*Добрый день уважаемая комиссия!*

Наша команда состоит из Корчагиной Василисы Алексеевны и Свердловой Екатерины Сергеевны, мы обе учимся на экономическом факультете. Наш проект - мобильное приложение Petty Clicker.

Это результат нескольких месяцев изучения android разработки, обсуждений, проб и ошибок, и, честно говоря, мы гордимся тем, что получилось.

По реализации могут быть вопросы в некоторых местах мы до конца не разобрались, но старались делать максимально хорошо исходя из знаний, что получили на курсе.

**0. Как мы работали в команде**

Мы работали над проектом вдвоём. С самого начала распределили роли, чтобы каждый занимался тем, что получается лучше всего.

Я взяла на себя техническую часть — разрабатывал архитектуру, писал код, настраивал базу данных, внедрение зависимостей через Hilt, а также логику автокликера и сохранения данных в Room.

Катя занималась структурой проекта и документацией — писал отчёт, готовил презентацию, оформлял выступление, проверял логику с точки зрения пользователя и помогал тестировать интерфейс. Также он оформлял PNG-графику и участвовал в разработке визуального стиля.

Когда возникали сложности с кодом или тестами, мы вместе разбирались в проблемах: читали документацию, искали решения и обсуждали, как лучше реализовать тот или иной функционал.

Благодаря такому разделению мы смогли быстрее двигаться, не перегружать друг друга, а главное — довести проект до законченного состояния.

**1. Введение и мотивация**

Мы выбрали формат clicker-игры, потому что это отличный способ потренироваться в архитектуре, логике, работе с данными и визуальном оформлении.

Приложение сочетает простую механику с визуально привлекательными элементами и возможностью собирать питомцев.

**2. Основной функционал**

* Клик по экрану начисляет очки
* Очки можно тратить на покупку питомцев разных категорий
* Есть коллекция, где питомцев можно переименовать и выбрать активного
* Легендарные питомцы активируют автокликер
* Встроены анимации кликов и отображение случайных цитат
* Есть тёмная и светлая тема оформления

**3. Демонстрация экранов (если включена презентация)**

* Главный экран — кнопка клика, счёт, питомец
* Магазин — карточки питомцев с фильтрацией
* Коллекция — галерея купленных животных

**4. Архитектура**

Проект построен на MVVM:

* ViewModel управляет состоянием и логикой
* Room хранит клики и коллекцию
* Hilt внедряет зависимости
* Состояния управляются через StateFlow

**5. Автокликер**

Если в коллекции есть хотя бы один легендарный питомец, запускается автокликер, который каждые 5 секунд начисляет бонусные клики.

Сумма бонуса зависит от количества и ценности питомцев. А в UI появляется анимация +X.

**6. Визуальный стиль**

Все визуальные элементы — это PNG-графика: фоны, питомцы, плашки карточек.

Вместо стандартной темизации используются отдельные изображения для светлой и тёмной темы.

**7. Тестирование**

Чтобы убедиться в надёжности логики, я написала юнит-тест, который проверяет: если пользователь покупает питомца, то приложение действительно отправляет команду сохранить его в базу.

Для этого используется мок-DAO — поддельный объект базы, который позволяет отследить вызов метода сохранения.

Это важно: если в коде случайно уберут этот вызов, тест покажет ошибку.

Так мы уверены, что логика покупки работает корректно и данные не теряются.

**8. Где и как использовали технологии**

В проекте мы использовали современный стек Android-разработки. Каждую технологию старались не просто подключить, а понять, зачем она нужна и как работает на практике.

* Jetpack Compose — весь UI построен на Compose. Это позволило быстро создавать экраны, реализовать анимации кликов, а также оформить коллекцию питомцев с помощью LazyColumn и кастомных карточек.
* Room — отвечает за хранение кликов и коллекции питомцев. Мы создали две сущности: ClickerEntity и PurchasedImageEntity, реализовали DAO и репозитории. Все действия с данными проходят через ViewModel.
* Hilt — используется для внедрения зависимостей. Настроены модули, с помощью которых мы передаём DAO и репозитории во ViewModel. Это сильно упростило структуру кода и сделало её более масштабируемой.
* StateFlow — управляет состоянием интерфейса. Например, количество кликов обновляется в реальном времени при каждом действии пользователя или автоклике. Compose подписан на эти изменения и реагирует автоматически.
* Корутины (Coroutines) — используются для фоновых операций, включая автокликер. Мы запускали таймер автокликера внутри viewModelScope, чтобы начислять бонусы каждые 5 секунд.
* Моковые DAO и юнит-тесты — для проверки бизнес-логики. Мы реализовали тест, который проверяет, что при покупке питомца репозиторий действительно вызывает метод сохранения в базу. Это защищает от случайных ошибок при рефакторинге.
* PNG-графика — в качестве фонов, карточек и питомцев. Вместо стандартной темизации мы подгружаем отдельные изображения в зависимости от темы, что создаёт визуальный стиль ближе к игровому.
* Navigation (интенты) — для переходов между экранами. Мы использовали несколько активити, между которыми передавались параметры, например — имя активного питомца.

**9. Перспективы**

В будущем мы хоти добавить:

* Достижения и прогрессию
* Облачную синхронизацию
* Монетизацию
* Мультиплатформенность (через Compose Multiplatform)
* Локализацию

**10. Личный опыт**

Этот проект помог нам освоить Android-разработку на практике — от UI до архитектуры. Мы научилась принимать технические решения, организовывать логику и тестировать результат.

***Спасибо за внимание! Готовы ответить на ваши вопросы — по архитектуре, технологиям, реализации, тестированию или любым другим аспектам проекта.***

**Возможные вопросы на защите и как на них ответить**

*Общие вопросы по проекту*

**1. Почему вы выбрали именно жанр clicker?** 👉 Clicker — простой и увлекательный жанр, в котором легко продемонстрировать архитектуру, работу с базой данных, UI и асинхронную логику. Кроме того, он подходит для реализации анимаций и прогрессии, что делает его интересным для пользователя.

**2. Почему ваше приложение называется Petty Clicker?** 👉 Название играет словами: "pet" (питомец) и "clicker" (кликер). Суть игры — собирать питомцев, кликая по экрану. "Petty" также подчёркивает милую и лёгкую стилистику.

**3. Что было самым сложным при разработке?** 👉 Больше всего времени ушло на реализацию автокликера с реактивным поведением. Также сложными оказались моменты с Hilt и Room — особенно в связке с Flow.

**4. Что бы вы добавили, если бы было больше времени?** 👉 Онлайн-синхронизацию, push-уведомления, достижения, платёжную систему и более развитую систему прокачки питомцев.

*Архитектура и технологии*

**5. Почему вы выбрали архитектуру MVVM?** 👉 Она чётко разделяет ответственность между UI, логикой и данными. Это упрощает сопровождение, масштабирование и тестирование приложения.

**6. Для чего в проекте используется Hilt?** 👉 Hilt внедряет зависимости автоматически. Например, ViewModel получает нужный репозиторий без ручного создания — это сокращает связность кода и улучшает тестируемость.

**7. Почему Room, а не обычная SQLite?** 👉 Room — это удобный ORM, он предоставляет аннотации, автоматическую генерацию кода и типобезопасность. Это ускоряет работу и снижает количество ошибок.

**8. Как вы реализовали смену темы?** 👉 Используется DataStore для сохранения состояния. Тема применяется через MaterialTheme. Переключатель находится в интерфейсе, и всё обновляется в реальном времени.

*База данных и бизнес-логика*

**9. Как устроена база данных?** 👉 Есть две таблицы: clicker\_data (для хранения количества кликов) и purchased\_images (для хранения питомцев). DAO-интерфейсы управляют чтением и записью данных.

**10. Как рассчитывается бонус при клике?** 👉 Если питомец дешевле 500 — за каждый повторный питомец даётся +2. Если дороже — +4 за каждого. Таким образом, коллекция напрямую влияет на бонус.

**11. Как работает автокликер?** 👉 Если у пользователя есть легендарный питомец, запускается корутина, которая раз в 5 секунд начисляет бонус. Проверка идёт через StateFlow, а событие autoClickEvent сигнализирует UI, что произошёл автоклик.

*UI и пользовательский опыт*

**12. Почему Jetpack Compose?** 👉 Современный declarative UI-фреймворк от Google, который позволяет быстро и удобно создавать адаптивный и анимированный интерфейс.

**13. Как пользователь понимает, что автокликер сработал?** 👉 На экране появляется анимация +X, которая исчезает через 1 секунду. Это визуально подтверждает начисление бонуса.

**14. Что пользователь может сделать с питомцем**? 👉 Купить, просматривать в коллекции, выбрать как активного, дать кастомное имя.

Тестирование и производительность

**15. Как вы тестировали приложение?** 👉 Ручное тестирование на эмуляторе и реальном устройстве. Также реализованы unit-тесты для расчёта бонусов и сохранения кликов.

**16. Насколько производительно приложение?** 👉 Поддерживает 60 fps, работает при низком потреблении памяти (~110 МБ). Быстро запускается и не перегружает устройство.

*Курсовые темы (Room, Hilt, Flow и т.п.)*

**17. Что такое ViewModel и зачем она нужна?** 👉 Это компонент архитектуры, который хранит состояние и логику. Он переживает повороты экрана и не связан с жизненным циклом UI.

**18. Что такое Flow и StateFlow?** 👉 Flow — асинхронный поток данных, StateFlow — его реализация с сохранением текущего состояния. Используются для реактивного обновления UI.

**19. Что такое suspend-функция?** 👉 Это функция, которую можно приостановить и возобновить — применяется в корутинах, чтобы не блокировать основной поток.

**20. Как реализована навигация между экранами?** 👉 Используются Intent и startActivity, переходы между MainActivity, LkActivity, PurchasedImagesActivity.

**Ответы на возможные вопросы по коду проекта**

*Архитектура*

**1. Почему вы выбрали архитектуру MVVM?** 👉 MVVM обеспечивает разделение логики и UI: ViewModel содержит состояние и бизнес-логику, UI (Jetpack Compose) подписывается на изменения. Это улучшает читаемость и тестируемость.

**2. Какие слои в проекте и что в них находится?** 👉

* Presentation: Compose UI, активности, экраны.
* ViewModel: логика UI и состояния.
* Repository: обработка и агрегация данных.
* Data (Room): сущности, DAO, база данных.

*База данных (Room)*

**3. Какие сущности реализованы?** 👉

* ClickerEntity — хранит текущее количество кликов.
* PurchasedImageEntity — купленные питомцы: имя, тип, цена, количество, имя пользователя и ID ресурса.

**4. Как устроена работа с Room**? 👉 Через DAO-интерфейсы: ClickerDao, PurchasedImageDao. Они предоставляют suspend-функции и Flow-потоки для реактивной работы.

**5. Как осуществляется чтение и запись**? 👉 Репозитории используют DAO и возвращают Flow, которые подписаны во ViewModel. Все операции выполняются в viewModelScope с использованием Dispatchers.IO.

*Логика автокликера*

**6. Как реализован автокликер?** 👉 В ClickerViewModel есть функция startLegendaryAutoClicker(), которая запускает корутину. Коррутина каждые 5 секунд проверяет наличие легендарных питомцев и начисляет бонусные клики.

**7. Что запускает автокликер?** 👉 Условие: в коллекции пользователя должен быть хотя бы один питомец типа Legendary. Проверка выполняется при запуске приложения и обновлении коллекции.

**8. Как рассчитывается бонус?** 👉 В функции calculateBonusClicks():

+2 клика за каждого повторяющегося питомца

+4 клика за питомца с ценой > 500

Итоговая сумма начисляется пользователю.

**9. Как пользователь узнаёт, что сработал автокликер?** 👉 Через MutableSharedFlow autoClickEvent, который ловится в UI и вызывает анимацию +X.

*UI и Jetpack Compose*

**10. Почему использовали Jetpack Compose**? 👉 Он позволяет писать UI декларативно, гибко и компактно. Поддерживает анимации, темы, реактивность. Лучше интегрируется с StateFlow, чем XML.

**11. Какие элементы Compose вы использовали?** 👉

* LazyColumn — списки (коллекция, магазин)
* AnimatedVisibility и Box — анимации клика
* remember, LaunchedEffect, produceState — управление состоянием
* MaterialTheme — темы и цветовая палитра

**12. Где сохраняется кастомное имя питомца?** 👉 В PurchasedImageEntity.customName. Оно вводится пользователем и сохраняется в базе данных через DAO и Repository.

*Внедрение зависимостей (Hilt)*

**13.**  **Как вы настраивали Hilt**? 👉

Все DAO и репозитории помечены аннотациями @Inject

Используется @HiltViewModel

Модули настроены в AppModule.kt, где через @Provides создаются Room-база, DAO, репозитории.

**14. Что даёт Hilt в вашем проекте?** 👉 Упрощает создание зависимостей и их жизненный цикл. Например, ViewModel получает ClickerRepositoryавтоматически, без ручной инициализации.

*Асинхронность и StateFlow*

**15. Почему использовали StateFlow, а не LiveData?** 👉 StateFlow — современный и более гибкий API, особенно удобный для Compose. Он легче обрабатывает множественные подписки, а также совместим с корутинами.

**16.**  **Как реализовано обновление UI при кликах?** 👉 В ClickerViewModel есть clickerState: StateFlow<Int>. Compose-экран подписан на него и автоматически отображает новое значение.

**17. Как устроен поток автокликера?** 👉

* В viewModelScope запускается while (isActive)
* Каждые 5 секунд — проверка коллекции
* Если условие выполнено, начисляется бонус

*Тестирование*

**18. Что вы тестировали в проекте?** 👉 Мы написали юнит-тест, который проверяет, что при покупке изображения питомца вызывается метод сохранения в базу данных. Это ключевой элемент бизнес-логики — важно не просто отобразить покупку, но и гарантировать, что она сохраняется.

**19. Почему вы используете мок-объект, а не реальную базу**? 👉 Мок-объект (поддельная версия DAO) позволяет изолировать логику и проверить, была ли вызвана нужная команда. Это быстрее, надёжнее и не требует доступа к настоящей базе, что идеально подходит для юнит-теста.

**20. Что именно проверяет ваш тест?** 👉 Мы вызываем метод покупки с тестовым питомцем и проверяем, что insertPurchasedImage() действительно был вызван у DAO. То есть: если пользователь купил питомца, приложение должно попытаться сохранить его.

**21. Зачем это нужно, если база и так работает?** 👉 Такой тест нужен для защиты от случайных ошибок. Если кто-то в будущем случайно удалит или закомментирует вызов сохранения, тест тут же покажет, что логика нарушена. Это помогает поддерживать стабильность приложения при любых изменениях.

**22. Вы проверяете, как база хранит данные?** 👉 Нет, это не цель юнит-теста. Мы не проверяем внутреннюю работу базы — только факт вызова нужной функции. Хранение проверяется отдельно, например, в интеграционных или UI-тестах.

**23. Как бы вы протестировали, что питомец действительно отображается после покупки?** 👉 Это уже задача UI-теста: нужно проверить, что после покупки питомец появляется в коллекции. Такие тесты можно реализовать с помощью инструментов типа Espresso или Compose UI Testing, но мы пока сосредоточились на бизнес-логике.

*Ошибки и их решения*

**24. С какими трудностями в коде столкнулись?** 👉

* Настройка Hilt и его модулей
* Правильная отмена корутин (автокликер) — решалось через isActive
* Изоляция логики от UI — через ViewModel

**25. Как обеспечили отсутствие утечек памяти?** 👉 Все корутины запускаются в viewModelScope, подписки на Flow отменяются автоматически при уничтожении ViewModel. Анимации запускаются с таймером.