ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - ВАРНА

ФАКУЛТЕТ ПО ИЗЧИСЛИТЕЛНА ТЕХНИКА И АВТОМАТИЗАЦИЯ

****

Катедра „Софтуерни и интернет технологии“

**КУРСОВ ПРОЕКТ**

**Тема:**

Проектиране и реализиране на система за отчитане на ежедневни разходи

**Изготвили:**

Михаел Бердж Агопян ФН: 18621765

Георги Грозданов Георгиев ФН: 18621746

**Специалност:** Софтуерни и интернет технологии

ТУ Варна, 2021 г. Ръководител:

/ас. Д.Димитров/

Съдържание

[Увод 3](#_Toc91798728)

[Анализ на проблема 4](#_Toc91798729)

[Избор на технологии 6](#_Toc91798730)

[Проектиране 8](#_Toc91798731)

[UX Research 8](#_Toc91798732)

[Анализ на конкуренция 11](#_Toc91798733)

[Персони 16](#_Toc91798734)

[Карта на съпричастност 18](#_Toc91798735)

[Essential Use Case Diagram 19](#_Toc91798736)

[UML Diagram 20](#_Toc91798737)

[Coremodels 21](#_Toc91798738)

[Sitemap 23](#_Toc91798739)

[Moodboard 24](#_Toc91798740)

[Лого 25](#_Toc91798741)

[Реализация 26](#_Toc91798742)

[Архитектура на системата 26](#_Toc91798743)

[Entity 27](#_Toc91798744)

[DAO (Data Access Object) 30](#_Toc91798745)

[Repository 32](#_Toc91798746)

[Viewmodel 38](#_Toc91798747)

[LiveData 41](#_Toc91798748)

[Тестове и резултати 42](#_Toc91798749)

[Заключение 43](#_Toc91798750)

# Увод

В днешно време, управлението на финансите е доста важна част от нашето битово ежедневие. С всяка година ежедневието ни става все по-забързано, появяват се нови и различни отговорности и сякаш се налага да вършим всичко с по-голяма скорост и прецизност от преди. За щастие, благодарение на бързото развитие на технологията, ни се предоставят все повече нови възможности и функционалности, както имаме и свободен достъп до все повече информация, което създава предпоставки за по-високо качество, продуктивност и увереност в нашите действия.

Навременната, актуална и точна информация е от особено важно значение за управлението на нашите финанси. До голяма степен финансовото ни състояние се определя и от стратегическия финансов план, който сме изградили, а не само от получаването на приходи. От изключително голяма важност е човек, да умее да управлява правилно и най-вече ефективно своите разходи. Важно е да бъдат избегнати ненужните, непродуктивни действия и също така е от изключителна важност намаляването на шансовете за допускане на грешка до минимум, тъй като това би означавало загуба на финансови средства за нас. Скоростта на калкулацията на сметките трябва да бъде по възможност най-бърза и правилна. Актуалната и навременна информация също е от ключово значение, тъй като винаги бихме искали да разполагаме с наличното към настоящия момент, а не да се изчисляват приблизителни суми.

Ефективността при управлението на личните ни финанси, би могла да бъде постигната, чрез използването на определени инструменти за това. Като един от най-удобните и ефективни начини за постигане на тази съответна цел, е чрез използването на мобилно приложение, за управление на разходите и приходите. Такъв тип мобилно приложение, ще ни предоставя необходимата сигурност, точност, коректност и актуална, и навременна информация във всеки един момент. С този инструмент си гарантираме повишаването на продуктивността и ефективността при изчисляването на нашите разходи и приходи.

# Анализ на проблема

С проектирането и реализирането на мобилно приложение за управление на личните финанси, бихме искали до някаква степен да автоматизираме и улесним процеса по калкулация и изчисляване на разходи и приходи, както и да се погрижим за безопасното и сигурно съхранение на тази информация. Важно е информацията да бъде винаги лесно достъпна, да бъде актуална, точна, сигурна и правилна. Представянето на статистика под формата на кръгови диаграми също е от голямо значение, с цел по-лесното възприемане на информацията от човешкото око. Приложението е съсредоточено към повишаване на продуктивността и ефективността при процеса на управление на финансите, с цел помагане и улесняване на усилията на крайните потребители.

Първия проблем, който се решава, използвайки това мобилно приложение е свързан с правилните изчисления на разходи или приходи, като това е от особено важно значение, тъй като по този начин се елиминира шанса за грешни суми и съответно крайните потребители се предпазват от некоректни или неточни бъдещи инвестиции или планове.

Следващата полза от използването на тази система за отчитане на ежедневни разходи или приходи е именно дигиталния процес. Спестяват се влагането на ръчни усилия, за пресмятането на сумите и съответно записването и съхранението им на хартиен носител, което е несигурно, непродуктивно и неефективно. Чрез използването на технологията, информацията се съхранява на сигурно място, като винаги е налична във всеки един момент. Автоматизира се процеса на пресмятане, като това спестява доста усилия и време на крайните потребители, като най-важната особеност, е че се повишава продуктивността. Друга положителна черта, е че се изключва използването на хартиени носители, следствие на това се решават проблеми свързани със съхранението на тези носители, конфиденциалността на информацията и съответно ненавременното наличие на информацията при нужда.

Друга отличителна особеност на мобилното приложение е, че е лесно за използване с опростен интерфейс, включващ всички необходими функционалности за използване и управление на нашите финанси. Системата предоставя възможности за категоризация на нашите приходи и разходи, и съответно подкатегоризация с наличието на съответни икони, индикиращи целта или по-скоро визуалното представяне на сферата или категорията, към която попада нашия приход или разход.

Предоставят се възможности за създаване на различни сметки или пера, в различни валути за проследяване на определени финансови средства в различни категории или сфери. Като има възможност за представяне на информацията за разходите и приходите като статистика за определени сметки под формата на кръгови диаграми. Допълнителна функционалност е наличието на търсене и филтрация на информацията, което значително ускорява процеса по търсене на конкретните сметки или осъществени транзакции (приходи или разходи).

# Избор на технологии

За реализацията на нашата система за управление на личните финансови средства, сме избрали следните технологии:

* **Мобилна платформа ( Android OS )** – Реализацията на нашата система, решихме да я изградим под формата на мобилно приложение за Андроид Операционна система. Избрахме мобилната платформа тъй като в днешно време рязко нараства броя на потребители, които използват смартфони и съответно все по-голяма част от потребители предпочитат използването на мобилни приложения вместо използването на настолен компютър или лаптоп, тъй като това е доста по-удобно и ефективно в някои случаи. Смартфоните стават все по-популярни като все повече потребители инвестират време използвайки тях, вместо другите платформи. Избрахме да реализираме приложението на Android OS, тъй като в днешно време за крайните потребителите е много по-лесно откъм финансова гледна точка да се сдобият с Андроид телефон отколкото iPhone с неговия iOS. Към Юли 2021 г. повече от 70% от мобилните устройства са поддържани на Android OS. Следователно пазарът тук е доста по-голям. Друга негативна черта за реализацията на iOS е бариерата за навлизане за разработка на приложение. Наложително е използването на скъпа техника, която поддържа MacOS както и наличие на iPhone за истинска тестова среда.
* **Интегрирана среда за разработка ( Android Studio IDE )** – Избрахме разработката на нашето мобилно приложние да се осъществи на **Android Studio IDE,** защото е софтуер разработен и препоръчан за използване от **Google**, както и факта че предоставя функционалности като Code Completion, поддържа Gradle, възможност за преглеждане едновременно на XML кода и самия XML Layout на приложението, докато при Eclipse това не е възможно, трябва да се избере само единия вариант при проектиране на интерфейс и да се сменя постоянно между двете възможности, за да се проследяват промените по интерфейса и XML кода.
* **Език за програмиране ( JAVA 8 )** – Избрахме за разработка на нашето мобилно приложение да използваме **JAVA 8**, тъй като към настоящия момент разполагаме с повече знания и умения на този програмен език и бихме искали да се съсредоточим повече към разработката на приложението вместо изучаването другия популярен програмен език **KOTLIN**. Той е сравнително нов език, който тепърва навлиза и поддържа ООП и Функционално програмиране, но времето за компилация или Compilation Time на JAVA е около 15-20 % по-бързо, също така има много повече информация и ресурси на разположение при проблеми възникнали на JAVA, тъй като това е програмен език на 26 години. Избрахме версия 8 на JAVA, защото разполагаме с голям брой библиотеки, които можем да използваме наготово, тъй като от версия 9 и нагоре, навлиза концепцията за модулно програмиране и голяма част от библиотеките липсват и се налага ръчното изтегляне и конфигуриране. Ограничаваме ме се откъм функционалности като например, „Ламбда изрази” и „променливи с автоматичен тип”, но не е нещо изключително фатално.
* **База данни ( SQLite )** – За нашите цели, **SQLite база данни** идеално се вписва, поради факта, че ние създаваме за момента офлайн мобилно приложение с локална база данни, а не система с достъп до сървър. **SQLite** е free open-source и не е нужна инсталация на допълнителен софтуер за използването на тази технология. Също така, тази база данни е доста лесна за използване, бърза е и е значително малка и лека като размер, като е нужно само импортирането на определени библиотеки за използването на тази технология.
* **Room Persistence Library** – Това е **ORM библиотека** ( **оbject Relational Mapping library** ), представена от **Google**, която предоставя цялостен абстрактен слой над **SQLite База данни,** който се грижи за конвертирането на информацията в Java обекти и обратно, като улеснява максимално работата с база данни откъм начална конфигурация и започване на действителния процес на работа.
* **Система за контрол на версиите ( GitHub ) –** Позволява на много програмисти да работят едновременно върху един и същ проект, без да си пречат взаимно, като се запазва история на всички направени промени по текущия проект, като се разполага и със функционалности за разрешаване на конфликти при работа върху едни и същи файлове.

# Проектиране

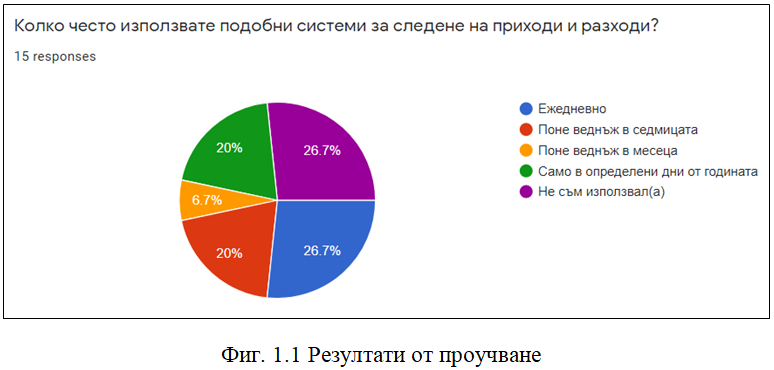
## UX Research

Изследването на потребителските нагласи за потребителското изживяване е ключова част от разработката на софтурния ни продукт и използва метода за UX research *Customer feedback* (отзиви на клиентите/потребителите). Бе извършено с помощта на Google Forms за събиране на отговорите на потребителите и извеждане на резултатите в графичен вид.

Изследваната група се състои от 15 студенти от 4 курс от специалност Софтуерни и интернет технологии в Технически университет – Варна, които отговориха на 5 въпроса със затворен отговор. Първите 4 от тях бяха задължителни, а 5-ят получиха само някои от анкетираните.

Изследването даде следните резултати:

1. На въпроса колко често използват системи за следене на приходите и разходите си (Фиг. 1.1), малко над ¼ от анкетираните заявиха, че ги използват ежедневно. Същият процент никога не са използвали подобни системи.



1. На въпроса коя платформа е най-подходяща за едно приложение за отчитане на приходи и разходи (Фиг. 1.2), 4/5 от анкетираните посочват мобилната.



1. На въпроса каква операционна система използва телефонът им (Фиг. 1.3), над 85% посочват Android.



1. На въпроса дали са склонни да заплатят за подобно приложение (Фиг.1.4), анкетираните имаха възможност да посочат и повече от един отговор. Преобладаващият отговор е „Не бих платил(а)“ с 40%. Малко над ¼ биха заплатили, ако приложението е досатъчно качествено или ако чрез заплащане се премахват рекламите. 1/5 биха заплатили, ако това би отключило нови функционалности.



Анкетираните, отговорили, че операционната система на смартфона им е Android, получиха допълнителен въпрос.

1. На въпроса коя е версията на Android, която използват (Фиг. 1.5), с най-голям дял (30%) се откроява Android 10. На второ място се нареждат Android 9 Pie и Android 11 с по 23%. Никой от анкетираните не използва версия 5 Lollipop или по-ниска.



## Анализ на конкуренция

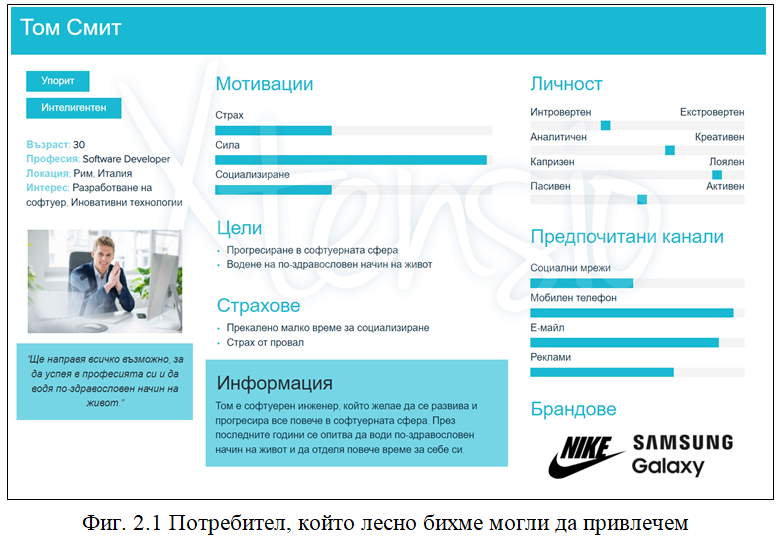
## 

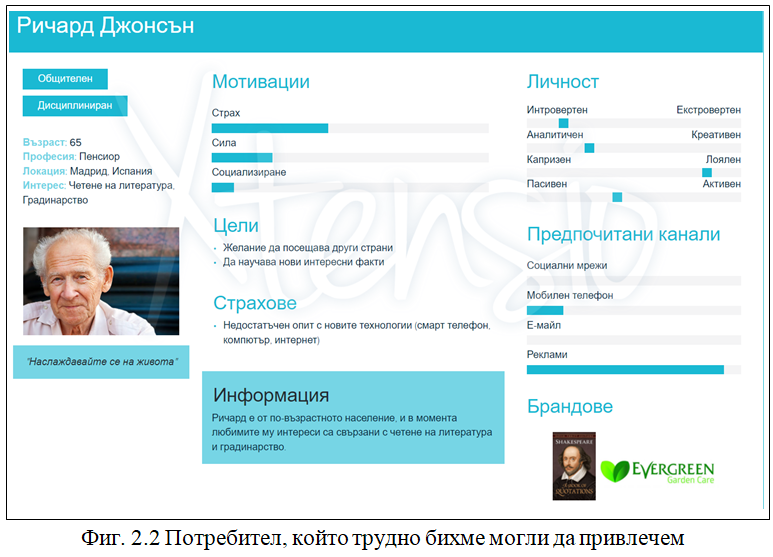
|  |  |  |
| --- | --- | --- |
| Критерии | Monefy – Budget Manager and Expense Tracking App на *Reflectly* | Money Manager Expense & Budget на *Realbyte Inc.* |
| Информация за състоянието на системата | Системата предоставя семпъл и ефективен интерфейс с добре групирани елементи, който позволява интуитивна работа с нея дори за потребители, които използват подобно приложение за пръв път. Основна функционалност е добавянето на записи за разходи и приходи и категоризирането им. Предоставя лесен начин за справки за период от време. Функционалности като синхронизания между множество устройства, задаване на валути, както и козметични промени като Dark mode не са включени в безплатната версия. Има функция за експорт на записите в избран формат | Системата предоставя възможност за отчитане на приходи и разходи по категории, както и за прехвърляне на средства от една сметка в друга. Записите могат да се представят графично за избран период от време. Всички функционалности са включени в безплатната версия, като платената единствено премахва рекламите. Освен начини за експорт на записите в избран формат, системата позволява изпращане по имейл, като автоматично прикачва .sqlite файл, който може да бъде restore-нат на друго устройство. Друг начин за backup е синхронизиране с Google Drive с възможност за restore от същия |
| Съвпадение на нуждите на реалния свят с предоставяните от приложението | Системата предоставя основни функционалности като избиране на вида на разходите и приходите, но лиспват подвидове, напр. има разходи за автомобил, но няма начин потребителят да конкретизира точно за какво е разхода – За ремонт? Гориво? И т.н. Има начин за добавяне на нови категории, но в платената версия, и дори това не решава проблема с липсата на йерархия (подкатегории). Липсата на избор на валути може да отблъсне потребители, които имат сметки в различни валути. | Приложението предоставя повечето функционалности, с които да покрие потребителските нужди, но липсват подкатегории на категориите приходи и разходи. Функционалността за добавяне на нови категории е полезна, но не заменя нуждата от йерархична структура. |
| Потребителски контрол и свобода на действията | Потребителите имат свободата да избират категория на разходите/приходите, да избират интервал от време за справки, но въпреки че системата предоставя меню с около 20 опции, повечето от тях (за синхронизация, backup, избиране на валута, опции за по-голяма сигурност) остават заключени в безплатната версия | Потребителите имат свободата да избират категория на разходите/приходите, да избират интервал от време за справки, да избират цветова гама на интерфейса на приложението. Има налични всички валути и за всяка сметка те могат да се променят. |
| Консистентно представяне на информацията и стандарти | Информацията е нагледно представена в основния, начален екран на приложението. Разходите и приходите са представени графично в кръгова диаграма с празен център с текст в него, която не е идеалният вариант за целта и би било добре да има поне опция за представяне с друг вид диаграма | Името на таба в navigation bar за преглед на списъка и добавяне на нов приход/разход е Transactions, но тъй като е дълга дума е съкратено на „Trans.”, което не е удачно за толкова ключов елемент от интерфейса.  Записите на същия основен екран са представени в списък, но както в него, така и на други места в приложението текстът на места е труден за четене, в сив вместо черен цвят, с малък размер и шрифт, който в тази ситуация прави текста още по-труден за четене  Графично представените справки са под формата на кръгова диаграма с добре обозначено това кой цвят какво преставлява |
| Предотвратяване на грешки и тяхното прихващане | Системата не позволява добавяне на записи без минимална въведена информация (в случая сумата на прихода/разхода – категорията му се избира предварително) | Системата не позволява добавяне на записи без минимална въведена информация (в случая сумата на прихода/разхода, категорията и разхода) |
| Осигуряване на интерфейс пред действие по памет | Основните функционалности са лесно достъпни, нагледни и говорят за себе си | Основните функционалности са лесно достъпни, нагледни и говорят за себе си |
| Гъвкавост и ефективност | Липсват подкатегории за приходите и разходите, и няма функционалност да се добавят такива. Дори в платената верия, която предоставя множество други подобрения, на фона на липсата на подкатегории те изглеждат тривиални. Има пълна гъвкавост при отчетите за период – за деня/седмицата/месеца/годината, както и за зададен интервал от-до дата. Липсата на валути в безплатната версия и това намалява гъвкавостта й за една част от потребителите | Липсват подкатегории за приходите и разходите, и няма функционалност да се добавят такива. Налични са всички валути. Има пълна гъвкавост при отчетите за период – за деня/седмицата/месеца/годината, както и за зададен интервал от-до дата. Броят сметки, които могат да се добавят е ограничен в безплатната версия. Няколкото начина за backup дават свобода на избор и са обяснени нагледно в секцията за помощ. Към запис за приход/разход може да се добави снимка |
| Минималистичен дизайн | Приложението има минималистичен, на места твърде семпъл дизайн. Анимации при интеракция с основните бутони за добавяне на приход или раход лиспват, те са представени като големи кръгове с оцветен border и бяла сърцевина с текст, което не е оптимално имайки в предвид предназначението им. На места са използвани излишни анимации като леко подскачащ бутон при отваряне на форма. Някои от иконките, с които са представени видовете разходи са неудачни, като например живачен термометър за разходи за здраве | Приложението има минималистичен дизайн, но цветове, различни от бял фон с черен текст или иконки със сиви очертания почти не се срещат. На основния екран текста за стойността на приходите е в тъмносин цвят вместо по-интуитивния зелен. Разходите са в червено, но същото червено се използва като основен цвят на приложението за обозначаване на текущия таб и на други места и основния бутон за добавяне на запис също е в червено, при положение че записа може да бъде не само разход, но и приход или трансфер. Липсва логика в избора на цветове. На почти половината от екраните има реклами, за чиито премахване може да се заплати. Рекламите не са удачни за такъв вид приложение. Текстът на места е труден за четене. Анимации липсват, има бързи transition-и при превключване между табовете, но като цяло усещането за интерактивност липсва |
| Обратна връзка | Възможно е единствено оценяване на приложението в Google Play Store, за което има бутон в едно от менютата на приложението | Освен оценяване на приложението в Google Play Store, то има и бутон Feedback, който води до екран с уебсайта и имейла на компанията-разработчик и опции за Feedback, при което се избира приложение, което да се отвори, за да продължим. При отваряне с Gmail автоматично се зарежда темплейтно писмо с уникален код и информация за версията и устройството, но в дългия списък с възможни приложения влизат и такива като Viber, които при избиране се отварят и не водят до нищо. |
| Помощ и документация | При първо добавяне на запис има напътствия под формата на подсказки под полета и бутони, но след това лиспва начин да се видят помощни ресурси | Системата няма въвеждащи подсказки при първо използване, но има секция Help, в която има описани промените за всеки ъпдейт, както и параграфи с често срещани въпроси, които нагледно с помощта и на скрийншоти обясняват как да ползваме приложението и разясняват за какво служи всеки елемент от интерфейса |

## Персони

На етапа на проектирането на системата за отчитане на ежедневни разходи и приходи е изключително важно да бъдат идентифицирани персоните, за потребителят който най-лесно бихме могли да привлечем (Фиг 2.1) и съответно този който най-трудно бихме могли да привлечем (Фиг 2.2), тъй като това би ни дало отговори на доста важни въпроси и съответно би определило и бъдещото развитие и разработване на нашия продукт.

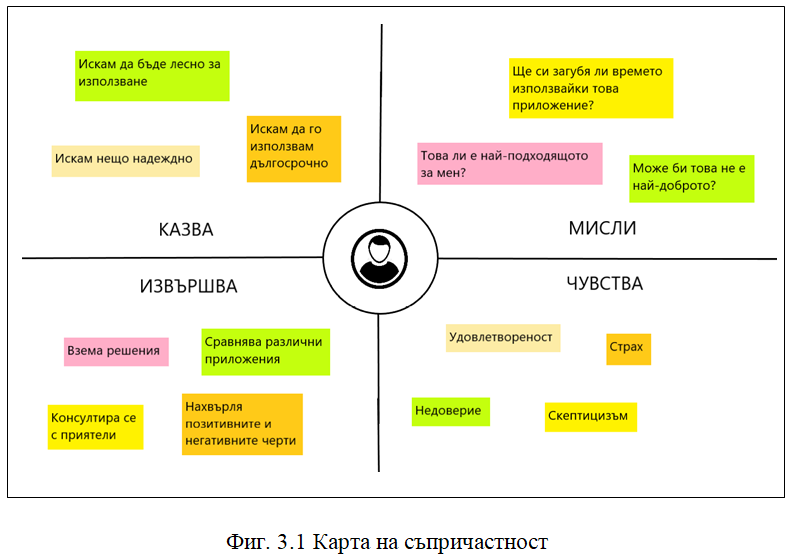
Този тип инструмент ни позволява да опознаем аудиторията, за която разработваме продукта на едно доста по-добро ниво, като се стремим да се съобразяваме с техните интереси и нужди, и съответно да предоставим едно възхитително потребителско изживяване. Колкото по-детайлни и информативни са персоните, толкова по-лесно ще можем да анализираме и да удовлетворим нуждите на крайните потребители.





## Карта на съпричастност

Това е инструмент, който ни помага да разберем действителните мисли и актуалното поведение на крайният потребител (Фиг. 3.1), като това до голяма степен помага на екипа ни, разработващ системата за отчитане на ежедневни разходи и приходи да навлезе в мислите на групата от потребители, за да бъдем по-информирани и подготвени при разработването на системата.



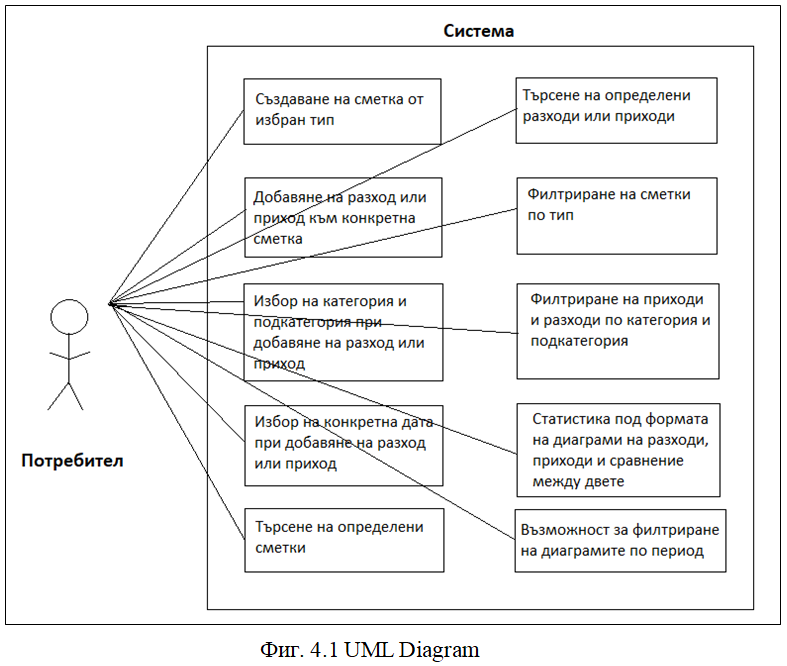
## Essential Use Case Diagram

При разработването на един софтуерен продукт, е от изключителна важност да бъдат уточнени детайлно изискванията и функционалностите от клиентите, още при началния етап на проектиране. За тази цел инструменти като Essential Use Case Diagram, са подходящи , за да се нахвърлят базовите изисквания и функционалности, както и кои са ключовите ни актьори и достъп до каква функционалност ще имат те. Този етап не бива да се пренебрегва тъй като липсата на пълна информация за изискванията на клиентите води до удължаване на крайните срокове, загуба на ресурси като пари и време, и съответно разочарование от страна на клиента.

|  |  |  |
| --- | --- | --- |
| **Функционалност** | **Потребител** | **Система** |
| Създаване на сметка от избран тип | Да | Осигурява функционалността |
| Добавяне на разход или приход към конкретна сметка | Да | Осигурява функционалността |
| Избор на категория и подкатегория при добавяне на разход или приход | Да | Осигурява функционалността |
| Избор на конкретна дата при добавяне на разход или приход | Да | Осигурява функционалността |
| Търсене на определени сметки | Да | Осигурява функционалността |
| Търсене на определени разходи или приходи | Да | Осигурява функционалността |
| Филтриране на сметки по тип | Да | Осигурява функционалността |
| Филтриране на приходи и разходи по категория и подкатегория | Да | Осигурява функционалността |
| Статистика под формата на диаграми на разходи, приходи и сравнение между двете | Да | Осигурява функционалността |
| Възможност за филтриране на диаграмите по период | Да | Осигурява функционалността |

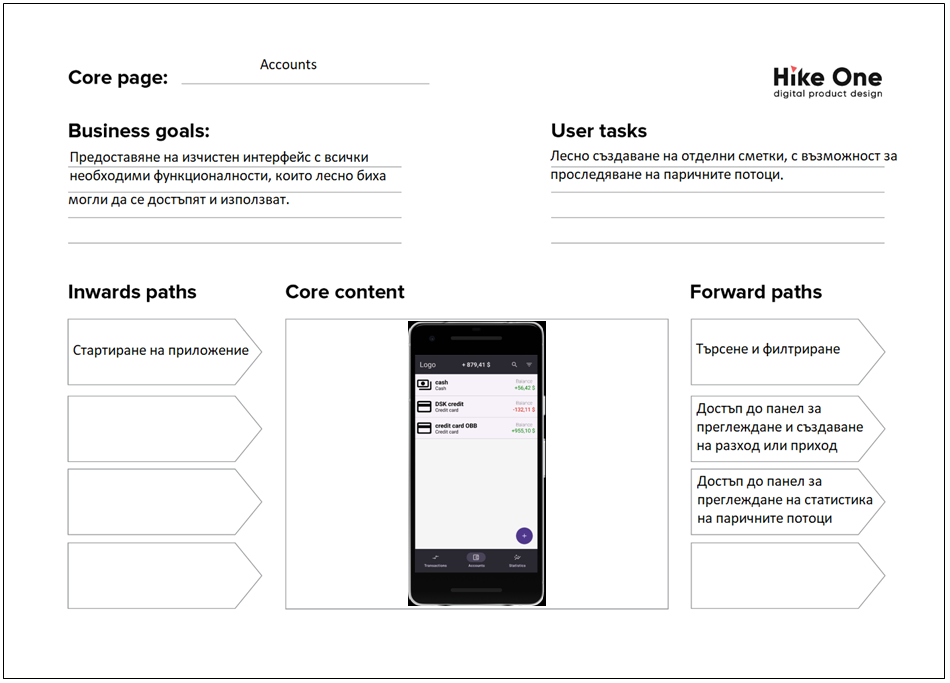
## UML Diagram

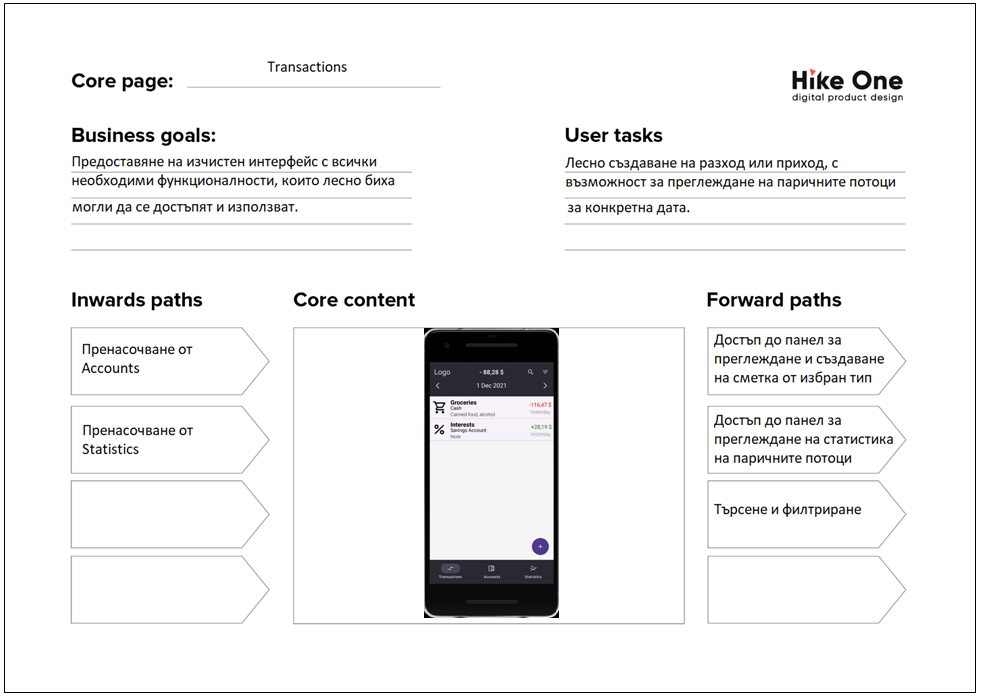
Този тип диаграма (Фиг. 4.1) е също част от съвкупността от инструменти, които бихме могли да използваме при начална идентификация на изискванията и функционалностите, като разликата тук е, че имаме по-добро визуално представяне, с цел улесняване разбирането на системата от крайните потребители. Тук заявяваме, кой е нашият главен актьор, в нашия случай това е само един и това е крайният потребител, който ще използва системата, за удовлетворяване на своите нужди. Системата е другият актьор, която е представена на диаграмата като правоъгълник, който обхваща цялата функционалност и осигурява нейното възпроизвеждане, при иницииране на действие от страна на потребителя.

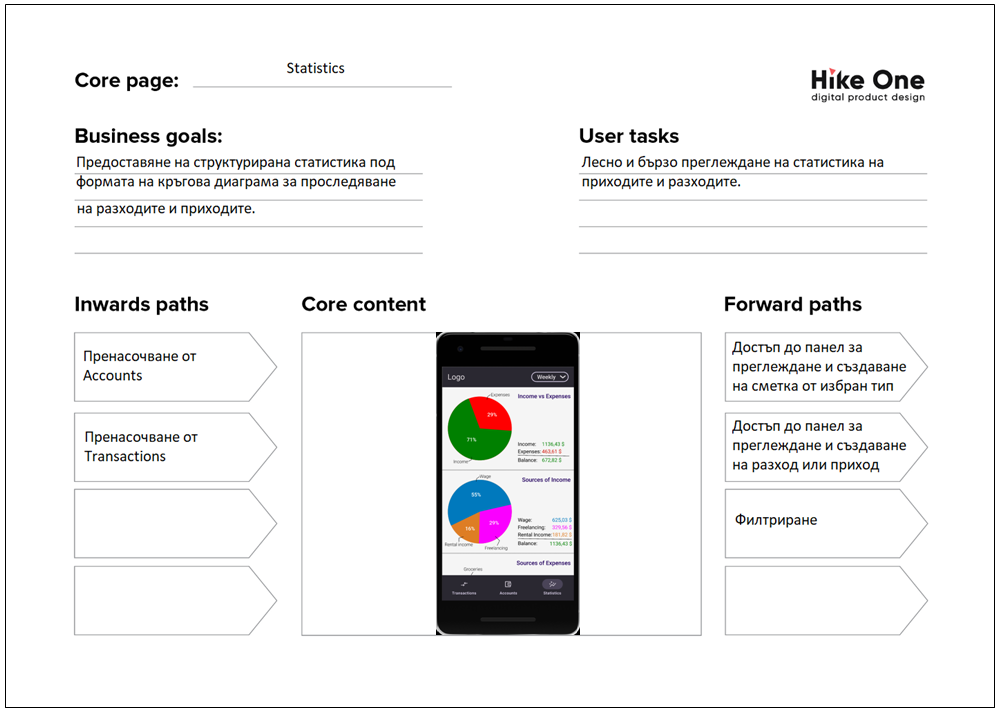


## Coremodels

Следващата стъпка от проектиране на системата, е разбработването на така наречените „Coremodels”, чрез които нахвърляме от лявата страна бизнес целта на този модул, пътищата, които водят до него, а от другата страна съответно пътищата до които, той, ще ни отведе както и информация за целта на потребителя, която бихме искали да удовлетворим. Като по средата се представя визуално как би изглеждал прототипа на нашата система. Този инструмент е доста полезен при бъдещо изграждане на прототип, а в следствие и от полза за изграждане на „Sitemap”, за което ще поговорим по-късно. В нашия случай, системата ни има 3 основни визуални части или табове, като това са съответно за Сметки (Фиг. 5.1), Транзакции (Фиг. 5.2) и Статистика (Фиг. 5.3). Като главният или „Main” таб е този за сметките на потребителите.

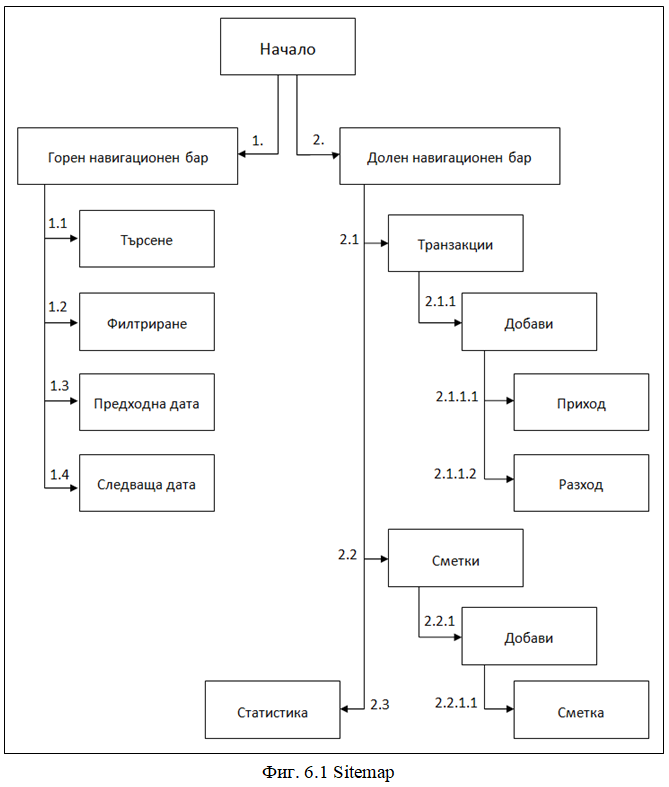






## Sitemap

Друг полезен инструмент на етапа на проектиране, който сме използвали е „Sitemap” (Фиг. 6.1), чрез който след като имаме готови „Coremodels” и прототип, бихме могли да разработим една дървовидна структура, която пряко да рефлектира или да бъде отражение на нашата система, и да служи като ориентировачен инструмент за навигиране на клиентите в системата.



## Moodboard

Това е инструмент, чрез който искаме да представим визуално (Фиг. 7.1) един от проблемите, с които често се сблъскваме, а именно процеса на управление на нашите финанси. Независимо дали се занимаваме професионално във финансовата сфера или просто изчисляваме нашите лични финанси, всеки се сблъсква с този проблем, като бихме искали да разработим и представим една автоматизирана система, която да бъде от ключова полза на крайните потребители и да решава по един ефективен начин този проблем. Именно, чрез мобилно приложение, което всеки би могъл да използва от телефона си по всяко време и да разполага с актуална и навременна информация за своите финансови средства. Като управлението на средствата ще бъде лесно и най-вече продуктивно и ефективно.

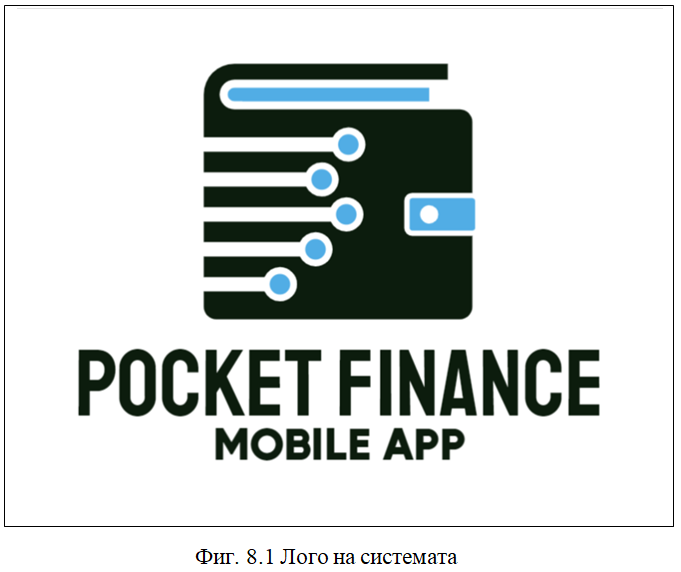
Чрез този „Moodboard” също така представяме именованието на нашето мобилно приложение, както и цветовата гама, която предпочитаме да използваме за графичния интерфейс.



## Лого

Чрез своето лого бихме искали да заявим своята идентичност и лице на нашия дигитален софтуерен продукт, тъй като в самото начало, когато нашия продукт не е познат на потребители, логото е средството, чрез което заявяваме кои сме и какво разработваме. Изборът на подходящи изразни средства като форми, цветове и шрифт помагат да се постигне търсеното внушение. Логото може да бъде средство, чрез което продукта ни да се отличи от своите конкуренти и да демонстрира своята уникалност. Използвайки различно графично решение за своя знак, може да се акцентира върху преимущество, ценност или специално отношение, поради които клиентът да избере нашия продукт пред останалите. Изборът на различен цвят, форма, дори нетипичен за бранша символ (ако разбира се това е добре обосновано), допринася за това, и да бъдат запомнени по-лесно. Предварителното проучване и анализ на логата на конкурентите е задължителна стъпка в дизайна на лого.

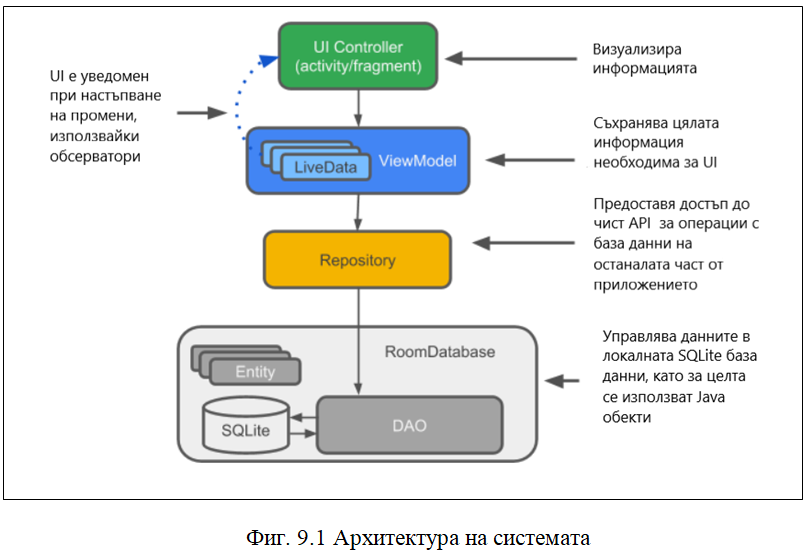
Това е нашето лого, идентичност и заявката ни за качествен продукт (Фиг. 8.1), на който крайните потребители биха могли да се доверят и използват.



# Реализация

## Архитектура на системата

Операционната система Android управлява ресурсите агресивно, за да работи добре на огромен набор от устройства и понякога това е предизвикателство да се изградят стабилни приложения. Компонентите за архитектурата на Android предоставят насоки за архитектурата на приложението, с библиотеки за общи задачи като управление на жизнения цикъл и запазване на данните. Архитектурните компоненти ни помагат да структурираме приложението си по начин, който е стабилен, тестван и поддържан с по-малко ръчно написан код. Архитектурните компоненти предоставят прост, гъвкав и практичен подход, който ни освобождава от някои често срещани проблеми, за да можем да се съсредоточим върху изграждането на страхотни приложения. На фиг. 9.1 е представена архитектурата на нашата система.



## Entity

За да работи Room с обект, трябва да дадете информация на Room, която свързва съдържанието на Java класа на обекта (например AccountsEntity) с това, което искате да представите в таблицата на базата данни. Правите това с помощта на анотации.

Това са някои често използвани анотации:

* @Entity(tableName =**"account"**) - Всеки екземпляр @Entity представлява обект в таблица. Посочете името на таблицата, ако искате то да е различно от името на класа.
* @PrimaryKey - Всеки обект се нуждае от първичен ключ. За да генерирате автоматично уникален ключ за всеки обект, добавете и анотирайте първичен целочислен ключ с autoGenerate=true, както е показано в кода по-долу.
* @NonNull - Означава, че връщаната стойност на параметър, поле или метод никога не може да бъде Null. Първичният ключ винаги трябва да използва анотацията @NonNull. Използвайте тази анотация за задължителни полета във вашите редове.
* @ColumnInfo(name =**"account\_name "**) - Посочете името на колоната в таблицата, ако искате името на колоната да е различно от името на променливата-член.

Всяко поле, което се съхранява в базата данни, трябва или да е публично, или да има метод за получаване, за да може Room да има достъп до него. Кодът по-долу предоставя метод getWord() "getter", вместо да излага директно променливи-членове.

Това е анотиран код, представляващ Accounts Entity:

@Entity(tableName = "account",  
 foreignKeys = @ForeignKey(entity = AccountType.class, parentColumns = "id", childColumns = "acc\_type\_id", onDelete = *CASCADE*, onUpdate = *CASCADE*),  
 indices = { @Index(name = "Account\_PK", value = "id"),  
 @Index(name = "Account\_FK", value = "acc\_type\_id")} )  
public class Account implements Parcelable { // Parcelable? Serializable?  
  
 @PrimaryKey(autoGenerate = true)  
 private int id;  
  
 private double balance;  
  
 private String name;  
  
 @ColumnInfo(name = "acc\_type\_id")  
 private int accountTypeId;  
  
 // Currency ID zasega lipsva, zashtoto shte dobavi dopulnitelna slojnost  
  
 //////////////////////////  
  
 public Account(double balance, String name, int accountTypeId) {  
 this.balance = balance;  
 this.name=name;  
 this.accountTypeId = accountTypeId;  
 }  
  
 protected Account(Parcel in) {  
 id = in.readInt();  
 balance = in.readDouble();  
 name = in.readString();  
 accountTypeId = in.readInt();  
 }  
  
 public static final Creator<Account> *CREATOR* = new Creator<Account>() {  
 @Override  
 public Account createFromParcel(Parcel in) {  
 return new Account(in);  
 }  
  
 @Override  
 public Account[] newArray(int size) {  
 return new Account[size];  
 }  
 };  
  
 public int getId() {  
 return id;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
  
 public double getBalance() {  
 return balance;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public void setBalance(double balance) {  
 this.balance = balance;  
 }  
  
 public int getAccountTypeId() {  
 return accountTypeId;  
 }  
  
 public void setAccountTypeId(int accountTypeId) {  
 this.accountTypeId = accountTypeId;  
 }  
  
 @Override  
 public int describeContents() {  
 return 0;  
 }  
  
 @Override  
 public void writeToParcel(Parcel dest, int flags) {  
 dest.writeInt(id);  
 dest.writeDouble(balance);  
 dest.writeString(name);  
 dest.writeInt(accountTypeId);  
 }

## DAO (Data Access Object)

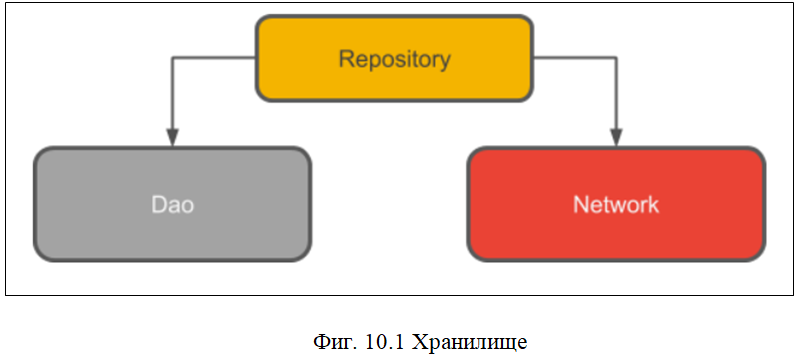
За да получите достъп до данните на приложението си чрез библиотеката за постоянство на стаята, работите с обекти за достъп до данни или DAO. Набор от Dao обекти (DAOs) формира основния компонент на Room database. Всеки DAO включва методи, които предлагат абстрактен достъп до базата данни на вашето приложение. Анотирате DAO, за да укажете SQL заявки и да ги свържете с извиквания на методи. Компилаторът проверява SQL за грешки, след което генерира заявки от анотациите. За често срещани заявки библиотеките предоставят удобни анотации, като @Insert, @Delete и @Update.

Това е код, който демонстрира тези пояснения заедно с различни типове заявки:

@Dao  
public interface AccountTypeDao {   
  
 @Transaction  
 @Insert  
 long insertAccountType(AccountType accountType);  
  
 @Transaction  
 @Insert  
 long insertAccount(Account account);  
  
 @Transaction  
 @Update  
 void updateAccountType(AccountType accountType);  
  
 @Transaction  
 @Update  
 void updateAccount(Account account);  
  
 @Transaction  
 @Delete  
 void deleteAccountType(AccountType accountType);  
  
 @Transaction  
 @Delete  
 void deleteAccount(Account account);  
  
 @Transaction  
 @Query("DELETE FROM account\_type")  
 void deleteAllAccountTypes();  
  
 @Transaction  
 @Query("DELETE FROM account")  
 void deleteAllAccounts();  
  
 //////////////////////////////////////////////////////////////////////////////////////////////////////  
 // SELECT & SELECT WHERE id  
 // for:  
 // - AccountType (with Accounts ||| RELATIONSHIP)  
 // - Account (SIMPLE ||| NO RELATIONSHIP) ---> for relationship with Transactions see \*TransactionDao\*  
  
 @Transaction  
 @Query("SELECT \* FROM account\_type")  
 LiveData<List<AccountType>> getAllAccountTypes();  
  
 @Transaction  
 @Query("SELECT \* FROM account\_type WHERE id = :arg0")  
 LiveData<AccountType> getAccountTypeByID(int arg0);  
  
 @Transaction  
 @Query("SELECT \* FROM account")  
 LiveData<List<Account>> getAllAccounts(); // N  
  
 @Transaction  
 @Query("SELECT \* FROM account WHERE id = :arg0")  
 LiveData<Account> getAccountByID(int arg0);  
  
 @Transaction  
 @Query("SELECT \* FROM account\_type")  
 LiveData<List<AccountTypeWithAccounts>> getAllAccountTypesWithAccounts();  
  
 @Transaction  
 @Query("SELECT \* FROM account\_type where id = :id")  
 LiveData<AccountTypeWithAccounts> getAccountTypeWithAccountsById(int id);  
}

## Repository

Хранилището е клас, който абстрахира достъпа до множество източници на данни (Фиг. 10.1). Хранилището не е част от библиотеките на компонентите на архитектурата, но е препоръчителна най-добра практика за разделяне на кода от архитектурата. Клас Repository обработва операции с данни. Той предоставя чист API за останалата част от приложението за работа с база данни. Хранилището е мястото, където бихте поставили кода, за да управлявате нишките на заявки и да използвате множество бекендове, ако е подходящо. Обичайната употреба за хранилище е да се внедри логиката за вземане на решение дали да се извличат данни от мрежа или да се използват резултати, кеширани в базата данни.

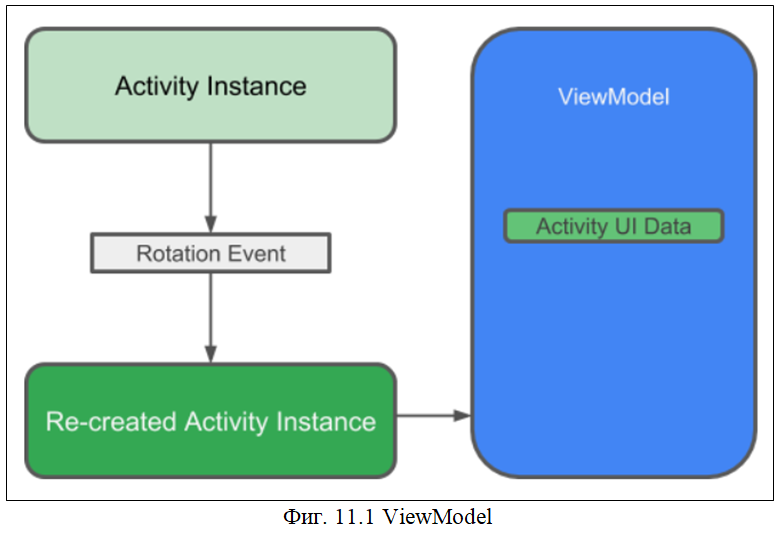


Това е примерен код за основно хранилище:

public class AccountTypeRepository {  
  
 private AccountTypeDao oAccountTypeDao;  
 private LiveData<List<Account>> oLiveDataListAllAccounts;  
 private LiveData<List<AccountType>> oLiveDataListAllAccountTypes;  
 private LiveData<List<AccountTypeWithAccounts>> oLiveDataListAccountTypesWithAccounts;  
  
 public AccountTypeRepository(Application application) {  
 AppDatabase oAppDatabase = AppDatabase.*getInstance*(application);  
  
 oAccountTypeDao = oAppDatabase.accountTypeDao();  
  
 oLiveDataListAllAccounts = oAccountTypeDao.getAllAccounts();  
 oLiveDataListAllAccountTypes = oAccountTypeDao.getAllAccountTypes();  
 oLiveDataListAccountTypesWithAccounts = oAccountTypeDao.getAllAccountTypesWithAccounts();  
 }  
  
 public LiveData<List<AccountTypeWithAccounts>> getAllAccountTypesWithAccounts() {  
 return oLiveDataListAccountTypesWithAccounts;  
 }  
  
 public LiveData<Account> getAccountByID(int nID){  
 new getAccountByIDAsyncTask(oAccountTypeDao).execute(nID);  
 return getAccountByIDAsyncTask.*getAccount*();  
 }  
  
 public int insertAccount(Account oAccount){  
 new InsertAccountAsyncTask(oAccountTypeDao).execute(oAccount);  
 return InsertAccountAsyncTask.*getAccountID*();  
 }  
  
 public void updateAccount(Account oAccount){  
 new UpdateAccountAsyncTask(oAccountTypeDao).execute(oAccount);  
 }  
  
 public void deleteAccount(Account oAccount){  
 new DeleteAccountAsyncTask(oAccountTypeDao).execute(oAccount);  
 }  
  
 public void deleteAllAccounts(){  
 new DeleteAllAccountsAsyncTask(oAccountTypeDao).execute();  
 }  
  
 public LiveData<List<Account>> getAllAccounts() {  
 return oLiveDataListAllAccounts;  
 }  
  
 public LiveData<AccountType> getAccountTypeByID(int nID){  
 new getAccountTypeByIDAsyncTask(oAccountTypeDao).execute(nID);  
 return getAccountTypeByIDAsyncTask.*getAccountType*();  
 }  
  
 public int insertAccountType(AccountType oAccountType){  
 new InsertAccountTypeAsyncTask(oAccountTypeDao).execute(oAccountType);  
 return InsertAccountTypeAsyncTask.*getAccountTypeID*();  
 }  
  
 public void updateAccountType(AccountType oAccountType){  
 new UpdateAccountTypeAsyncTask(oAccountTypeDao).execute(oAccountType);  
 }  
  
 public void deleteAccountType(AccountType oAccountType){  
 new DeleteAccountTypeAsyncTask(oAccountTypeDao).execute(oAccountType);  
 }  
  
 public void deleteAllAccountTypes(){  
 new DeleteAllAccountTypesAsyncTask(oAccountTypeDao).execute();  
 }  
  
 public LiveData<List<AccountType>> getAllAccountTypes() {  
 return oLiveDataListAllAccountTypes;  
 }  
  
 private static class getAccountByIDAsyncTask extends AsyncTask<Integer, Void, LiveData<Account>> {  
 private AccountTypeDao oAccountTypeDao;  
 private static LiveData<Account> *oAccount*;  
  
 private getAccountByIDAsyncTask(AccountTypeDao oAccountTypeDao){  
 this.oAccountTypeDao = oAccountTypeDao;  
 }  
  
 @Override  
 protected LiveData<Account> doInBackground(Integer... integers) {  
 *oAccount* = oAccountTypeDao.getAccountByID(integers[0]);  
 return oAccountTypeDao.getAccountByID(integers[0]);  
 }  
  
 @Override  
 protected void onPostExecute(LiveData<Account> oAccount) {  
 this.*oAccount* = oAccount;  
 }  
  
 public static LiveData<Account> getAccount() {  
 return *oAccount*;  
 }  
 }  
  
 private static class InsertAccountAsyncTask extends AsyncTask<Account, Void, Long>{  
 private AccountTypeDao oAccountTypeDao;  
 private static int *nAccountID*;  
  
 private InsertAccountAsyncTask(AccountTypeDao oAccountTypeDao){  
 this.oAccountTypeDao = oAccountTypeDao;  
 }  
  
 @Override  
 protected Long doInBackground(Account... accounts) {  
 return oAccountTypeDao.insertAccount(accounts[0]);  
 }  
  
 @Override  
 protected void onPostExecute(Long aLong) {  
 this.*nAccountID* = aLong.intValue();  
 }  
  
 public static int getAccountID() {  
 return *nAccountID*;  
 }  
 }  
  
 private static class UpdateAccountAsyncTask extends AsyncTask<Account, Void, Void>{  
 private AccountTypeDao oAccountTypeDao;  
  
 private UpdateAccountAsyncTask(AccountTypeDao oAccountTypeDao){  
 this.oAccountTypeDao = oAccountTypeDao;  
 }  
  
 @Override  
 protected Void doInBackground(Account... accounts) {  
 oAccountTypeDao.updateAccount(accounts[0]);  
 return null;  
 }  
 }  
  
 private static class DeleteAccountAsyncTask extends AsyncTask<Account, Void, Void>{  
 private AccountTypeDao oAccountTypeDao;  
  
 private DeleteAccountAsyncTask(AccountTypeDao oAccountTypeDao){  
 this.oAccountTypeDao = oAccountTypeDao;  
 }  
  
 @Override  
 protected Void doInBackground(Account... accounts) {  
 oAccountTypeDao.deleteAccount(accounts[0]);  
 return null;  
 }  
 }  
  
 private static class DeleteAllAccountsAsyncTask extends AsyncTask<Void, Void, Void>{  
 private AccountTypeDao oAccountTypeDao;  
  
 private DeleteAllAccountsAsyncTask(AccountTypeDao oAccountTypeDao){  
 this.oAccountTypeDao = oAccountTypeDao;  
 }  
  
 @Override  
 protected Void doInBackground(Void... voids) {  
 oAccountTypeDao.deleteAllAccounts();  
 return null;  
 }  
 }  
  
 private static class getAccountTypeByIDAsyncTask extends AsyncTask<Integer, Void, LiveData<AccountType>> {  
 private AccountTypeDao oAccountTypeDao;  
 private static LiveData<AccountType> *oAccountType*;  
  
 private getAccountTypeByIDAsyncTask(AccountTypeDao oAccountTypeDao){  
 this.oAccountTypeDao = oAccountTypeDao;  
 }  
  
 @Override  
 protected LiveData<AccountType> doInBackground(Integer... integers) {  
 return oAccountTypeDao.getAccountTypeByID(integers[0]);  
 }  
  
 @Override  
 protected void onPostExecute(LiveData<AccountType> oAccountType) {  
 this.*oAccountType* = oAccountType;  
 }  
  
 public static LiveData<AccountType> getAccountType() {  
 return *oAccountType*;  
 }  
 }  
  
 private static class InsertAccountTypeAsyncTask extends AsyncTask<AccountType, Void, Long>{  
 private AccountTypeDao oAccountTypeDao;  
 private static int *nAccountTypeID*;  
  
 private InsertAccountTypeAsyncTask(AccountTypeDao oAccountTypeDao){  
 this.oAccountTypeDao = oAccountTypeDao;  
 }  
  
 @Override  
 protected Long doInBackground(AccountType... accountTypes) {  
 return oAccountTypeDao.insertAccountType(accountTypes[0]);  
 }  
  
 @Override  
 protected void onPostExecute(Long aLong) {  
 this.*nAccountTypeID* = aLong.intValue();  
 }  
  
 public static int getAccountTypeID() {  
 return *nAccountTypeID*;  
 }  
 }  
  
 private static class UpdateAccountTypeAsyncTask extends AsyncTask<AccountType, Void, Void>{  
 private AccountTypeDao oAccountTypeDao;  
  
 private UpdateAccountTypeAsyncTask(AccountTypeDao oAccountTypeDao){  
 this.oAccountTypeDao = oAccountTypeDao;  
 }  
  
 @Override  
 protected Void doInBackground(AccountType... accountTypes) {  
 oAccountTypeDao.updateAccountType(accountTypes[0]);  
 return null;  
 }  
 }  
  
 private static class DeleteAccountTypeAsyncTask extends AsyncTask<AccountType, Void, Void>{  
 private AccountTypeDao oAccountTypeDao;  
  
 private DeleteAccountTypeAsyncTask(AccountTypeDao oAccountTypeDao){  
 this.oAccountTypeDao = oAccountTypeDao;  
 }  
  
 @Override  
 protected Void doInBackground(AccountType... accountTypes) {  
 oAccountTypeDao.deleteAccountType(accountTypes[0]);  
 return null;  
 }  
 }  
  
 private static class DeleteAllAccountTypesAsyncTask extends AsyncTask<Void, Void, Void>{  
 private AccountTypeDao oAccountTypeDao;  
  
 private DeleteAllAccountTypesAsyncTask(AccountTypeDao oAccountTypeDao){  
 this.oAccountTypeDao = oAccountTypeDao;  
 }  
  
 @Override  
 protected Void doInBackground(Void... voids) {  
 oAccountTypeDao.deleteAllAccountTypes();  
 return null;  
 }  
 }  
}

## Viewmodel

ViewModel е клас, чиято роля е да предоставя данни на потребителския интерфейс и да оцелее при промени в конфигурацията. ViewModel действа като комуникационен център между хранилището и потребителския интерфейс. Можем също да използваме ViewModel за споделяне на данни между фрагменти. ViewModel е част от библиотеката на жизнения цикъл. ViewModel съхранява данните за потребителския интерфейс на нашето приложение по начин, съобразен с жизнения цикъл, който оцелява при промени в конфигурацията. Разделянето на данните за потребителския интерфейс на нашето приложение от нашите класове Activity и Fragment ни позволява по-добре да следваме принципа на единна отговорност: нашите дейности и фрагменти са отговорни за изтеглянето на данни на екрана, докато нашият ViewModel е отговорен за съхраняването и обработката на всички данни, необходими за потребителския интерфейс.



Това е примерен код за Viewmodel:

public class AccountTypeViewModel extends AndroidViewModel {  
  
 private AccountTypeRepository oAccountTypeRepository;  
 private LiveData<List<Account>> oLiveDataListAllAccounts;  
 private LiveData<List<AccountType>> oLiveDataListAllAccountTypes;  
 private LiveData<List<AccountTypeWithAccounts>> oLiveDataListAccountTypesWithAccounts;  
  
 public AccountTypeViewModel(@NonNull Application application) {  
 super(application);  
 oAccountTypeRepository = new AccountTypeRepository(application);  
 oLiveDataListAllAccounts = oAccountTypeRepository.getAllAccounts();  
 oLiveDataListAllAccountTypes = oAccountTypeRepository.getAllAccountTypes();  
 oLiveDataListAccountTypesWithAccounts = oAccountTypeRepository.getAllAccountTypesWithAccounts();  
 }  
  
 public LiveData<AccountType> getAccountTypeByID(int nID){  
 return oAccountTypeRepository.getAccountTypeByID(nID);  
 }  
  
 public int insertAccountType(AccountType oAccountType){  
 return oAccountTypeRepository.insertAccountType(oAccountType);  
 }  
  
 public void updateAccountType(AccountType oAccountType){  
 oAccountTypeRepository.updateAccountType(oAccountType);  
 }  
  
 public void deleteAccountType(AccountType oAccountType){  
 oAccountTypeRepository.deleteAccountType(oAccountType);  
 }  
  
 public void deleteAllAccountTypes(){  
 oAccountTypeRepository.deleteAllAccountTypes();  
 }  
  
 public LiveData<List<AccountType>> getAllAccountTypes(){  
 return oLiveDataListAllAccountTypes;  
 }  
  
 public LiveData<Account> getAccountByID(int nID){  
 return oAccountTypeRepository.getAccountByID(nID);  
 }  
  
 public int insertAccount(Account oAccount){  
 return oAccountTypeRepository.insertAccount(oAccount);  
 }  
  
 public void updateAccount(Account oAccount){  
 oAccountTypeRepository.updateAccount(oAccount);  
 }  
  
 public void deleteAccount(Account oAccount){  
 oAccountTypeRepository.deleteAccount(oAccount);  
 }  
  
 public void deleteAllAccounts(){  
 oAccountTypeRepository.deleteAllAccounts();  
 }  
  
 public LiveData<List<Account>> getAllAccounts(){  
 return oLiveDataListAllAccounts;  
 }  
  
 public LiveData<List<AccountTypeWithAccounts>> getAllAccountTypesWithAccounts(){  
 return oLiveDataListAccountTypesWithAccounts;  
 }  
}

## LiveData

Всеки път, когато данните се променят, методът onChanged() на нашия наблюдател се извиква. В най-основния случай това може да актуализира съдържанието на TextView, както е показано в този код:

final Observer<String> nameObserver = new Observer<String>() {

@Override

public void onChanged(@Nullable final String newName) {

// Update the UI, in this case, a TextView.

mNameTextView.setText(newName);

}

};

Друг често срещан случай е показването на данни в изглед, който работи с адаптер. Например, ако показваме данни в RecyclerView, методът onChanged() актуализира данните, кеширани в адаптера:

oViewModel.getAllAccountTypesWithAccounts().observe(getViewLifecycleOwner(), new Observer<List<AccountTypeWithAccounts>>() {  
 @Override  
 public void onChanged(@Nullable final List<AccountTypeWithAccounts> accTypesWithAccounts) {  
 // Update the cached copy of the words in the adapter.  
 adapter.setData(accTypesWithAccounts);  
 }  
});

# Тестове и резултати

# Заключение

Нуждата от правилно управление на личните финанси и повишаване на продуктивността и ефективността, са ключови фактори, поради които се зароди идеята и целта да се проектира и реализира мобилно приложение за управление на финансовите средства. С разработването на тази система се надяваме да улесним ежедневието на крайните потребители и да успеем да предоставим решение на един от техните проблеми, с които се сблъскват всеки ден. Нуждата от управление на паричните средства, никога не е била толкова сериозна, както в днешните времена, и правилната стратегия и подходи откъм инвестиции на средствата в бъдеще са от ключово значение, за нашия живот. Разработването на тази система и предоставянето и на крайните потребители е първата стъпка към решаването на един от многото проблеми в ежедневието ни и предоставяне на нужната ефективност, продуктивност, надеждност и сигурност, при управление на финансовите ни средства.