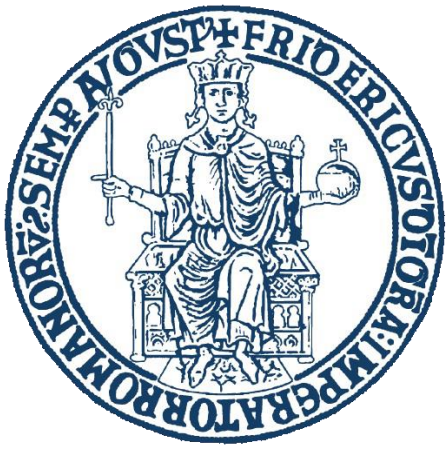


UNIVERSITÀ DEGLI STUDI
DI NAPOLI FEDERICO II

INGEGNERIA DEL SOFTWARE II

Android Application

By Guillermo Arce

0001/16313

Summary

The scope of the following project is to develop an Android application and test it using testing frameworks.

For that purpose, Android Studio and Java coding language are going to be used.

Table of Contents

Introduction

This application will allow the user to have a general overview of his expenses and a quick, easy and instant access to a breakdown of those expenses by categories. The application will help to manage daily expenses and to improve spending control.

Being very practical and intuitive, this application tries to help users to save money and time. People want to control how much they spend but they don't want to waste time with tedious or difficult accounting procedures.

"Time is gold!"

Objective

The application is designed to be very handy in such a way that it can be used every day. In addition, there is an special attention on providing a very simple use so that all type of public can interact with it without difficulties. The basis of the project relies on three main characteristics:

User-Friendly, Fast and Simple

User friendly because it has an intuitive user interface and is easy to use. The user should feel comfortable navigating through the app.

Fast because it is a daily use application. It must be light and fast to avoid wasting users' time. For that purpose, it doesn't stablish any communication with the Internet; it acts local in order not to depend on the network.

Simple because people don't want to spend time learning and dealing with all the functionalities of the app, they just want to achieve their goal: keep track of the expenses.

In order to achieve this, some components like the *navigation* have been used. Also, the functionalities provided were also selected in such a way that the user can do whatever necessary: add expenses, check current situation or delete expenses.

Finally, as a key choice, the first view of the application should be the adding expense process. The 90% of the times that the user is going to enter the application is going to be for adding an expense. For that reason, it is on the foreground of the application.

Design

For the development of this application, quite a lot of Google lately advices have been followed. That is, the application is a **single Activity** application and uses Android Jetpack components. This way of developing relies on the use of a single Activity and instead use Fragments as a reusable component handled with the FragmentManager.

However, the core of the application is based on the **Android Architecture Components**. Android architecture components are a collection of libraries that help us design robust, testable, and maintainable apps. Starting with classes for managing the UI component lifecycle and handling data persistence.

In order to use these components and build a robust architecture, the *Guide to App architecture* of Android official documentation has been key. As recommended on this guide, the project follows the two main architectural principles:

Separation of concerns: As the most important concept, it is based on the need of dividing the application into different modules so that each of them is "independent" from the rest. Doing this, the application becomes easier maintainable and easier to understand.

In addition, it should be known that the Activity or Fragment lifecycle doesn't depend completely on the developer. The OS can destroy them at any time based on user interactions or because of system conditions like low memory. To provide a satisfactory user experience and a more manageable app maintenance experience, it's best to minimize the dependency on them.

Drive UI from a model: As the title suggests, the idea is to separate the data handling with the UI. For that, model components are used. They're independent from the View objects and app components in the app, so they're unaffected by the app's lifecycle and the associated concerns. In such a way, we can have a consistent and independent mechanism to provide or extract data from the UI whenever necessary.

The easiest way to understand this architecture is to see *Figure 1*.

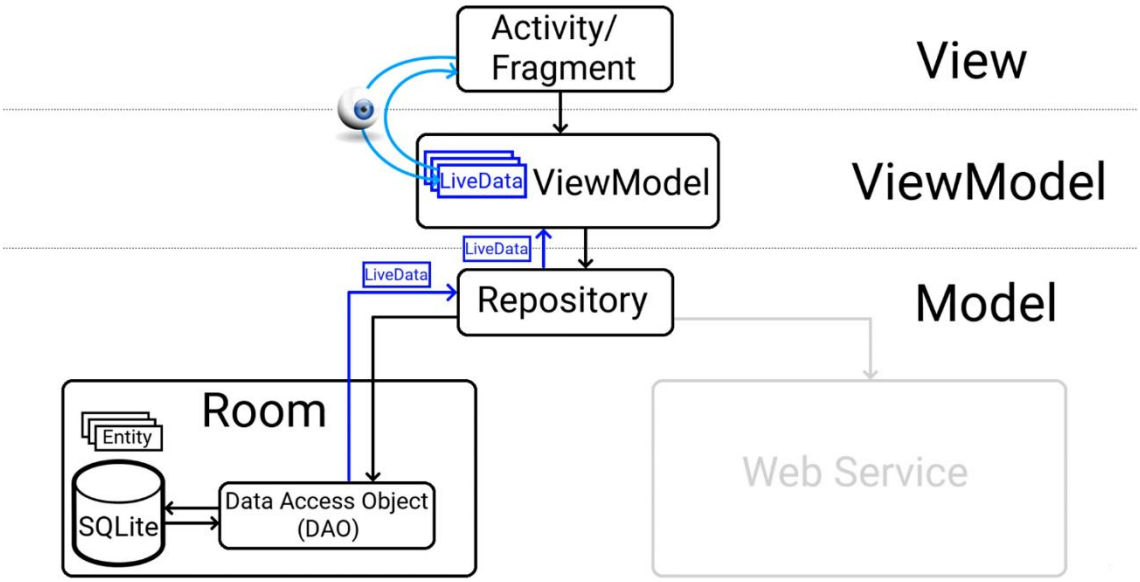


Figure 1

This is the representation of the architecture that our project is following. Nevertheless, the LiveData component (introduced on the Android Jetpack components) is not used because it wasn't needed; the input and output of data were on different fragments than its representation so it wasn't necessary to listen on data changes, as they will be loaded when loading the fragment.

As seen on *Figure1*, the components used are the following:

View:

- Activity/Fragments: As stated at the beginning, there is a single activity and various fragments. They act as the intermediary with the Android OS and as the View.

  In this project, the activity is *MainActivity* and the fragments are the following:

  - *HomeFragment*: Deals with home screen and it is the start point of the navigation component (explained in a following chapter). Thanks to the navigation, three more fragments can be deployed from it: *I_ExpenseTypeFragment*, *II_PaymentMethodFragment* and *III_ConfirmExpenseFragment*.

  - *AnalysisFragment*: Provides a tab layout for the representation of the analysis of the expenses, which is fulfilled by two more fragments: *AnalysisMonthFragment* and *AnalysisYearFragment*. Each one dealing with a different tab on the *AnalysisFragment* parent.

  - *SettingsFragment*: Deals with the settings section.

  For the user interaction among these fragments, a navigation drawer is implemented.

ViewModel:

- The ViewModel object provides the data for a specific UI component, such as a fragment or activity, and contains data-handling business logic to communicate with the model. In the project, it is represented by *ExpenseViewModel* class.

Model:

- Repository: It is also used a Repository class that works as another abstraction layer between the ViewModel and the underlying data model. They provide a clean API so that the rest of the app can retrieve this data easily.
  They can be considered to be mediators between different data sources, such as persistent models (our case) or web services. It makes reference to *ExpenseRepository* class on the project.
- Room: For the backend of the app it is used the Room Persistence Library, which works as a wrapper around SQLite and helps us reduce the difficulty of the code by making extensive use of Annotations.
  Instead of creating an SQLiteOpenHelper, Java classes are turned into "entities" (Expense class) to create tables, and use "Data Access Objects" (DAO) to query these tables and make operations on them.
  Room also provides compile time verification for SQL statements, so we run into fewer runtime exceptions, caused by typos and invalid queries.
  In the application, the database has only one table (*expense_table*) which refers to the *Expense* entity class.

Together, this whole structure constitutes an **MVVM** (Model-View-ViewModel) architecture, which follows the single responsibility and separation of concerns principles.

Components/Libraries

To develop the application in the most efficient and professional way, some libraries and components have been used. They provide method and functionalities that would be very difficult to implement from scratch.

The first and most important is the **Android Navigation Components**.

As mentioned on the objectives of the application, a user-friendly behaviour was one of the main requirements. For that, a very interesting option is an intuitive user navigation through the process of adding an expense.

As a recent release on the Android Jetpack components, Android Navigation Components helps the developer to implement navigation. Navigation refers to the interactions that allow users to navigate across, into, and back out from the different pieces of content within the app.

The Navigation component consists of three key parts:

- Navigation graph: XML resource that contains all navigation-related information in one centralized location. In the project, it is *nav_graph* xml.

  If using an updated version of Android Studio, the navigation graph graphical representation can be seen, as in *Figure 2.*

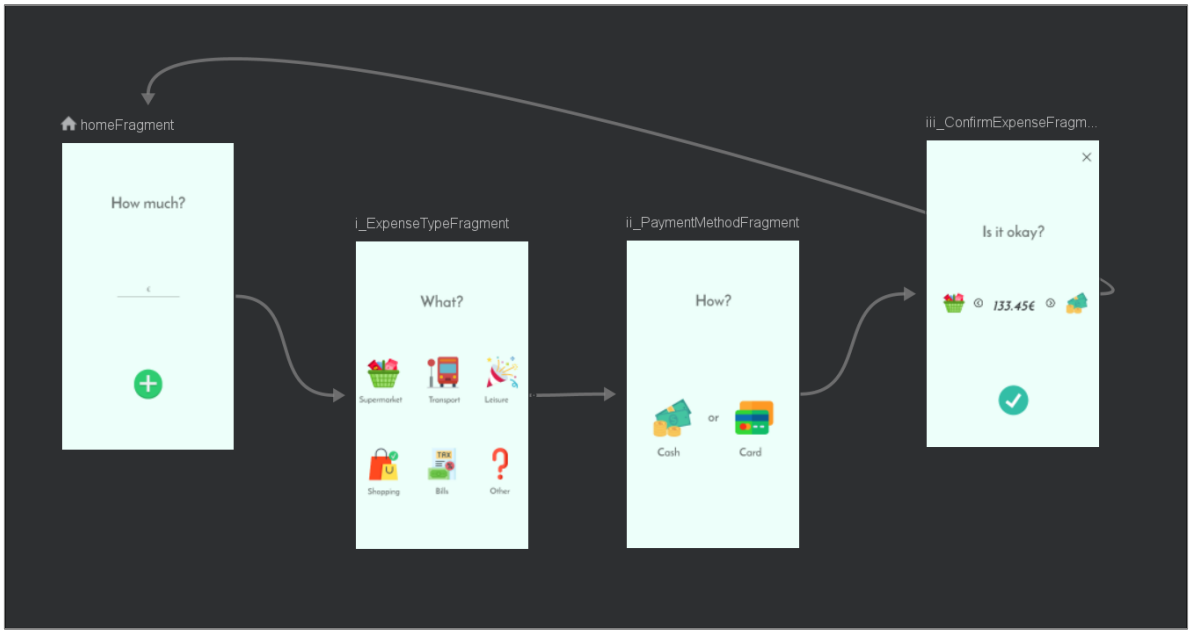


Figure 2

- NavHost: An empty container that displays destinations from your navigation graph. In this project, it forms part of the *activity_main* xml.
- NavController: An object that manages app navigation within a NavHost. The NavController orchestrates the swapping of destination content in the NavHost as users move throughout your app.

In the application, supporting the navigation, a *SharedViewModel* class has been developed in order to share the data among the fragments. In that way, data introduced by the user is continuously accessible from the fragments.

Following with the components/libraries used, as an important part of the application there is the analysis section. For a simple and clear analysis, the use of graphics is almost compulsory. For that purpose, the **MPAndroid Chart** library has been used.

MPAndroid Chart is a free Android chart / graph view library which eases the representation of pie, line or bars, for example, charts. It also supports scaling, dragging and animations; it is available via GitHub.

Thanks to this library, some nice charts are displayed on the analysis section in order to improve the user interaction and make it easier to check. For example, the one in charge of displaying the whole view of the expenses through the month (selected on a spinner) is shown in *Figure 3*.

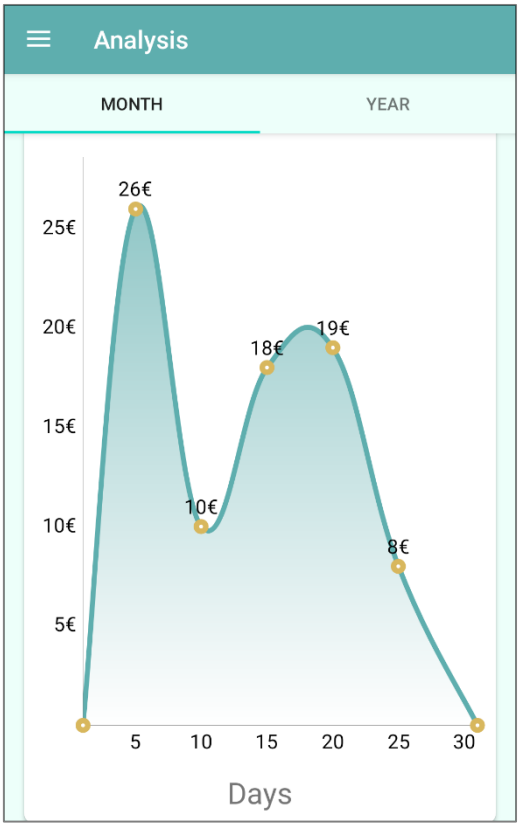


Figure 3

In order to represent the expenses on the charts, some classes to handle the data retrieved from the database had to be created. The operations like sum or grouping of expenses are done with queries, not programmatically. In such a way the performance is improved, because the operations based on the data done on the database are faster. If not, some loops organizing the data would be needed and it can slow-down the application.

Finally, as the last library, already presented on the architecture, we have **Room**. As part of the Android Jetpack components, the Room persistence library provides an abstraction layer over SQLite to allow for more robust database access while harnessing the full power of SQLite.

For the use of all these components/libraries, the correspondent dependencies had to be added to Gradle.

Testing

Testing the app is an integral part of the app development process. By running tests against the app consistently, we can verify the app's correctness, functional behaviour, and usability before the release.

In Android, it is specially interesting to test because of its complexity. An Android app is basically an interactive application subjected to: user events (touch events, signals from sensors) and system events (interruptions, broadcast signals). That means that the application is susceptible to these types of events, so it should be prepared.

For that purpose, the application has been tested, both at a unit level (unit tests) and at system level (instrumentation tests).

Unit testing is a level of software testing where individual units/components of a software are tested. The purpose is to validate that each unit of the software performs as intended. A unit is the smallest testable part of any software.

Unit testing applied to Android refers to the test of single components or classes, like an Activity or a regular java class. However, in order to test an Activity on isolation we need an Android environment that simulates the deploy of the Activity on it. To accomplish these dependency relationships either Robolectric or a mocking framework, such as Mockito, can be used.

In the current project, **Robolectric** has been chosen as the unit-testing framework so all the tests will run on a JVM-powered development machine. It simulates the runtime for Android 4.1 (API level 16) or higher and provides community-maintained fakes called **shadows**. This functionality allows us to test code that depends on the framework without needing to use an emulator or mock objects.

Each shadow can modify or extend the behaviour of a corresponding class in the Android OS. When an Android class is instantiated, Robolectric looks for a corresponding shadow class, and if it finds one it creates a shadow object to associate with it.

Robolectric supports the following aspects of the Android platform:

- Component lifecycles
- Event loops
- All resources

In fact, with the 4.0 version of Robolectric, the one used in this project, it is intended to be fully compatible with Android's official testing libraries. In such a way, using it, wouldn't imply a big effort in terms of learning.

An example of Robolectric taken from the project would be the following:

```
activityController = Robolectric.buildActivity(MainActivity.class)
                .create().start().resume();

activity = activityController.get();

Toolbar toolbar = activity.findViewById(R.id.toolbar);
```

```
assertNotNull(toolbar);
```

In this example, the Robolectric *buildActivity* method is called in order to create, start and resume an Activity. As explained before, Robolectric supports components lifecycles, here is the prove. In addition, after the creation of the Activity, the existence of a toolbar is tested.

Robolectric supports this kind of testing (unit testing), however, this is not enough if we want to test the interaction between modules, or the user interaction with the system.

For that purpose, **instrumentation testing**, **Espresso** framework has been chosen.

Instrumented tests are tests that run on physical devices and emulators, and they can take advantage of the Android framework APIs and supporting APIs, such as AndroidX Test. Instrumented tests provide more fidelity than local unit tests, but they run much more slowly. We can run instrumented unit tests on a physical device or emulator, in our case the Nexus 5X API 29.

This form of testing involves significantly slower execution times than local unit tests, however, as said before, they are necessary to test the interaction with the real system.

Espresso allows us to write Junit tests that can be executed through the AndroidJUnitRunner executor who runs the project under test "instrumented" by the test classes. These types of tests are placed on the folder *androidTest* of the project, separated from the unit tests that are on the *test* one.

The main components of Espresso are the following:

- Espresso: Entry point to interactions with views (via *onView()* and *onData()*).
- ViewMatchers: A collection of objects that implement the Matcher interface. You can pass one or more of these to the *onView()* method to locate a view within the current view hierarchy.
  In case of problems locating a view within a view hierarchy, Espresso provides a very handy representation of the view hierarchy on the console which helps understanding and finding the desired view. In addition, it provides all the attributes of each view so that the user can find the desired one.
  Example of a summary of a view hierarchy provided by Espresso:

```
+>DecorView{id=-1, visibility=VISIBLE, width=480, height=800,     has-
focus=true, has-focusable=true,... has-input-connection=false, x=0.0,
y=0.0, child-count=2}
|
+->LinearLayout{id=-1, visibility=VISIBLE, width=480, height=800, has-
focus=true, has-focusable=true, has-window-focus=true, is-
clickable=false, is-enabled=true, is-focused=false, is- ... x=0.0,
y=0.0, child-count=2}
|
+-->ViewStub{id=16909225 ... has-window-focus=true, is-
clickable=false, is-enabled=true, is-focused=false, is-
focusable=false, is-layout-requested=true, is-selected=false, root-is-
layout-requested=false, has-input-connection=false, x=0.0, y=0.0}
|
+-->FrameLayout{id=-1, visibility=VISIBLE, width=480, height=764, has-
focus=true, has-focusable=true, ... is-layout-requested=false, is-
```

```
selected=false, root-is-layout-requested=false, has-input-
connection=false, x=0.0, y=36.0, child-count=1}
|
+--->ActionBarOverlayLayout{id=2131427395, res-
name=decor_content_parent, visibility=VISIBLE, width=480, height=764,
has-focus=true, ... root-is-layout-requested=false, has-input-
connection=false, x=0.0, y=0.0, child-count=2}
```

- ViewActions: A collection of ViewAction objects that can be passed to the ViewInteraction.perform() method, such as *click()*.

- ViewAssertions: A collection of ViewAssertion objects that can be passed the ViewInteraction.*check()* method. Most of the time, you will use the matches assertion, which uses a View matcher to assert the state of the currently selected view.

In order to see it clearly, an example of both ViewActions and ViewAssertions extracted from the project is provided.

First, an example of ViewActions *click()* on a back button of the navigation:

```
//Go back!

ViewInteraction backButton2 = onView(
                        allOf(withContentDescription(
                        activity.getString(R.string.backButton)),
                        isDisplayed()));

backButton2.perform(click());
```

As we can see, first it gets the button through onView and a ViewMatcher on ContentDescription. Then It performs the ViewAction *click()*.

For the ViewAssertion examples the following code has been extracted:

```
//Check title

ViewInteraction textView5 = onView(
                        allOf(withId(R.id.textPaymentMethod),
withText(activity.getString(R.string.payment_method_title)),
                        isDisplayed()));

textView5.check(matches(withText(activity.getString(R.string.payment_m
ethod_title))));
```

In this case, the check is done over a TextView. Its content is checked through *withText*(…).

Another interesting feature used on this project is the **Espresso Test Recorder**. This tool allows the recording of UI tests directly performed by the developer on the device. By recording a test

scenario, it is possible to record our interactions with a device and add assertions to verify UI elements in particular snapshots of your app.

In *Figure 4*, for example, the *Share* option on the navigation drawer is checked on an assertion.

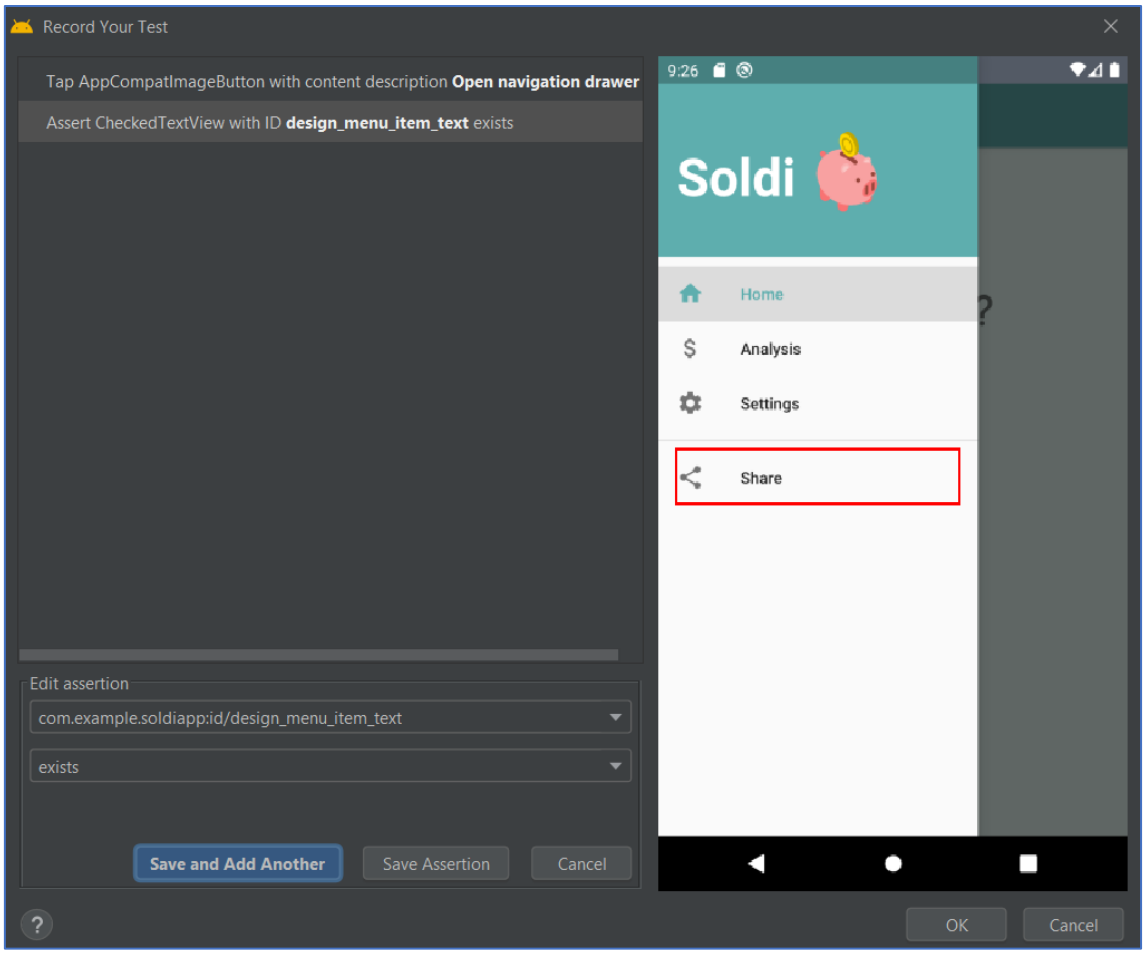


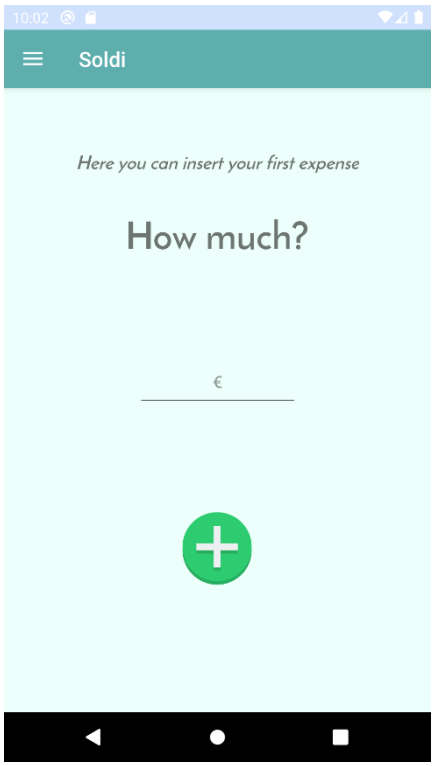
Figure 4

Espresso Test Recorder then takes the saved recording and automatically generates a corresponding UI test that you can run to test your app. It is a very useful tool in order to autogenerate lots of tests and UI routines to test. However, as probably with all autogenerate testing tools, there are always some sort of problems and most of the time tests fail. So it is always advisable to edit the code after it is generated.

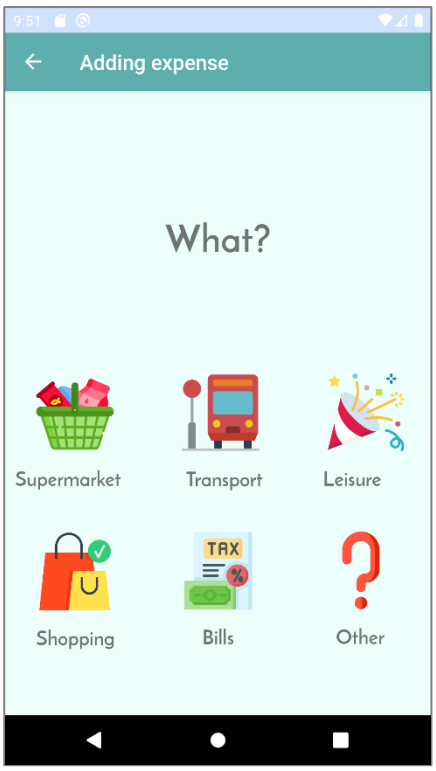
Anyway, it also helps you to increase the test coverage and add extra checks if you edit the test cases, as done in the project.

Application review

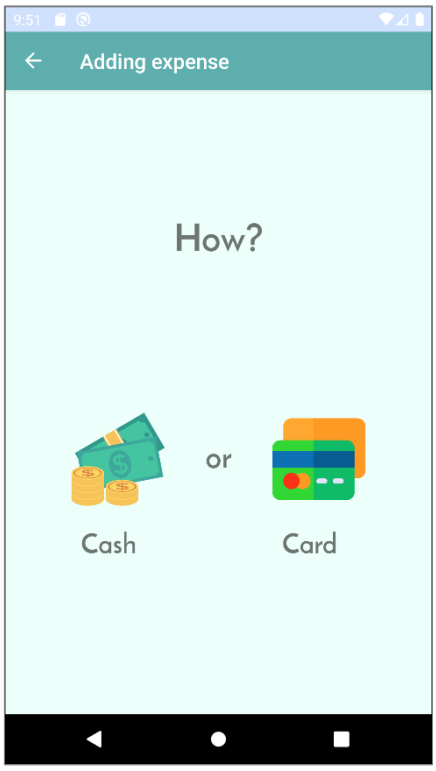
To conclude with the report, a review of the main screens of the application is going to be presented.



Position 1



Position 2

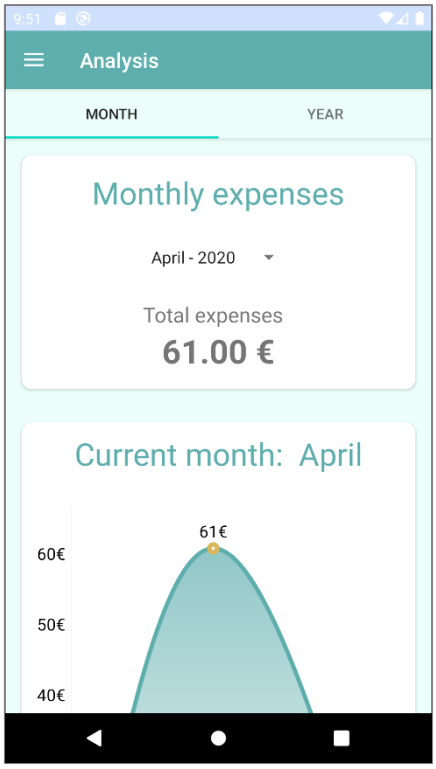


Position 3

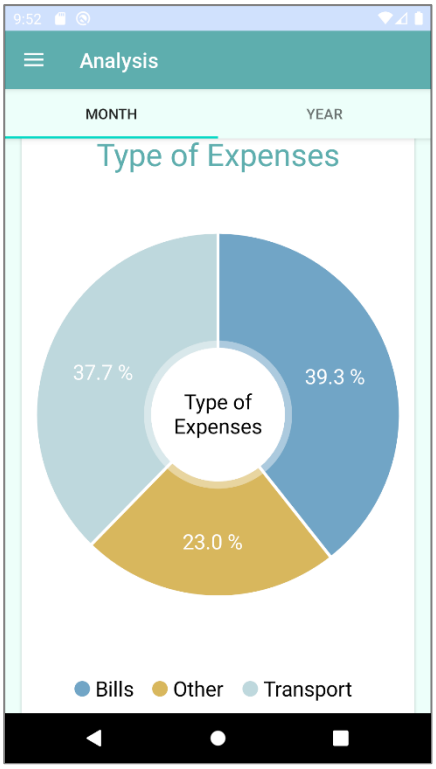
Position 4

Process of adding an expense is described on the above screens. After confirming, the user will be back on Home.

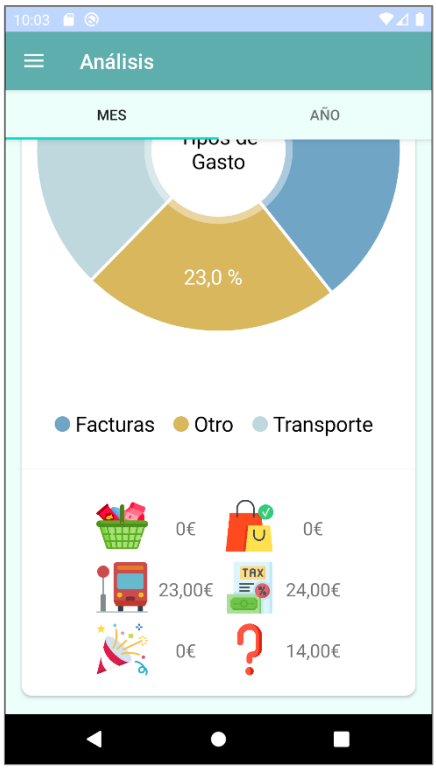
In position 1, if user has already inserted an expense, the advice *Here you can...* will not appear.



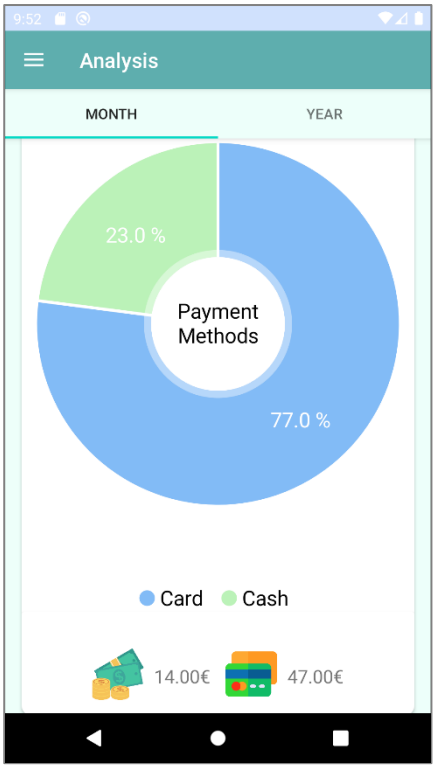
Position 1



Position 2

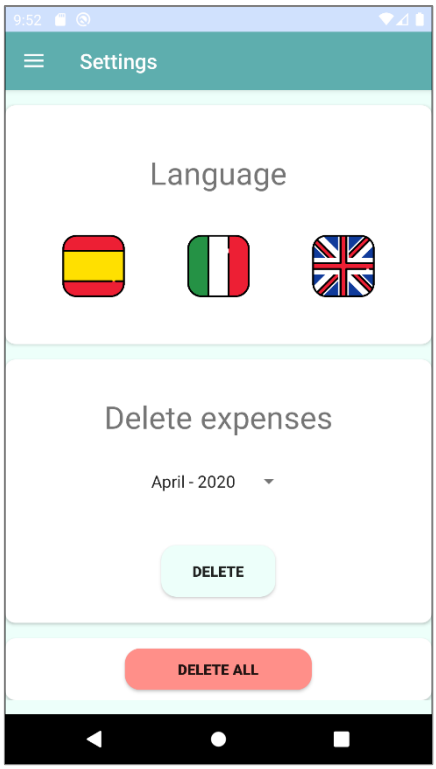


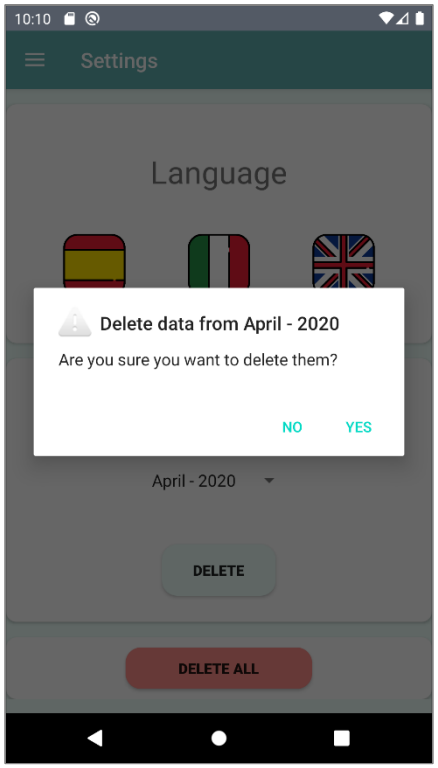
Position 3



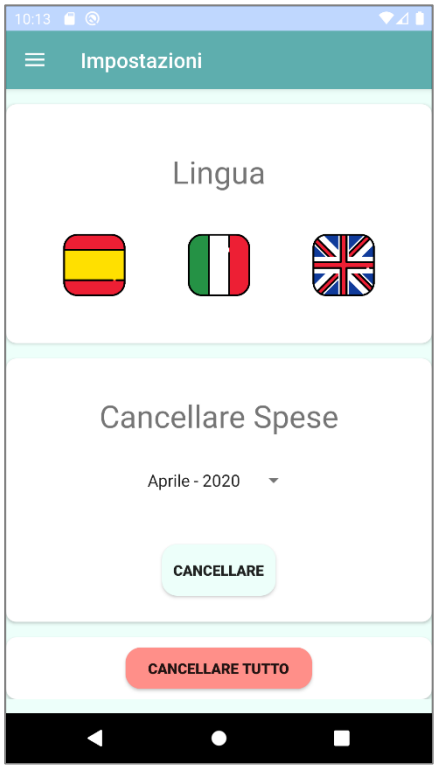
Position 4

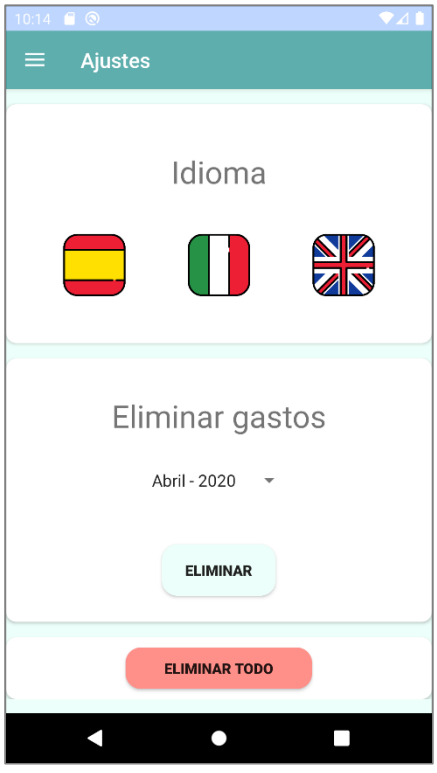
Some screens of the Analysis section. All pie charts have a legend supporting them.



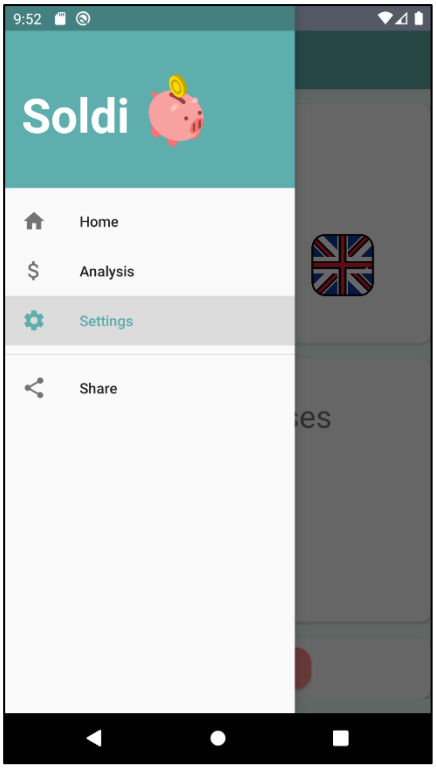


Settings section. Second image shows the confirmation dialog when pressing *Delete* button.





Settings section, first on Italian and then on Spanish.



Navigation drawer. Share option allows the user to share the application with a personalized text:

Time is gold! Use Soldi app to manage your expenses in the simplest way :) Take a look at it and save your time: https://play.google.com/store/apps/details?id=com.project.soldiapp