

**Разработка мобильного приложения для персонализированного выбора и предложения банковских карт с максимальным cashback пользователю в зависимости от места оплаты.**

IT-специалист:

Программист мобильных устройств

Киряков А.И.

Калининград

2024

**Дипломный проект «Разработка мобильного приложения для персонализированного выбора и предложения банковских карт с максимальным cashback пользователю в зависимости от места оплаты»**

**Содержание**

1. Введение2
   1. Формулировка проблемы или потребности, которую решает приложение 2
   2. Цель и задачи дипломной работы 3
   3. Описание методов и инструментов разработки 4
2. Анализ требований7
   1. Определение функциональных и нефункциональных требований приложения 7
   2. Изучение потребностей и предпочтений целевой аудитории 8
3. Проектирование приложения10
   1. Создание архитектуры и дизайна мобильного приложения 10
      1. Use-case диаграмма11
      2. UI/UX диаграмма 12
      3. ER диаграмма (сущность-связь)13
      4. Диаграмма модулей 14
      5. Диаграмма классов 15
4. Разработка приложения18
   1. Реализация функциональности приложения согласно разработанной архитектуре и дизайну 18
      1. Создание основных экранов и компонентов21
      2. Обработка событий и взаимодействие с пользователем (ветка «Карты»)26
      3. Обработка событий и взаимодействие с пользователем (ветка «Категории»)30
      4. Управление данными: взаимодействие с базой данных33
      5. Работа с внешними сервисами и API35
      6. Использование шаблонов35
5. Заключение40
   1. Суммирование результатов разработки и достижений целей дипломной работы 40
   2. Предложение дальнейших возможностей для развития приложения 40
6. Список литературы и ресурсов42

Приложения

1. **Введение**
   1. **Формулировка проблемы или потребности, которую решает приложение.**

В современном мире уже никого не удивить наличием нескольких платежных карт, обычно, это как минимум – «зарплатная, потому что на работе дали», «кредитная, потому что на большие покупки надо», «дебетовая, потому что бонусы какие-то есть», «вэбка для интернета» и т.п. Это карты не обязательно одного банка и даже не одной платёжной системы[[1]](#footnote-1).

Банки в жестокой конкурентной борьбе стараются привлечь клиентов на свою сторону, зачастую предлагая различные «фишки» к своим картам – это и процент на остаток, и интересные условия по накопительному счету, и мили, и, конечно же, кэшбек по различным категориям, что в свою очередь является MCC-кодом[[2]](#footnote-2). Вот на нём мы и остановимся.

В дипломной работе мы будем применять два определения – «категория кэшбека» (применительно к карте) и «место оплаты» (применительно к выбору пользователя).

Итак, примем, что практически у каждого из нас есть как минимум 3 платежных карты и на каждой из них по 3 категории с кэшбеком, а также то, что в современном мире все больше людей пользуются смартфонами для оплаты своих покупок. Принимая во внимание минимальное количество категорий (как минимум 9 в нашем случае), не мудрено, что при выборе способа оплаты, пользователи могут пропускать возможность получить дополнительные выгоды от кэшбека, так как удержать в голове всё и оперативно сориентироваться при оплате не всегда получается.

В связи с этим, формируется потребность в удобном инструменте, который бы автоматически предлагал пользователям наиболее выгодную карту с наибольшим кэшбеком в зависимости от места, где они собираются совершить оплату.

Идея приложения заключается в том, чтобы помочь пользователям выбрать наиболее выгодную карту с наибольшим кэшбеком перед оплатой смартфоном. Приложение будет предлагать различные варианты карт в зависимости от места оплаты, указанного пользователем. Это поможет клиентам не упускать возможность получить дополнительные бонусы и максимизировать свои выгоды при каждой покупке.

* 1. **Цель и задачи дипломной работы.**

Целью дипломной работы по разработке мобильного приложения будет создание функционального и эффективного приложения, которое в версии, описанной в данном материале, поможет пользователям выбрать наиболее выгодную карту с наибольшим кэшбеком перед оплатой смартфоном.

Для достижения этой цели можно поставить следующие задачи:

1. Исследование и анализ существующих мобильных приложений и сервисов, связанных с выбором оптимальной карты перед покупкой.

   - Изучение функциональности и возможностей существующих приложений.

   - Анализ и сравнение сильных и слабых сторон данных приложений.

2. Изучение основ и принципов работы мобильных платежей и кэшбек-систем.

   - Понимание принципов начисления и использования кэшбека в различных банках и платежных системах.

   - Изучение технических аспектов мобильных платежей и их интеграции с банковскими картами (в будущих релизах).

3. Проектирование архитектуры и дизайна приложения.

   - Определение функциональных и нефункциональных требований к приложению.

   - Создание удобного и интуитивно понятного пользовательского интерфейса.

4. Разработка и тестирование функциональности приложения.

   - Реализация выбора наиболее выгодной карты на основе места оплаты.

- Применение на практике полученных знаний, а также параллельное усовершенствование приложения.

   - Интеграция с платежными системами и банками для получения информации о кэшбеке (в будущих релизах).

5. Оценка эффективности и качества разработанного приложения.

   - Тестирование приложения на различных устройствах для проверки совместимости и производительности.

   - Сравнение полученных результатов с ожиданиями и улучшение приложения, если это необходимо.

6. Документирование процесса разработки и результатов исследования.

   - Написание дипломной работы, включающей описание каждого этапа разработки и полученных результатов.

   - Подготовка презентации дипломной работы и ее защита перед комиссией.

Основная задача дипломной работы заключается в разработке полнофункционального мобильного приложения, которое будет удовлетворять потребности пользователей в выборе оптимальной карты для максимизации получаемого кэшбека при оплате товаров и услуг с использованием смартфона.

* 1. **Описание инструментов разработки**

1. *Аппаратные средства:*

При разработке использовался ноутбук Lenovo 17 с процессором AMD Ryzen 3 3250U, 8 ГБ оперативной памяти и SSD диском для повышения производительности при запуске Xcode.

1. *Операционная система:*

Для разработки мобильного приложения под iOS использовалась операционная система macOS 11.7.10 (20G1427). Выбор обусловлен необходимостью написания кода на языке Swift, который используется для разработки приложений для устройств Apple.

Операционная система MacOS, разработанная компанией Apple, является операционной системой, специально предназначенной для устройств этой компании, таких как MacBook, iMac, Mac Pro и других. ОС macOS предлагает удобное и интуитивно понятное пользовательское интерфейс, высокую стабильность и производительность.

В контексте разработки мобильных приложений для iOS, MacOS является предпочтительной операционной системой благодаря интеграции с инструментами разработки Apple, такими как Xcode, который используется для создания приложений под iOS и macOS. MacOS поддерживает язык программирования Swift, который является основным языком для создания приложений для устройств Apple.

1. *Инструменты виртуализации:*

В связи с отсутствием оборудования (персонального компьютера, ноутбука) производства Apple принято решение развернуть «виртуальную машину» с macOS (11.7.10 Big Sur) на оборудовании с ОС Windows с помощью VMware Workstation.

VMware Workstation - это программное обеспечение для виртуализации, которое позволяет пользователям создавать и запускать виртуальные машины на своем компьютере. С помощью VMware Workstation возможно создавать различные виртуальные среды для запуска различных операционных систем, включая macOS, на обычном компьютере под управлением Windows.

VMware Workstation обладает высокой степенью гибкости и производительности, позволяет управлять ресурсами виртуальных машин, устанавливать необходимые настройки и проводить тестирование программного обеспечения в виртуальной среде.

1. *Языки программирования:*

Для разработки приложения с учетом обучения на курсе специализации безальтернативно выбран язык программирования Swift version 13.1 (13A1030d). Использование именно этой версии связано с ограничениями использования Swift на macOS 11.7.10.

Swift - это современный язык программирования, разработанный компанией Apple для создания приложений под iOS, macOS, watchOS и tvOS. Swift является мощным, безопасным и быстрым языком, который предлагает простой и понятный синтаксис, что делает его пригодным для как начинающих, так и опытных разработчиков.

Swift поддерживает различные парадигмы программирования, такие как процедурное, объектно-ориентированное и функциональное программирование. Он основан на современных концепциях и включает в себя множество инновационных функций, чтобы упростить процесс разработки и улучшить производительность.

1. *Фреймворки и библиотеки:*

При написании приложения применялись фреймворки и библиотеки, которые обеспечили функциональность, производительность и удобство разработки приложения, помогая реализовать необходимый функционал и достичь поставленных целей.

- UIKit: Основной фреймворк для разработки пользовательского интерфейса приложения под iOS. UIKit предоставляет широкий набор классов и элементов управления для построения интерфейса приложения.

- CoreData: Фреймворк для работы с базами данных на устройствах Apple. CoreData обеспечивает удобный доступ к данным, их хранение и управление в приложении.

1. *Инструменты управления версиями:*

Для управления версиями и контроля изменений в проекте был использован инструмент Git в сочетании с платформой GitHub.

- Git является распределенной системой контроля версий, обеспечивающей отслеживание изменений в исходном коде проекта. С помощью Git разработчики могут фиксировать изменения, создавать ветки для разработки новых функций, объединять изменения от разных участников и управлять историей проекта.

- GitHub - это веб-платформа для хостинга проектов с использованием системы контроля версий Git. GitHub предоставляет возможность хранить репозитории проектов, управлять задачами, просматривать код, создавать отчеты об ошибках и совместно работать над проектами в команде.

Использование Git и GitHub обеспечивает эффективное управление версиями и совместную разработку проекта, обеспечивая прозрачность изменений, возможность отката к предыдущим версиям, а также облегчая совместную работу между разработчиками.

1. *Дизайн и прототипирование:*

Для проектирования дизайна приложения и построения диаграмм использовались следующие инструменты:

Дизайн - Balsamiq.cloud

Диаграммы - Draw.io

Использование указанных инструментов для дизайна и прототипирования помогло в создании пользовательского интерфейса приложения до начала его разработки.

1. **Анализ требований**
   1. **Определение функциональных и нефункциональных требований приложения**

*Функциональные требования – описание основных функций и возможностей, которые должны быть реализованы в приложении.*

Функционал для текущего релиза:

Приложение, размещенное локально на устройстве, по выбору пользователем места оплаты (категории кэшбека) предлагает ему платёжные карты с наибольшим кэшбеком для выбранной и отображает важные параметры категории для принятия окончательного решения пользователем.

Основные функции, реализованные в текущем релизе:

* демонстрация работы бизнес-логики приложения, а именно – приложение отображает данные на экране устройства в соответствии с требованиями (демонстрирует места оплаты – уникальные категории кэшбека, которые присутствуют хотя бы у одной карты), а также при выборе того или иного места оплаты предлагает выбрать карту с максимальным кэшбеком. Данные внесены «жестко» с применением структур данных.
* демонстрация списка карт и функция добавления карт с применением паттерна MVC, с использованием структур данных и CoreData для их хранения. Методы удаления и редактирования не реализованы.
* демонстрация списка категорий кэшбека и функция добавления новой категории с применением паттерна Obsrver, с использованием структур данных и CoreData для их хранения. Методы удаления и редактирования не реализованы.

Дальнейшие шаги по разработке кода для получения рабочего релиза:

* настройка связей в сущностях для дальнейшей реализации создания конкретных мест оплаты в автоматическом режиме (путем правильно выстроенных связей сущностей карт и категорий)
* реализация функций удаления и редактирования карт и категорий с учетом связей БД.

Таким образом будет получена полнофункциональная обработка данных по картам и категориям, а с их корректной настройкой связей – создана сущность для хранения кэшбека.

Возможная масштабируемость функционала:

1. Добавление статистики (реализация сущности deal для хранения данных об оплате);
2. Создание напоминания для смены категорий;
3. Контроль ограничений, установленных банками;
4. При реализации взаимодействия с ПС – контроль MCC-кодов на совпадение, отправка сообщений о мискодинге в ПС, автоматическое «подтягивание» информации о кэшбеках от ПС, формирование системы рекомендаций по геолокации.

*Нефункциональные требования – ограничения и критерии производительности, которым должно соответствовать приложение.*

Масштабируемость для текущего релиза:

* Расширение свойств карты (в текущем релизе: наименование, картинка для визуализации карты, категории кэшбека по карте, процент кэшбека):

- периодическая смена категорий: для создания напоминания – требуется ли смена категорий (да/нет), если да – периодичность (ежемесячно/ ежеквартально/ ежегодно), число месяца смены.

- подсчет минимально требуемой суммы покупок за период для начисления кэшбека (важное условие – наличие подключения к процессингу платежной системы): минимальная требуемая сумма покупок в месяц, текущая сумма покупок в течение месяца

- прекращение выдачи карты в подборку при достижении максимальной величины кэшбека (важное условие – наличие подключения к процессингу платежной системы): максимальная сумма кэшбека за месяц, текущая сумма кэшбека за месяц.

* Расширение свойств категории кэшбека (в текущем релизе: наименование, иконка для визуализации категории).
* Расширение за счет организации взаимодействия с другими приложениями (интерфейс), в первую очередь Мир Pay. Сбор данных для НСПК на предмет мискодинга (подмены МСС-кода торгового предприятия) – если выбранная категория по геолокации не совпадает с MCC постоянно, то стоит обратить внимание.
* Расширение за счет взаимодействия с процессингом платежной системы (интерфейс) – формирование аналитических, статистических и отчетных данных.
  1. **Изучение потребностей и предпочтений целевой аудитории**

Для анализа и определения потребностей и предпочтений целевой аудитории при разработке приложения было важно провести исследование, которое включает в себя следующие шаги:

1. *Определение целевой аудитории:*

Пользователи относятся к широкой возрастной группе – от 14 лет (первый опыт пользования банковскими картами и приложениями, зарождение финансовых навыков) до 50 лет (уверенный пользователь смартфона, все карты в «цифре», постоянное повышение финансовой грамотности либо склонность к экономии на регулярных покупках).

У разной возрастной группы свой интерес: финансовая грамотность, интерес к инвестициям или склонность к экономии.

1. *Сбор обратной связи:*

По результатам опроса среди друзей и знакомых, а также с коллегами из финансовой сферы – подобное приложение востребовано. Начальный функционал должен остаться основным, а остальное – крутиться «под капотом». Основная просьба – приложение должно остаться простым и понятным.

1. *Анализ конкурентов:*

На сегодняшний момент пока не сталкивался с аналогичными приложениями.

Изучение потребностей и предпочтений целевой аудитории позволяет разработать приложение, которое будет соответствовать ожиданиям пользователей, решать их проблемы и предлагать удобный и привлекательный пользовательский опыт.

1. **Проектирование приложения**

Проектирование — важный этап, на котором принимается решение о том, какой функционал реализовать в приложении, какие технологии, как правильно продумать его структуру, а также уместить все это в оговоренные сроки. Он требует наибольшей вовлеченности и высокого уровня коммуникации со стороны заказчика.

Когда требуется создать нестандартное приложение с особенным, уникальным функционалом — просто начать писать код сразу не получится. Вы рискуете совершить много ненужных действий, вносить бесконечные правки, потратить много лишних времени и денег. В конечном итоге все это может привести к провалу проекта в целом.

Проектирование помогает сформировать понимание того, что должно быть разработано, зачем, как и в какой срок.

* 1. **Создание архитектуры и дизайна мобильного приложения**

**теория, ТЗ**

Архитектура приложения - это планирование и организация распределения компонентов, их взаимодействия и структуры внутри программного приложения. Она определяет, как различные части приложения будут взаимодействовать друг с другом и какие принципы и шаблоны будут использоваться для разработки и поддержки приложения.

Архитектура приложения помогает создать структуру, которая обеспечивает:

*1. Разделение ответственности (Separation of Concerns):*

- хорошо спроектированная архитектура разделяет функциональность приложения на отдельные компоненты, каждый из которых отвечает за узкую область функций;

- позволяет разработчикам сосредоточиться на конкретной задаче или компоненте, не затрагивая остальную часть приложения;

- эффективное разделение ответственности позволяет легче понимать и поддерживать код, а также дает возможность для распараллеливания работы разработчиков.

*2. Модульность (Modularity):*

- архитектура приложения помогает разделить его на модули, что позволяет легко добавлять, изменять или удалять функциональность без значительного влияния на остальную часть приложения;

- модули могут быть независимыми и легко поддерживаемыми, также они могут быть повторно использованы в других проектах.

*3. Расширяемость (Extensibility):*

- хорошо спроектированная архитектура обеспечивает гибкость и расширяемость приложения;

- позволяет добавлять новую функциональность или модифицировать существующую, минимизируя влияние на другие части приложения.

Это особенно полезно при развитии приложения с течением времени, когда требования и потребности могут меняться.

*4. Тестируемость (Testability):*

- хорошо спроектированная архитектура облегчает написание юнит-тестов и интеграционных тестов, что помогает убедиться в надежности и правильной работе приложения;

- хорошо структурированный код позволяет изолировать компоненты приложения для тестирования, не требуя выполнения длительных и сложных настроек, таких как базы данных или сетевые подключения.

Существует множество архитектурных шаблонов и подходов в разработке приложений, таких как MVC (Model-View-Controller), MVVM (Model-View-ViewModel), VIPER и другие. Каждый из них имеет свои сильные и слабые стороны и может быть выбран в соответствии с требованиями и особенностями проекта.

Важно отметить, что архитектура приложения – не конкретная технология или инструмент, а скорее организационный и проектный принцип, который помогает разработчикам создавать масштабируемые, гибкие и легко обслуживаемые приложения.

С целью отражения полученных знаний и результата самообразования в нашем приложении использовались два паттерна – по отношению к Картам MVC, а по отношению к Категориям Observer. Пришлось дополнительно потратить время на изучение паттерна Observer и его интеграцию в код. Схематически применение паттернов непосредственно в нашем приложении отобразил в Приложениях 1 и 2.

* + 1. **Use-case диаграмма**

**теория, описательная часть диаграммы для разработки, приложение**

Use-case диаграмма (диаграмма вариантов использования) - это инструмент моделирования, который используется для идентификации различных актеров (пользователей) и их взаимодействия с системой через различные сценарии (use-case'ы). Это позволяет определить функциональные требования к системе и понять, как пользователи будут использовать приложение.

Описательная часть Use-case диаграммы включает в себя следующие элементы:

1. *Актеры (Actors):*

Определение: Актеры представляют различные участники или роли, взаимодействующие с системой (пользователи, внешние системы и т. д.).

В нашем случае актерами могут быть пользователи, система обработки запросов платежной системы и т. д.

*2. Use-case'ы (Use Cases):*

Определение: Use-case описывает конкретное действие или функцию системы, которую пользователь может выполнять.

Нашими Use-case'ами могут быть "Пользователь просматривает доступные карты с кэшбеком", "Пользователь выбирает место оплаты" и другие.

*3. Отношения между актерами и Use-case'ами:*

Ассоциация (Association): Линия, соединяющая актера с Use-case, показывает, что актер взаимодействует с этим Use-case.

Включение (Inclusion): Показывает, что один Use-case включает в себя функциональность другого.

Расширение (Extension): Показывает расширение одного Use-case другим при определенных условиях.

Use-case диаграмма является важным инструментом на начальных этапах разработки приложения, так как она помогает определить основные сценарии использования и функциональные требования к системе. После создания Use-case диаграммы, можно перейти к детальному проектированию функциональности приложения и разработке.

Use-case диаграмма проекта представлена в Приложении 3.

* + 1. **UI/UX диаграмма**

**теория, описательная часть диаграммы для разработки, приложение**

UI/UX диаграмма - это инструмент, который помогает визуализировать пользовательский интерфейс (UI) и пользовательский опыт (UX) при разработке приложения. Она включает в себя элементы дизайна, компоненты интерфейса, путь пользователей и другие аспекты пользовательского опыта.

Описательная часть UI/UX диаграммы обычно включает в себя следующие элементы:

*1. Элементы дизайна:*

Наполнения: цвета, шрифты, иконки, изображения, графика и другие дизайн-элементы, которые используются для создания интерфейса приложения.

Формы: различные формы и элементы взаимодействия, такие как кнопки, поля ввода, переключатели и другие.

*2. Макеты экранов:*

Изображения макетов: визуализации различных экранов приложения с расположением элементов интерфейса и визуальным оформлением.

Исследование макетов: анализ различных вариантов дизайна экранов для определения наилучших решений.

*3. Интерактивность и навигация:*

Mockups или прототипы: создание интерактивных прототипов, которые показывают навигацию по приложению и взаимодействие пользователя.

Поток пользователей: описание пути пользователя через приложение, включая взаимодействие с различными экранами.

*4. Тестирование и итерации:*

Тестирование дизайна: оценка дизайна интерфейса с участием тестировщиков или фокусных групп пользователей.

Итерации и улучшения: исправление или улучшение дизайна на основе обратной связи и результатов тестирования.

UI/UX диаграмма помогает дизайнерам и разработчикам получить общее представление о дизайне приложения, определить пользовательский опыт и обеспечить удобство использования для конечных пользователей.

В данном проекте для создания UI/UX диаграммы использовался бесплатный период на Balsamiq.cloud с помощью которого был «нарисован» первоначальный дизайн программы на основе личного опыта и знаний (Приложение 4), а затем уже непосредственно после получения знаний программирования на Swift и в процессе создания приложения, был сформирован окончательный вид (Приложение 5).

* + 1. **ER диаграмма (сущность-связь)**

**теория, описательная часть диаграммы для разработки, приложение**

ER (Entity-Relationship) диаграмма является инструментом моделирования данных, который используется для отображения сущностей (entities) и их взаимосвязей в базе данных. Она помогает описать структуру данных, атрибуты сущностей и связи между ними.

Описательная часть ER диаграммы включает в себя следующие аспекты:

*1. Сущности (Entities):*

- Описание: Сущности представляют основные объекты, о которых хранится информация в базе данных. Каждая сущность имеет уникальный идентификатор (Primary Key) и атрибуты, которые характеризуют сущность.

В текущей версии нашего приложения такие сущности: Карты (CardEntity), Категории (CategoryEntity) и Кэшбек (CashbackEntity). Подразумеваются и новые сущности, в частности Сделка (DealEntity).

*2. Отношения (Relationships):*

- Описание: Отношения определяют связи между сущностями. Они показывают, как сущности взаимодействуют и как связаны друг с другом.

В текущем релизе сущности связаны следующим образом: Карты с Кэшбеком и Категории с Кэшбеком. По сути сущность Кэшбек – хранилище ID из сущностей Карты и Категории с добавлением персонификации экземпляра Кэшбек.

*3. Атрибуты (Attributes):*

- Описание: Атрибуты определяют характеристики или свойства сущностей. Они представляют информацию, которая хранится в базе данных.

Атрибуты сущностей нашего приложения отображены непосредственно в самой ER диаграмме в Приложении 6.

*4. Ключи (Keys):*

- Описание: Ключи используются для однозначной идентификации записей в базе данных. Основные ключи (Primary Keys) уникально идентифицируют каждую запись, в то время как внешние ключи (Foreign Keys) связывают разные сущности.

В нашем приложении ключами являются ID Карт, Категорий и Кэшбека.

ER диаграмма помогает визуализировать структуру данных приложения, связи между объектами и обеспечить правильное хранение и управление информацией в базе данных.

* + 1. **Диаграмма модулей**

**теория, описательная часть диаграммы для разработки, приложение**

Диаграмма модулей (Module diagram) - это структурная диаграмма, которая отображает различные модули, компоненты или блоки (modules) системы, их взаимосвязи, зависимости и взаимодействия между ними. Эта диаграмма помогает визуализировать структуру системы как совокупность отдельных модулей и понять их взаимодействие.

Описательная часть Диаграммы модулей включает в себя следующие аспекты:

*1. Модули (Modules):*

- Описание: Модули представляют собой логически или функционально связанные компоненты или блоки программного обеспечения. Они могут включать в себя набор функций, классов, процедур и т.д.

*2. Связи и зависимости между модулями:*

- Описание: Диаграмма показывает, как модули связаны между собой, какие зависимости между ними существуют и как они взаимодействуют друг с другом.

*3. Интерфейсы модулей:*

- Описание: Диаграмма может включать указание интерфейсов и методов, которые предоставляются различными модулями для взаимодействия и передачи данных между ними.

Диаграмма модулей помогает в понимании архитектуры системы, определении модульной структуры приложения, их взаимодействии и связях, что способствует легкости поддержки и дальнейшему развитию приложения.

Диаграмма модулей нашего приложения представлена в Приложении 7.

* + 1. **Диаграмма классов**

**теория, описательная часть диаграммы для разработки, приложение**

Диаграмма классов (Class diagram) является частью модели UML (Unified Modeling Language) и используется для визуализации структуры классов в системе, их атрибутов, методов и отношений между классами. Диаграмма классов позволяет представить основные объекты и взаимосвязи в объектно-ориентированной системе.

Описательная часть Диаграммы классов включает в себя следующие аспекты:

*1. Классы (Classes):*

- Описание: Классы представляют объекты или сущности в системе, имеющие определенный набор атрибутов (полей) и методов (операций).

*2. Атрибуты (Attributes):*

- Описание: Атрибуты класса определяют характеристики этого класса. Они представляют данные, которые принадлежат классу.

*3. Методы (Methods):*

- Описание: Методы класса представляют операции, которые класс может выполнять. Они определяют поведение класса.

*4. Отношения между классами:*

- Описание: Диаграмма классов показывает связи между классами, такие как ассоциация, наследование, агрегация и композиция.

Диаграмма классов помогает визуализировать структуру классов и их взаимосвязи в вашем приложении, что облегчает понимание архитектуры и дизайна системы. Диаграмма классов отражена в Приложении 8.

1. **Разработка приложения**

При разработке программного обеспечения каждый этап, начиная с идеи и заканчивая реализацией, играет важную роль в создании сложных систем. Этот пункт нашего дипломного проекта является описанием специализированной части – этапа реализации, а конкретнее разработки.

Пройдя путь от идеи (воплощенной в простейшей блок-схеме в Приложении 9) до проектирования архитектуры и конкретной функциональности, мы сфокусируемся на важных этапах, которые включают создание экранов и компонентов, взаимодействие с пользователем, интеграцию с внешними сервисами, управление данными, использование шаблонов дизайна и описание ключевых функций приложения.

Итак, приступим к этапу разработки приложения, где каждый шаг имеет значение для создания продукта, соответствующего требованиям и ожиданиям пользователей.

* 1. **Реализация функциональности приложения согласно разработанной архитектуре и дизайну**

Прежде чем приступить к разработке полноценного приложения с UI интерфейсом, мной было принято важное решение о написании бизнес-логики в соответствии с алгоритмом на линейном коде. Это позволило прежде всего убедиться в правильности и эффективности работы алгоритма поиска карты с наибольшим кэшбеком, который был написан в онлайн IDE.

Прежде всего был написан текстовый алгоритм для бизнес-логики:

*1. Найти карту с наибольшим кэшбеком из набора карт.*

*2. Изначально установить максимальный кэшбек в 0.*

*3. Для каждой карты в наборе карт:*

*а. Если кэшбек текущей карты больше максимального кэшбека:*

*I. Обновить максимальный кэшбек значением кэшбека текущей карты.*

*II. Запомнить текущую карту как карту с наибольшим кэшбеком.*

*4. Вернуть карту с наибольшим кэшбеком.*

Затем на основе текстового алгоритма разработана блок-схема (Приложение 10) работы приложения.

После выполнения подготовительных шагов, а также вместе с изучением языка программирования Swift я приступил к написанию кода для реализации бизнес-логики будущего приложения.

//

//  main.swift

//  CommandLineVersion001

//

//  Created by Mac on 11.01.2024.

//

import Foundation

struct PlacePay: Equatable {

    let id: Int

    let name: String

    let icon: String

    static func ==(lhs: PlacePay, rhs: PlacePay) -> Bool {

        return lhs.id == rhs.id && lhs.name == rhs.name && lhs.icon == rhs.icon

    }

}

struct Card {

    let id: Int

    let name: String

    let placePay: [PlacePay]

    let cashbackPercentage: Double

    let icon: String

}

func findBestCard(for placePay: PlacePay, from cards: [Card]) -> Card? {

    var bestCard: Card?

    var maxCashbackPercentage: Double = 0

    for card in cards {

        if card.placePay.contains(placePay) && card.cashbackPercentage > maxCashbackPercentage {

            maxCashbackPercentage = card.cashbackPercentage

            bestCard = card

        }

    }

    return bestCard

}

let restaurantPay = PlacePay(id: 1, name: "Ресторан", icon: "restaurant\_icon")

let supermarketPay = PlacePay(id: 2, name: "Супермаркет", icon: "supermarket\_icon")

let cards = [

    Card(id: 1, name: "Card1", placePay: [restaurantPay], cashbackPercentage: 5.0, icon: "card\_icon1"),

    Card(id: 2, name: "Card2", placePay: [restaurantPay], cashbackPercentage: 7.5, icon: "card\_icon2"),

    Card(id: 3, name: "Card3", placePay: [restaurantPay], cashbackPercentage: 3.0, icon: "card\_icon3")

]

print("\*\*\* Реализация выбора карты с макс процентом кэшбека. Применение структур, методов. Вывод в консоль \*\*\* \n")

if let bestCard = findBestCard(for: restaurantPay, from: cards) {

    print("Выбрана карточка \(bestCard.name) с процентом кэшбека \(bestCard.cashbackPercentage)%")

} else {

    print("Не удалось найти подходящую карточку для данного места оплаты")

}

Первый релиз линейного

кода бизнес-логики

В первой версии кода были применены полученные на тот момент знания языка Swift, а именно:

- Структуры данных (структура PlacePay представляет место платежа с уникальным идентификатором, названием и иконкой, структура Card описывает карту с идентификатором, названием, списком мест платежа, процентом кэшбека и иконкой);

- Функции/методы (функция findBestCard принимает место платежа и список карт, и возвращает наилучшую карту с максимальным процентом кэшбека для этого места);

- Объекты и переменные (создаются объекты restaurantPay и supermarketPay для мест оплаты, задаются карточки с различными процентами кэшбека для ресторанов);

- Вызов функции findBestCard для места оплаты ресторан и выводится результат выбора наилучшей карты на консоль.

Этот код иллюстрирует использование структур данных, методов и итерацию по массиву карт, демонстрируя выбор наилучшей кредитной карты для конкретного места оплаты на основе процента кэшбека.

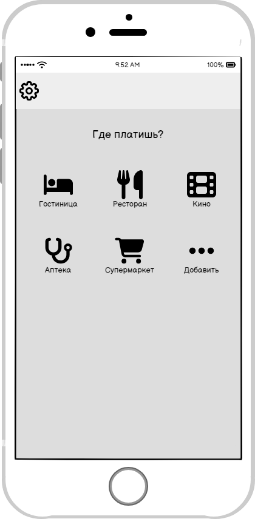
По мере расширения знаний при обучении на платформе и саморазвитию, первоначальный код постоянно подвергался изменениям, связанным с тестированием бизнес-логики, включением в нее опционалов и исключений, разделением зон ответственности между различными участками кода, применением паттерна MVC насколько это возможно в линейном коде. В итоге получился итоговый код бизнес-логики, представленный в Приложении 11.

Создание и контроль качества бизнес-логики на ранних этапах разработки помогает гарантировать стабильность и эффективность работы приложения. Теперь можно приступать к реализации архитектурных решений.

Начинался проект с бизнес-логики в линейном коде из одного файла менее 50 строк, а закончился файловой структурой из 31 директории и 63 файлов (Приложение 12).

* + 1. **Создание основных экранов и компонентов**

Формирование UI/UX – диаграммы строилось на основе личного опыта и предпочтений. Предполагалось, что будет основной экран, с которого пользователем будет производиться выбор места оплаты, а приложение будет «под капотом» проводить все необходимые расчеты и выдавать соответствующий результат. Все настройки – списки карт, категорий, взаимодействие с пользователем, предполагалось вынести на кнопку «Шестеренка».

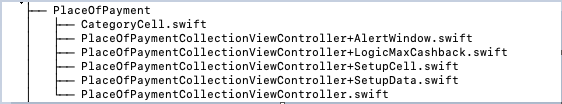


Первоначальный вариант основного окна приложения

В соответствии с этим решением и было реализовано основное окно, которое демонстрирует работу приложения, его бизнес-логику, а также корректно производит отбор мест оплаты, так как это не просто все категории, а уникальные категории, связанные с картой, то есть кэшбек. Приложение не будет отображать категории, экземпляры которых не существуют вместе с картами.

Для реализации основного окна были созданы соответствующие файлы, содержащие в себе:

1. Графическое представление окна (файл PlaceOfPaymentCollectionViewController.swift с расширениями[[3]](#footnote-3), которые вынесены в отдельные файлы и отделяются от основного файла через знак «+»):

[[4]](#footnote-4)

* CategoryCell.swift

Этот код представляет ячейку коллекции, которая отображает каждую категорию в списке с иконкой и названием, готовую к отображению в пользовательском интерфейсе.

* PlaceOfPaymentCollectionViewController.swift

Этот класс используется для отображения списка категорий оплаты в приложении, управляет данными, регистрирует и настраивает элементы интерфейса коллекции. Расширение делегата UICollectionViewDelegateFlowLayout дает контроль над определением размеров ячеек.

* PlaceOfPaymentCollectionViewController+AlertWindow.swift

Методы в этом расширении PlaceOfPaymentCollectionViewController предназначены для обработки действий пользователя по выбору карты оплаты, отображении информации о выборе карты и предоставлении возможности оплаты с дальнейшей логикой обработки. Код также использует встроенные контроллеры UIAlertController для отображения всплывающих сообщений и диалоговых окон в iOS приложении.

* PlaceOfPaymentCollectionViewController+LogicMaxCashback.swift

Метод findCardsWithMaxCashback из этого расширения предоставляет функционал по поиску и определению карт с максимальным кэшбэком для конкретной категории. Он использует данные о категории, кэшбэке и картах для выполнения соответствующих вычислений и формирования результатов поиска. Это та самая бизнес-логика, которая «обкатывалась» в линейном коде.

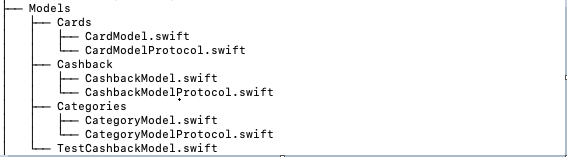
* PlaceOfPaymentCollectionViewController+SetupCell.swift

Это расширение добавляет функционал для настройки и управления ячейками коллекции в классе PlaceOfPaymentCollectionViewController. Оно определяет, как данные отображаются в ячейках и как обрабатывать пользовательские действия в коллекции при прокрутке и нажатиях на элементы. Еще один элемент логики, касающейся отображения данных – демонстрация только уникальных категорий, имеющих связь с картами.

* PlaceOfPaymentCollectionViewController+SetupData.swift

Это расширение инициализирует начальные данные для отображения в коллекции мест оплаты. Оно заполняет информацию о категориях, картах и кэшбеках, необходимых для правильного отображения и работы функционала коллекции. Кроме того, оно помогает определить уникальные категории для дальнейшего использования в приложении.

1. Модели

[[5]](#footnote-5)

* CardModel.swift
* CardModelProtocol.swift

Протокол определяет свойства, которые должны быть реализованы объектами, соответствующими этому протоколу, а структура CardModel реализует протокол CardModelProtocol. В структуре определены свойства id, name и image, соответствующие свойствам протокола. Также имеется инициализатор, который принимает id, name и image, чтобы установить значения этих свойств при создании экземпляра CardModel.

* CashbackModel.swift
* CashbackModelProtocol.swift

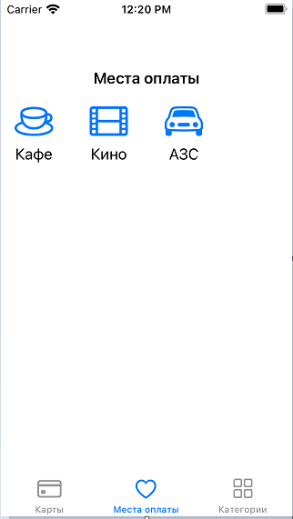
Протокол CashbackProtocol определяет свойства, которые должны быть реализованы объектами, соответствующими этому протоколу, а структура CashbackModelр еализует протокол CashbackProtocol. Определяет свойства id, categoryID, cardID и percentage для модели кэшбека. А также имеет инициализатор, который устанавливает значения свойств модели при создании экземпляра CashbackModel.

* CategoryModel.swift
* CategoryModelProtocol.swift

Протокол CategoryModelProtocol описывает свойства, которые должны быть реализованы объектами, соответствующими этому протоколу. Структура CategoryModel реализует протокол CategoryModelProtocol, определяет свойства id, name, icon и mcc для объектов модели категории. А конструктор init принимает параметры id, name, icon и mcc, чтобы установить значения свойств при создании экземпляра CategoryModel.

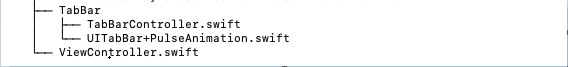
В соответствии с архитектурой предполагалось «навешивание» на это окно кнопки для настроек приложения. Однако, по мере изучения возможностей UI дизайна Swift и инструментария, предоставляемого UIKit[[6]](#footnote-6), было принято решение использовать инструмент UITabBarController[[7]](#footnote-7). С его помощью имеется возможность демонстрации нескольких основных окон за счет использования меню с визуализацией элементов, а именно в нашем приложении – «Карты» (демонстрация списка карт и управления ими), «Места оплаты» (основной экран для диалога с пользователем в части выбора места оплаты, соответствующего той или иной категории), «Категории» (демонстрация списка категорий и управления ими).

Таким образом основное окно приложения претерпело изменения, как и код.



Итоговый вариант основного окна приложения

С целью реализации таких изменений были добавлены некоторые файлы, которые позволили добавить TabBar на основное окно, а также настроить его соответствующим образом.



* ViewController.swift

Код этого класса инициализирует и добавляет TabBarController в качестве дочернего контроллера к ViewController для дальнейшего отображения табов и навигации на экране.

* TabBarController.swift

Этот класс инициализирует и настраивает TabBar с различными вкладками, устанавливает порядок вкладок и вызывает анимацию для TabBar при отображении.

* UITabBar+PulseAnimation.swift

Это расширение обогащает UITabBar новой функциональностью, позволяя добавить анимацию «выезжания» TabBar из левой стороны, чтобы продемонстрировать навыки работы с анимацией, а также сделать интерфейс более динамичным и привлекательным для пользователей.

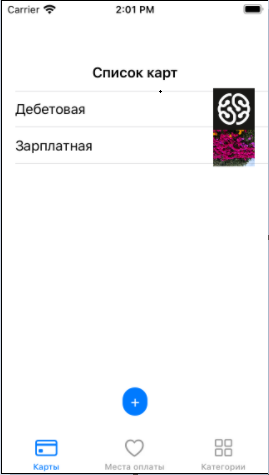
Таким образом мы получили основные окна и дальнейшее направление развития приложения:

1. Основное окно приложения – Места оплаты
2. Окно со списком Карты
3. Окно со списком Категории.

Переключение между этими окнами реализовано посредством контроллера UITabBarController фреймворка UIKit. Дальнейшее развитие приложение шло в соответствии с этим маршрутом – реализация функционала модуля/ветки «Карты», реализация функционала модуля/ветки «Категории», организация хранения данных при помощи CoreData[[8]](#footnote-8), настройка связей между сущностями БД, реализация функционала «Кэшбек» внутри модуля/ветки «Карты».

* + 1. **Обработка событий и взаимодействие с пользователем (ветка «Карты»)**

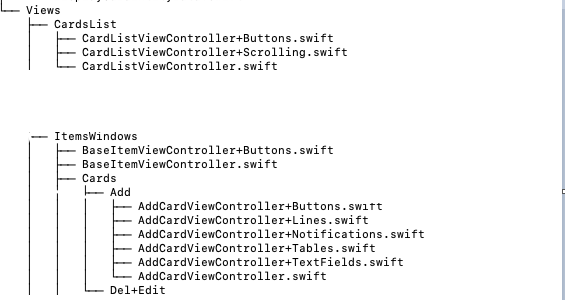
Одна из кнопок TabBar отвечает за демонстрацию списка карт. Это одно из трех основных окон приложения, которое показывает список карт, а также позволяет вызвать диалоговое окно для добавления новой карты. При реализации этого окна использовался подкласс UITableViewController[[9]](#footnote-9).



Окно со списком карт

Для реализации данного функционала в код было добавлено несколько файлов, отвечающих за различный функционал:

1. Контроллеры, отвечающие за визуализацию



Демонстрация списка карт с элементами взаимодействия с пользователем:

* CardListViewController.swift

Этот класс, по сути, управляет отображением списка карт, обновлением данных и реагированием на действия пользователя в связи с каждой картой. Предполагается, что при нажатии на ячейку (строку) таблицы открывается карточка карты в которой отображается полная информация о карте и кэшбеках, привязанных к ней. Внутри карточки должен быть реализован функционал редактирования и удаления карты.

* CardListViewController+Buttons.swift

Расширение CardListViewController позволяет легко добавить кнопку для добавления карты на экран контроллера и обрабатывать нажатие на эту кнопку для открытия окна добавления карты с последующим обновлением списка и прокруткой таблицы. Кстати, при создании и настройки кнопки применялся собственный шаблон для круглой кнопки.

* CardListViewController+Scrolling.swift

Это расширение обеспечивает автоматическую прокрутку таблицы, чтобы пользователь мог видеть последние добавленные карточки без необходимости ручного прокручивания.

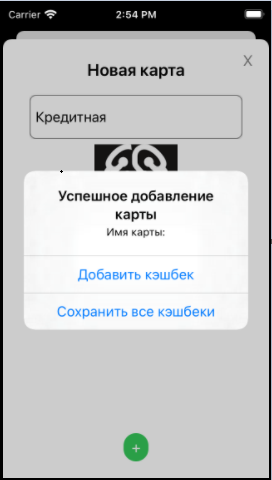
Демонстрация окон для работы с элементами списка карт (добавление, редактирование, удаление):

* BaseItemViewController.swift
* BaseItemViewController+Buttons.swift

Этот класс и расширение предоставляют базовую структуру для контроллеров элементов приложения, предварительно настраивая интерфейс с заголовком и кнопкой закрытия. При необходимости другие контроллеры могут наследовать и расширять этот базовый функционал для конкретных элементов.

* ├── Cards
* ├── Add
* AddCardViewController.swift
* AddCardViewController+Buttons.swift
* AddCardViewController+Lines.swift
* AddCardViewController+Notifications.swift
* AddCardViewController+Tables.swift
* AddCardViewController+TextFields.swift

Класс AddCardViewController с расширениями отвечает за создание новой карты, включает элементы интерфейса для ввода информации о карте, добавления кэшбэков, изображения и управления ими[[10]](#footnote-10).



Окно добавления новой карты и кэшбеков к ней

* └── Del+Edit

Эта папка создана для реализации визуализации окна с карточкой карты, а также с элементами управления и взаимодействия с пользователем при удалении карты или ее редактировании[[11]](#footnote-11).

1. Менеджеры, отвечающие за работу с данными



* CardManagerProtocol.swift
* CardManager.swift

Протокол и его реализация в CardManager предоставляют структуру для управления объектами карт, включая их получение, добавление, редактирование и удаление. Они следуют принципам разделения ответственности и позволяют эффективно управлять картами в приложении[[12]](#footnote-12).

* CardDataProvider.swift

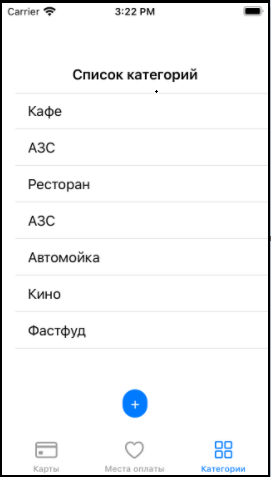
Этот класс отвечает за доступ и манипуляции с данными карт в CoreData.

При реализации ветки «Карты» я старался придерживаться паттерна MVC[[13]](#footnote-13), хотя при раскладке схемы взаимодействия с пользователем и наложением на нее диаграммы методов, могло показаться, что реализация ближе к MVVM[[14]](#footnote-14).

* + 1. **Обработка событий и взаимодействие с пользователем (ветка «Категории»)**

Третьим основным окном является окно – Категории. По сути, ветка практически полностью копирует функционал ветки «Карты», в связи с чем я решил учесть предыдущие ошибки и параллельно, предварительно самостоятельно изучив, применить паттерн Observer[[15]](#footnote-15) для реализации обновления списка карты при внесении изменений в БД.

Вообще, существует список категорий, используемый банками и агентами платежных систем, который предоставляется платежной системой для настройки платежных терминалов и банковских сервисов. Насколько мне известно, что этот сервис платный и только для зарегистрированных в ПС агентов. Это один из возможных путей реализации взаимодействия с внешними структурами через API[[16]](#footnote-16), например, для автоматического наполнения списка категорий.



Окно со списком категорий

Файловая структура части проекта, реализующая этот функционал выглядит следующим образом:

1. Контроллеры, отвечающие за визуализацию



* CategoryListViewController.swift

Этот класс управляет отображением списка категорий, обновлением данных и реагированием на действия пользователя в связи с каждой категорией. Предполагается, что при нажатии на ячейку (строку) таблицы открывается карточка категории в которой отображается полная информация о ней. Внутри карточки должен быть реализован функционал редактирования и удаления категории.

* CategoryListViewController+Buttons.swift

Расширение CategoryListViewController позволяет легко добавить кнопку для добавления категории на экран контроллера и обрабатывать нажатие на эту кнопку для открытия окна добавления категории с последующим обновлением списка и прокруткой таблицы.

* CategoryListViewController+Scrolling.swift

Это расширение обеспечивает автоматическую прокрутку таблицы, чтобы пользователь мог видеть последние добавленные карточки без необходимости ручного прокручивания.

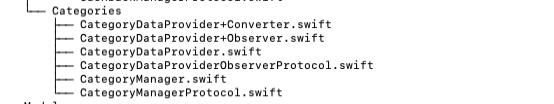
* └── Category
* ├── Add
* AddCategoryViewController.swift
* AddCategoryViewController+Buttons.swift
* AddCategoryViewController+TextFields.swift

Класс AddCategoryViewController с расширениями отвечает за создание новой категории, включает элементы интерфейса для ввода информации о категории.

* └── Del+Edit

Эта папка создана для реализации визуализации окна с карточкой категории, а также с элементами управления и взаимодействия с пользователем при удалении категории или ее редактировании[[17]](#footnote-17).

1. Менеджеры, отвечающие за работу с данными и обеспечивающие взаимодействие в рамках паттерна Observer



* CategoryManagerProtocol.swift
* CategoryManager.swift

CategoryManager и CategoryManagerProtocol предоставляют логику для управления категориями с возможностью получения, добавления, редактирования и удаления категорий. Благодаря использованию протокола и разделению ответственности, код становится более модульным и расширяемым.

* CategoryDataProvider.swift

Этот класс CategoryDataProvider отвечает за доступ, добавление и уведомление об изменениях в данных категорий. При добавлении новой категории он обновляет записи в хранилище, уведомляя наблюдателей о внесенных изменениях.

* CategoryDataProvider+Converter.swift

Расширение класса CategoryDataProvider для конвертации сущности CategoryEntity в модель CategoryModel.

Метод внутри расширения помогает понятным образом конвертировать объекты из одного формата в другой, облегчая передачу данных между различными типами объектов

* CategoryDataProvider+Observer.swift

Это расширение класса CategoryDataProvider для управления наблюдателями.

Эти методы внутри класса позволяют добавлять наблюдателей и уведомлять их о произошедших изменениях, обеспечивая реакцию наблюдателей на изменения данных в CategoryDataProvider.

* CategoryDataProviderObserverProtocol.swift

Протокол CategoryDataProviderObserverProtocol выступает в качестве своеобразного интерфейса для классов, которые хотят наблюдать за изменениями данных категорий в CategoryDataProvider.

При реализации ветки «Категории» архитектурный паттерн остался прежним – MVC, однако был добавлен поведенческий паттерн Observer, который позволил производить обновление списка категорий в зависимости от изменений, вносимых в БД через CategoryDataProvider.

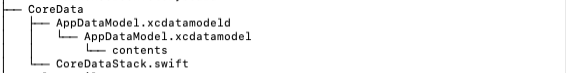
* + 1. **Управление данными: взаимодействие с базой данных**

После реализации веток «Карты» и «Категории» через менеджеры, которые взаимодействуют со структурами из моделей, появилась необходимость сохранения данных и дальнейшей работы с ними. Было принято решение использовать фреймворк CoreData, который предоставляет удобный способ управления объектами в приложении и их хранения в базе данных. CoreData предоставляет ряд возможностей, которые делают его широко используемым в мобильной и десктопной разработке. Вот несколько ключевых возможностей CoreData:

1. Управление объектами данных
2. Отслеживание изменений
3. Кеширование данных
4. Отображение данных в пользовательском интерфейсе
5. Работа с различными источниками данных
6. Миграция данных
7. Интеграция с Core Data Stack

Использование CoreData позволяет разработчикам создавать приложения с более удобным управлением данными, улучшенной производительностью и возможностью интеграции данных в пользовательский интерфейс.

Файловая структура CoreData в нашем проекте имеет следующий вид:



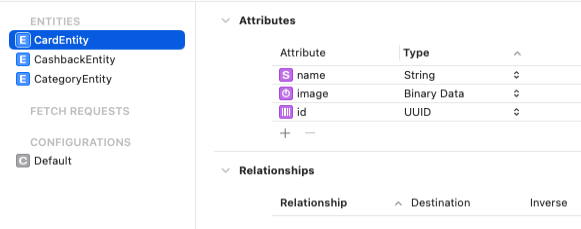
* AppDataModel.xcdatamodeld

Это файл CoreData Data Model, который содержит описание модели данных в приложении. В этом файле определяются сущности (Entities), их атрибуты (Attributes), отношения (Relationships) и другие характеристики данных, которые ваше приложение будет использовать.

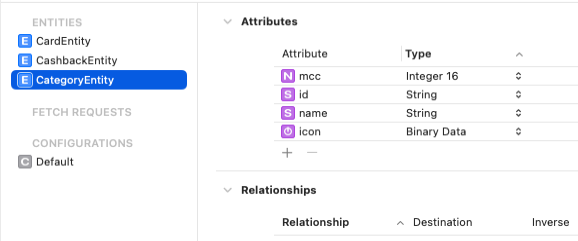
В нашем приложении здесь определены 3 сущности:

- CardEntity

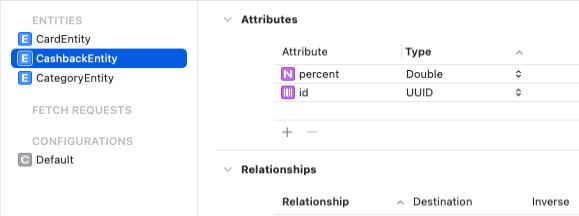
с атрибутами id, name, image



- CategoryEntity

с атрибутами id, name, image, mcc

- CashbackEntity

с атрибутами id, percent

Дальнейшие настройки AppDataModel.xcdatamodeld заключаются в определении отношений (Relationships) между сущностями.

* CoreDataStack.swift

CoreDataStack.swift[[18]](#footnote-18) является реализацией класса, который представляет стек CoreData для работы с базой данных в приложениях iOS. Его код организует основную структуру и функциональность для работы с базой данных через CoreData в iOS приложениях. Он обеспечивает удобное управление контекстом CoreData, сохранение данных и обработку уведомлений о жизненном цикле приложения для поддержки целостности данных.

* + 1. **Работа с внешними сервисами и API**

На данном этапе разработки в приложении отсутствует взаимодействие с внешними сервисами и API. Подразумевается следующее дальнейшее развитие в этом направлении:

1. Взаимодействие с платежной системой для автоматического получения списка категорий с MCC посредством API.
2. Взаимодействие с платежной системой для получения информации при совершении транзакции, а именно MCC и TID устройства на котором проведена операция.
3. При реализации модуля сравнения MCC из списка мест оплаты и MCC, полученного от терминала, отправка «наружу» сообщения о несоответствии MCC с указанием TID устройства.
4. Взаимодействие с платежной системой для получения TID всех устройств для дальнейшей реализации определения места оплаты при помощи геолокации.

Разработка этих модулей и расширение функционала приложения возможно только после интеграции приложения с платежным сервисом или получением статуса агента платежной системы.

* + 1. **Использование шаблонов**

При написании кода приложения нашлось место и кастомным шаблонам, которые были разработаны под определенные нужды и цели.

1. Класс BaseItemViewController.

Этот код представляет базовый класс контроллера элемента с настраиваемыми элементами интерфейса для отображения заголовка и кнопки закрытия элемента. В дальнейшем применяется как базовый класс для классов создания окон добавления карты и категории, а также просмотра их карточек.

class BaseItemViewController: UIViewController {

    /// Лейбл для отображения заголовка элемента

    lazy var titleLabel: UILabel = {

        let label = UILabel()

        label.textAlignment = .center

        label.font = UIFont.boldSystemFont(ofSize: 20.0)

        return label

    }()

    /// Кнопка закрытия элемента

    let closeButton: RoundButtonWithSettings = {

        let button = RoundButtonWithSettings(title: "X", backgroundColor: .white, titleColor: .gray)

        return button

    }()

    override func viewDidLoad() {

        super.viewDidLoad()

        view.backgroundColor = .white

        setupUI()

    }

    /// Настройка UI элементов

    func setupUI() {

        view.addSubview(titleLabel)

        ConstraintsConstructor.setupViewConstraints(view: titleLabel, contentView: view, distanceFromYMultiplier: -0.4, distanceFromXMultiplier: 0)

        setupCloseButton()

    }

}

1. Класс RoundButtonWithSettings.

Этот класс представляет собой кастомную круглую кнопку, унаследованную от UIButton, с добавлением специфических настроек для внешнего вида, а именно:

- title: Название (один символ) на кнопке

- backgroundColor: Цвет кнопки

- titleColor: Цвет шрифта

Посредством этого шаблона можно создать круглую кнопку нужного цвета, с нужным символом посередине и цветом шрифта символа.

Пример использования:

*let openAddCategoryWindowButton = RoundButtonWithSettings(title: "+", backgroundColor: .systemBlue, titleColor: .white)*

class RoundButtonWithSettings: UIButton {

    private var originalBackgroundColor: UIColor?

    init(title: String, backgroundColor: UIColor, titleColor: UIColor) {

        super.init(frame: .zero)

        // Сохраняем оригинальный цвет фона кнопки

        originalBackgroundColor = backgroundColor

        // Настройка внешнего вида кнопки

        self.setTitle(title, for: .normal)

        self.backgroundColor = backgroundColor

        setTitleColor(titleColor, for: .normal)

        self.translatesAutoresizingMaskIntoConstraints = false

        //self.widthAnchor.constraint(equalTo: self.heightAnchor).isActive = true

        addTarget(self, action: #selector(buttonPressed), for: .touchDown)

        addTarget(self, action: #selector(buttonReleased), for: [.touchUpInside, .touchUpOutside])

    }

    required init?(coder: NSCoder) {

        super.init(coder: coder)

    }

    override func layoutSubviews() {

        super.layoutSubviews()

        layer.cornerRadius = bounds.width / 2

    }

    @objc private func buttonPressed() {

        backgroundColor = UIColor.lightGray

    }

    @objc private func buttonReleased() {

        backgroundColor = originalBackgroundColor

    }

}

1. Класс PopupStub.

Данный класс отображает всплывающее окно с заданным заголовком и сообщением. Создан для использования при формировании заглушек. Параметры настройки:

- title: Заголовок всплывающего окна

- message: Сообщение, отображаемое во всплывающем окне

- viewController: ViewController, на котором нужно отобразить всплывающее окно

class PopupStub {

    static func showPopup(title: String, message: String, viewController: UIViewController) {

        let alert = UIAlertController(title: title, message: message, preferredStyle: .alert)

        let attributedTitleString = NSAttributedString(string: title, attributes: [

            NSAttributedString.Key.foregroundColor: UIColor.red

        ])

        alert.setValue(attributedTitleString, forKey: "attributedTitle")

        alert.addAction(UIAlertAction(title: "OK", style: .default, handler: nil))

        viewController.present(alert, animated: true, completion: nil)

    }

}

1. Метод из вспомогательного класса ConstraintsConstructor – setupViewConstraints().

Этот метод позволяет осуществлять настройку констрейнтов для кнопки, размещаемой с учетом отступа от нижнего края представления. Параметры:

- button: Кнопка, для которой настраиваются констрейнты.

- contentView: Представление, к которому добавляется кнопка.

- distanceFromBottomMultiplier: Множитель, определяющий отступ кнопки от нижнего края представления. Значение 0.0 означает размещение у самого нижнего края, 1.0 - размещение у верхнего края представления.

Пример использования:

*ConstraintsConstructor.setupViewConstraints(view: addNewCategoryButton, contentView: view, distanceFromYMultiplier: 0.4, distanceFromXMultiplier: 0)*

static func setupViewConstraints(view: UIView, contentView: UIView, distanceFromYMultiplier: CGFloat, distanceFromXMultiplier: CGFloat) {

        view.translatesAutoresizingMaskIntoConstraints = false // Отключаем автоматическую генерацию констрейнтов для вида

        //let safeGuide = contentView.safeAreaLayoutGuide // Получаем Safe Area Layout Guide

        let centerY = contentView.centerYAnchor // Центрирование по вертикали

        let centerX = contentView.centerXAnchor // Центрирование по горизонтали

        NSLayoutConstraint.activate([

            // Устанавливаем констрейнты с использованием Safe Area для правильного позиционирования вида

            view.centerYAnchor.constraint(equalTo: centerY, constant: contentView.bounds.height \* distanceFromYMultiplier), // Размещение вида относительно центра по вертикали

            view.centerXAnchor.constraint(equalTo: centerX, constant: contentView.bounds.width \* distanceFromXMultiplier), // Размещение вида относительно центра по горизонтали

        ])

    }

Применение шаблонов при разработке приложений играет ключевую роль в обеспечении эффективности, удобства и согласованности в процессе создания программного продукта. Вот несколько ключевых моментов, подводящих итог значимости применения шаблонов:

1. Удобство и повторное использование кода:

- шаблоны позволяют удобно организовать и стандартизировать код, что ускоряет разработку новых функций и упрощает его поддержку.

1. Согласованность дизайна и пользовательского опыта:

- применение шаблонов в дизайне помогает создать единое визуальное восприятие приложения, что улучшает пользовательский опыт и делает интерфейс более интуитивным.

1. Повышение производительности и качества:

- использование шаблонов способствует улучшению качества кода, уменьшению вероятности ошибок и облегчает тестирование, что в итоге повышает производительность приложения.

1. Эффективное масштабирование и поддержка:

- хорошо спроектированные шаблоны упрощают добавление новых функций, расширение приложения и его обслуживание на протяжении всего жизненного цикла.

1. Создание структурированного и модульного кода:

- применение шаблонов помогает создать структурированный код, разделить его на независимые модули и упростить его понимание и сопровождение.

1. Сокращение времени разработки:

- использование готовых шаблонов позволяет ускорить процесс разработки, так как не приходится писать код "с нуля", а можно использовать проверенные и оптимизированные решения.

1. Повышение профессионализма и стандартизации:

- применение шаблонов способствует созданию профессионального и структурированного кода, соответствующего стандартам и лучшим практикам разработки.

В итоге, применение шаблонов в разработке приложений не только улучшает качество и производительность программного продукта, но также обеспечивает команду разработчиков средствами для эффективного и согласованного творчества.

1. Заключение

В ходе выполнения этого дипломного проекта мне удалось достаточно глубоко в рамках данных знаний и ограничения по времени исследовать тему, связанную с применением языка программирования Swift в разработке мобильного приложения, а также иных аспектов и направлений IT индустрии, связанных в единый комплекс мероприятий от развертывания виртуальной машины до создания архитектуры приложения. В ходе реализации диплома удалось разработать и придать основные очертания для моей идеи в части создания приложения, помогающего пользователю с выбором наиболее выгодной платежной карты.

Этот дипломный проект не только позволил мне приобрести новый опыт и знания, но и продемонстрировал мою готовность к решению сложных задач и стремление к профессиональному росту.

Я бы хотел довести свой программный продукт до логического завершения, так как знаю куда надо двигаться и какие ресурсы для этого использовать.

* 1. **Суммирование результатов разработки и достижений целей дипломной работы**

В результате работы над дипломным проектом, пусть и не полностью, но успешно были достигнуты поставленные цели и задачи. Удалось разработать приложение, которое отвечает требованиям и архитектуре, хотя и с некоторыми отступлениями, но как показала практика – без них никуда. Приложение содержит необходимую функциональность, а также имеет достаточный потенциал для развития и расширения.

Работа над дипломом позволила убедиться в необходимости совершения всех предварительных мероприятий, направленных на подготовку к разработке. Они позволили не переписывать код по несколько раз, а создавать вполне рабочий продукт постепенно, как бы обволакивая логику «под капотом» в гибкую обертку с помощью различных фреймворков.

* 1. **Предложение дальнейших возможностей для развития приложения**

Для дальнейшего развития приложения необходимо для начала реализовать весь базовый функционал (добавить удаление и добавление элементов сущностей), корректно настроить работу CoreData с учетом связей между сущностями, реализовать менеджеры и вью (в ветке «Карты») для работы с сущностью CashbackEntity.

Что бы я продолжил реализовывать дальше:

1. Формирование сущности DealEntity. Для ведения статистики, дальнейшей обработки данных по каждой сделке и т.п.
2. После дополнительного изучения внесение изменений в интерфейс – по моему мнению его можно сделать более лаконичным в части работы с элементами системы (карты, категории).
3. После консультаций и более детального изучения вопроса – подключение к платежной системе с целью получения через API списка категорий, а также данных по платежу.
4. Реализация модуля отправки сообщений в платежную систему.

Эти шаги могут помочь приложению получить законченный вид, продолжить свое развитие и оставаться актуальным и конкурентоспособным на рынке.

1. Список литературы и ресурсов
2. Книга "Swift Programming: The Big Nerd Ranch Guide" - <https://www.bignerdranch.com/books/swift-programming/>
3. Coursera курс "Swift for Beginners" - <https://www.coursera.org/learn/swift-for-beginners>
4. Официальные руководства GitHub - <https://guides.github.com/>
5. GitHub Learning Lab - <https://lab.github.com/>
6. Ресурсы Ray Wenderlich - <https://www.raywenderlich.com/>
7. Ресурсы AppCoda - <https://www.appcoda.com/>
8. URLSession ресурсы от Hacking with Swift - <https://www.hackingwithswift.com/read/5/overview>
9. Codable на Swift by Sundell - <https://www.swiftbysundell.com/basics/codable/>

Список приложений:

Приложение 1. Паттерн MVC

Приложение 2. Паттерн Observer

Приложение 3. Use-case диаграмма

Приложение 4. Первоначальный вариант UI/UX

Приложение 5. UI/UX рабочего релиза

Приложение 6. ER диаграмма (сущность-связь)

Приложение 7. Диаграмма модулей

Приложение 8. Диаграмма классов

Приложение 9. Простейшая блок-схема приложения

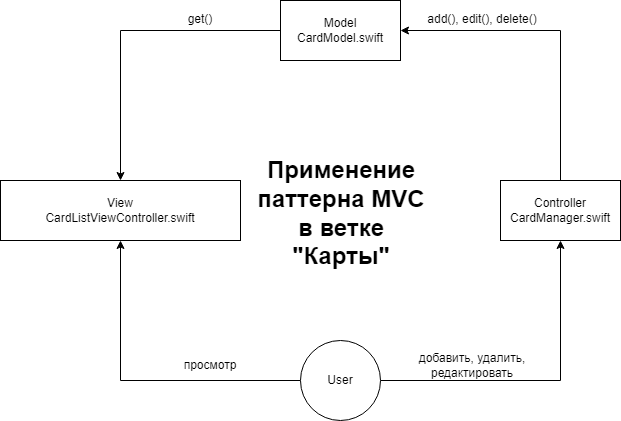
Приложение 10. Блок-схема бизнес-логики

Приложение 11. Линейный код бизнес-логики

Приложение 12. Файловая структура проекта

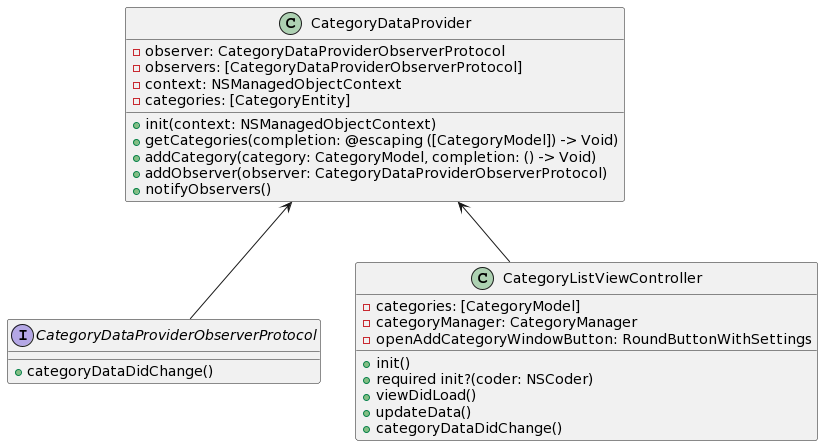
Приложение 1

Паттерн MVC



Приложение 2

Паттерн Observer



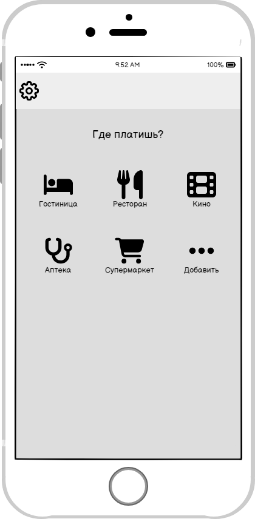
Приложение 3

Use-case диаграмма

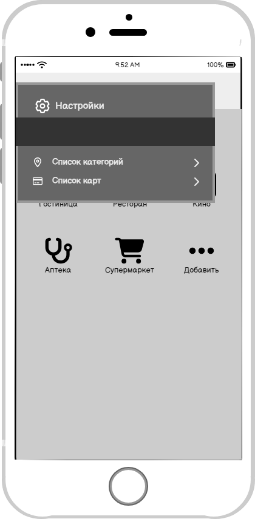


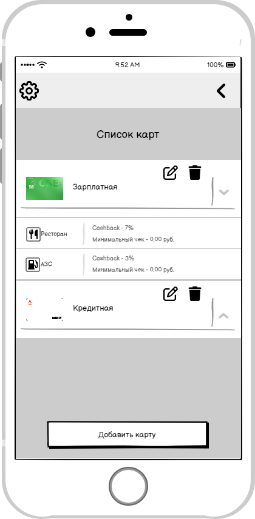
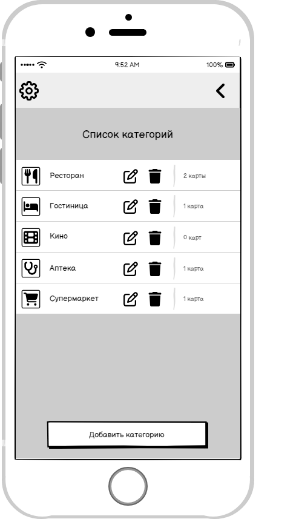
Приложение 4

Первоначальный вариант UI/UX



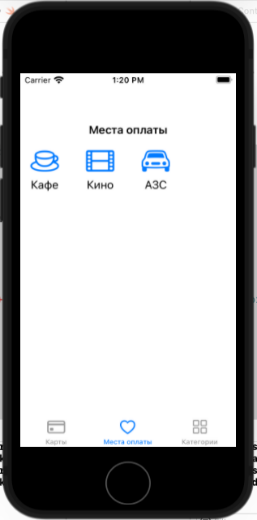
Результат



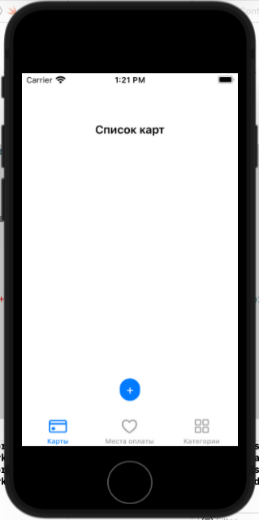
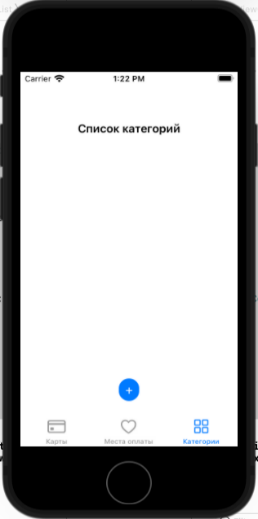


Приложение 5

UI/UX рабочего релиза

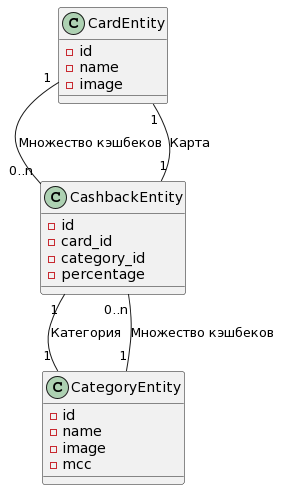


Результат



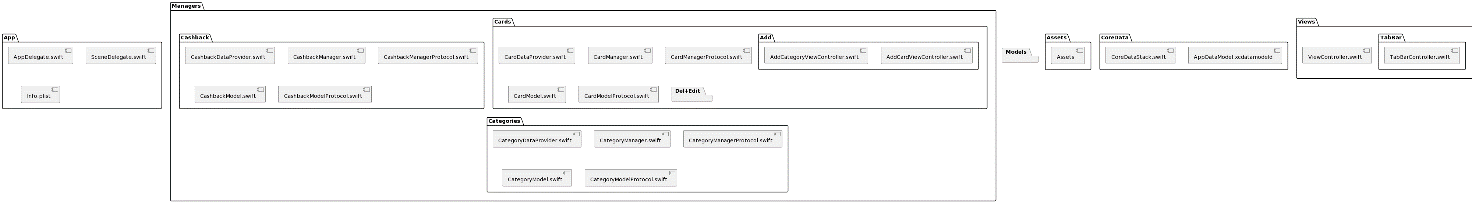
Приложение 6

ER диаграмма (сущность – связь)



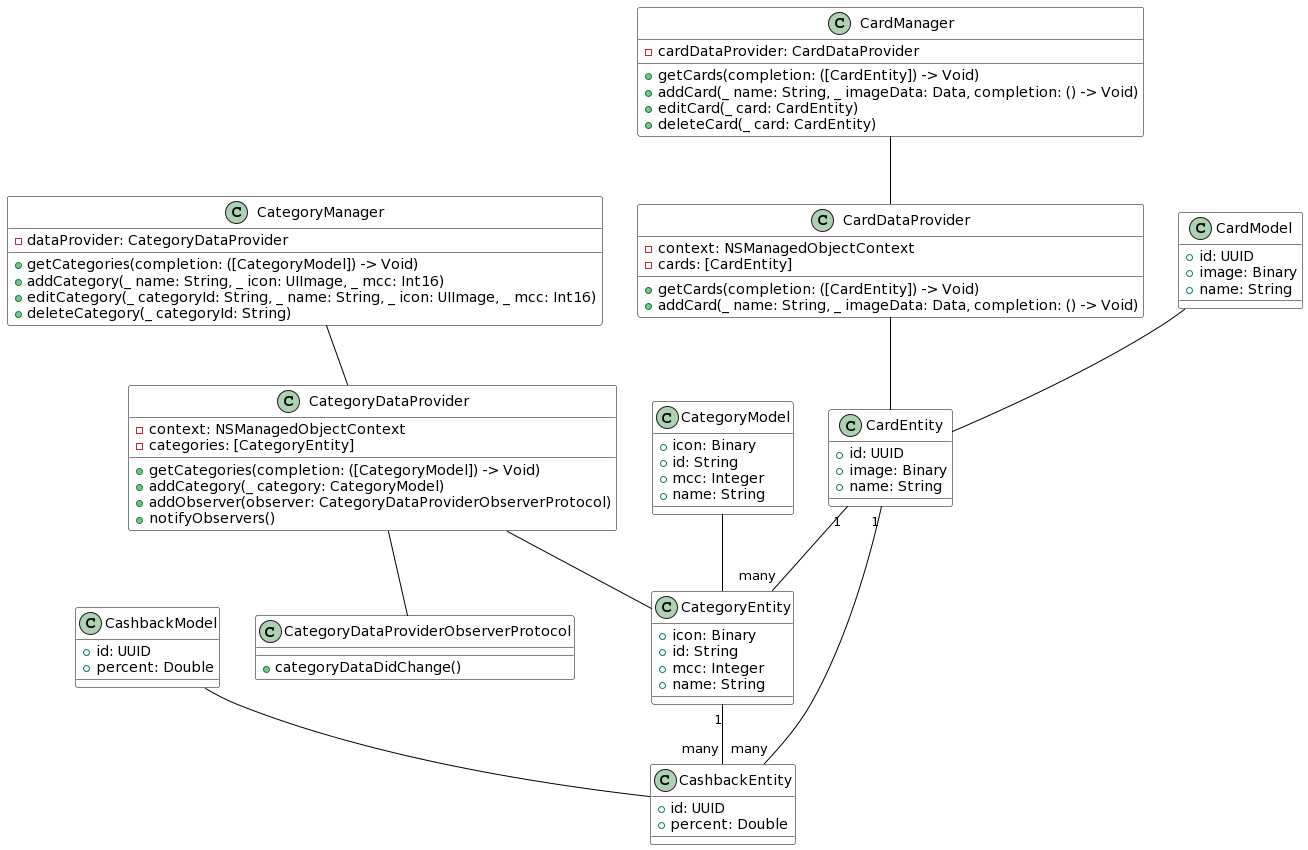
Приложение 7

Диаграмма модулей



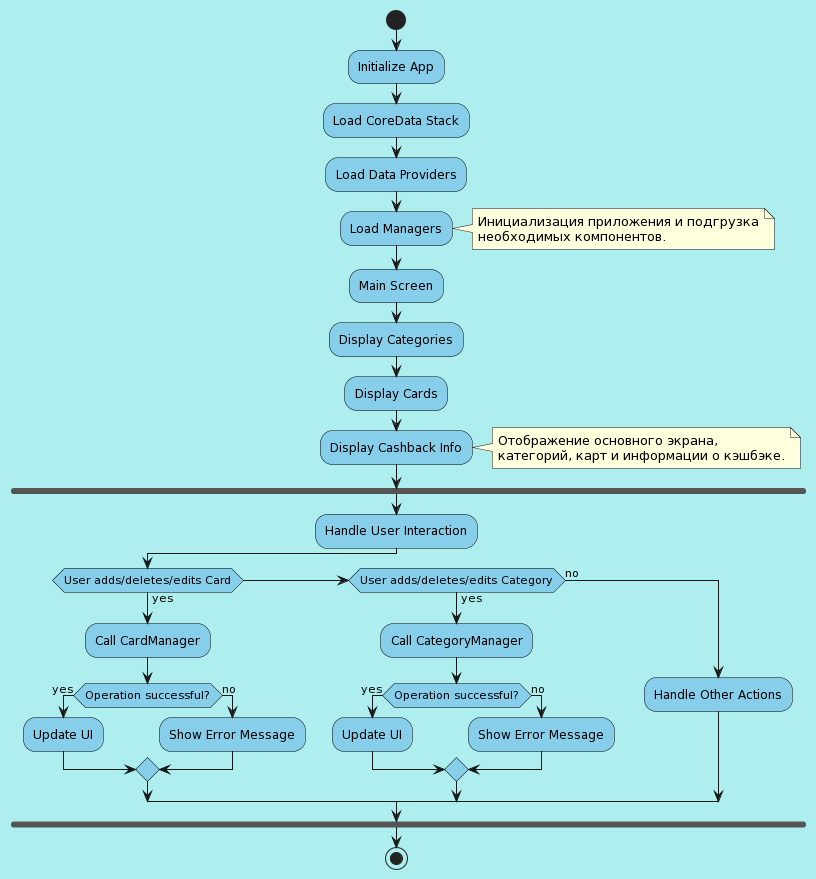
Приложение 8

Диаграмма классов



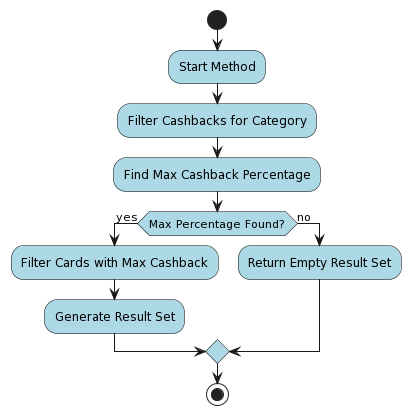
Приложение 9

Простейшая блок-схема приложения



Приложение 10

Блок-схема бизнес-логики



Приложение 11

Линейный код бизнес-логики

//

//  main.swift

//  CommandLineFinalVersion100

//

//  Created by Mac on 11.01.2024.

//

import Foundation

/\*Model (Модель)\*/

/\*КАРТА\*/

/\*\*

Протокол для класса Card

\*/

protocol CardProtocol {

    var id: Int {get}

    var name: String {get}

    var icon: String {get}

    var paymentCategories: [PaymentCategory] {get}

}

/\*\*

Класс с параметрами и инициализацией карты

\*/

class Card {

    let id: Int // идентификатор карты

    let name: String // наименование карты

    let icon: String // изображение карты (пока в таком виде)

    //var variableProperties: [String: Any] // словарь для хранения дополнительных свойств карты

    var paymentCategories : [PaymentCategory] // массив категорий с процентами кэшбека

    /\*\*

    Инициализация объекта класса

    - Parameters:

        - id: идентификатор карты

        - name: наименование карты

        - icon: изображение карты

    \*/

    init(id: Int, name: String, icon: String) {

        self.id = id

        self.name = name

        self.icon = icon

    //    self.variableProperties = [:]

        self.paymentCategories = []

    }

}

/\*\*

Расширение класса Card

\*/

extension Card: CardProtocol {}

/\*КАТЕГОРИИ КЭШБЕКА\*/

/\*\*

Протокол для структуры PaymentCategory

\*/

protocol PaymentCategoryProtocol {

    var category: String { get }

    var cashbackPercentage: Double { get }

    var minimumPurchaseAmount: Double { get }

}

/\*\*

Структура категорий и процентов кэшбека

\*/

struct PaymentCategory {

    let category: String // наименование категории/места оплаты

    let cashbackPercentage: Double // процент кэшбека для категории

    let minimumPurchaseAmount: Double // размер минимального чека

}

/\*СБОРКА КАРТЫ\*/

/\*\*

Протокол для класса создания карты

\*/

protocol CardBuildProtocol {

    //func setID(\_ id: Int) -> Self

    func setName(\_ name: String) -> Self

    func setIcon(\_ icon: String) -> Self

    func addPaymentCategory(\_ category: PaymentCategory) -> Self

    func build() -> Card?

}

/\*\*

Класс для создания карты

В этом классе используются опциональные типы переменных, т.к. не все параметры обязательны

для указания при создании карты. Опциональный тип позволяет установить значение параметра,

если оно передано, или оставить его как nil, если данные отсутствуют.

\*/

class CardBuilder {

    private var id: Int? // идентификатор карты

    private var name: String? // наименование карты

    private var icon: String? // изображение карты

    //private var variableProperties: [String: Any] = [:]

    private var paymentCategories: [PaymentCategory] = [] // массив категорий с процентами кэшбека

    enum CardBuildError: Error {

        case missingRequiredProperties

    }

}

extension CardBuilder: CardBuildProtocol {

// сеттеры

    func setID(\_ id: Int) -> Self {

        self.id = id

        return self

    }

    func setName(\_ name: String) -> Self {

        self.name = name

        return self

    }

    func setIcon(\_ icon: String) -> Self {

        self.icon = icon

        return self

    }

// метод добавления дополнительных свойств карты

    // func addVariableProperty(variableKey: String, variableValue: Any) -> CardBuilder {

    //     self.variableProperties[variableKey] = variableValue

    //     return self

    // }

// метод добавления категории cashback в карту

    func addPaymentCategory(\_ category: PaymentCategory) -> Self {

        // проверка на дублирование категорий (по имени) на одной карте

        guard !paymentCategories.contains(where: { $0.category == category.category })

        else {

            print("Внимание: Категория \(category.category) уже присутствует на карте \(name!) и не будет добавлена\n")

            return self

        }

        self.paymentCategories.append(category)

        return self

    }

// метод «сборки» карты с проверкой на наличие обязательных полей и обработкой исключений

    func build() -> Card? {

        do {

            guard let id = id, let name = name, let icon = icon else {

                throw CardBuildError.missingRequiredProperties

        }

        let newCard = Card(id: id, name: name, icon: icon)

        //newCard.variableProperties = variableProperties

        newCard.paymentCategories = paymentCategories

        // проверка процента кэшбека на положительное значение

        for category in newCard.paymentCategories {

            if category.cashbackPercentage <= 0 {

                print("Внимание: Категория \(category.category) карты \(name) имеет некорректный cashback (\(category.cashbackPercentage)%) и была удалена\n")

                newCard.paymentCategories.removeAll { $0.category == category.category }

                }

            }

            return newCard

        } catch CardBuildError.missingRequiredProperties {

            print("Ошибка: Не удалось создать карту – отсутствуют обязательные свойства!")

            return nil

        } catch {

            print("Ошибка: Не удалось создать карту из-за неизвестной ошибки: \(error)")

            return nil

        }

    }

}

/\*УПРАВЛЕНИЕ КАРТАМИ\*/

/\*\*

Протокол для класса управления списком карт.

Определяет методы для добавления, получения списка карт и поиска карт с максимальным кэшбеком по категории.

\*/

protocol CardManagerProtocol {

    // Свойство для хранения списка карт

    var cards: [CardProtocol] { get set }

    // Метод добавления карты в список

    func addCard(\_ card: CardProtocol)

    // Метод получения списка карт

    func getCards() -> [CardProtocol]

    // Метод получения списка карт и поиска карт с мксимальным кэшбеком по категории

    func getCardWithMaxCashback(forCategory category: String) -> [CardProtocol]

}

/\*\*

Класс для управления списком карт

\*/

class CardManager {

    // Список карт

    var cards: [CardProtocol]

    // Инициализация

    init() {

        self.cards = []

    }

}

/\*\*

Расширение класса, содержащее методы протокола

\*/

extension CardManager: CardManagerProtocol {

    // Метод добавляет карту в список

    func addCard(\_ card: CardProtocol) {

        cards.append(card)

    }

    // Метод возвращает список карт

    func getCards() -> [CardProtocol] {

        return cards

    }

    // Метод возвращает карты, соответствующих условию: макс кэшбек для указанной категории из общего списка

    func getCardWithMaxCashback(forCategory category: String) -> [CardProtocol] {

        // Получение макс кэшбека для указанной категории

        guard let maxCashback = cards.flatMap({ $0.paymentCategories })

            .filter ({ $0.category == category })

            .max (by: { $0.cashbackPercentage < $1.cashbackPercentage })?.cashbackPercentage

        else {

            return []

        }

        // Фильтрация карт по макс кэшбеку

        return cards.filter {

            card in card.paymentCategories.contains {

                $0.category == category && $0.cashbackPercentage == maxCashback

            }

        }

    }

}

/\*View (Представление)\*/

/\*ПРОСМОТР ДАННЫХ КАРТ\*/

/\*\*

Протокол для класса отображения карты

\*/

protocol CardViewProtocol {

    func displayCard(\_ card: CardProtocol)

}

/\*\*

Класс для отображения свойств карты

\*/

class CardView {}

extension CardView: CardViewProtocol {

// метод, отображающий карту со всеми характеристиками

    func displayCard(\_ card: CardProtocol) {

        print("ID карты: \(card.id)")

        print("Название карты: \(card.name)")

        print("Изображение карты: \(card.icon)")

        //print(«Пользовательские свойства:»)

        //for (key, value) in card.variableProperties {

            //print(«\(key): \(value)»)

        //}

        print("Категории оплаты:")

        for paymentCategory in card.paymentCategories {

            print("Категория: \(paymentCategory.category), Кэшбек: \(paymentCategory.cashbackPercentage)%")

        }

        print("\*\*\*")

    }

}

/\*Controller (Контроллер)\*/

/\*ЛОГИКА\*/

/\*\*

Протокол контроллера

\*/

protocol CardControllerProtocol {

    func createCard()

}

/\*\*

Класс контроллера

\*/

class CardController {

    let cardManager: CardManager

    let cardView: CardView

    init(cardManager: CardManager, cardView: CardView) {

        self.cardManager = cardManager

        self.cardView = cardView

    }

}

/\*\*

Расширение контроллера с методами

\*/

extension CardController: CardControllerProtocol {

// метод, принимающий данные для создания экземпляров карт и создающий карты

    func createCard() {

        // карта 1

        if let card1 = CardBuilder()

            .setID(1)

            .setName("Зарплатная")

            .setIcon("card\_icon1")

            //.addVariableProperty(variableKey: «monthlyCashback», variableValue: true)

            //.addVariableProperty(variableKey: «minimumPurchaseAmount», variableValue: 1000.0)

            .addPaymentCategory(PaymentCategory(category: "Ресторан", cashbackPercentage: 7.0, minimumPurchaseAmount: 0.0))

            .addPaymentCategory(PaymentCategory(category: "АЗС", cashbackPercentage: 2.0, minimumPurchaseAmount: 0.0))

            // тест с дублированием категории на одной карте

            .addPaymentCategory(PaymentCategory(category: "АЗС", cashbackPercentage: 4.0, minimumPurchaseAmount: 0.0))

            // тест с нулевым кэшбеком у категории

            .addPaymentCategory(PaymentCategory(category: "Супермаркет", cashbackPercentage: 0.0, minimumPurchaseAmount: 0.0))

            .build() {

                cardManager.addCard(card1)

            } else {

                print("Ошибка при создании карты 1\n")

            }

        // карта 2

        if let card2 = CardBuilder()

            .setID(2)

            .setName("Кредитная")

            .setIcon("card\_icon2")

            //.addVariableProperty(variableKey: «monthlyCashback», variableValue: false)

            .addPaymentCategory(PaymentCategory(category: "Ресторан", cashbackPercentage: 7.0, minimumPurchaseAmount: 1000.0))

            // тест с отрицательным кэшбеком у категории

            .addPaymentCategory(PaymentCategory(category: "Аптека", cashbackPercentage: -5.0, minimumPurchaseAmount: 1000.0))

            .build() {

                cardManager.addCard(card2)

            } else {

                print("Ошибка при создании карты 2\n")

            }

        // карта 3

        if let card3 = CardBuilder()

            .setID(3)

            // тест с отсутствием обязательного параметра у карты

            //.setName(«Дебетовая»)

            .setIcon("card\_icon3")

            //.addVariableProperty(variableKey: «monthlyCashback», variableValue: true)

            //.addVariableProperty(variableKey: «minimumPurchaseAmount», variableValue: 1000.0)

            .addPaymentCategory(PaymentCategory(category: "Ресторан", cashbackPercentage: 5.0, minimumPurchaseAmount: 0.0))

            .addPaymentCategory(PaymentCategory(category: "Кино", cashbackPercentage: 2.0, minimumPurchaseAmount: 0.0))

            .addPaymentCategory(PaymentCategory(category: "Кафе", cashbackPercentage: 0.0, minimumPurchaseAmount: 0.0))

            .build() {

                cardManager.addCard(card3)

            } else {

                print("Ошибка при создании карты 3\n")

            }

    }

}

/\*Main\*/

print("\*\*\* Реализация кода для создания и поиска карты с макс кэшбеком с проверками. Применение протоколов, структур, методов, классов, расширений. Последний консольный релиз \*\*\* \n")

let cardManager = CardManager()

let cardView = CardView()

let cardController = CardController(cardManager: cardManager, cardView: cardView)

cardController.createCard()

// вывод в консоль всех карт с параметрами для анализа отработки исключений (CP>0, CP!=CP)

let allCards = cardManager.getCards()

for card in allCards {

    cardView.displayCard(card)

}

let selectedCategory = "Ресторан" // назначение переменной места оплаты (категории кэшбека)

let cardWithMaxCashback = cardManager.getCardWithMaxCashback(forCategory: selectedCategory) // формирование списка карт по условию

if !cardWithMaxCashback.isEmpty {

    print("\nКАРТЫ С НАИБОЛЬШИМ РАЗМЕРОМ CASHBACK ДЛЯ КАТЕГОРИИ '\(selectedCategory)':\n")

// вывод в консоль списка карт с нужными данными

    for card in cardWithMaxCashback {

        print("Имя карты: \(card.name)")

        if let paymentCategory = card.paymentCategories.first(where: { $0.category == selectedCategory }) {

            print("Процент cashback: \(paymentCategory.cashbackPercentage)%")

            print("Минимальный чек для начисления casback: \(paymentCategory.minimumPurchaseAmount)")

        }

        print("---")

    }

} else {

    print("Нет карт с категорией '\(selectedCategory)'")

}

Приложение 12

Файловая структура проекта

.

├── AppDelegate.swift

├── Assets.xcassets

│   ├── AccentColor.colorset

│   │   └── Contents.json

│   ├── AppIcon.appiconset

│   │   └── Contents.json

│   ├── Contents.json

│   ├── GB.imageset

│   │   ├── Contents.json

│   │   ├── GB-1.png

│   │   ├── GB-2.png

│   │   └── GB.png

│   └── GB.png

├── Base.lproj

│   └── LaunchScreen.storyboard

├── CoreData

│   ├── AppDataModel.xcdatamodeld

│   │   └── AppDataModel.xcdatamodel

│   │   └── contents

│   └── CoreDataStack.swift

├── Helper Files

│   ├── ConstraintsConstructor.swift

│   ├── PopupStub.swift

│   └── RoundButtonWithSettings.swift

├── Info.plist

├── Managers

│   ├── Cards

│   │   ├── CardDataProvider.swift

│   │   ├── CardManager.swift

│   │   └── CardManagerProtocol.swift

│   ├── Cashback

│   │   ├── CashbackDataProvider.swift

│   │   ├── CashbackManager.swift

│   │   └── CashbackManagerProtocol.swift

│   └── Categories

│   ├── CategoryDataProvider+Converter.swift

│   ├── CategoryDataProvider+Observer.swift

│   ├── CategoryDataProvider.swift

│   ├── CategoryDataProviderObserverProtocol.swift

│   ├── CategoryManager.swift

│   └── CategoryManagerProtocol.swift

├── Models

│   ├── Cards

│   │   ├── CardModel.swift

│   │   └── CardModelProtocol.swift

│   ├── Cashback

│   │   ├── CashbackModel.swift

│   │   └── CashbackModelProtocol.swift

│   ├── Categories

│   │   ├── CategoryModel.swift

│   │   └── CategoryModelProtocol.swift

│   └── TestCashbackModel.swift

├── SceneDelegate.swift

├── Tests

│   └── DisplayCardEntityData.swift

└── Views

├── CardsList

│   ├── CardListViewController+Buttons.swift

│   ├── CardListViewController+Scrolling.swift

│   └── CardListViewController.swift

├── CategoryList

│   ├── CategoryListViewController+Buttons.swift

│   ├── CategoryListViewController+Scrolling.swift

│   └── CategoryListViewController.swift

├── ItemsWindows

│   ├── BaseItemViewController+Buttons.swift

│   ├── BaseItemViewController.swift

│   ├── Cards

│   │   ├── Add

│   │   │   ├── AddCardViewController+Buttons.swift

│   │   │   ├── AddCardViewController+Lines.swift

│   │   │   ├── AddCardViewController+Notifications.swift

│   │   │   ├── AddCardViewController+Tables.swift

│   │   │   ├── AddCardViewController+TextFields.swift

│   │   │   └── AddCardViewController.swift

│   │   └── Del+Edit

│   └── Category

│   ├── Add

│   │   ├── AddCategoryViewController+Buttons.swift

│   │   ├── AddCategoryViewController+TextFields.swift

│   │   └── AddCategoryViewController.swift

│   └── Del+Edit

├── PlaceOfPayment

│   ├── CategoryCell.swift

│   ├── PlaceOfPaymentCollectionViewController+AlertWindow.swift

│   ├── PlaceOfPaymentCollectionViewController+LogicMaxCashback.swift

│   ├── PlaceOfPaymentCollectionViewController+SetupCell.swift

│   ├── PlaceOfPaymentCollectionViewController+SetupData.swift

│   └── PlaceOfPaymentCollectionViewController.swift

├── TabBar

│   ├── TabBarController.swift

│   └── UITabBar+PulseAnimation.swift

└── ViewController.swift

31 directories, 63 files

1. в свете последних геополитических событий, речь будет идти только о ПС «Мир» от НСПК [↑](#footnote-ref-1)
2. MCC-код используется платежными системами для маркировки вида деятельности организации, где устанавливается то или иное эквайринговое устройство, проще говоря – терминал оплаты. В соответствии с MCC-кодами банками выставляется комиссия на точку продаж (торговая уступка) в зависимости от различных параметров. Но определяющим является – MCC-код, т.е. для Ресторана это один код, а для АЗС – другой. [↑](#footnote-ref-2)
3. В языке программирования Swift такая техника обычно используется для разделения кода на разные файлы для более удобного управления и поддерживаемости проекта. Когда все части класса или структуры разделены на разные файлы, это упрощает работу разработчиков, особенно при коллективной разработке проекта.

   Таким образом, при использовании "Partial classes" или "Partial files" в Swift, каждый файл представляет собой часть целого класса или структуры, а в процессе компиляции Swift объединяет эти части в одну сущность. [↑](#footnote-ref-3)
4. Получить структуру папки в подобном виде можно посредством программы Terminal, встроенной в macOS:

   1. Откройте Terminal;

   2. Перейдите в директорию с проектом с помощью команды cd;

   3. Выведите структуру файлов проекта в древовидной форме с помощью команды tree.

   Если у вас нет установленной утилиты tree, установите ее через Homebrew путем выполнения команды brew install tree. [↑](#footnote-ref-4)
5. Выделить область экрана для копирования можно с помощью встроенного в macOS инструмента "Захват экрана" (Screenshot), который позволяет вам выделить определенную область экрана с помощью мыши или клавиатуры и скопировать ее.

   Открывается он при помощи – «Command + Shift + 5» на клавиатуре. [↑](#footnote-ref-5)
6. UIKit - это фреймворк, разработанный Apple для создания графического пользовательского интерфейса (GUI) в приложениях, работающих на платформах iOS, iPadOS, macOS и tvOS. [↑](#footnote-ref-6)
7. UITabBarController – это контроллер, предоставляемый фреймворком UIKit в iOS, который облегчает навигацию между различными представлениями пользовательского интерфейса через вкладки (табы) внизу экрана. Он позволяет организовать множество представлений в пользовательском интерфейсе и обеспечивает простой и интуитивно понятный способ переключения между разделами приложения с помощью табов. [↑](#footnote-ref-7)
8. CoreData — это фреймворк от Apple для управления объектами приложения в iOS и macOS. Он предоставляет возможность хранить, извлекать и управлять данными в приложении с помощью объектов, а не просто напрямую с базой данных. Это означает, что можно работать с данными, как если бы они были объектами, что сильно упрощает работу с хранилищем данных. [↑](#footnote-ref-8)
9. UITableViewController – это подкласс UIViewController, специально предназначенный для управления таблицами (UITableView) на iOS. Он уже содержит в себе таблицу и методы для управления данными таблицы. Так что не нужно тратить много времени на настройку таблицы вручную. [↑](#footnote-ref-9)
10. В текущем релизе функционал добавления кэшбека в карту не реализован. [↑](#footnote-ref-10)
11. В текущем релизе функционал отображения карточки карты, удаления и редактирования не реализован. [↑](#footnote-ref-11)
12. Предстоит переписать методы менеджеров, работающие с картами, чтобы чётко разделить ответственность:

    - CardManager работает с данными, полученными от пользователя, и взаимодействует с соответствующей моделью

    - CardDataProvider работает с данными из модели и доставляет их в нужном формате на хранение в БД. [↑](#footnote-ref-12)
13. MVC (Model-View-Controller), где:

    Model (Модель) отвечает за данные и бизнес-логику приложения, View (Представление) отображает данные для пользователя, Controller (Контроллер) обрабатывает пользовательский ввод, взаимодействует с моделью и управляет представлением. [↑](#footnote-ref-13)
14. MVVM (Model-View-ViewModel), где:

    Model (Модель) содержит данные и бизнес-логику приложения, View (Представление) отображает данные и предоставляет пользовательский интерфейс, ViewModel (Представление модели) содержит представление данных и бизнес-логику, которую представление отображает. [↑](#footnote-ref-14)
15. Паттерн Observer относится к поведенческим паттернам проектирования. В этом паттерне устанавливается зависимость «один ко многим» между объектами, где если один объект меняет свое состояние, все зависящие от него объекты автоматически уведомляются и обновляются. [↑](#footnote-ref-15)
16. API (Application Programming Interface) - это набор различных определенных взаимодействующих протоколов и инструментов для компьютерных программ. [↑](#footnote-ref-16)
17. В текущем релизе функционал отображения карточки категории, удаления и редактирования не реализован. [↑](#footnote-ref-17)
18. CoreDataStack в iOS-разработке обычно представляет собой класс, который инкапсулирует логику управления CoreData стеком. Этот стек отвечает за управление взаимодействием приложения с базой данных SQLite (или другим хранилищем) через фреймворк CoreData. [↑](#footnote-ref-18)