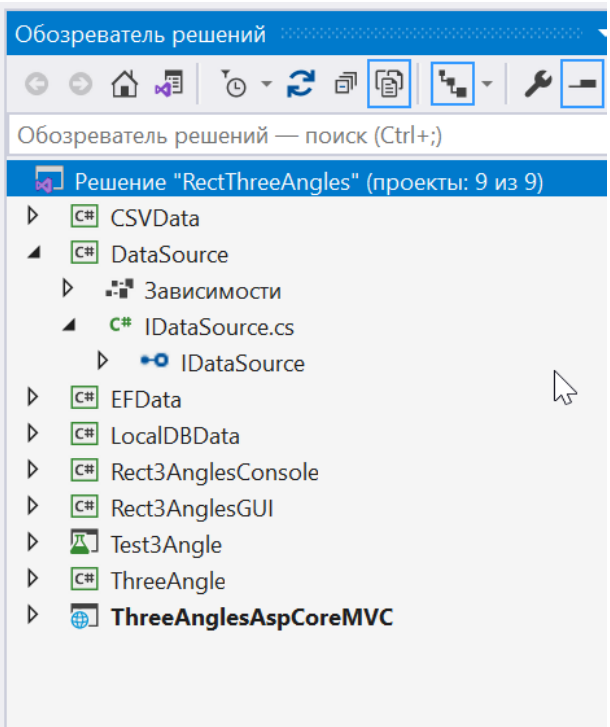


Рисунок 41 – Отображение конечного решения Visual Studio

Рекомендуемая литература

1. Черников, В. Разработка мобильных приложений на C# для iOS и Android : учебное пособие / В. Черников. — Москва : ДМК Пресс, 2020. — 188 с.
2. Умрихин, Е. Д. Основы разработки iOS-приложений на C# с помощью Xamarin : учебное пособие для вузов / Е. Д. Умрихин. — Санкт-Петербург : Лань, 2021. — 384 с.
3. Джанарсанам, С. Практическое руководство по разработке чат-интерфейсов : руководство / С. Джанарсанам. — Москва : ДМК Пресс, 2018. — 340 с.
4. Васильев, Н. П. Введение в гибридные технологии разработки мобильных приложений : учебное пособие для вузов / Н. П. Васильев, А. М. Заяц. — 2-е изд., стер. — Санкт-Петербург : Лань, 2021. — 160 с.
5. Архитектурные решения информационных систем : учебник / А. И. Водяхо, Л. С. Выговский, В. А. Дубенецкий, В. В. Цехановский. — 2-е изд., перераб. — Санкт-Петербург : Лань, 2021.
6. Нестеров, С. А. Основы информационной безопасности : учебник для вузов / С. А. Нестеров. — Санкт-Петербург : Лань, 2021. — 324 с.

Учебное издание

Андрей Евгеньевич Андреев
Михаил Андреевич Кузнецов

Перспективные Web-технологии

Учебное пособие

Волгоградский государственный технический университет.
400005, г. Волгоград, просп. В. И. Ленина, 28, корп. 1.