Final report LEEFO
RAD + SDD + Peer review

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

October 24, 2021

Requirements and Analysis Document for LEEFO

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

2021-10-23

Contents

1 Introduction

The purpose of this project is to create an Android application for finance tracking. The application is a tool created for anyone to get a better understanding and a clear picture of ones income and expenses. The user manually enters their transactions into the app and chooses what category it lies under. The application then keeps track of the users spending every month and displays how much money goes to different categories through diagrams and lists. Users can also create a monthly budget to get maximum control of their money.

1.1 Definitions, acronyms, and abbreviations

Transaction, an instance of income or expense. Example a food purchase or receiving salary. A transaction has associated with it, an amount, a date, a category and an optional description.

2 Requirements

2.1 User Stories

01 Enter transactions

As a user, I want to be able to enter new transactions so that I can keep track of my financial transactions.

IMPLEMENTED

Confirmation, functional

- Can I enter a new transaction?
- Can I set an amount?
- Can I set a date?
- Can I chose a category?
- Can I chose income/expense?
- Can I edit a transaction after adding it?
- Can I remove a transaction?

02 Overview

As a user I want a page were I can view general information on the same screen so that I can get an overview of my current financial situation.

IMPLEMENTED

Confirmation, functional

- Can I, on a single screen, see my total income, expense and balance for the current month?
- Can I, on the same screen, see some representation of my expenses for that month?

03 View transactions, list

As a user I want to be able to see a list of my previously entered transaction so that I can keep track of what I have entered.

IMPLEMENTED

Confirmation, functional

- Can I view my previously entered transactions this month in a list?

04 View transactions, pie chart

As a user, I want to be able to view my transactions in a pie chart, sorted by category to see how my spending is distributed between different areas.

IMPLEMENTED

Confirmation, functional

- Can I see my spending for each category in a pie chart?
- Can I clearly understand what ”slice” corresponds to what category?
- Can I see the percentage for a ”slice”/category?

05 View transactions, single category

As a user, I want to be able to view my transactions from a single category so that I can see more details of where the money goes in that category.

IMPLEMENTED

Confirmation, functional

- Can I chose which category I want too view?
- Can I see the transactions of the chosen category?

06 Edit/delete transactions

As a user, I want to be able to edit and delete my previous transactions so that I can correct wrongfully entered data.

IMPLEMENTED

Confirmation, functional

- Can I edit a transaction?
- Can I remove a transaction?

07 Add/edit/delete Categories

As a user, I want to able to edit the amount of categories available so that I can customize the categories to fit my own life.

IMPLEMENTED

Confirmation, functional

- Can I create custom categories?
- Can I edit existing categories?
- Can I remove a category?

Confirmation, nonfunctional

Avalability

- Can I remove a category that has transactions under it?

08 Monthly time periods

As a user I want to be able to see my transactions monthly, not just all history since the start, so that I can think of my economy in monthly intervals as humans commonly like to do.

IMPLEMENTED

Confirmation, functional

- Can I see my transactions from the current month?
- Can I see my transactions from previous months?

09 Set/edit budget

As a user I want to be able to set a budget for categories of my choosing so that I can plan and check up on my spending.

IMPLEMENTED

Confirmation, functional

- Can I a choose a goal amount for the categories I want?

10 View budget

As a user I want to be able to check how well I am following my budget so that I can know how much money I can afford to spend.

IMPLEMENTED

Confirmation, functional

- Can I see how well I am currently following my budget?
- Can I see budget history for previous month?

11 Budget grade

As a user I want to be able to get a grade for how well im doing with my budgetting so that I am incentivized to keep saving and budgetting.

IMPLEMENTED

Confirmation, functional

- Can I see the grade for a specific category for a specific month?
- Can I see a grade for my budgetting as a whole for a specific month?

12 Sort

As a user I want to be able to sort transactions in different orders so I can view my transactions easier.

IMPLEMENTED

Confirmation, functional

- Can I sort by oldest/newest date?
- Can I sort by largest/smallest amount?

13 Search

As a user I want to be able to search for transactions so that I can find them easier.

IMPLEMENTED

Confirmation, functional

- Can I search for transaction description?
- Can I search for transaction amount?

13 Filter

As a user I want to be able to filter out transactions so that I can view/find them easier.

IMPLEMENTED

Confirmation, functional

- Can I filter out expenses/incomes?

14 Category popularity

As a user I want my recently most used categories to be recommended first when adding a new transaction so that I don't need to look as much for the right category.

IMPLEMENTED

Confirmation, functional

- Does my recently most used categories show at the top of the list when adding a new transaction?

15 Streak

As a user I want to be able to achieve a streak for good spending habits so that I get encouraged to to spend less.

IMPLEMENTED

Confirmation, functional

- Can I see my streak for amount of days in a row where I have spent less than I do on a average day?
- Can I see my high score streak?

16 Compare

As a user I want to be able to compare spending between years/months so that I can see how my spending has changed between years/months.

NOT IMPLEMENTED

Confirmation, functional

- Can I compare my spending?
- Can I decide what time periods I would like to compare?
- Can I see the comparison displayed in a diagram?

17 Balance graph

As a user I want to be able to view my account balance history in a line graph so that I can see how my account balance has been impacted over time.

NOT IMPLEMENTED

Confirmation, functional

- Can I see a graph that displays my balance history?
- Can I set the time period that the graph is displaying?

2.2 Definition of Done

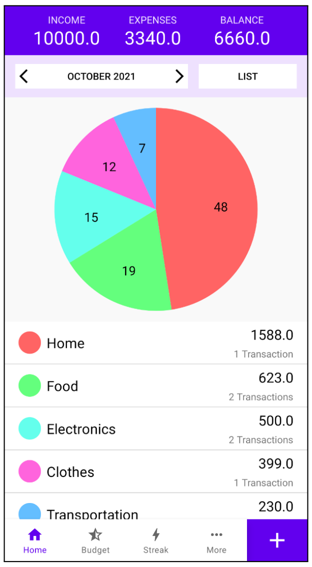
For a user story to be considered finished it must meet the following criteria.

- All added features are tested and have passed jUnit tests. (90+ percent coverage)
- User story confirmations are fulfilled as expected.
- Completed features are added to GitHub.
- The code is well commented. All public methods must have javadoc comments.

2.3 User interface

The user interface consists of four main pages, home, budget, streak and more, which are navigated with a navigation bar at the bottom of the screen. The navigation bar also contains a plus button which is used to add a new transaction. Due to this position it is possible to add a transaction from every part of the app.

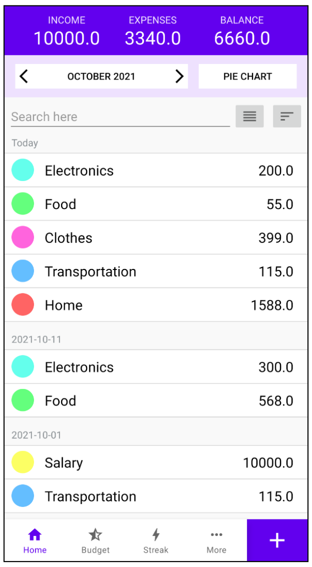
The "home" page is the start page and it contains an overview of the users transactions in a chosen time period. The user can change month or choose to see all transactions by pressing the month picker. On the home page the user can choose between two different visualizations of the transaction data. Either a list view which displays every transaction in a list, which can be sorted, filtered and searched in. Or a category view which contains a pie chart based on category as well as a more compact list ranking the categories from most to least amount spent in them. These list items are clickable and takes the user to another page with a list of every transaction in that specific category. The user can easily edit a transaction by pressing it. See figure 1 for some images of these screens.



INCOME	EXPENSES	BALANCE
10000.0	3340.0	6660.0

Home	1588.0 1 Transaction
Food	623.0 2 Transactions
Electronics	500.0 2 Transactions
Clothes	399.0 1 Transaction
Transportation	230.0

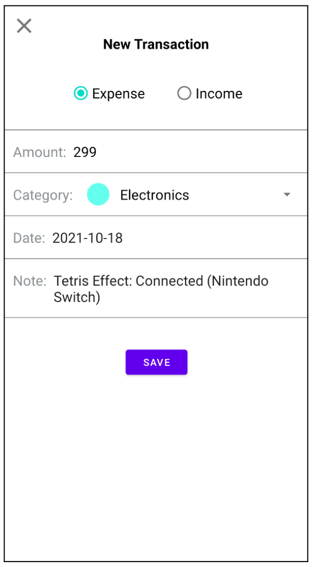
(a) Pie chart view



INCOME	EXPENSES	BALANCE
10000.0	3340.0	6660.0

Today	
Electronics	200.0
Food	55.0
Clothes	399.0
Transportation	115.0
Home	1588.0
2021-10-11	
Electronics	300.0
Food	568.0
2021-10-01	
Salary	10000.0
Transportation	115.0

(b) List view



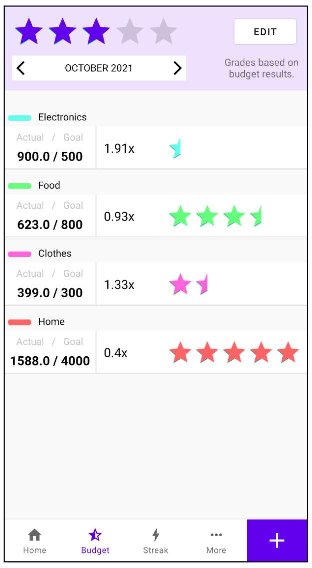
[✓ Expense] Income

Amount: 299
Category: Electronics
Date: 2021-10-18
Note: Tetris Effect: Connected (Nintendo Switch)

(c) New transaction

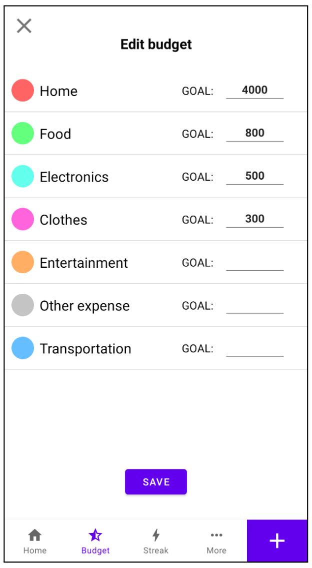
Figure 1: Home + new transaction

The "budget" page is where the user can create a budget and see how well the budgets spending goals are met. The user can choose which categories they want to include in their budget. Every category then receives a grade based on how well the user is following the spending goal. See figure 2.



Category	Actual / Goal		
Electronics	900.0 / 500	1.91x	
Food	623.0 / 800	0.93x	
Clothes	399.0 / 300	1.33x	
Home	1588.0 / 4000	0.4x	

(a) Grade

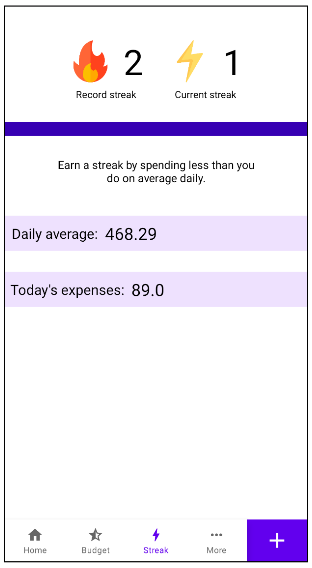


Home	GOAL: 4000
Food	GOAL: 800
Electronics	GOAL: 500
Clothes	GOAL: 300
Entertainment	GOAL:
Other expense	GOAL:
Transportation	GOAL:

(b) Edit

Figure 2: Budget

The ”streak” page displays the users current and record streak. Here the user also gets data about their daily average spending as well as today’s spending. See figure 3.



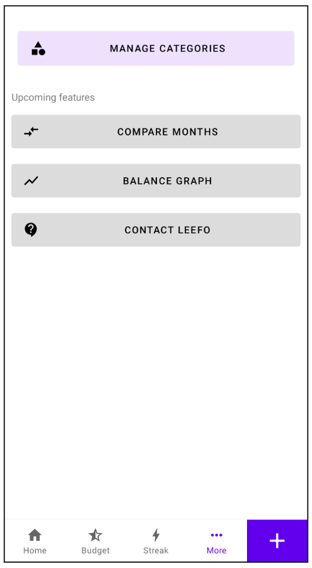
Earn a streak by spending less than you do on average daily.

Daily average: 468.29

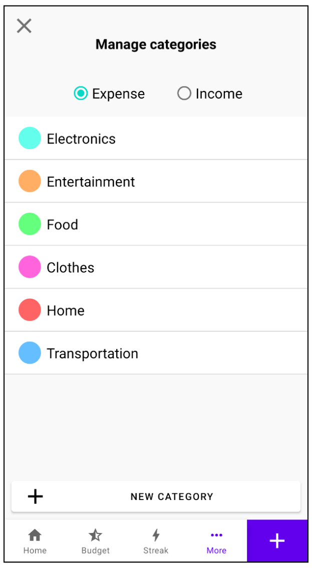
Today's expenses: 89.0

Figure 3: Streak

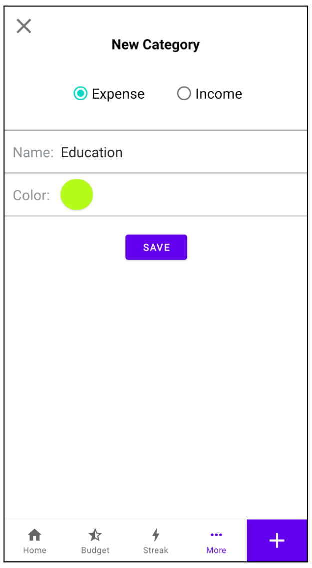
The ”more” page contains additional features which are not as commonly used. Here the user can press ”manage categories” to edit categories and add new ones. To edit a category you press on it in the list. See figure 4.



(a) More



(b) Manage categories



Name: Education

Color:

(c) New category

Figure 4: More

3 Domain model

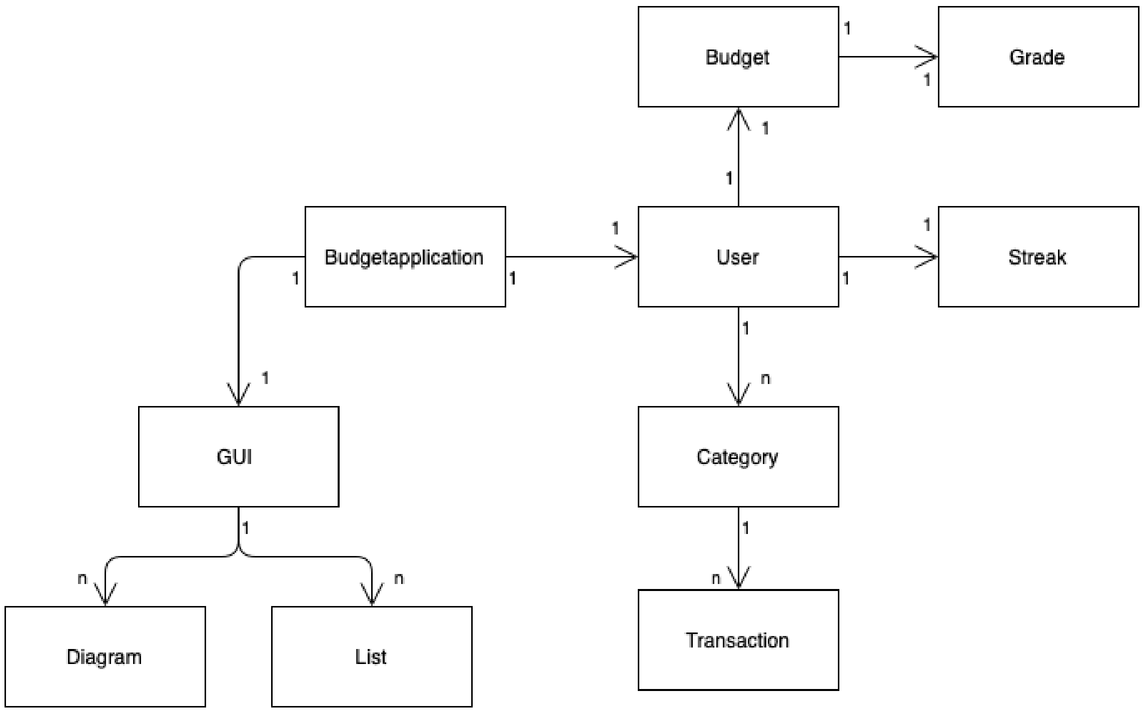


Figure 5: Domain model

3.1 Class responsibilities

Budgetapplication - Represents the application and connects the other classes.

User - The user of the application.

Budget - A spending budget the user can set

Grade - The budget has a grade based on how well the user is following the budget goal.

Category - Categorisation of incomes and expenses, every single transaction falls under one category chosen by the user. The user can edit the number of transactions and customize their names and colors.

Transaction - A container for the user's transaction data. The user can add a unlimited number of transactions.

Streak - A spending streak the user can earn by spending less than it's daily average several days in a row.

GUI - Represents the graphical user interface of the application.

Diagram - Displays the users transaction data through diagrams in the GUI.

List - Displays the user transaction data through lists in the GUI.

System design document for LEEFO

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

2021-10-23

Contents

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.

Singleton design pattern The Singleton pattern is a design pattern that ensures that only one instance exists of a class and also provides global access to that instance [4].

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 [6]. 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 [8].

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 [9]. 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 [10].

JUnit JUnit is a framework used for unit testing in Java [11]. 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 [12].

PMD PMD is a tool used for analyzing code to find common programming mistakes [13].

2 System architecture

LEEFO can be described as a standalone application under the runtime of the application. Still the application is mostly focused on storing information than visualisation. The app is mostly built to help the user manage their own transactions and not about visualising them. With these thoughts it was decided that the application would need some kind of service to save the data. The application runs with the help of a database which makes storing data in the application much smoother.

2.1 Database SQLITE

The database is used with the help of a class inbuilt in android studio which manages the creation of the database which uses SQLite. SQLite is a C-language library that implements a part of the SQL database engine. This library is mostly used for mobile phone development [5].

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 saved before the shut down 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. And when starting the application the app retrieves all the information that was 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.

2.2 MVC

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. See figure 1.
The basic purpose of these packages are:

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

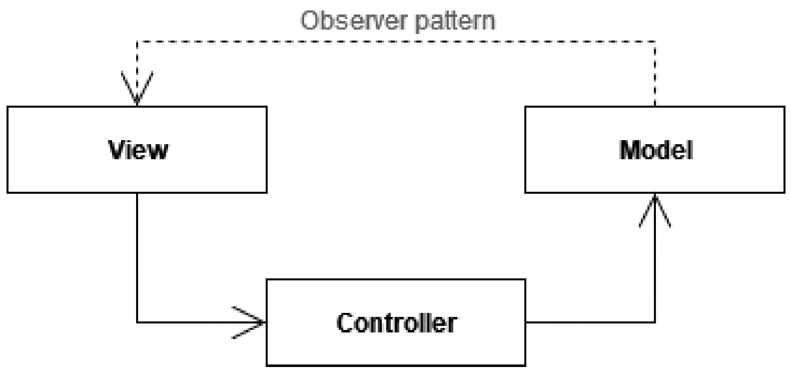


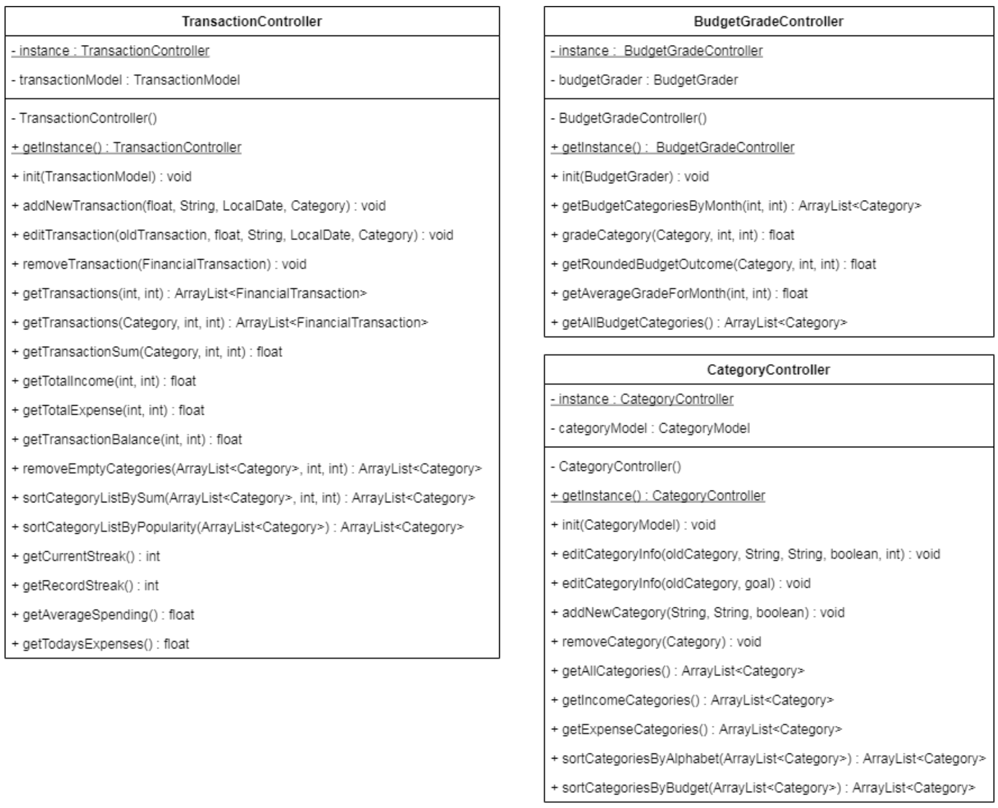
Figure 1: Application packages

From an object oriented programming view it is very important to apply a software architecture pattern. In this project the MVC pattern is used through having a Controller, a Model and a View. The Controller is meant to connect the View and the Model. It gets notified by the View of the user's behavior and acts upon it and calls the methods needed in the model. The Model on the other hand is the part containing all the logic. As written before the database is not meant to be a part of the model but in our case the database has some logic in it. Nevertheless the database is more of a service than a model which is why a part of the database is counted as a part of the Model. Finally the View is where the users interactions are sent to, and where the application interacts with the user by showing and visualising data retrieved from the Model depending on the user's input [6].

3 System design

3.1 Controller package

The Controller package is split into three different classes to separate responsibilities and thus make the code more extensible. The basic purpose of those classes is to parse any interaction the user might have with the view. The controller classes are also responsible for calling on the appropriate methods in the model, to execute whatever task the user wants done. See figure 2.



TransactionController
- instance : TransactionController
- transactionModel : TransactionModel
- TransactionController()
+ getInstance() : TransactionController
+ init(TransactionModel) : void
+ addNewTransaction(float, String, LocalDate, Category) : void
+ editTransaction(oldTransaction, float, String, LocalDate, 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
+ removeEmptyCategories(ArrayList<Category>, int, int) : ArrayList<Category>
+ sortCategoryListBySum(ArrayList<Category>, int, int) : ArrayList<Category>
+ sortCategoryListByPopularity(ArrayList<Category>) : ArrayList<Category>
+ getCurrentStreak() : int
+ getRecordStreak() : int
+ getAverageSpending() : float
+ getTodaysExpenses() : float
BudgetGradeController
- instance : BudgetGradeController
- budgetGrader : BudgetGrader
- BudgetGradeController()
+ getInstance() : BudgetGradeController
+ init(BudgetGrader) : void
+ getBudgetCategoriesByMonth(int, int) : ArrayList<Category>
+ gradeCategory(Category, int, int) : float
+ getRoundedBudgetOutcome(Category, int, int) : float
+ getAverageGradeForMonth(int, int) : float
+ getAllBudgetCategories() : ArrayList<Category>
CategoryController
- instance : CategoryController
- categoryModel : CategoryModel
- CategoryController()
+ getInstance() : CategoryController
+ init(CategoryModel) : void
+ editCategoryInfo(oldCategory, String, String, boolean, int) : void
+ editCategoryInfo(oldCategory, goal) : void
+ addNewCategory(String, String, boolean) : void
+ removeCategory(Category) : void
+ getAllCategories() : ArrayList<Category>
+ getIncomeCategories() : ArrayList<Category>
+ getExpenseCategories() : ArrayList<Category>
+ sortCategoriesByAlphabet(ArrayList<Category>) : ArrayList<Category>
+ sortCategoriesByBudget(ArrayList<Category>) : ArrayList<Category>

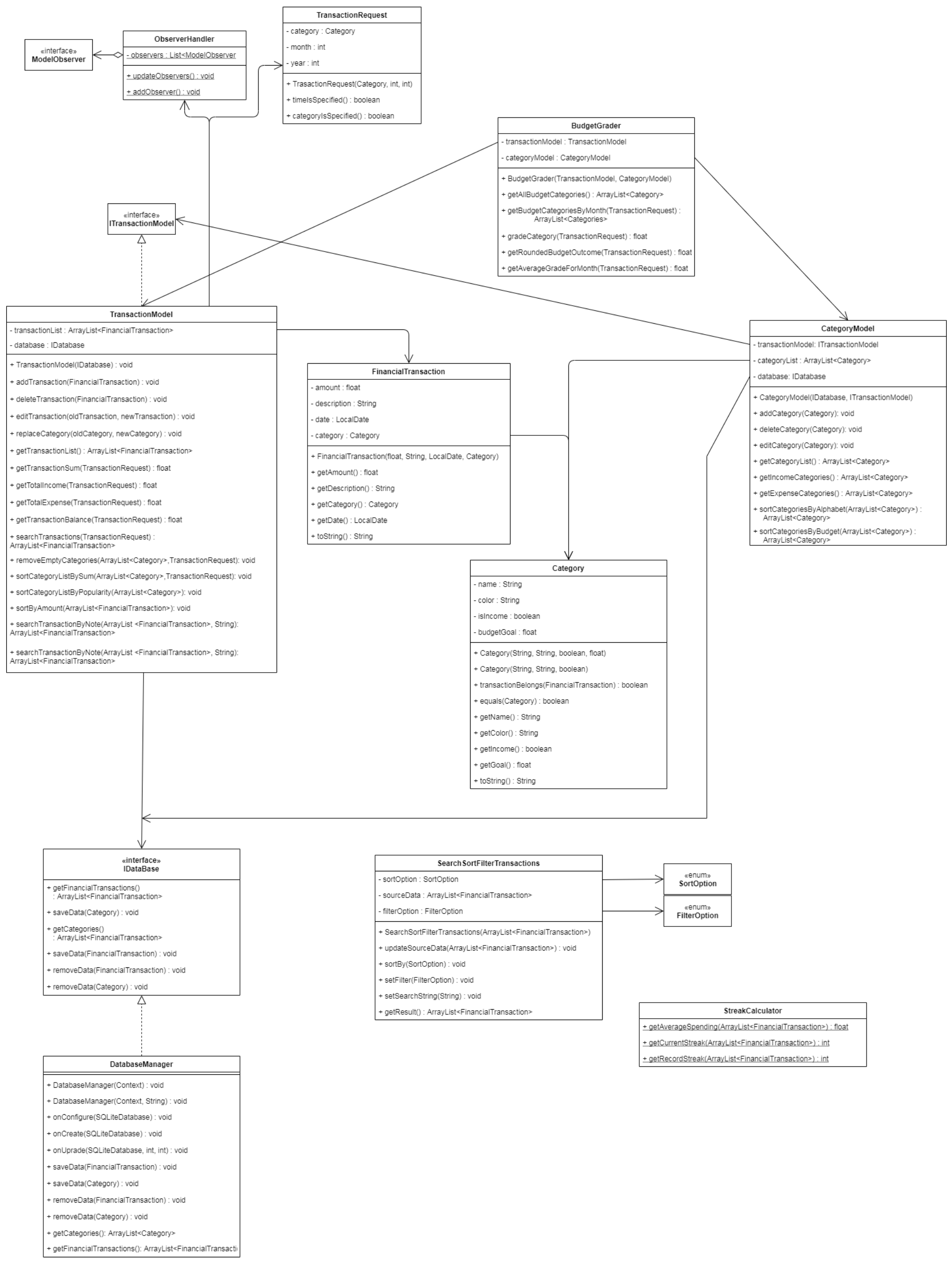
Figure 2: Controller package design

Each controller class is implemented using the Singleton Pattern. This means that there is only a single instance of the class and that instance can only be accessed through the getInstance() method. The choice to use the Singleton pattern was made to give fragments access to the controllers without having to pass controller objects to the fragments as this is not easy or preferable to do in android development. In order to avoid direct calls with "new" operator, the constructor is private and empty.

The controllers are 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 as well as the logic for budgets and streaks. See figure 3.



«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
BudgetGrader
- transactionModel : TransactionModel
- categoryModel : CategoryModel
+ BudgetGrader(TransactionModel, CategoryModel)
+ getAllBudgetCategories() : ArrayList<Category>
+ getBudgetCategoriesByMonth(TransactionRequest) : ArrayList<Categories>
+ gradeCategory(TransactionRequest) : float
+ getRoundedBudgetOutcome(TransactionRequest) : float
+ getAverageGradeForMonth(TransactionRequest) : float
«interface» ITransactionModel
TransactionModel
- transactionList : ArrayList<FinancialTransaction>
- database : IDatabase
+ TransactionModel(IDatabase) : void
+ addTransaction(FinancialTransaction) : void
+ deleteTransaction(FinancialTransaction) : void
+ editTransaction(oldTransaction, newTransaction) : void
+ replaceCategory(oldCategory, newCategory) : void
+ getTransactionList() : ArrayList<FinancialTransaction>
+ getTransactionSum(TransactionRequest) : float
+ getTotalIncome(TransactionRequest) : float
+ getTotalExpense(TransactionRequest) : float
+ getTransactionBalance(TransactionRequest) : float
+ searchTransactions(TransactionRequest) : ArrayList<FinancialTransaction>
+ removeEmptyCategories(ArrayList<Category>,TransactionRequest): void
+ sortCategoryListBySum(ArrayList<Category>,TransactionRequest): void
+ sortCategoryListByPopularity(ArrayList<Category>): void
+ sortByAmount(ArrayList<FinancialTransaction>): void
+ searchTransactionByNote(ArrayList <FinancialTransaction>, String): ArrayList<FinancialTransaction>
+ searchTransactionByNote(ArrayList <FinancialTransaction>, String): ArrayList<FinancialTransaction>
FinancialTransaction
- amount : float
- description : String
- date : LocalDate
- category : Category
+ FinancialTransaction(float, String, LocalDate, Category)
+ getAmount() : float
+ getDescription() : String
+ getCategory() : Category
+ getDate() : LocalDate
+ toString() : String
CategoryModel
- transactionModel: ITransactionModel
- categoryList : ArrayList<Category>
- database: IDatabase
+ CategoryModel(IDatabase, ITransactionModel)
+ addCategory(Category): void
+ deleteCategory(Category): void
+ editCategory(Category): void
+ getCategoryList() : ArrayList<Category>
+ getIncomeCategories() : ArrayList<Category>
+ getExpenseCategories() : ArrayList<Category>
+ sortCategoriesByAlphabet(ArrayList<Category>) : ArrayList<Category>
+ sortCategoriesByBudget(ArrayList<Category>) : ArrayList<Category>
Category
- name : String
- color : String
- isIncome : boolean
- budgetGoal : float
+ Category(String, String, boolean, float)
+ Category(String, String, boolean)
+ transactionBelongs(FinancialTransaction) : boolean
+ equals(Category) : boolean
+ getName() : String
+ getColor() : String
+ getIncome() : boolean
+ getGoal() : float
+ toString() : String
«interface» IDataBase
+ getFinancialTransactions() : ArrayList<FinancialTransaction>
+ saveData(Category) : void
+ getCategories() : ArrayList<FinancialTransaction>
+ saveData(FinancialTransaction) : void
+ removeData(FinancialTransaction) : void
+ removeData(Category) : void
SearchSortFilterTransactions
- sortOption : SortOption
- sourceData : ArrayList<FinancialTransaction>
- filterOption : FilterOption
+ SearchSortFilterTransactions(ArrayList<FinancialTransaction>)
+ updateSourceData(ArrayList<FinancialTransaction>) : void
+ sortBy(SortOption) : void
+ setFilter(FilterOption) : void
+ setSearchString(String) : void
+ getResult() : ArrayList<FinancialTransaction>
«enum» SortOption
«enum» FilterOption
StreakCalculator
+ getAverageSpending(ArrayList<FinancialTransaction>) : float
+ getCurrentStreak(ArrayList<FinancialTransaction>) : int
+ getRecordStreak(ArrayList<FinancialTransaction>) : int
DatabaseManager
+ DatabaseManager(Context) : void
+ DatabaseManager(Context, String) : void
+ onConfigure(SQLiteDatabase) : void
+ onCreate(SQLiteDatabase) : void
+ onUprade(SQLiteDatabase, int, int) : void
+ saveData(FinancialTransaction) : void
+ saveData(Category) : void
+ removeData(FinancialTransaction) : void
+ removeData(Category) : void
+ getCategories(): ArrayList<Category>
+ getFinancialTransactions(): ArrayList<FinancialTransacti

Figure 3: Model package design

FinancialTransaction and Category classes

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.

TransactionModel and CategoryModel

The FinancialTransaction 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 Lists, along with other things required for the model to operate smoothly.

The CategoryModel has the same area of responsibility as the TransactionModel, the only difference being that it handles Category objects instead.

DatabaseManager

The Transaction- and CategoryModel 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 MainActivity class on startup.

Observer handling

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 is changed, the TransactionModel calls on updateObservers() which sends a notification to all stored observers, who can then execute whatever functionality is specified.

TransactionRequest

The controllers usually interacts with the model package through simply calling on a method in the Model. Sometimes, however, the request from the controllers 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.

Search, sort and filter

The class SearchSortFilterTransactions is responsible for combining searching, sorting and filtering of a list. The class provides methods for entering said list and to set different sort and filter options and a search string as well as getting the result list. The class is used directly by the view.

StreakCalculator

The purpose of the streak calculator is to calculate the spending streak that the user is currently on, as well as the longest spending streak that the user has ever had. A spending streak is a sequence of days where the user has spent less than they do on an average day (calculated based on all transactions the user has ever made).

BudgetGrader

The BudgetGrader class handles the logic to calculate the grade of a category with a budget goal. The grade is based on the quotient between the actual spending in a month and the spending goal of that category. The class also calculates the average grade for each month.

3.3 View package

The view packages 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 classes and displays it in various ways in the GUI. The view listens to the user's interactions and then uses the controllers to tell the model what the user wants done. See figure 4.

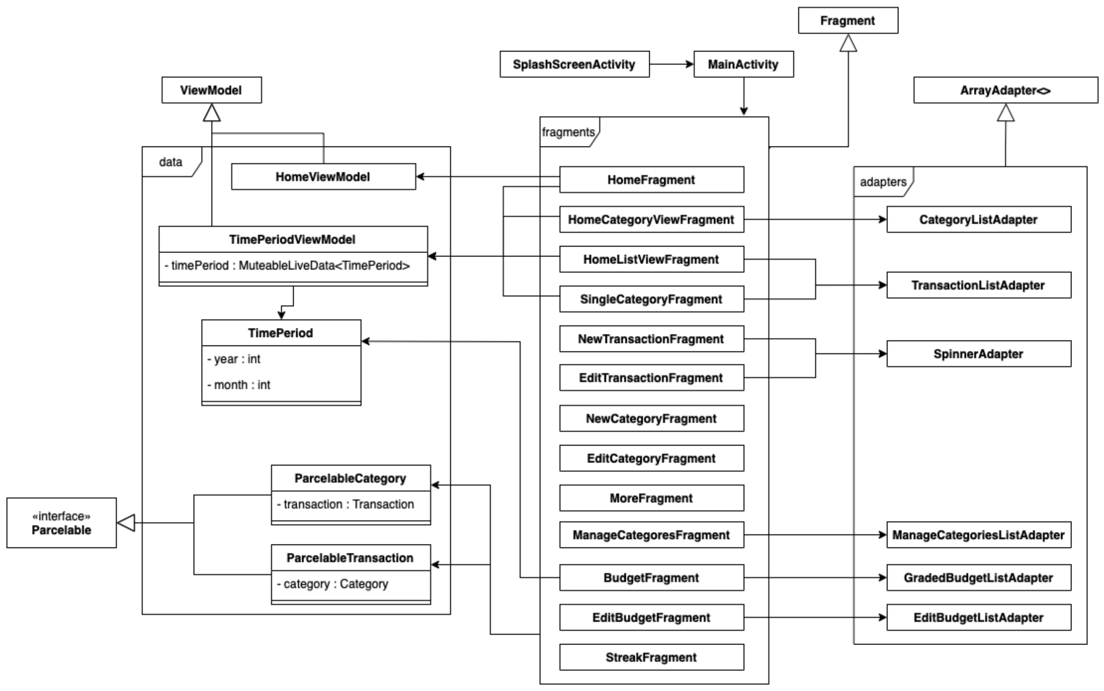


Figure 4: View package design

The view's structure is made up of two activities: SplashScreenActivity and MainActivity, along with with three packages: adapters, data and fragements.

When the application is launched, a start screen generated by SplashScreenActivity will be visible. A few moments later, the MainActivity will be created and from there, the rest of the application is initialized and started. An activity represents one smartphone window in which the UI is located. LEEFO uses one activity while several fragments representing different pages of the app are placed inside it.

Each fragment, adapter and activity uses a corresponding layout resource file (XML) which decides the design and what components it has.

Adapters package

The adapter package is made up of six custom adapter classes being used for making lists and spinners look and work as desired.

Fragments package

The fragments package consists of all the fragments containing the UI. The fragment classes handle initialising of components, setting onClickListeners and communicating with the controllers.

Navigation between fragments is operated from frameLayout inside the MainActivity. One fragment, at a time, can be opened in the frameLayout. When the user wants to navigate to a different fragment, the current fragment is replaced with a new one. When a fragment is being replaced, it is destroyed and every time a fragment is opened it will be created anew. Due to this, the fragment's current data will be lost, when replaced. This operation removes the need for the view to be notified of changes in the model, since every time a fragment is opened, it gets the current and updated data from the model.

An observer pattern is currently being implemented but not used. The pattern exists for the future features' needs, where the user can make a change and view it in the same fragment.

Data package

The data package consist of classes the fragments use for saving and sharing view associated data. Example which of two fragments was opened most recent. (One could think that these are logic classes and should therefore belong in the model. However, we believe that the model should be separate and not know anything of any view, therefore we can't put these classes in the model just because the view needs them.)

Two ViewModel classes are used to store fragment data that needs to survive longer than the fragment life cycle and to share data between the fragments. (These classes are not ViewModels in the sense of the MVVM pattern, we use the ViewModel classes because that is the way to store view data in android.) The TimePeriodViewModel is used for keeping track of what time period the user currently has chosen. HomeViewModel is used to save information for HomeFragment.

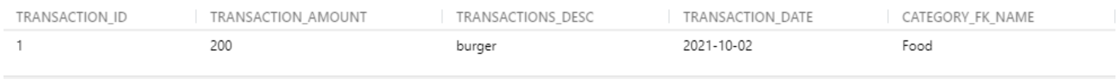
Another way the fragments share data is when fragment A needs to send data to fragment B when the user navigates from fragment A to B. In this case, a method, that attaches arguments from fragment A to fragment B with the use of Bundle, is being called. Objects, that are sent between fragments, need to implement the Parcelable interface and that's the reason classes ParcelableTransaction and ParcelableCategory exist.

4 Persistent data management

During runtime the application uses non persistent data like storing objects in list objets. The app uses persistent data management when starting the application due to 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 used as a service and not as a model class which makes using the database under runtime an invalid option.

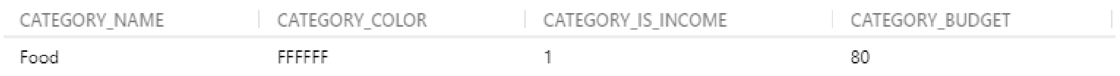
The application starts by requesting the information in the database stored in two tables as seen in figures 5 and 6. For example adding a financial transaction starts by adding the transaction into a list in the model and then adding the transaction to the transaction table in the database. This is 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 the model lists. For example, when starting the application, the model checks if there are any transactions saved in the database by calling a method of getting all transactions 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: Transaction table



CATEGORY_NAME	CATEGORY_COLOR	CATEGORY_IS_INCOME	CATEGORY_BUDGET
Food	FFFFFF	1	80

Figure 6: 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 and have 93% method coverage in the model package:

- 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, searching and filtering the information in the application.
- Tests for calculating total expense, total income and balance.
- Tests for calculating and retrieving streaks.
- Tests for calculating and retrieving budget grades.

5.2 Known issues

The application has some known issues. The issues mostly arise because of the limited time with the project as well as the project not being an application with the purpose of actually reaching a market.

- The database could possibly benefit from being split into separate tables for categories and transactions. Instead of creating everything from within the single database you can create separate classes. If more tables are needed it would be good to separate but as it stands, with only two tables, it is not a major problem.
- If the project would be made for an actual market the grade limits for the budget outcomes would be based on user research and not arbitrarily chosen as it is now.
- The implementation of the observer pattern is never actually used. It could be useful in the future if a new addition needs it, but as it stands it is never used.

- The GUI does not always translate well between different screens. The design is made to be somewhat dynamic for many screens but on some screens it can look a little off.
- There is no exception handling in the code. The program is designed with checks so that no exceptions based on user input should be possible, though some exception handling could still be useful to catch possible errors that those checks do not.
- Some information, such as streaks and budget grades, is calculated every time that view is selected. In a real application where a user has a lot of data after many years of use this could be resource heavy and storing that information in a database would be preferable.
- Sorting, searching and filtering is done in the model. This is a more object oriented approach because that was one of the main purposes of the assignment, but it may be less effective than doing it in the database since one of the main purposes of using a database is to search and sort.

5.3 STAN results

STAN is a tool used for showing dependencies in a Java project [12]. In figure 7 you can see the dependencies in the project for LEEFO. The dependencies in the middle right of the figure from ".budgetapplication" are from the JUnit tests so these can be disregarded.

There are no circular dependencies in the project structure.

The View package depends on the Controller package in order to send requests to the Model package but there is no dependency the other way because the controllers do 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.

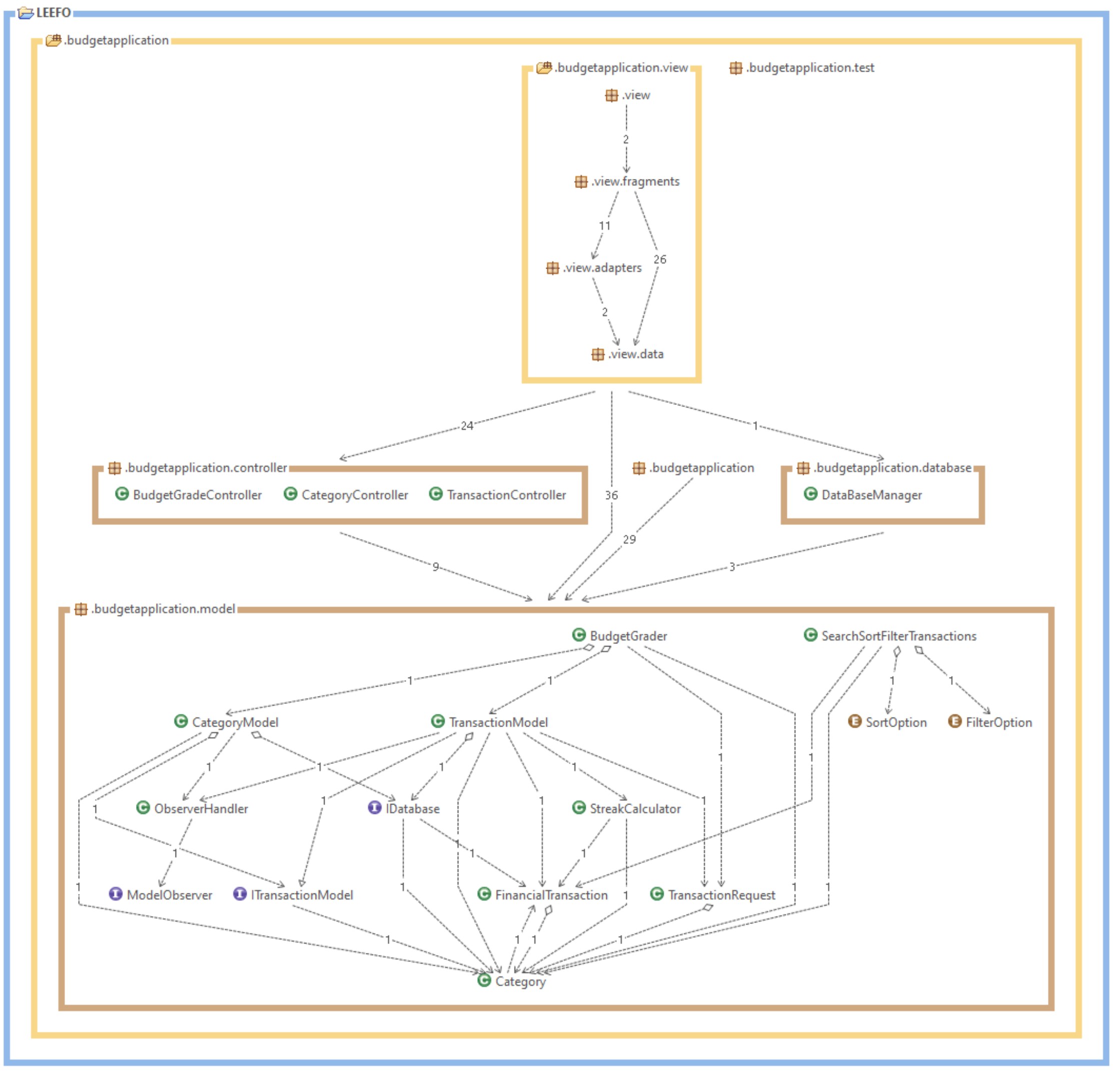


Figure 7: STAN results.

5.4 PMD results

PMD is a tool used for analyzing source code to find common programming mistakes [13].

The PMD analysis found some minor issues such as comments in the code possibly being 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. The TransactionModel class has been refactored into separate classes before, though the PMD analysis still sees it as a possible issue. However, we believe that the class is fine as it is in its current state.

Full PMD analysis can be downloaded *here*.

6 References and libraries

Libraries used in the application

MPAndroidChart - Chart library for Android.
`https://github.com/PhilJay/MPAndroidChart`

Ambilwarna - Android color picker library.
`https://github.com/yukuku/ambilwarna`

Core Library Desugaring, to be able to use a number of Java 8 language APIs without requiring a minimum API level for your app. LEEFO needs it for the use of LocalDate. `https://developer.android.com/studio/write/java8-support`

Lottie-android - Animation library.
`https://github.com/airbnb/lottie-android`

JUnit 5 - Testing framework.
`https://github.com/junit-team/junit5/`

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] Refactoring.Guru, ″Singleton,″ 2021. [online] Available at: `https://refactoring.guru/design-patterns/singleton` [Accessed 22 October 2021].

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

[6] SQLite, ″What is SQLite?,″ 2021. [online] Available at: `https://www.sqlite.org/index.html` [Accessed 19 October 2021].

[7] Medium, ″Android Architecture Patterns Part 1: Model-View-Controller,″ 2021. [online] Available at: `https://medium.com/upday-devs/android-architecture-patterns-part-1-model-view-controller-3baecef5f2b6` [Accessed 19 October 2021].

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

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

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

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

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

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

Peer review of the project “Nollans första dag”

Review made by group LEEFO: Linus Lundgren, Emelie Edberg, Eugene Dvoryankov, Felix Edholm and Omar Sulaiman

- *Do the design and implementation follow design principles?*

The code does have some issues regarding the single responsibility principle. One example of this is the Orientation enum, which is supposed to represent which direction the player is facing. In this implementation however, it is also used for determining the speed at which the player moves, which gives it multiple areas of responsibility.

Another example of this is the CollisionChecker class, which based on its name is supposed to check whether the player is colliding with any obstacles or if they collide with any side of the map. Currently however, it also handles switching between maps, which should probably be handled by another class such as the MapModel.

- *Does the project use a consistent coding style?*

The code does use a consistent style through most of the classes. Nonetheless some classes have long methods when others have very short methods and same thing for documentation, some classes are well documented when others are documented using some swedish words “svengelska” which also applies to names of variables and methods..

- *Is the code reusable?*

As it stands, there are quite a lot of instances where the same code is basically reused (copy/pasted) in several different places. Examples of this is in the setup in MainGame, in the PlayerController where it checks if the player is pressing a button, in the NPC classes, where there’s a class for every unique NPC or in the player model where there’s a method for every direction the player can move (these methods are unused however, so they should probably be removed). This makes it hard to reuse since expanding basically just means adding a bunch of new code, not reusing already existing code.

- *Can we easily add/remove functionality?*

The use of separate MVC structures for each task makes it easy to add or remove tasks without impacting other tasks. This is good!

- *Are design patterns used?*

Factory: the NPC classes are created through a Factory class, called NPCFactory. The pattern as it is currently implemented however makes some of the other code useless. Currently there is a unique class for every NPC, but the point of each of these classes is just to set their respective attributes to specific values. This is unnecessary, and makes it harder to add new NPCs since you have to create a new class every time. Instead of this you can use the Factory pattern with a method for each NPC which returns an NPC object (which would no longer be an abstract class) with their corresponding attributes. This makes adding new NPCs easier since you just have to add a function to the Factory, instead of creating a new class.

State: Each map is a state which holds a method for transition to the next map. The map transition is happening in MapModel using a Mapstate attribute.

MVC: Each task has its own model, its own view and controller. Those model, views and controllers are independent from other tasks.

- *Is the code documented?*

The code is somewhat documented using JavaDoc comments. Some classes, like the class BeerChuggingController, have comments for every method and attribute but other classes, like the BeerChuggingModel class, have JavaDoc comments for some attributes and methods but not all. Other classes are not documented at all though we understand that this is a work in process and not fully finished yet. @Author tag is missing from most files. There are also quite a few grammar mistakes and misspelling in the JavaDoc comments that should probably be fixed.

- *Are proper names used?*

Some names may be improper to use out of a programming view. It is mostly fine to use names that shows a view of what the class is capable of but when having names like BeerChugging*, it makes sense to use because it is what the task includes. Though this can make the class a bit unclear to what the task is programmatically. Same thing when thinking about the names of some methods like “loopGreenThingyLocation”. This does not give any identification to the method and makes more needless complications.

- *Is the code well tested?*

There are currently no runnable tests in the project. There is one test class for though this is commented out.

- *Is the code easy to understand? Does it have an MVC structure, and is the model isolated from the other parts?*

We don't know much about working with slick2D but it looks like the model may not be completely separated and contains more than domain logic. The playerModel makes use of Animation which may not belong in the model but in the view instead, because the view should be responsible for how the model is displayed.

- *Can the design or code be improved? Are there better solutions?*

Something small that could be improved other than what is explained above would be to make the controls clearer for a new player as we needed to check the code to understand that you should press F to start a task.

Overall it is a fun and ambitious project, especially for us as other Chalmers students. The familiarity of campus is there and the graphics are very nice. It is obviously still not fully finished but we look forward to being able to play the full game when it is done!