System design document for LEEFO

Eugene Dvoryankov, Emelie Edberg, Felix Edholm, Linus Lundgren, Omar Sulaiman

2021-10-05
version 1

1 Introduction

LEEFO is an Android application made for keeping track of a users finances. The application lets the user know how their income and expenses change on a month to month basis as well as letting the user create a budget. The purpose of the application is to make the user get a better understanding of their spending habits and assist them in changing them if they would like to.

This system design document details the design choices made in the development process of the application, the overarching design of the application as well as the system structure. The document also covers how the application deals with persistence and the quality of the final product and how it was tested.

1.1 Definitions, acronyms, and abbreviations

MVC Model-View-Controller, a design pattern of high abstraction used to break an application into three parts: the model, the view and the controller [1]. The purpose of the model is to manage all application logic and data handling. The purpose of the view is to display information to the user, information sent from/retrieved from the model. Finally, the purpose of the controller is to handle all user input, and converting it into specific commands/method calls in the model.

Command design pattern A design pattern used when parameters for methods get more complicated [2]. The implementation of a 'Command class' is a class used to create objects storing parameters for whatever method should be called. An example of this would be an object storing a list of search parameters for a function that searches through data.

Observer design pattern The observer design pattern is a way of implementing how the view responds to changes in the model [3]. An 'Observer' is a class implementing an abstract interface consisting of methods required by whatever the model requires it to have. The most basic example of this would be an interface with an update() method, which is called every time some data in the model changes.

SQLite SQLite is a database management system embedded in the application where it is used, instead of implementing a client-server structure that is common with other database management systems [4]. It is used to store/access data with persistence, and includes most of the functionality you would expect from normal SQL databases.

Android activity In developing applications for android, an activity is basically a page that you can view while using the app. It is the highest level 'container' for the user interface [5].

Android fragment A fragment is a lower level UI 'container' than the activity, and is usually used to represent some specific part of an activity [6]. These fragments can be switched out with other fragments, making the user interface more modular in functionality.

XML XML is a markup language used in android applications for designing the user interface [7].

JUnit JUnit is a framework used for unit testing in Java [8]. Unit testing means testing individual parts of a program to see if they are functioning properly.

STAN STAN is a tool used to visualize dependencies between packages and classes inside of a Java project [9].

PMD PMD is a tool used for analyzing code to find common programming mistakes [10].

2 System architecture

LEEFO can be described as a standalone application at first glance. The application runs with the help of a database which makes storing data in the application much smoother.

The goal of using the database is to store data when the application shuts down. That does not mean LEEFO does not work without the database. The application works just fine without the database but when the application shuts down all the data in the lists will be lost. Under runtime the application only stores and deletes from the database and when shutting the system down all the data needed is already in the database. Then when starting the application the app gets all the information that were already stored in the database. The reason for using the database is mostly to store and delete, as the fact is that the database is meant to be a service and not a model in the MVC pattern.

3 System design

From the most abstract perspective, the application consists of three main packages: the view, the controller and the model. The diagram below describes the relationship between these packages.

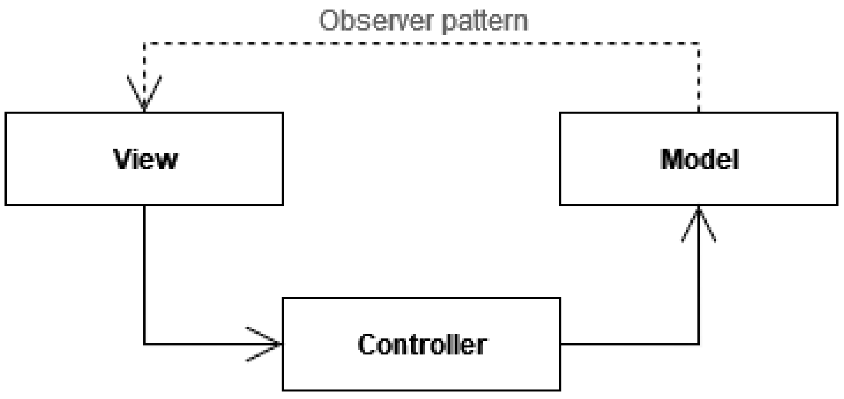


Figure 1: Application packages

The basic purpose of these packages are:

- View - handles all user interfaces.
- Controller - handles user interaction with the view.
- Model - handles all application logic

3.1 Controller package

The basic purpose of the controller package is to first of all parse any interaction the user might have had with the view. It then calls on the appropriate methods in the model to execute whatever task the user wants done.

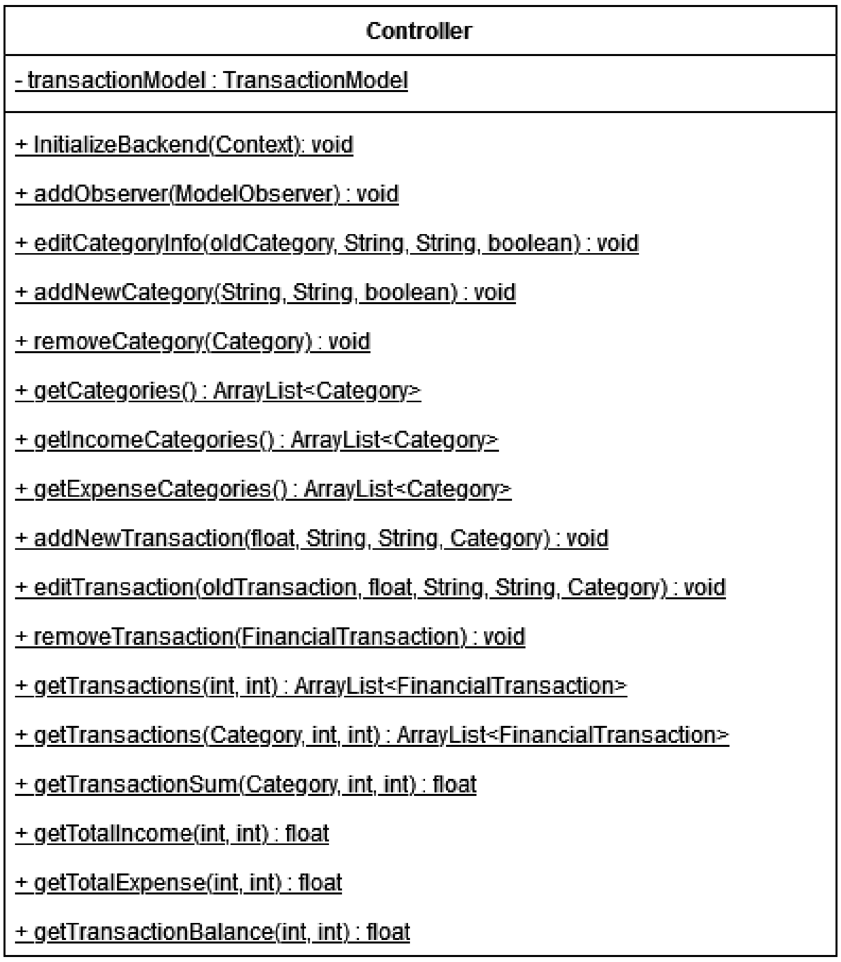

Controller
- transactionModel : TransactionModel
+ InitializeBackend(Context): void
+ addObserver(ModelObserver) : void
+ editCategoryInfo(oldCategory, String, String, boolean) : void
+ addNewCategory(String, String, boolean) : void
+ removeCategory(Category) : void
+ getCategories() : ArrayList<Category>
+ getIncomeCategories() : ArrayList<Category>
+ getExpenseCategories() : ArrayList<Category>
+ addNewTransaction(float, String, String, Category) : void
+ editTransaction(oldTransaction, float, String, String, Category) : void
+ removeTransaction(FinancialTransaction) : void
+ getTransactions(int, int) : ArrayList<FinancialTransaction>
+ getTransactions(Category, int, int) : ArrayList<FinancialTransaction>
+ getTransactionSum(Category, int, int) : float
+ getTotalIncome(int, int) : float
+ getTotalExpense(int, int) : float
+ getTransactionBalance(int, int) : float

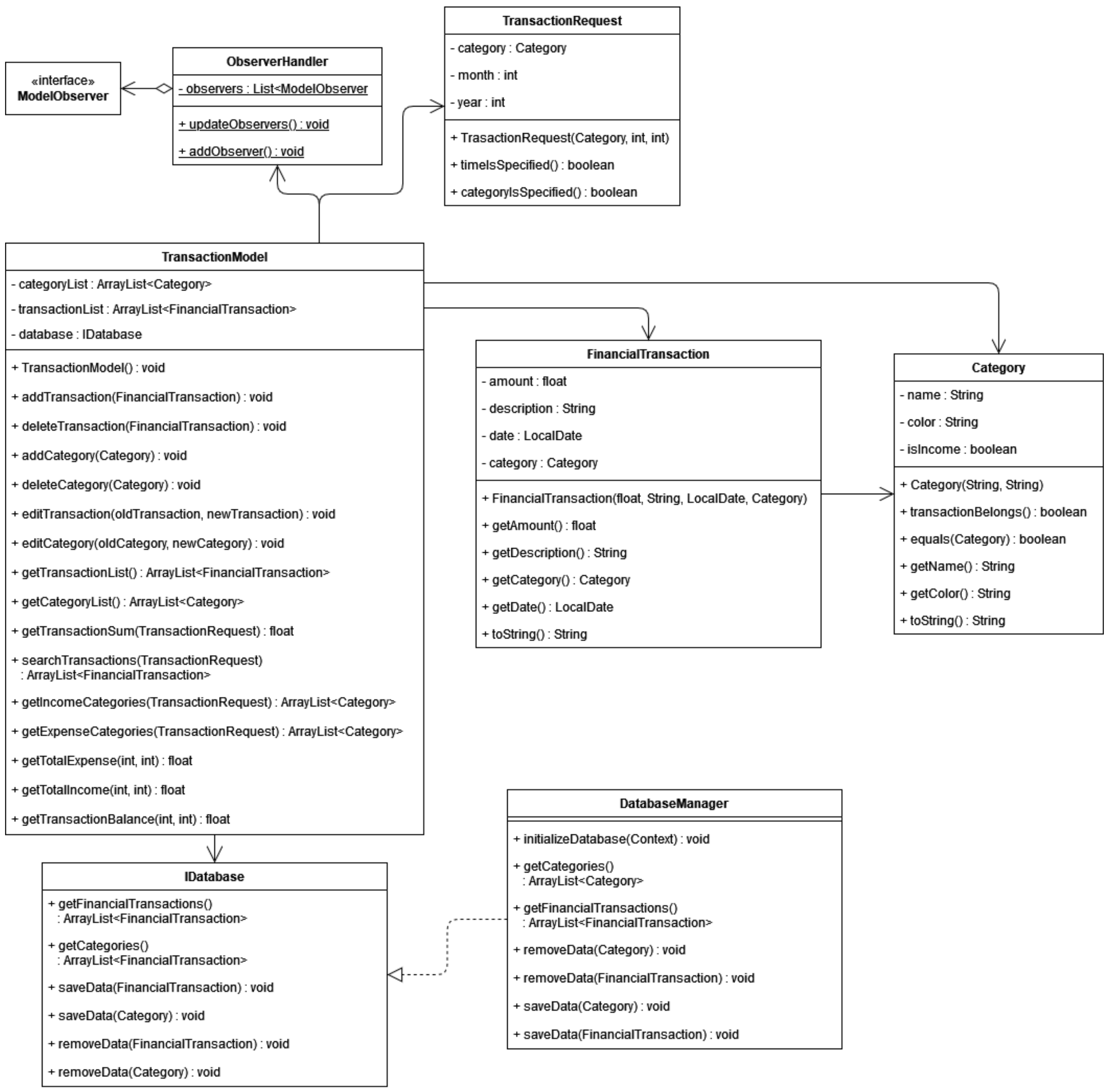
Figure 2: Controller package design

Our implementation of the controller is a static class which is accessed by the view through any of the public methods seen in the diagram above. The single static variable transactionModel is the object through which the model is told what to do.

In our implementation of the MVC-pattern the controller is fairly thin, consisting mostly of delegation to the model, with a few exceptions where objects such as TransactionRequests (which is a class from the model) are created so that the Model knows what transactions are specified.

3.2 Model package

The purpose of the model in the Model-View-Controller design pattern is to handle all domain logic. In our case that means managing financial transactions and categories to which these transactions can belong.



«interface»
ModelObserver
ObserverHandler
- observers : List<ModelObserver
+ updateObservers() : void
+ addObserver() : void
TransactionRequest
- category : Category
- month : int
- year : int
+ TrasactionRequest(Category, int, int)
+ timeIsSpecified() : boolean
+ categoryIsSpecified() : boolean
TransactionModel
- categoryList : ArrayList<Category>
- transactionList : ArrayList<FinancialTransaction>
- database : IDatabase
+ TransactionModel() : void
+ addTransaction(FinancialTransaction) : void
+ deleteTransaction(FinancialTransaction) : void
+ addCategory(Category) : void
+ deleteCategory(Category) : void
+ editTransaction(oldTransaction, newTransaction) : void
+ editCategory(oldCategory, newCategory) : void
+ getTransactionList() : ArrayList<FinancialTransaction>
+ getCategoryList() : ArrayList<Category>
+ getTransactionSum(TransactionRequest) : float
+ searchTransactions(TransactionRequest) : ArrayList<FinancialTransaction>
+ getIncomeCategories(TransactionRequest) : ArrayList<Category>
+ getExpenseCategories(TransactionRequest) : ArrayList<Category>
+ getTotalExpense(int, int) : float
+ getTotalIncome(int, int) : float
+ getTransactionBalance(int, int) : float
FinancialTransaction
- amount : float
- description : String
- date : LocalDate
- category : Category
+ FinancialTransaction(float, String, LocalDate, Category)
+ getAmount() : float
+ getDescription() : String
+ getCategory() : Category
+ getDate() : LocalDate
+ toString() : String
Category
- name : String
- color : String
- isIncome : boolean
+ Category(String, String)
+ transactionBelongs() : boolean
+ equals(Category) : boolean
+ getName() : String
+ getColor() : String
+ toString() : String
IDatabase
+ getFinancialTransactions() : ArrayList<FinancialTransaction>
+ getCategories() : ArrayList<FinancialTransaction>
+ saveData(FinancialTransaction) : void
+ saveData(Category) : void
+ removeData(FinancialTransaction) : void
+ removeData(Category) : void
DatabaseManager
+ initializeDatabase(Context) : void
+ getCategories() : ArrayList<Category>
+ getFinancialTransactions() : ArrayList<FinancialTransaction>
+ removeData(Category) : void
+ removeData(FinancialTransaction) : void
+ saveData(Category) : void
+ saveData(FinancialTransaction) : void

Figure 3: Model package design

At the center of this entire application lies the way in which transaction and category data is packaged. Transactions are handled by the FinancialTransaction class, which contains basic information about the transactions well as a reference to whatever Category object represents the category that the transaction belongs to.

These FinancialTransaction and Category objects are handled by the TransactionModel, whose main objective is to deal with requests sent by the controller. The logic implemented to make sure this is done correctly includes sorting, searching transactions, making sure the database is updated along with the ArrayLists, along with other things required for the model to operate smoothly.

The TransactionModel interfaces with the database through the IDatabase interface and is only used for basic data persistence, as we wanted to keep most domain logic object-oriented. The DatabaseManager object for which the database field is used as an interface is instantiated in the controller class on startup.

TransactionModel also interacts with the ObserverHandler class, which is a static class whose purpose is to store a list of observers that are interested in being notified whenever something changes inside the model. For example, when a transaction or category is changed, the TransactionModel calls on updateObservers() which sends a notification to all stored observers, who can then execute whatever functionality is specified.

The controller usually interacts with the model package through simply calling on a method in the TransactionModel. Sometimes however, the request from the controller might be a little more complex, such as when the view is requesting FinancialTransactions from a specific time period and of a specific category. In these instances, the controller makes use of the TransactionRequest class. This class is an implementation of the Command design pattern, and is basically used for creating objects that stores search parameters defining exactly what transactions the model should return.

3.3 View package

The view package purpose is to provide a way for the user to view and interact with the model through a graphical interface. The view package retrieves information about the model through the Controller class and displays it in various ways in the GUI. The view listens to the user's interactions and then uses the Controller to tell the model what the user wants done.

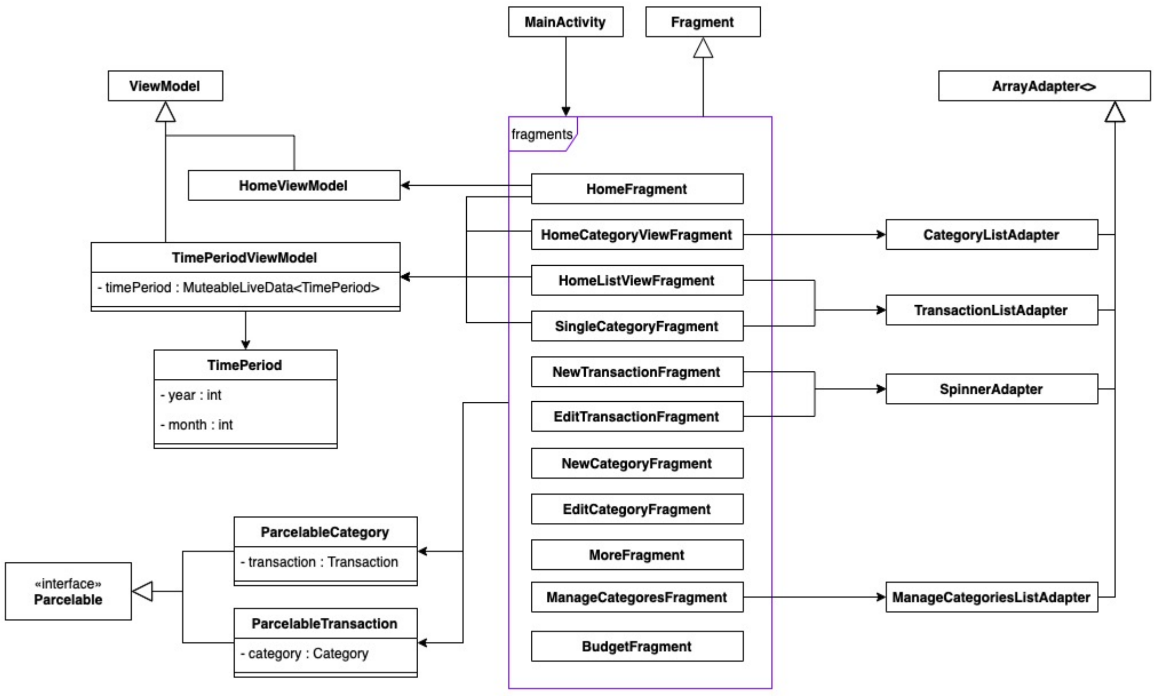


Figure 4: View package design

When running an android project, the MainActivity will be created first and from there the rest of the application is initialized and started. An activity represents one phone window in which the UI is placed. LEEFO uses one activity and several fragments representing different pages of the app placed in MainActivity.

Every fragment, adapter and activity uses a corresponding layout resource file (XML) which decides the design of the and what components it has. The fragment classes handles initialising of components, setting onClickListeners and communicates with the Controller.

Four custom adapter classes are being used for making list and spinners look and work as desired.

Navigation between fragments work like so that MainActivity has a frameLayout in which one fragment can be opened at a time. When the user wants to navigate to a different fragment the current one is replaced with the new one. When a fragment is replaced it is being destroyed and every time a fragment is opened it will be created anew. Due to this, a fragments current data will be lost when replaced. But this also has made it so that the view does not need to be updated by changes in the model. Because every time a fragment is opened it gets the current and updated data from the model. An observer pattern is currently implemented but not used. We have it because it may be needed in the future for features where the user can make a change and view it in the same fragment, but that is currently not the case.

Two ViewModel classes are used to store fragment data that needs to survive longer than the fragment life cycle and to share data between fragments. Only used for storing view associated data and not data from the model, like keeping track of what time period the user currently has chosen.

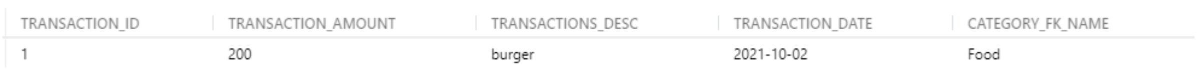
Another way the fragments share data is when fragment A need to send data to fragment B when you navigate from fragment A to B. Then we use a method where fragment A can attach arguments to fragment B with the use of Bundle. In order to send object this way they need to implement the interface Parcelable and that is why we have the classes ParcelableTransaction and ParcelableCategory.

4 Persistent data management

Under runtime the application uses non persistent data like storing objects in lists. The app does also use persistent data management when starting the application due to the the loss of the non persistent data after the shut down. LEEFO uses a database class called SQLiteOpenHelper.The database is used whenever adding or deleting any transaction or category object. It is also used as a service and not as a model which makes using the database under runtime an invalid option.

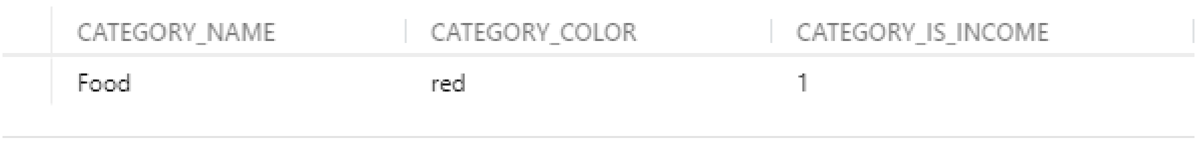
The application starts by requesting the information in the database stored in two tables as seen in figures 5 and 6. For example adding a financial transaction starts by adding the transaction into a list in the model and then adding the transaction in the transaction table in the database. Which is the same for all the operations that is related to the database. Thenceforth everything under runtime does the same thing continuously. That is until the application shuts down.

When the application shuts down, all the non persistent data stored in the lists in the model disappears. Which leaves the application without any data. That is where the database comes in. When the application starts, it checks if there is any data in the database and makes objects of the data as well as saving the objects in lists. For example when starting the application the model checks if there are any transactions saved in the database by calling getting all transactions method which returns a list of the transaction objects that were saved before the shutdown.



TRANSACTION_ID	TRANSACTION_AMOUNT	TRANSACTIONS_DESC	TRANSACTION_DATE	CATEGORY_FK_NAME
1	200	burger	2021-10-02	Food

Figure 5: Figure shows how the transactions are saved in the database transaction table



CATEGORY_NAME	CATEGORY_COLOR	CATEGORY_IS_INCOME
Food	red	1

Figure 6: Figure shows how the categories are saved in the database category table

5 Quality

5.1 Testing

The application is tested using JUnit tests. The tests along will the rest of the project are viewable at *this* GitHub repository.

The JUnit tests are in a separate folder in the project. The coverage goal for the tests has been over 90% coverage of all methods in the model package.

The following tests have been written:

- Tests for adding and removing transactions and categories.
- Tests for editing already added transactions and categories.
- Tests for retrieving information from specific conditions e.g. for a time period.
- Tests for sorting the information in the application.
- Tests for calculating total expense, total income and balance.

5.2 STAN results

STAN is a tool used for showing dependencies in a Java project [9]. In figure 7 you can see the dependencies in the project for LEEFO.

The dependencies on the left of the figure from ".budgetapplication" are from the JUnit tests so these can be disregarded.

There are no major circular dependencies in the project structure. There is only one circular dependency and that is between the view and the fragments. The view contains the object first created when the application is started which is responsible for creating some of the fragments, that is the dependency from the view to the fragments in the diagram. The dependency from the fragments to the view is mostly because the view includes some ViewModel classes that the fragments need to access shared data within the view. See chapter 3.3 View package for a more detailed diagram.

The View package depends on the Controller package in order to send requests to the model but there is no dependency the other way because the Controller does not need to know the details of who is sending the requests. The Controller package then depends on the Model package in order to forward the requests from the view and also to retrieve information from the Model for the view.

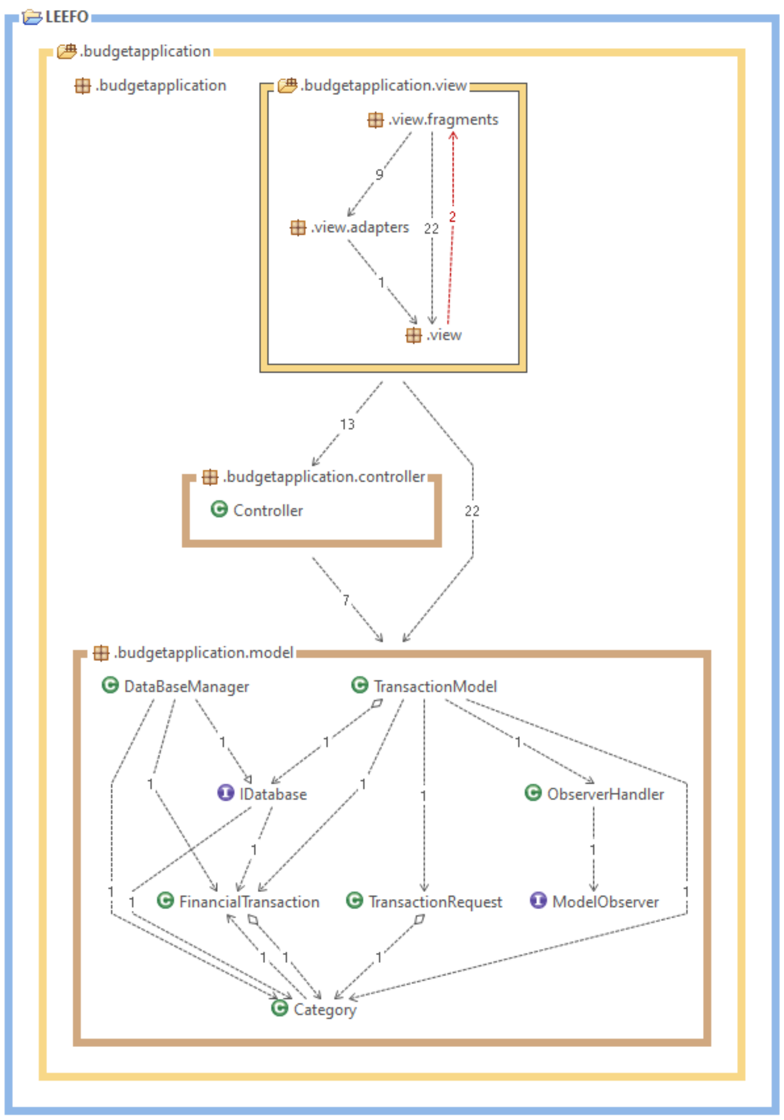


Figure 7: STAN results.

5.3 PMD results

PMD is a tool used for analyzing source code to find common programming mistakes [10].

The PMD analysis found some minor issues such as unused imports or comments in the code that are too long. Though, it did also highlight some other possible issues with the code such as the TransactionModel class being too big, with too many methods, possibly needing refactoring into separate classes.

See appendix for full PMD analysis.

6 References

References

[1] GeeksForGeeks, "MVC Design Pattern," 2018. [online] Available at: https://www.geeksforgeeks.org/mvc-design-pattern/ [Accessed 7 October 2021].

[2] Refactoring.Guru, "Command," 2021. [online] Available at: https://refactoring.guru/design-patterns/command [Accessed 7 October 2021].

[3] Refactoring.Guru, "Observer," 2021. [online] Available at: https://refactoring.guru/design-patterns/observer [Accessed 7 October 2021].

[4] SQLite, "About SQLite," 2021. [online] Available at: https://www.sqlite.org/about.html [Accessed 7 October 2021].

[5] Android Developers, "Introduction to Activites," 2019. [online] Available at: https://developer.android.com/guide/components/activities/intro-activities [Accessed 7 October 2021].

[6] Android Developers, "Fragments," 2020. [online] Available at: https://developer.android.com/guide/fragments [Accessed 7 October 2021].

[7] Liam Quin, "Extensible Markup Language (XML)," 2016. [online] Available at https://www.w3.org/XML/ [Accessed 7 October 2021].

[8] Marc Philipp, "FAQ," 2020. [online] Available at: https://github.com/junit-team/junit4/wiki/FAQ [Accessed 7 October 2021].

[9] Stan4j, "About," 2021. [online] Available at: http://stan4j.com/about [Accessed 8 October 2021].

[10] PMD, "About," 2021. PMD. [online] Available at: https://pmd.github.io/#about [Accessed 8 October 2021].