System design document for La-Luna

Ali Malla, Deaa Khankan, Ali Alkhaled, Bilal Al Malek

date 2021-10-24
version
2

Innehållsförteckning

1 Introduction

La Luna är en budget Applikation som hjälper användaren att ha koll på hur mycket pengar den spenderar varje månad. Applikationen är uppbyggt med syfte att den ska följa objekt orienterade principer. Exempelvis på dessa principer är MVVM som används för att separera logik från gränssnitt samt minska kopplingar mellan klasserna. Facade Pattern som döljer databasen från klientkoden samt Bridge Pattern som förenklar bytet av databasen och gör det möjligt att modellen blir oberoende av services (SQLite) vilket är ett krav på applikationen. Applikationen använder sig av SQlite som databas för att spara data som matas in av användaren. Applikationens modell har testets med JUnit samt med hjälp av PMD analys plug-in.

1.1 Definitions, acronyms, and abbreviations

- MVVM: Ett designmönster som strukturerar upp mjukvaruprojekt genom att uppnå det så kallat Seperation of Concern, vilket med andra ord innebär att gränsnitt, affärslogik och data ska skiljas och vara direkt oberoende av varandra.
- SOLID: En förkortning för fem viktiga designprinciper inom mjukvaruutveckling, vilka är Single responsibility principle, Open closed principle, Liskov substitution principle, Interface segregation principle och Dependency inversion principle.
- Databas: En samling information/data sparad i ett datorsystem.
- SQLite: Ett tillgänligt bibliotek för server-lös databashanterare.
- DBHandler: En klass som underlättar kommunikationen mellan databasen och andra delar i projektet

2 System architecture

La Luna är en Android applikation som är skriven i Javaspråket. Applikationen följer MVVM desigmönstret vilket är viktig för att strukturera applikationen på ett sätt som följer SOLID principer och ha en low coupling kod. Applikationen använder sig av SQLite för att spara de data som matas in av användare såsom utgifter och kategorier. Applikationens modell är oberoende av SQLite databas då kommunikationen sker genom en medellager klass ”DBhandler” som hanterar mellan modellen och service ”SQLite”. Genom detta kan man enkelt byta databasen vilket är viktig för att koden ska vara öppen för förlängning och stängd för modifiering.

När man kör applikationen, öppnas hem-sidan först. Där går man oftast för att se tidigare utgifter och lägga till nya eller modifiera redan existerade utgifter.Om

användaren till exempel vill lägga till en utgift, behöver hen klicka på knappen ”Add Expense” sedan behöver hen fylla utgiftsinformation. Därefter sparas utgiften i tabellen (SQLite databasen) med namn, värde, skapelsedatum och till vilken kategori den tillhör plus ID nummer. För att se en övergripande bild på utgifterna som är delade i olika kategorier går man till analsyis-sidan. Där också visualiseras Kategorierna genom cirkeldiagram (pie-charts). Dessutom kan man vara i behöv att lägga till nya kategorier vilket man gör i kategori sidan och datan sparas på liknande sätt som när sparas en utgiftsdata. När applikationen stängs och öppnas igen visas all data som lagts innan.

3 System design

3.1 UML package diagram

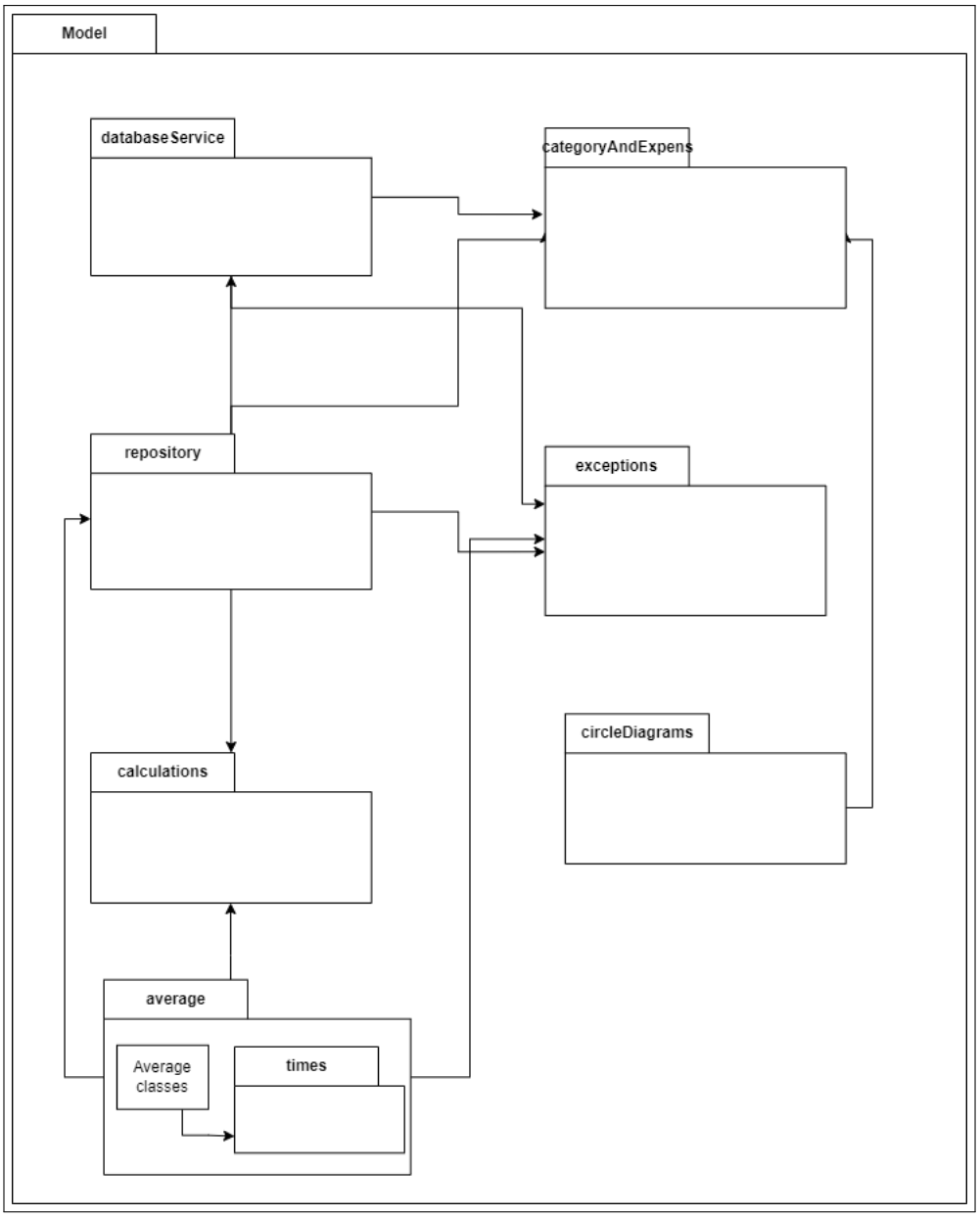

Figure 1: UML package diagram.

3.2 UML class diagram

3.2.1 model package class diagram

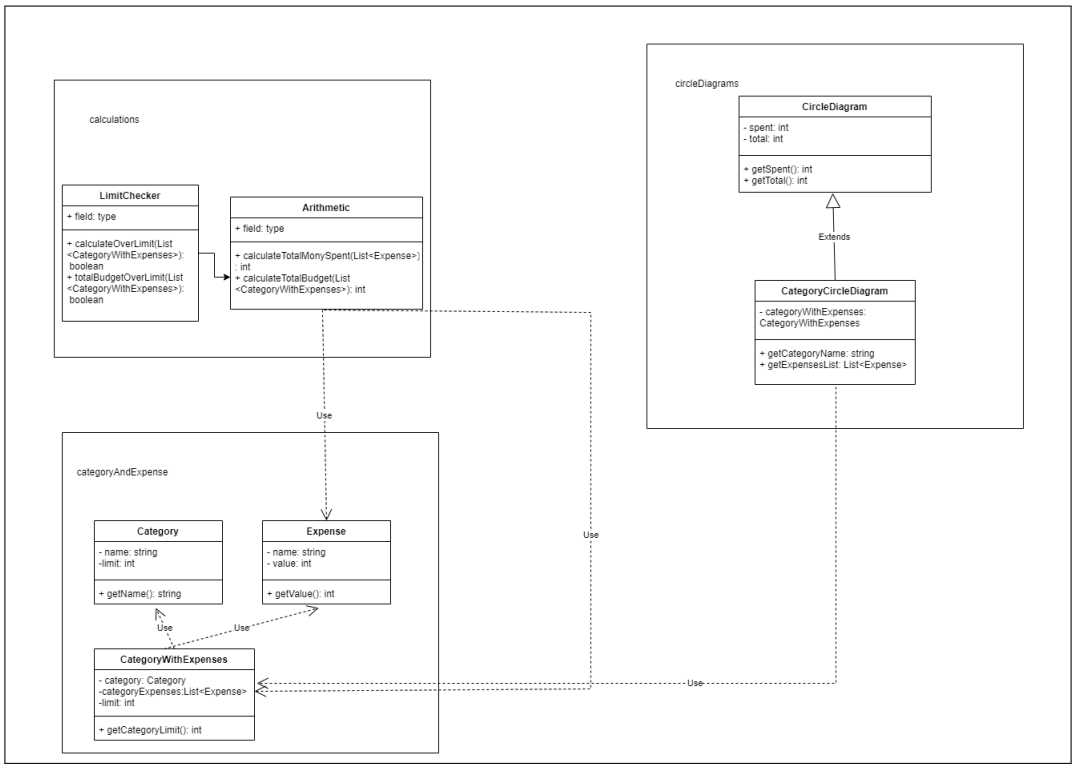

Figure 2: Model class diagram.

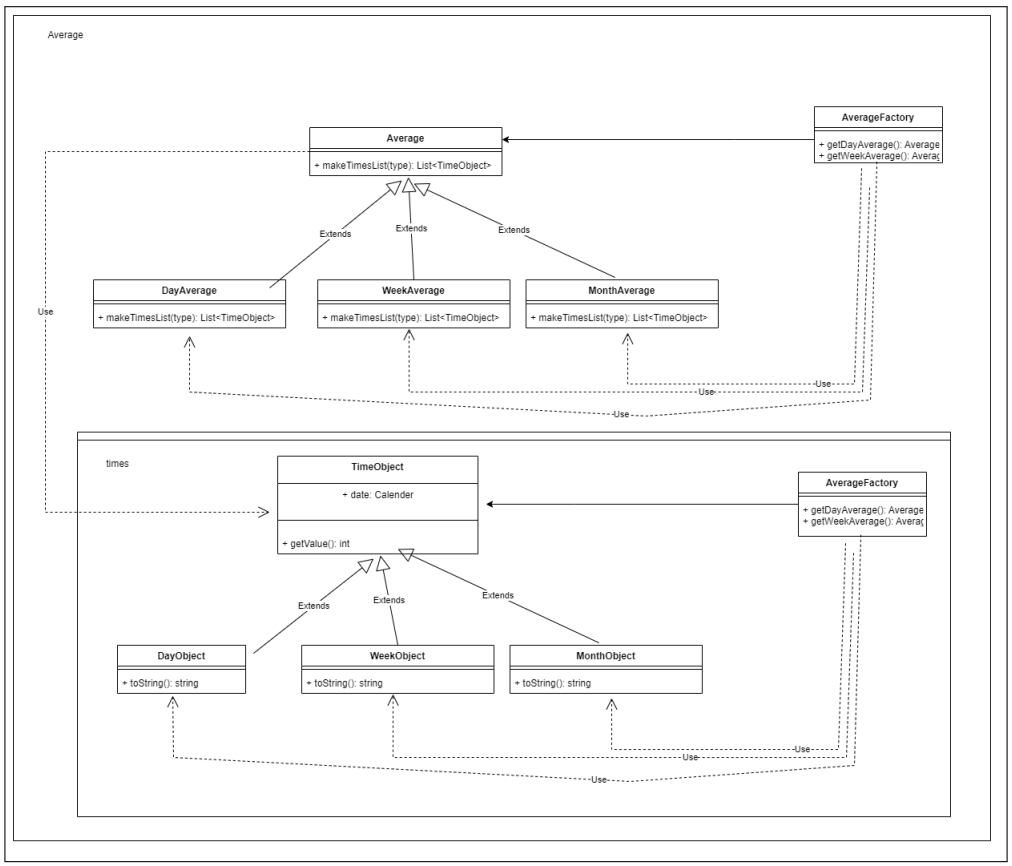

Figure 3: Model class diagram, average package.

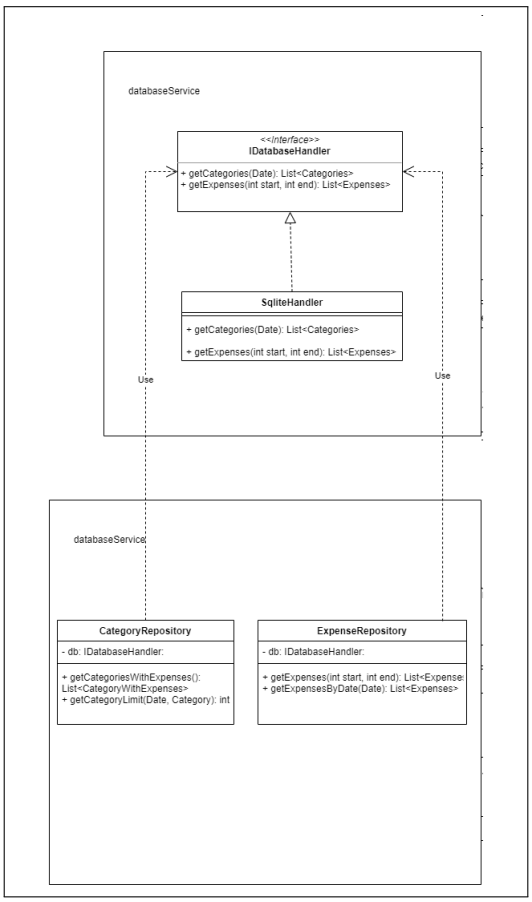

Figure 4: Model class diagram, database related classes.

3.2.2 Home package class diagram

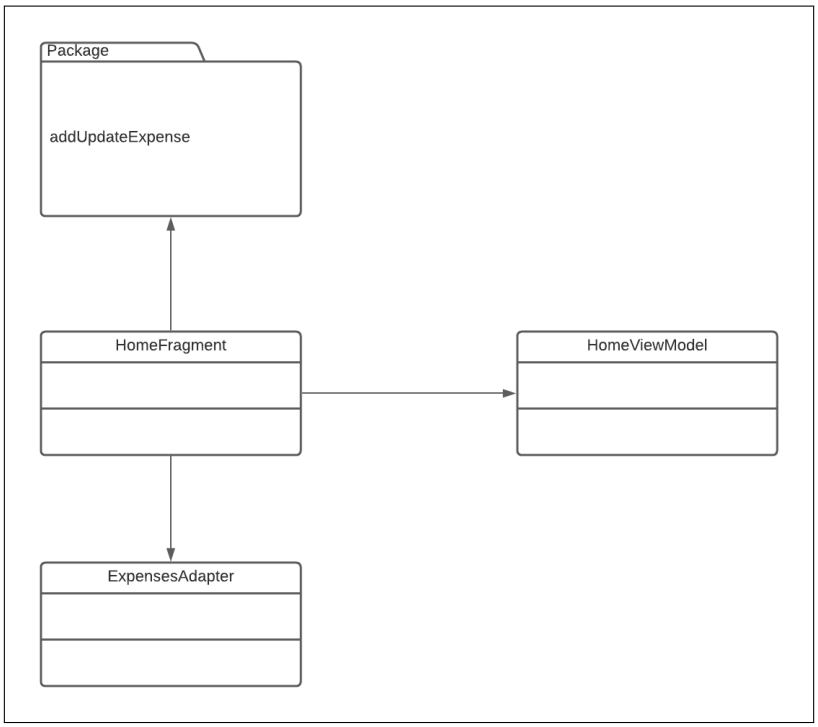

Figure 5: Home class diagram.

addUpdateExpense package class diagram

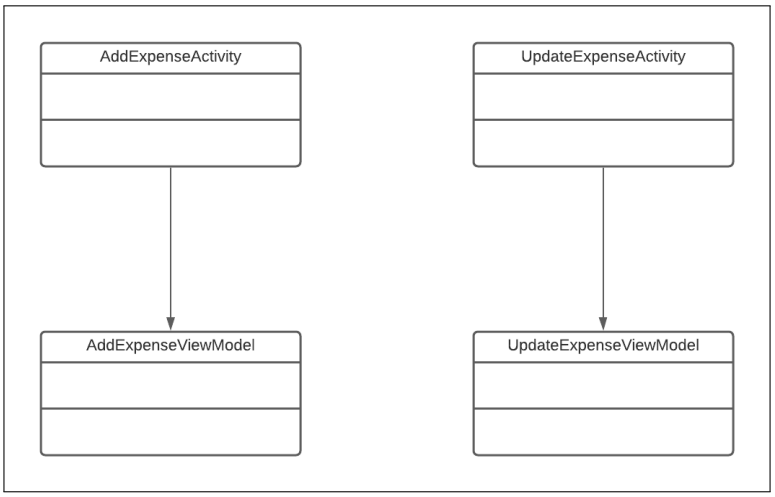

Figure 6: Add update class diagram.

3.2.3 Analysis package class diagram

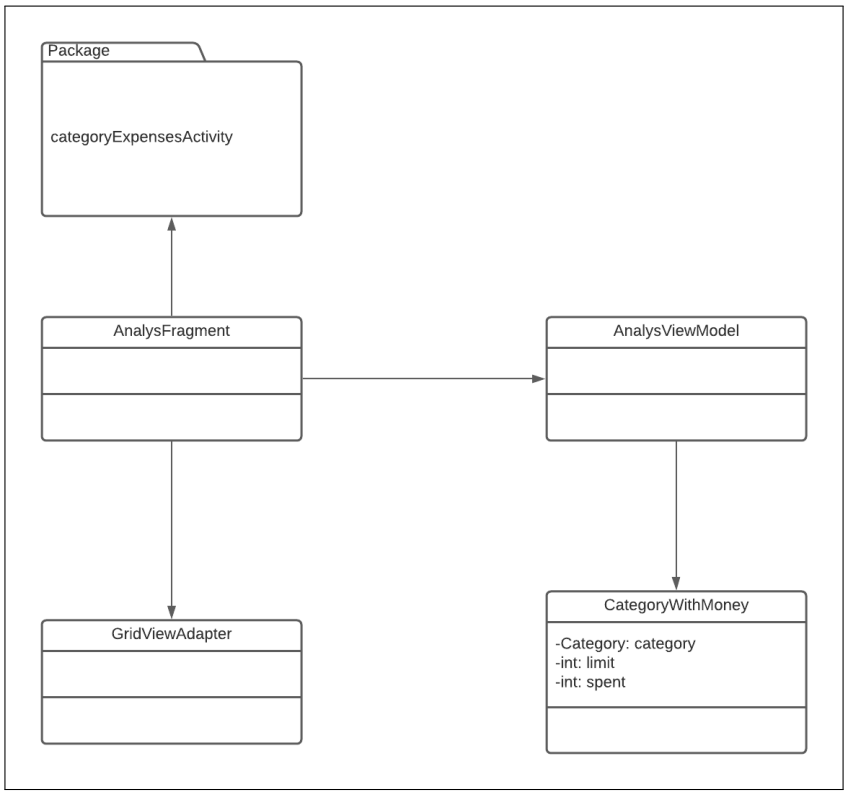

Figure 7: Analysis class diagram.

categoryExpensesActivity package class diagram

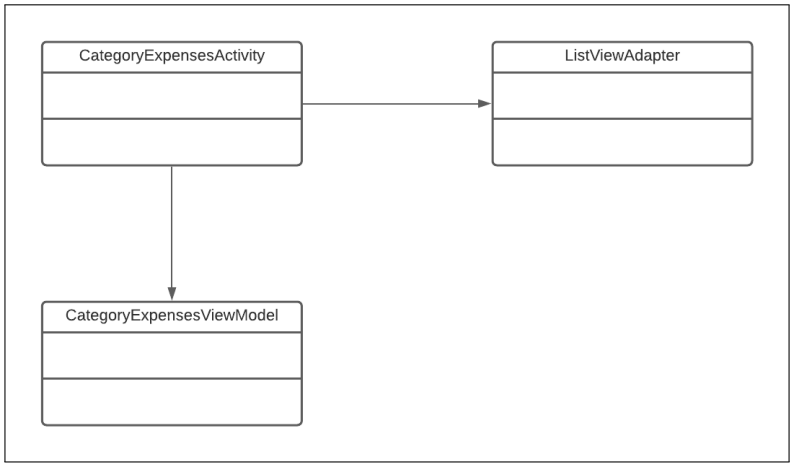

Figure 8: Category expenses class diagram.

3.2.4 Categories package class diagram

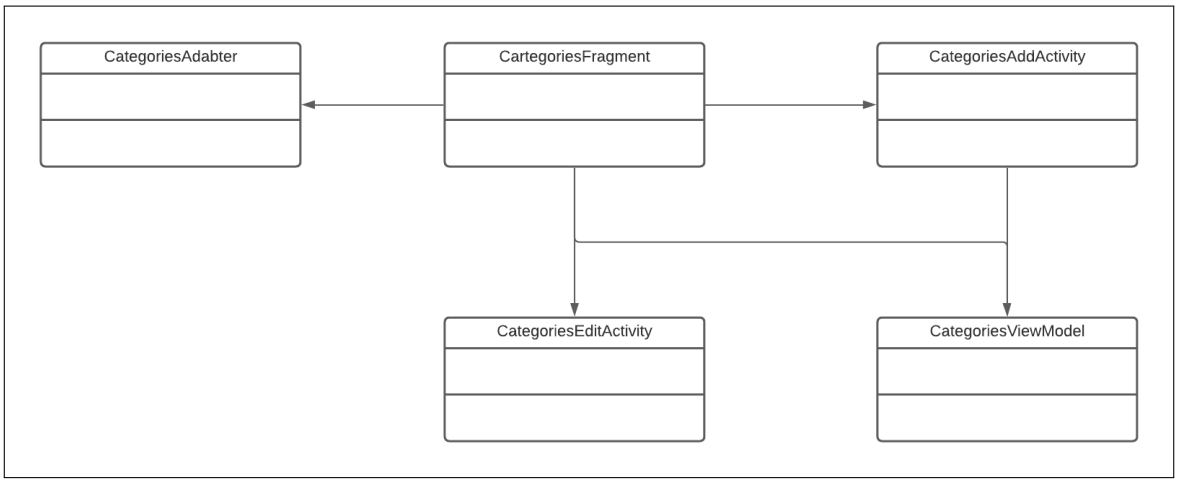

Figure 9: Categories class diagram.

3.3 Sequence diagram

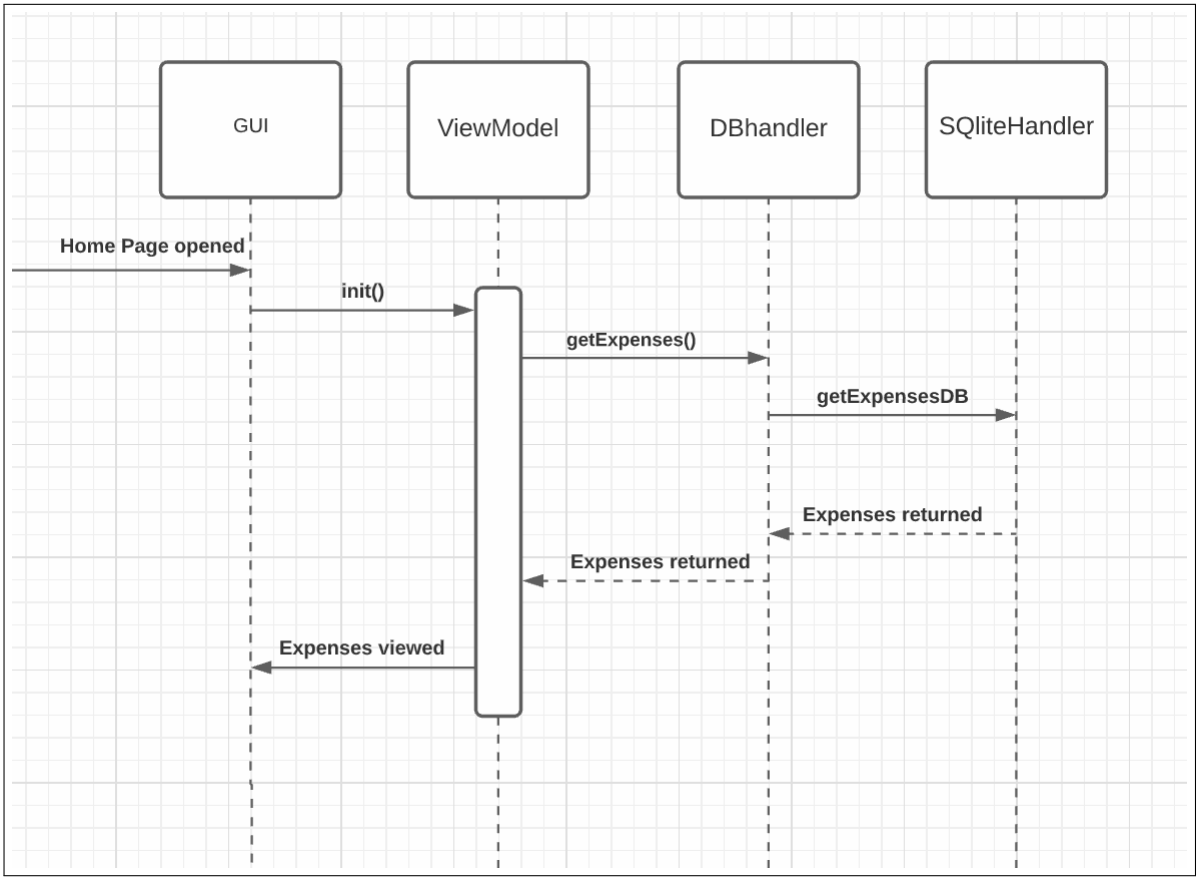

Figure 10: Home Sequence diagram.

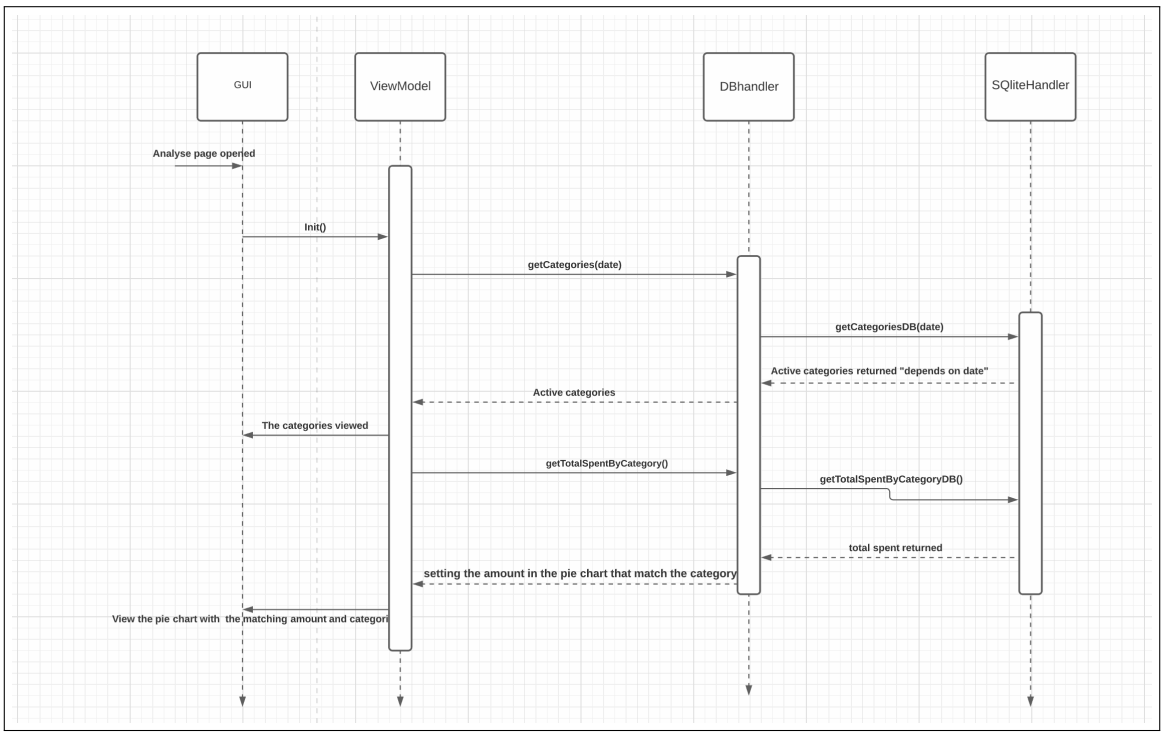

Figure 11: Analysis Sequence diagram.

3.4 Design patterns

- Model-View-ViewModel pattern: Applikationen följer MVVM designmönster. Databasen är inte direkt kopplat till klient koden. Klassen ”DBhandler” fungerar som ett lager mellan databasen och klientkoden. Men vi vill inte heller att DBhandler ska ha kopplingar med View klasser, dvs klasser som har renderingsansvar. Därför kommer ViewModel lager för att hämta data från databasen genom DBhandler och skicka den vidare till View. Dessutom kommer det att lyssna på de ändringar som sker i UI för att uppdatera motsvarande data i databasen. Varje View-klass kommer att ha sin egen ViewModel klass som har ovannämnda ansvar.

- Bridge pattern: Bridge designmönster används för att förenkla bytet av databasen. Vi har ett interface som har alla metoder som vår DBHandler behöver. SQLiteHandler klassen ärvar detta gränsnitt. DBhandler beror inte på SQLiteHandler där samtliga SQLite-instruktioner finns, men har istället ett attribut av IDatabasHandler (beror på abstrktion) som pikar i sin tur på ett SQLite-objekt vid runtime. Detta ger oss möjlighet att byta databasen när som helst då det bara är att skapa en ny databasklass och byta referensen som attributen pikar på till den nya databasen.

4 Persistent data management

Det finns data som inte ändras i applikationen. Applikationen kommer att skapa defaulta kategorier som användaren kommer inte att kunna ta bort eller ändra, den enda som användaren kommer att kunna ändra på defaulta kategroier är budgeten. De ska finnas som vanliga kategorier i kategorier tabellen men applikationen kommer att hardkodas så att den vet att tillexempel första fem kategorier ska inte ändras och viewn kommer inte att låta användaren ändra de.

Applikationen kommer att använda bilder också som inte ändras. Det ska finnas ett stort antal bilder som finns på resources mappen, de ska användas för kategorier alltså varje kategori ska ha en bild. Användaren ska se de när hen skapar en ny kategori där hen ska kunna välja en bild för den nya kategorin och varje kategori ska innehålla ett referens nummer i databasen som pekar på en specifik bild.

5 Quality

Test till DBHandler är gjort men vi kan inte se coverage procent av någon anledning. Detta ska lösas i kommande iterationer. Testerna är gjorda med Junit och de finns i klassen DatabaseClassTest. Samtliga metoder i DBhandler har testats på olika sätt för att täcka samtliga möjliga fall och därmed se till att dessa metoder beter sig som förväntat. PMD analys plug-in har använts för att undersöka om koden följer Java standard principer. Bild på resultat finns i referens, Figure 1.

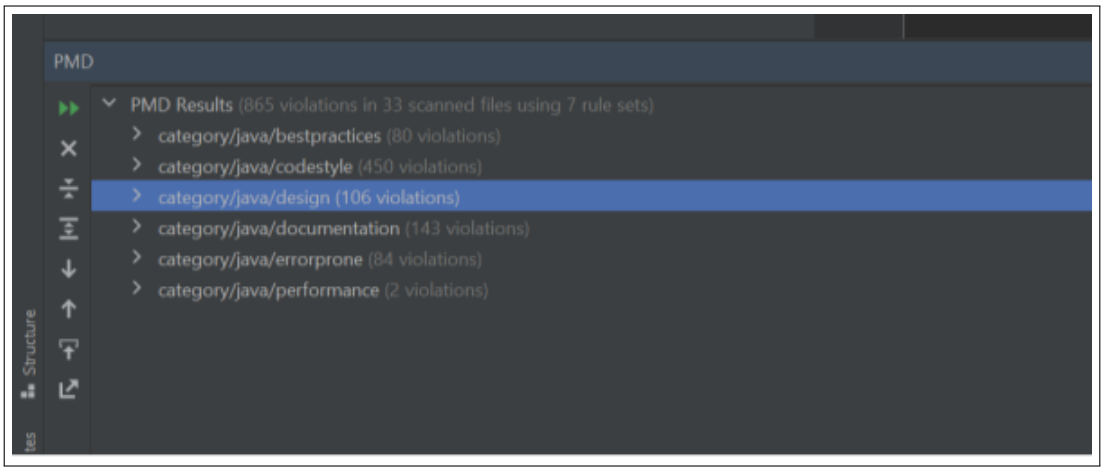

Figure 12: PMD.

6 References

https://refactoring.guru/design-patterns
https://codingwithmitch.com/blog/getting-started-with-mvvm-android/