Министерство образования и науки российской федерации

Федеральное государственное автономное образовательное учреждение
высшего профессионального образования
«ЮЖНО-УРАЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(Национальный исследовательский университет)
Высшая школа электроники и компьютерных наук
Кафедра «Информационно-аналитическое обеспечение управления в социальных и экономических системах»

ПРОВЕРЕНО
Рецензент,
разработчик мобильных
приложений
ООО «РЭД»
________ / В.Д. Огородникова /
«___» ________________ 2020 г.

ДОПУСТИТЬ К ЗАЩИТЕ
Заведующий кафедрой,
д.т.н., профессор
________ / О.В. Логиновский /
«___» ________________ 2020 г.

# Автоматизация процесса тестирования функциональности Android-приложений

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ
ЮУрГУ – 09.04.01.2020.671 ПЗ ВКР

Руководитель работы
д.т.н., профессор
________ / О.В. Логиновский /
«___» ________________ 2020 г.

Автор работы
студент группы КЭ-221
________ / В.А. Сысуев /
«___» ________________ 2020 г.

Нормоконтролер,
к.т.н., доцент,
________ / В.Н. Любицын /
«___» ________________ 2020 г.

Челябинск 2020

# АННОТАЦИЯ

Сысуев В.А. Автоматизация процесса тестирования функциональности Android-приложений: ЮУрГУ (НИУ), ВШ ЭКН; 2020, 79 с. 21 ил., библиогр. список – 29 наим.

В результате выполнения работы рассмотрена сущность Android-приложений, проведен анализ главных компонентов операционной системы, рассмотрены проблемы, которые могут встречаться при тестировании, проведен анализ актуальных инструментов и методологий для тестирования Android-приложений, рассмотрена сущность автоматизированного тестирования и разработаны автоматизированные тесты для Android-приложения.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ........ 8

1 АРХИТЕКТУРА ОПЕРАЦИОННОЙ СИСТЕМЫ ANDROID И ПРОБЛЕМЫ ТЕСТИРОВАНИЯ........ 9

1.1 Архитектура операционной системы Android........ 9

1.1.1 Уровень приложений........ 11

1.1.2 Уровень библиотек........ 14

1.1.4 Уровень абстрагирования оборудования........ 16

1.1.5 Ядро Linux........ 16

1.2 Проблемы, связанные с тестированием Android-приложений........ 17

1.2.1 Четыре типа компонентов........ 17

1.2.2 Уникальные жизненные циклы компонентов Android........ 21

1.2.3 Интенсивное использование XML-файлов........ 26

1.2.4 Контекстно-зависимые характеристики........ 28

1.2.5 Два типа ориентации экрана........ 29

1.2.6 Разрешения приложений Android........ 30

1.2.7 Разнообразные сетевые подключения........ 32

1.2.8 Ограниченное время автономной работы........ 33

Выводы по разделу один........ 35

2 ИССЛЕДОВАНИЕ ИНСТРУМЕНТОВ И МЕТОДОЛОГИЙ ТЕСТИРОВАНИЯ ANDROID-ПРИЛОЖЕНИЙ........ 36

2.1 Введение в тестирование программного обеспечения........ 36

2.2 Жизненный цикл программного обеспечения........ 36

2.3 Жизненный цикл тестирования программного обеспечения........ 38

2.2 Типы тестов Android-приложений........ 41

2.2.1 Unit-тесты........ 41

2.2.2 Интеграционные тесты........ 43

2.2.3 Тесты пользовательского интерфейса........ 44

2.3 JUnit-фреймфорк ... 46
2.4 Espresso-фреймворк ... 47
2.5 Разработка через тестирование ... 49
Выводы по разделу два ... 52
3 РАЗРАБОТКА И АВТОМАТИЗАЦИЯ ТЕСТОВЫХ СЦЕНАРИЕВ ... 53
3.1 Автоматизированное тестирование ... 53
3.2 Создание тестовых случаев Android-приложения для ведения и управления задач ... 57
3.3 Создание автоматизированных сценариев тестирования ... 61
3.4 Анализ результатов тестирования ... 70
Выводы по разделу три ... 75
ЗАКЛЮЧЕНИЕ ... 76
БИБЛИОГРАФИЧЕСКИЙ СПИСОК ... 77

# ВВЕДЕНИЕ

*Актуальность исследования.* Наряду со значительным распространением мобильных устройств, мобильные приложения также доминируют на мировом рынке с точки зрения пользователей, разработчиков, выпусков приложений и загрузок. Тем не менее, качество приложений является растущей и значительной проблемой. Многие приложения выпускаются на рынок с серьезными ошибками программного обеспечения. Тестирование приложений отличается от тестирования традиционных программ, поскольку включают новые функции и структуры программирования, которые раньше никогда не использовались.

*Цель.* Целью выпускной квалификационной работы является исследование области тестирования мобильных приложений, изучение методов и инструментов тестирования мобильных приложений, изучение проблем, с которыми сталкивается разработчик при тестирование мобильных приложений и способы их решения, а также создание тестовых случаев и разработка автоматизированных сценариев тестирования мобильного приложения для управления задачами проекта, которые обеспечат высокое качество программного продукта.

*Задачи*, которые необходимо выполнить для достижения поставленной цели исследования:

1. Изучить и определить архитектуру Android-приложений, а также проблемы, связанные с тестированием приложений;
2. Исследовать методологии, подходы и инструменты для тестирования Android-приложений;
3. Создать автоматизированные сценарии тестирования Android-приложений.

*Объект исследования:* автоматизированное тестирование программного обеспечения.

*Предмет исследования:* мобильные приложения.

# 1 АРХИТЕКТУРА ОПЕРАЦИОННОЙ СИСТЕМЫ ANDROID И ПРОБЛЕМЫ ТЕСТИРОВАНИЯ

## 1.1 Архитектура операционной системы Android

Android является операционной системой для мобильных устройств, создание которой приписывается компании Android Inc. В 2005 году Google заявили миру о покупке Android [1]. В основе операционной системы измененное Linux ядро. Некоторые причастные к Open Handset Alliance (OHA) [2] и компания Google вели совместную работу с Android (проектирование, разработка, распространение). Android Open Source Project (AOSP) занимается управлением и этапами разработки Android [3].

В основе мобильной системы Android лежит измененное ядро Linux 2.6 [4], но тем не менее структура Linux претерпела некоторые изменения, некоторые из драйверов и библиотек создали заново, либо модифицированы и подстроены под Android для максимальной эффективности и оптимизации для мобильных устройств (смартфонов, планшетов). Несколько таких библиотек относятся к проектам с открытым исходным кодом. Из-за некоторых проблем с получением прав и лицензий руководство операционной системы приняло решение разработать свой собственный механизм для выполнения программ – Davlik Virtual Machine, который вошел в Android и написали свою собственную библиотеку – Bionik. По причине незначительных вычислительных способностей мобильных устройств в Android всегда заостряют внимание на оптимизации системы [5]. Для обеспечения полноценности операционной системы, был дополнительно разработан фреймворк. Android можно представить как совокупность отдельных элементов, включая саму систему, связующие программные средства и системные приложения.

Операционная системаAndroid может быть обозначена следующим образом:

1. Платформа с открытым исходным кодом;

2. Грамотное проектирование оборудования;
3. Система основана на измененном ядре Linux;
4. Среда выполнения;
5. Структура приложения и пользовательского интерфейса (UI).

На рисунке 1.1 представлена текущая (многоуровневая) архитектура Android. Измененное Linux ядро выполняет роль аппаратной абстракции, в которую входят драйвера для устройств, работа с памятью, работа с процессами, и некоторые функции для работы с сетью. Все библиотеки взаимодействуют через язык программирования Java и не совпадает с обычным дизайном Linux. На этом уровне так же находится набор библиотек libc (Bionic). Уровень, на котором происходит выполнение программ, содержит Davlik и необходимые библиотеки. Многие функции Android можно получить из набора основных библиотек.

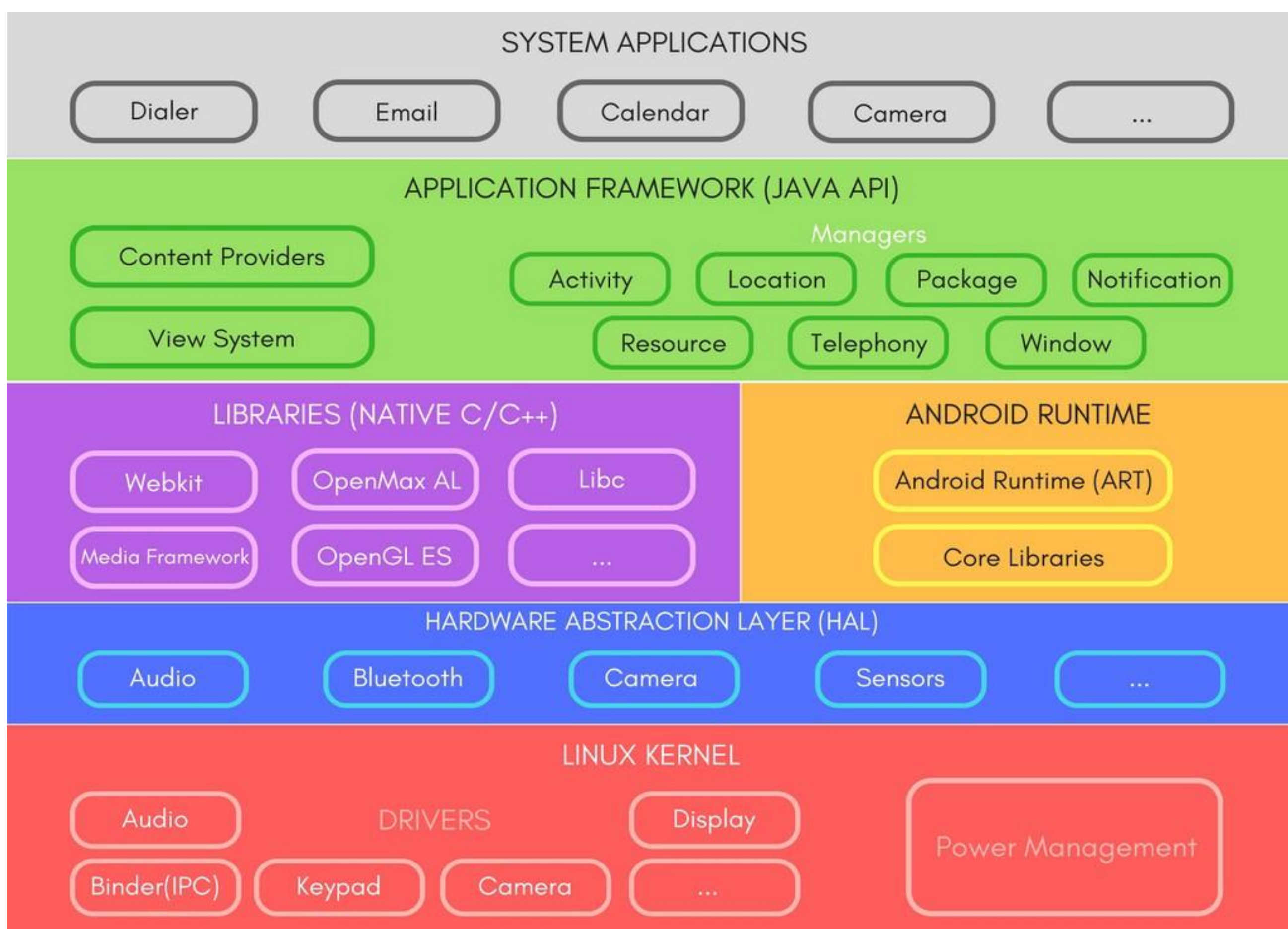


Рисунок 1.1 – Диаграмма архитектуры Android

### 1.1.1 Уровень приложений

Уровень приложений: это самый верхний уровень в архитектуре Android. Все приложения, такие как камера, карты Google, браузер, смс, календари, контакты являются родными приложениями. Эти приложения работают с конечным пользователем с помощью прикладной среды для работы. Фреймворк приложений – разрабатываемые приложения для Android, этот слой содержит необходимые классы и сервисы. Система представления - набор компонентов, которые используются для построения пользовательских интерфейсов приложения. Разработчики могут повторно использовать и расширять компоненты, уже присутствующие в API. На этом уровне есть менеджеры [5], которые позволяют приложению получать доступ к данным:

1. Activity manager: он управляет жизненным циклом приложений. Это позволяет правильно управлять всеми видами деятельности. Все действия контролирует activity manager;
2. Resource manager: обеспечивает доступ к ресурсам, таким как графика, анимация, звуки и т. д;
3. Content Providers – предоставляет возможность коммуницировать с другими приложения системы;
4. Notification manager: позволяет всем приложениям отображать пользовательские оповещения в строке состояния;
5. Location manager: выдает оповещения, когда пользователь входит или покидает указанное географическое местоположение.
6. Package manager: используется для получения данных об установленных пакетах на устройстве;
7. Window manager: он используется для создания видов и макетов;

8. Telephony manager: используется для управления настройками сетевого подключения и всей информацией об услугах на устройстве;
9. Android runtime: в этом разделе выполняются все приложения для Android. У Android есть собственная виртуальная машина, то есть DVM (Dalvik Virtual Machine), которая используется для запуска приложения Android. С этим DVK пользователи могут запускать несколько приложений одновременно.

Многие приложения, почти на всех устройствах Android поставляются со стандартным набор приложений от Google, к примеру Gmail, Google Maps, Google Chrome, Google Play Music, YouTube и другие приложения.

Приложения, которые расширяют функциональные возможности устройств, пишутся с использованием пакета разработки программного обеспечения Android (SDK) и, часто, языка программирования Java. Java можно комбинировать с C / C ++ вместе с выбором сред выполнения не по умолчанию, которые обеспечивают налучшую поддержку C ++. Язык программирования Go так же включен в список поддерживаемых, но имеет относительно скудный API (интерфейс прикладного программирования) Google сделали объявление в 2017 году о том, что язык программирования Kotlin становится приоритетным язык для разработки Android-приложений.

В Android SDK, как правило, входит весь набор инструментария для разработки приложений, такие как отладчик, виртуальные устройства, различные библиотеки, документация, примеры приложений, учебные материалы. Самой первой интегрированной средой разработки для Android приложений являлась Eclipse которая интегрировалась со специальным плагином Android Development Tools. А в 2014 году от Google появилась новая среда разработки называемая Android Studio, которая основывается на IntelliJ IDEA и стала основной средой для разработки Android-приложений. Так же имеются в доступе и другие средства для разработки, включая визуальную среду для неопытных разработчиков, набор

инструментов NDK для разработки на языках C или C++ , а так же множество кроссплатформенных решений для мобильных приложения для веба. В 2014 году Google разработали собственную платформу, на основе Apache Cordova, позволяющая переносить веб-приложения на Android.

В мире приложений Android происходит постоянный рост сторонних приложений, которые можно без труда установить на устройство, для этого нужно скачать и установить APK файл приложения либо воспользоваться специальным магазином мобильных приложений, с помощью которого можно устанавливать, удалять и обновлять приложения. Например, Google Play Store является основным магазином приложений, который предустановлен на устройствах Android, которые прошли сертификацию Google и имеют специальную лицензию Google Mobile Service. Магазин Google Play предоставляет возможность устанавливать и обновлять приложения от Google либо от сторонних разработчиков. По данным за 2020 год в магазине доступно почти три миллиона Android-приложений, а количество установок превышает двести миллиардов. Существуют мобильные операторы, которые предоставляют пользователям оплачивать покупки мобильных приложений, включая их стоимость в общую сумму за услуги связи. Больше миллиарда пользователей активных пользователей имеют приложения Google Play, Google Maps, Gmail и другие.

Поскольку Android является системой с открытым исходным кодом, помимо Google Play существуют и другие магазины приложений, нередко в таких магазинах представлены приложения, которые не доступны в Google Play из-за нарушений политики или по другой причине. Самыми популярными сторонними магазинами являются F-Droid, GetJar и Amazon Appstore, сторонние магазины стремятся предоставлять только те приложения, которые распространяются на условиях свободных лицензий и лицензий с открытым исходным кодом.

### 1.1.2 Уровень библиотек

Android имеет свои собственные библиотеки, написанные на C / C ++. Эти библиотеки не могут быть доступны напрямую. С помощью инфраструктуры приложения мы можем получить доступ к этим библиотекам. Существует много библиотек, таких как веб-библиотеки для доступа к веб-браузерам, библиотеки для android, видео форматов и т. д. Ниже представлено краткое описание некоторых основных библиотек Android, доступных для разработчика Android.

1. android.app – предоставляет доступ к модели приложения;
2. android.content – позволяет коммуницировать приложениям между собой и обмениваться данными;
3. android.database – предоставляет доступ к данным, и включает в себя инструменты для работы с базами данных SQLite;
4. android.opengl – набор инструментов для работы с трехмерной графикой.
5. android.os – дает доступ приложениям к системным функциям и службам;
6. android.text – предназначен для представления и управления текстовыми полями в приложении;
7. android.view – сюда включены основные компоненты для построения пользовательского интерфейса;
8. android.widget – представляет собой набор готовых элементов интерфейса;
9. android.webkit – набор инструментов для просмотра веб-страниц в приложении.

Расположенный на том же уровне, что и уровень библиотек, уровень среды выполнения Android также включает в себя набор библиотек языка

программирования Java. Разработчики Android-приложений создают свои приложения, используя язык программирования Java или Kotlin.

### 1.1.3 Виртуальная машина Dalvik

В операционную систему Android встроена собственная виртуальная машина под названием Davlik (Davlik Virtual Machine) [7]. Эта виртуальная машина использует свой байт-код, из-за этого программы написанные на языке программирования Java не могут сразу напрямую выполниться Android. Разработчики Android создали специальный инструмент (dx), позволяющий переводить файлы Java-классов в файлы, которые распознает Davlik, dex-файлы. Виртуальная машина DVM оптимизирована для мобильных устройств, чтобы обеспечить как можно более высокую производительность, ведь многие мобильные устройства имеют ограниченные ресурсы производительности, медленный процессов, малый объем памяти и скудную емкость батареи. Виртуальная машина разработана с учетом возможности эффективного запуска сразу нескольких таких машин на устройстве. Еще раз стоит отметить, что виртуальная машина использует измененное ядро Linux для выполнения многопоточный операций и других системных низкоуровневых операций. Версия 2.2 операционной системы Android претерпела ряд значительных модификаций в инфраструктуре JVM. До данной версии JVM выполнял роль интерпретатора. Не смотря на то, что в Android всегда был хороший интерпретатор, оно продолжало таким быть, поэтому новый код не создавался, с появление версии системы 2.2 туда добавили компилятор JIT, его задача заключается в том, чтобы переводить байт-код виртуальной машины Davlik в наиболее производительный машинный код. В будущем с Android будут развернуты дополнительные функции JIT и сборки мусора (GC), что еще больше снизит (потенциальную) совокупную производительность систем.

### 1.1.4 Уровень абстрагирования оборудования

Уровень абстрагирования оборудования (HAL) предоставляет стандартные интерфейсы, которые предоставляют аппаратные возможности устройства высокоуровневой структуре Java API. HAL состоит из нескольких библиотечных модулей, каждый из которых реализует интерфейс для определенного типа аппаратного компонента, такого как камера или модуль Bluetooth. Когда API-интерфейс платформы обращается к оборудованию устройства, система Android загружает модуль библиотеки для этого компонента оборудования.

### 1.1.5 Ядро Linux

Несмотря на то, что в основе Android лежит ядро Linux 2.6, стандартное ядро Linux Android не использует. Поэтому, нельзя утверждать, что Android является решением Linux. В список модификаций Android-ядра входят следующее:

1. Драйвер для управления разделяемой памятью;
2. Инструментарий для межпроцессного взаимодействия;
3. Решения для управления энергоэффективностью;
4. Инструменты для работы с памятью;
5. Регистратор.

Во время загрузки Android-системы ядро в первую очередь вызывает процесс init (такое же поведение наблюдает и в стандартном ядре Linux). Этот процесс взаимодействует со специальными файлами init.device.rc и init.rc. Посредством файла init.rc происходит запуск процесса zygote. Процесс zygote выполняет основные этапы обработки и загружает основные классы языка программирования Java. Все загруженные классы могут быть использованы Android-приложениями, соответственно этот этап ускоряет процесс загрузки. После того как процесс zygote загружен, он не используется и находится в режиме ожидания для дальнейших запросов.

Android-приложения запускаются изолированно в собственной среде процессов. Для межпроцессного взаимодействия, существует специальный драйвер, он обеспечивает IPC-связь. Всех необходимые объекты хранятся в общей памяти. С помощью разделяемой памяти, межпроцессная связь оптимизируется из-за того, что данных для передачи становится меньше. В отличии от многих сред Linux, в Android отсутствует место подкачки, поэтому вся виртуальная память эта память доступная на мобильном устройстве. [03].

## 1.2 Проблемы, связанные с тестированием Android-приложений

Большинство приложений Android написаны на Java, но приложения Android обладают уникальными характеристиками и дополнительными функциями, выходящими за рамки традиционного программного обеспечения, включая способ их разработки, тестирования, распространения и установки [10]. В частности, эти новые характеристики и функции делают весь процесс тестирования приложений Android разным и ставят перед разработчиками и тестировщиками несколько проблем при тестировании приложений Android. В этом разделе представлены уникальные характеристики и связанные с ними проблемы, выявленные в ходе исследования приложений Android, которые необходимо учитывать при разработке и тестировании приложений Android.

### 1.2.1 Четыре типа компонентов

Приложения Android содержат следующие компоненты: Content Providers Services, Broadcast Receivers, и Activities. Например, в файле манифеста приложения приложение определяет Activity для работы в качестве «основного класса» вместе с разрешениями безопасности и подписками на целевые трансляции. Эти компоненты написаны на Java и предоставляются комплектом разработчика программного обеспечения Android (SDK). Разработчики могут расширять эти суперклассы (компоненты) и предоставлять собственную реализацию на основе требований к программному обеспечению. Действие

представляет пользователю графический интерфейс пользователя (GUI) на основе одного или нескольких макетов. GUI может включать в себя виджеты, такие как кнопки, текстовые представления и другие расширенные артефакты. На рисунке 1.2 показан пример Activity.

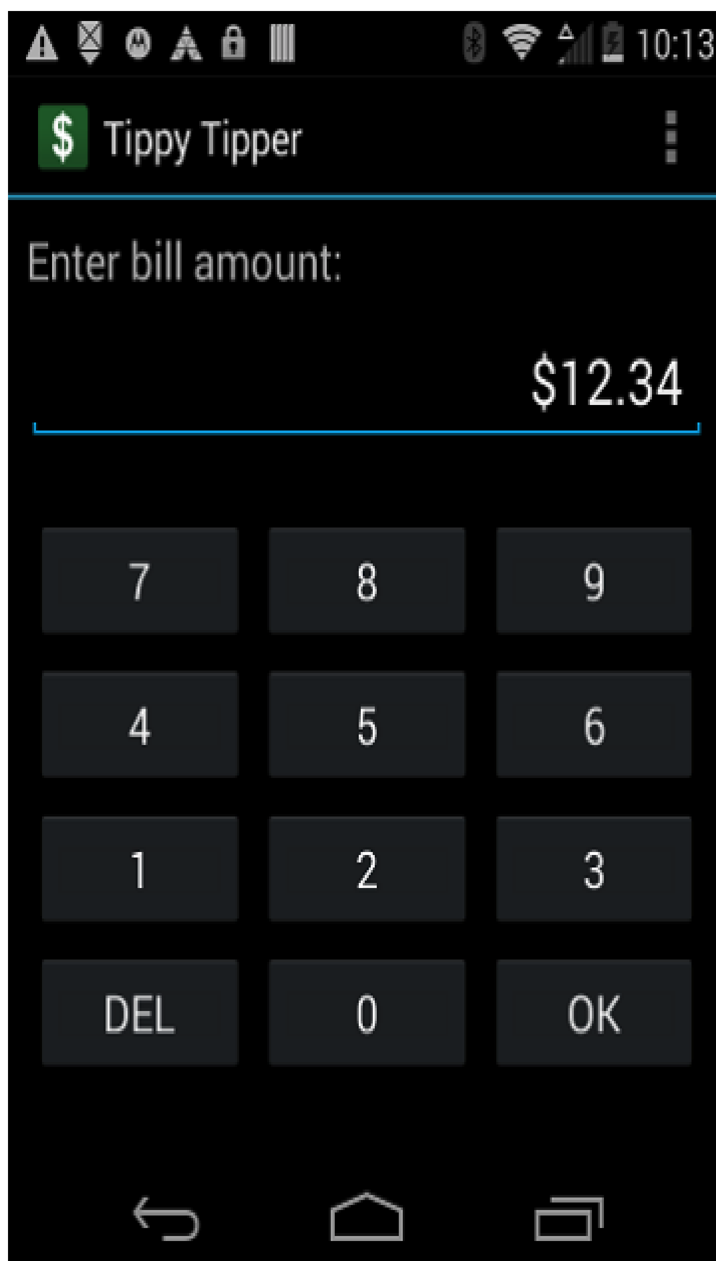


Рисунок 1.2 – Пример Activity

Разработчики заполняют графический интерфейс, переопределяя методы жизненного цикла Activity, и могут добавлять логику программирования для GUI. Кроме того, разработчики отделяют визуальную структуру от поведения с помощью расширяемого языка разметки (XML) для объявления макета приложения. Сервисы не имеют графического интерфейса и работают в фоновом режиме. Они не взаимодействуют с экраном. Таким образом, они обычно используются для выполнения долгосрочных задач, таких как воспроизведение музыки или запуск будильника. Служба может быть запущена другими компонентами, включая Activity и Broadcast Receivers. Broadcast Receiver используется для подписки приложения на Intents, транслируемое системой Android или другими приложениями, такими как низкий уровень заряда батареи. На рисунке 1.3 приведен пример Broadcast Receiver.

Рисунок 1.3 – Пример Broadcast Receiver

Content Provider предоставляет и управляет структурированными данными для других приложений. Эти данные хранятся в файловых системах или базах данных, включая календари, фотографии, контакты и сохраненную музыку. Например, на рисунке 1.4 показан пример использования Content Provider для доступа к файлам, хранящимся на устройстве. Взаимодействуя друг с другом, поставщики и клиенты представляют собой интерфейс для доступа и взаимодействия с данными, и так же обрабатывают взаимодействие между процессами и защищают данные.

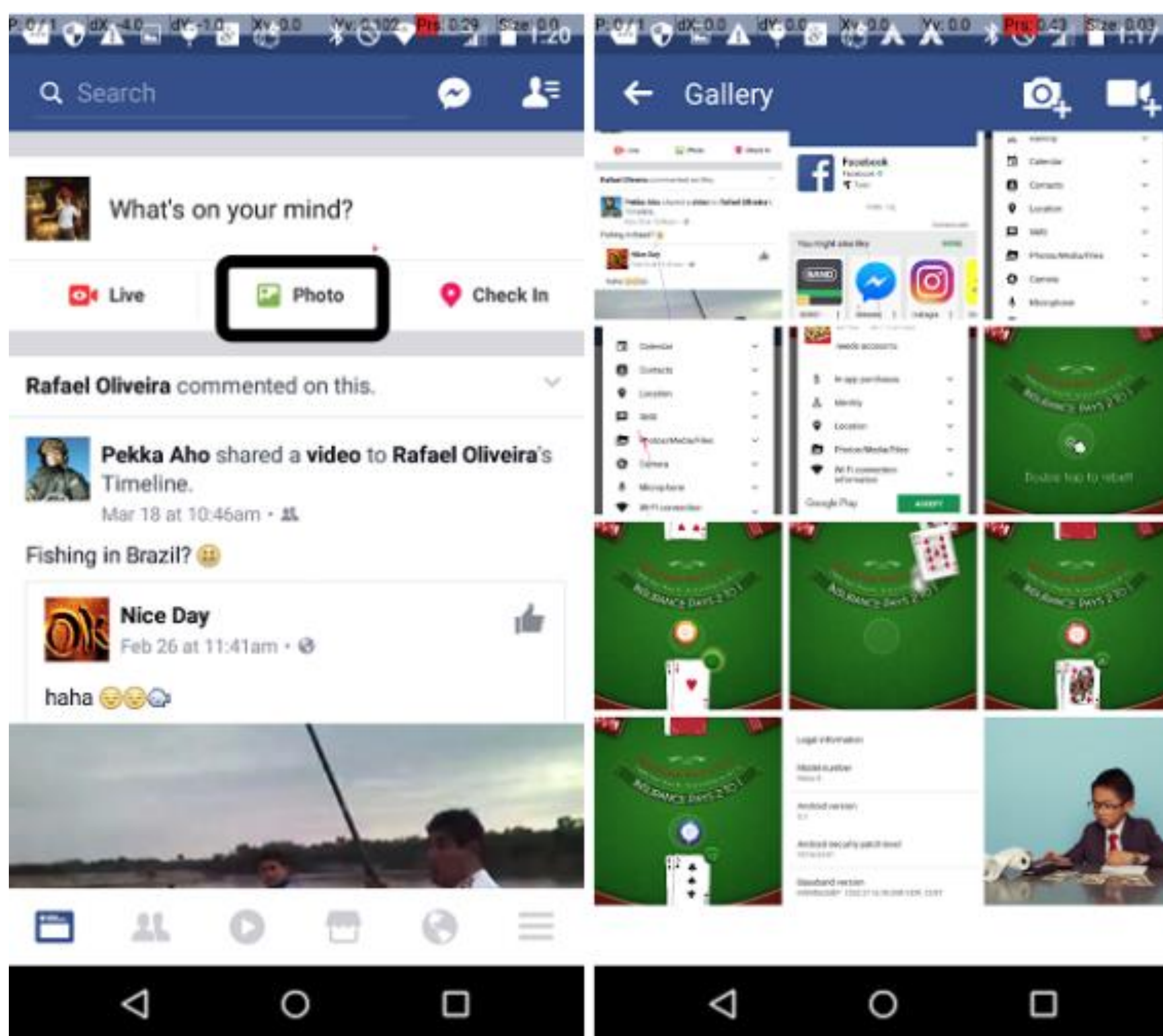


Рисунок 1.4 – Пример использования Content Provider

Когда пользователь нажимает кнопку «Фото», вызывается поставщик контента, который предоставляет все фотографии, сохраненные на устройстве.

Затем пользователь может выбрать один и загрузить его на Facebook. Некоторые исследования, такие как MobiGUITAR [11], рассматривают приложения Android так же, как программное обеспечение GUI, и применяют методы тестирования GUI для тестирования приложений Android. Эти подходы могут быть пригодны для тестирования действий, так как они отображаются на экране. Однако они не могут тестировать другие компоненты всесторонне.

### 1.2.2 Уникальные жизненные циклы компонентов Android

Жизненный цикл Android-приложения тщательно мониториться операционной системой и подстраивается под необходимые запросы и нужны пользователя и зависит от доступных ресурсов. Если пользователь захотел открыть веб-страницу в браузере, решение о запуске определенного приложения висит на системе. В свою очередь система соблюдает четко прописанные и логичные правила, которые определяют, что нужно загрузить, либо приостановить, либо прекратить процесс.

Система дает преимущества открытым на переднем плане приложениям, остальные процессы система может завершить, если определяет, что нужно высвободить ресурсы для более производительной работы системы, таким образом будут закрыты приложения с меньшим приоритетом, это происходит потому что ресурсы, как правило, ограничены и нужно их экономить.

В отличие от других типов программного обеспечения, операционной системе Android требуется, чтобы основные компоненты приложений Android работали в соответствии с заранее определенным жизненным циклом [3].

Если жизненный цикл компонента не обрабатывается надлежащим образом, это может привести к неожиданным проблемам, когда пользователи переключаются между различными приложениями, приостанавливают, затем возобновляют работу приложения или включают мобильное устройство из спящего режима [12]. В частности, поток непрерывности в некоторых приложениях для

Android имеет решающее значение для пользователей, таких как игры. Например, на рисунке 1.5 показано игровое приложение, которое неправильно обрабатывает жизненный цикл Activity. Картинка слева показывает ход игры, где пользователь собирается выиграть карточную игру. Он принимает телефонный звонок, однако, когда пользователь заканчивает вызов и возвращается в игру, приложение не может правильно обрабатывать жизненный цикл, вызывая правильные методы, чтобы восстановить ход игры с того места, где он был прерван. Вместо этого он перезапускается с начала, как показано на рисунке справа.

Рисунок 1.5 – Пример неправильной обработки жизненного цикла Activity

На рисунке 1.6 показан жизненный цикл activity в форме состояний и переходов событий. Activity имеет три состояния, связанные различными условиями: «Выполнено», «Приостановлено» и «Остановлено».

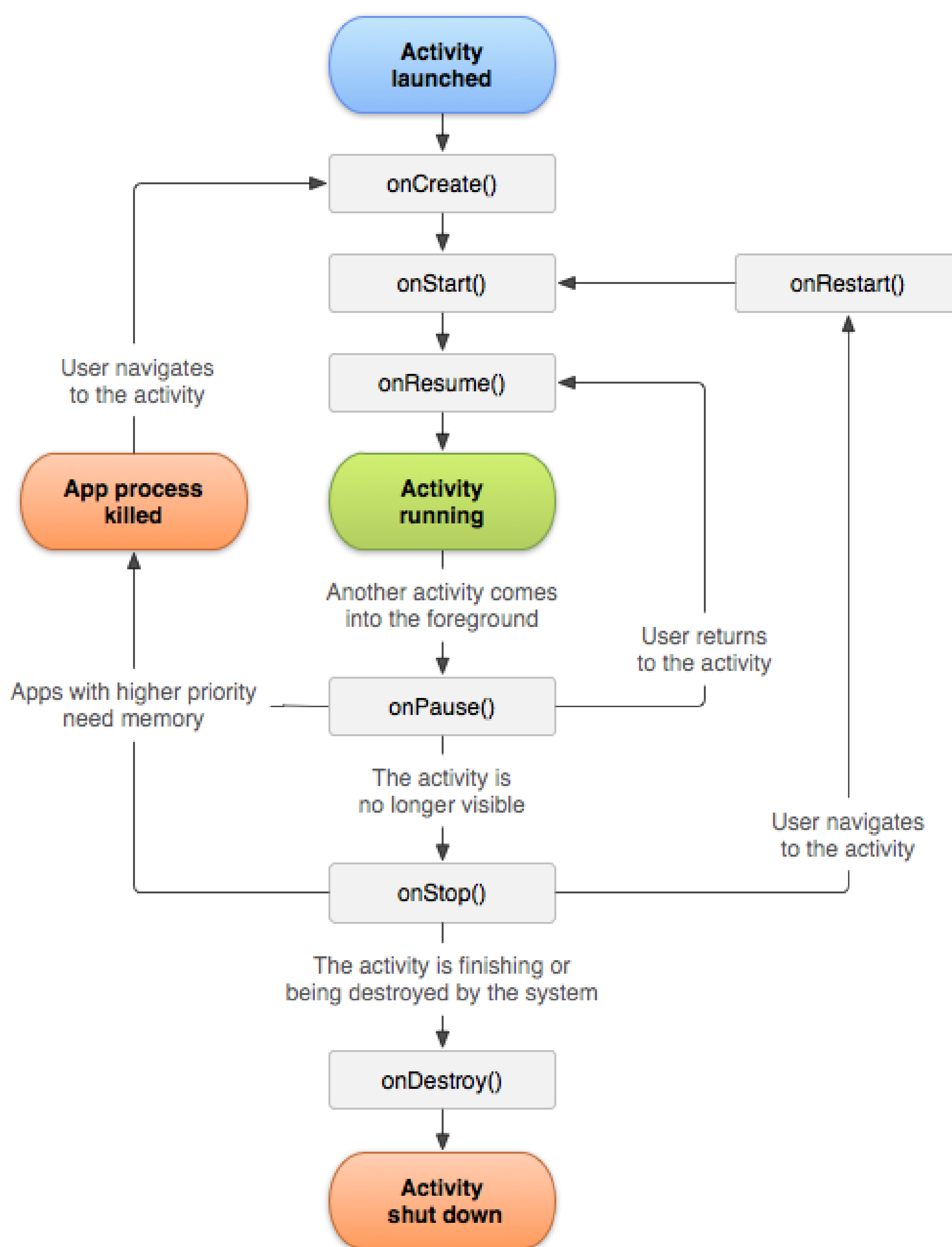


Рисунок 1.6 – Жизненный цикл activity

После запуска Activity необходимо последовательно вызвать три метода onCreate(), onStart() и onResume(), прежде чем будет достигнуто состояние Running, т. е. пользователь может видеть активность на экране. Метод onPause() отправляет

Activity в состояние Paused, где он может вернуться в состояние Running с помощью onResume(), или переходит в состояние Stopped с помощью onStop(). Когда пользователь хочет вернуться к остановленному Activity, необходимо использовать три метода: onRestart(), onStart() и onResume() они последовательно вызываются, чтобы вернуть активность на экран. Когда Activity больше не будет использоваться, он может завершиться с помощью метода onDestroy(). В отличие от activity, которое отображает экран для пользователя, services невидимы и выполняют долгосрочные задачи в фоновом режиме, такие как воспроизведение музыки. Services также ведут себя в соответствии с заранее определенным жизненным циклом, но он не совпадает с жизненным циклом действия. На рисунке 1.7 показан жизненный цикл activity в виде набора методов и состояний событий.

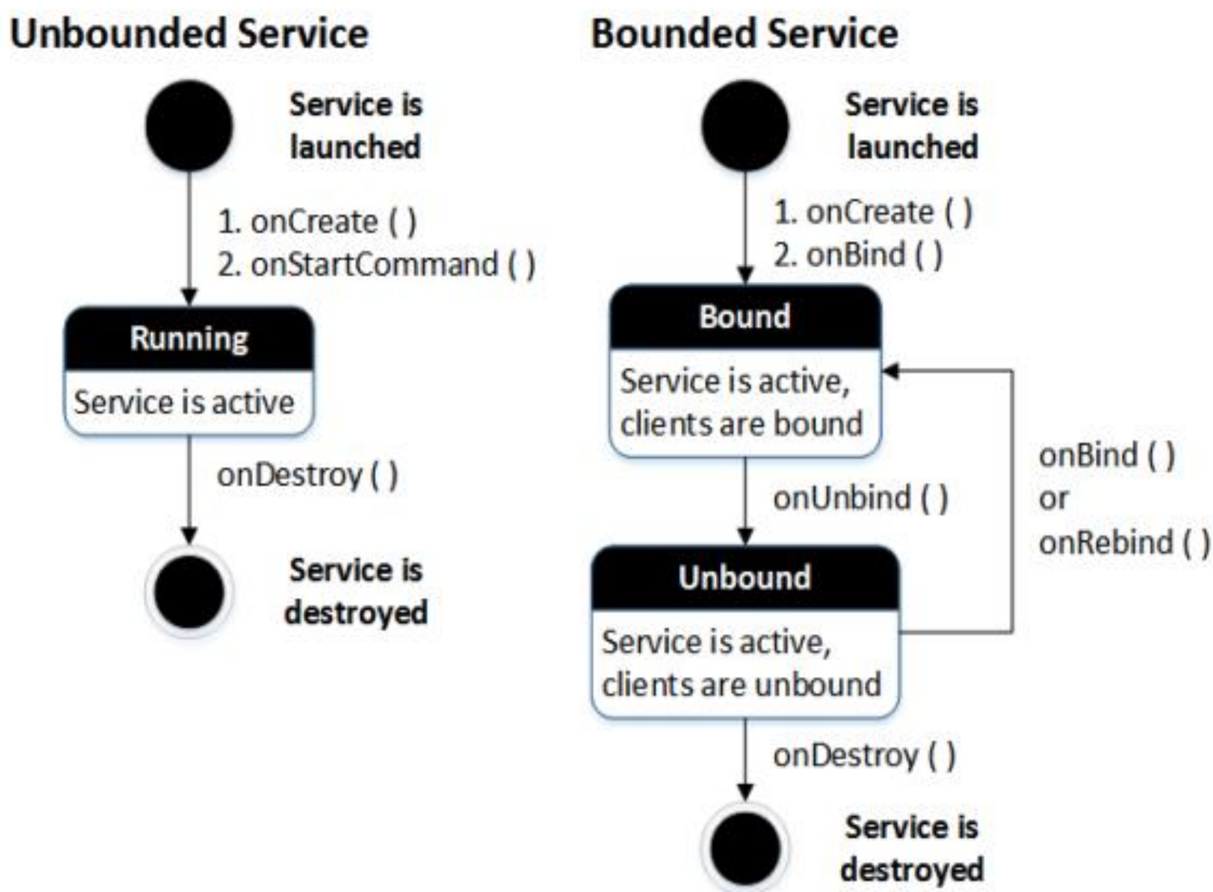


Рисунок 1.7 - Жизненный цикл activity

Services бывают двух типов: неограниченные (unbounded) и ограниченные (bounded). Неограниченный service запускается после того, как система Android выполняет методы service onCreate() и onStartCommand(), обычно, когда другие компоненты приложения запрашивают service. Затем service остается в фоновом режиме и выполняет свою работу. Как только service завершает свою задачу, она может остановить себя или быть остановлена своим клиентом.

Система Android вызывает метод onDestroy() для прекращения работы service. Ограниченная service запускается, когда клиентский компонент запрашивает привязку к нему. Система Android вызывает свои методы onCreate() и onBind() для запуска service. Service может принимать запросы на привязку от нескольких клиентов одновременно. Если service является чисто ограниченной service, то есть запущена с запросом привязки клиента, после отсоединения всех клиентов от service, система Android вызывает методы onUnbind() и onDestroy() для остановки service. Кроме того, две формы service не являются взаимоисключающими. Ограниченная service также может быть запущена с помощью метода onStartCommand(). Затем клиент может снова выполнить привязку к service после вызова метода onUnbind(), пока service все еще активна. Система Android вызовет либо метод onRebind(), либо метод onBind() для повторного связывания service. Однако, если эта service больше не нужна, ее нужно либо остановить самостоятельно, либо остановить ее клиентом, но не системой Android. Поскольку связь между service и другими компонентами имеет решающее значение для бесперебойного выполнения приложений Android, особенно для приложений, требующих правильного запуска services, таких как музыкальные проигрыватели, почтовые клиенты и будильники, разработчики должны надлежащим образом осуществлять service в соответствии с жизненными циклами service. В исследовании приложения с будильником были обнаружены программные ошибки в его services, из-за которых приложение работало в неправильное время, а некоторые приводили к сбоям во время выполнения.

### 1.2.3 Интенсивное использование XML-файлов

Несмотря на то, что исходный код большинства приложений Android написан на Java, XML-файлы также интенсивно используются приложениями Android для конфигурации программы, спецификации пользовательского интерфейса (макета) и временного хранения данных. Например, на рисунке 1.8 показан пример файла макета XML в приложении Android. Этот XML-файл определяет общую структуру макета текущей Activity как RelativeLayout, а затем создает TableLayout внутри него. Несколько виджетов Button и TextView с различными размерами и шрифтами также определены в Activity.

Язык программирования Java использует различные подходы к использованию XML-файлов, например объектная модель документов (Document Object Model – DOM) или SAX (Simple Api for XML), но она не поддерживается в Android, но в состав Android входит похожая библиотека для работы с XML-файлами которая не уступает по возможностям.

В зависимости от дизайна различных проектов приложение для Android может включать в себя даже больше файлов XML, чем исходных файлов Java. Однако использование файлов XML для этих целей является относительно новым.

Например, программы с графическим интерфейсом пользователя (GUI), реализованные с Java или C #, редко используют или включают файлы XML. Следовательно, никакие методы тестирования, разработанные для тестирования традиционных программ на Java, не учитывают исходный код, отличный от Java, в одном и том же проекте. Кроме того, не существует критерия покрытия теста дляизмерения информации о покрытии для файлов XML, используемых в приложениях Android.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
    <TableLayout
    android:id="@+id/level_table"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:stretchColumns="0,1">
    <TableRow>
      <Button android:id="@+id/up_button"
      android:layout_marginTop="5px"
      android:text="@string/up_level"/>
      <Button android:id="@+id/up_gear_button"
      android:layout_marginTop="5px"
      android:text="@string/up_gear_level"/>
    </TableRow>
    <TableRow>
      <TextView android:id="@+id/current_level"
      android:layout_width="wrap_content"
      android:layout_gravity="center"
      android:textSize="70sp"
      android:textStyle="bold"
      android:text="1"/>
    <TextView android:id="@+id/current_gear_level"
      android:layout_width="wrap_content"
      android:layout_gravity="center"
      android:textSize="70sp"
      android:textStyle="bold"
      android:text="0"/>
    </TableRow>
    <TableRow>
      <Button android:id="@+id/down_button"
      android:text="@string/down_level"/>
      <Button android:id="@+id/down_gear_button"
      android:text="@string/down_gear_level"/>
    </TableRow>
    </TableLayout>
    <TextView android:id="@+id/total_level"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:gravity="center"
    android:layout_below="@id/level_table"
    android:textSize="140sp"
    android:textStyle="bold"
    android:text="1"/>
</RelativeLayout>
```

Рисунок 1.8 – Пример файла макета XML в приложении Android

Для данного теста мы можем легко наблюдать, выполняется оператор или логическая ветвь или нет. Однако для XML-файла в том же проекте у нас нет

методов оценки его охвата. Действительно, не тестирование XML-файлов приложений Android может привести к неожиданным сбоям.

### 1.2.4 Контекстно-зависимые характеристики

Еще одна важная особенность приложений Android находится в том, что они представляют собой контекстно-зависимые приложения. Приложения получают разнообразные входные данные из своей физической среды через различные датчики, такие как линейное ускорение, определение местоположения в системе глобального позиционирования (GPS) и вращение. Например, контекстно-зависимые приложения ведут себя по-разному, когда телефон перемещается в автомобиле, а не сидит за столом. На рисунке 1.9 показан еще один реальный пример с широко используемым приложением для Android, Yelp.

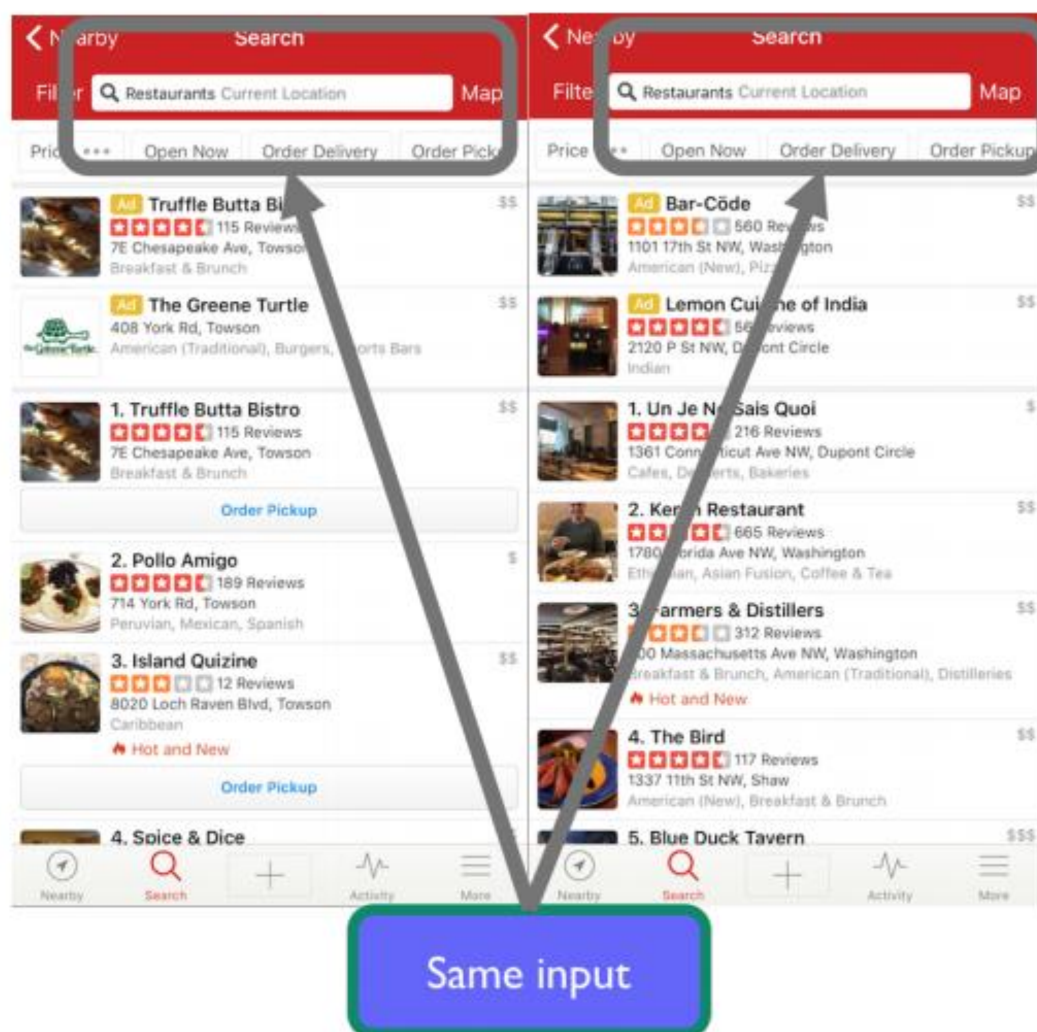


Рисунок 1.9 – Пример контекстно-зависимого ввода

Учитывая точно такой же тестовый ввод, выбор «Рестораны», «Текущее местоположение», затем «Поиск» в разных местах приводит к совершенно разным результатам. Несомненно, данные о местоположении, полученные от устройства, напрямую влияют на результаты. И эти типы данных не поступают от пользователей, а от датчиков в самом устройстве. Кроме того, это различие в поведении не отражается непосредственно в исходном коде приложения; скорее разница в том, как часто приложение получает уведомление о событии.

В некотором смысле эти уведомления о событиях также являются входными данными и должны моделироваться как часть теста. Однако существующие методы испытаний не учитывают эти типы входных данных.

### 1.2.5 Два типа ориентации экрана

При создании первых мобильных устройств за основу брали операционную системы персональных компьютеров и дополнялась, и видоизменялась для мобильных устройств. Функции, которые не требовались убирали из системы, оставляя лишь необходимое, изначально, никто не додумался, что телефон может менять ориентацию и только спустя несколько версий, появились системный функции для поворота экрана.

Возможность менять ориентацию является ключевой характеристикой мобильных устройств. Практически у каждого устройства есть два типа ориентации экрана: альбомная и книжная. Ориентация экрана переключается автоматически, когда устройство обнаруживает изменение в том, как оно удерживается пользователем, если оно не заблокировано пользователем вручную. Многие приложения для Android разработаны таким образом, чтобы иметь различные пользовательские интерфейсы для адаптации к событию изменения ориентации. Например, на рисунке 1.10 показано, что простой калькулятор (слева) с книжной ориентацией становится научным калькулятором (справа) с альбомной ориентацией.

Рисунок 1.10 – Пример адаптации приложений Android к изменению ориентации

В отличие от традиционного программного обеспечения, которое не учитывает ориентацию, тестирование приложений Android должно учитывать эту уникальную особенность изменения ориентации. Потому что с высокой вероятностью это может привести к различным типам отказов, таким как немедленный сбой после изменения ориентации [13].

### 1.2.6 Разрешения приложений Android

Для защиты безопасности операционной системы Android и конфиденциальности пользователей каждое приложение Android выполняется в отдельной изолированной программной среде с ограниченными разрешениями. Если приложение должно получить доступ к системным ресурсам или данным пользователя, он должен явно объявить запрошенные разрешения в своем файле AndroidManifest.xml. Например, на рисунке 1.11 показана часть файла приложения AndroidManifest.xml, который запрашивает три разрешения: WAKE LOCK, MODIFY AUDIO SETTINGS и VIBRATE.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  <uses-permission android:name="android.permission.WAKE_LOCK" />
  <uses-permission android:name="android.permission.MODIFY_AUDIO_SETTINGS" />
  <uses-permission android:name="android.permission.VIBRATE">
  </uses-permission>
</manifest>
```

Рисунок 1.11 – Android-приложение запрашивает разрешения

Обычно, когда приложение установлено на устройстве Android, система Android запрашивает у пользователя решение о предоставлении разрешений. На снимке экрана слева на рисунке 1.12 перечислены разрешения, запрошенные Facebook, включая календарь, контакты, местоположение, файлы, камера и т. д.

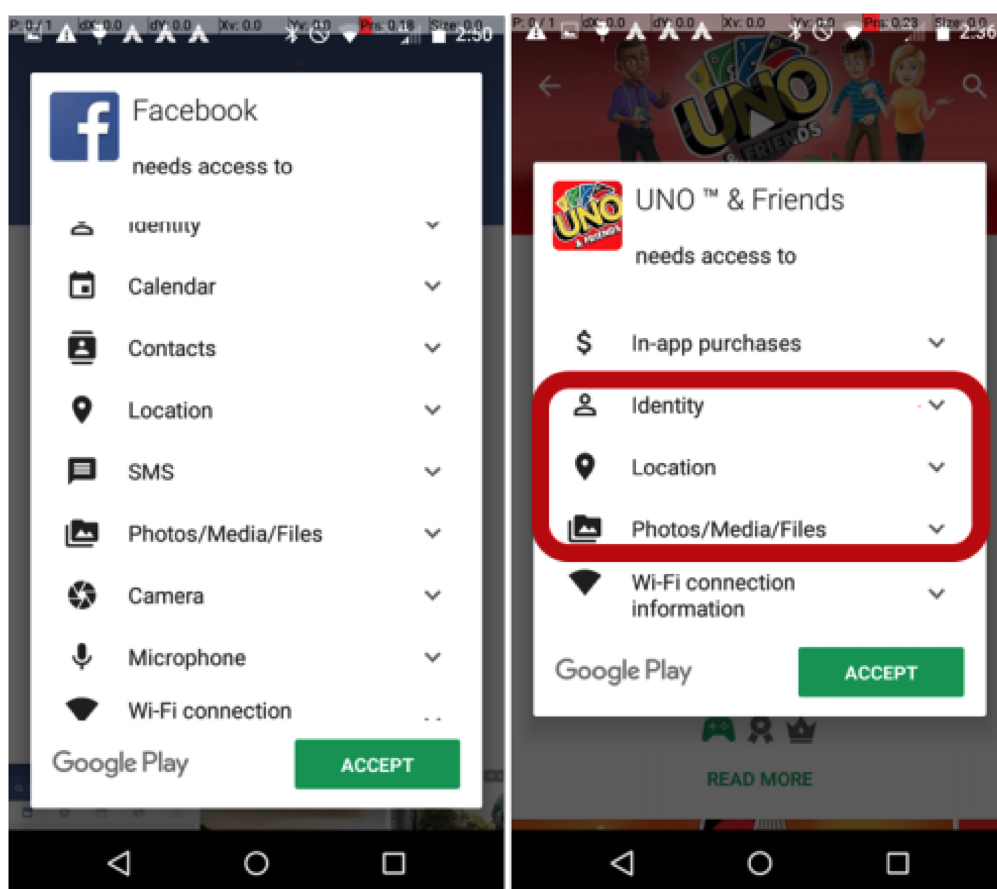


Рисунок 1.12 – Пример установки приложений Android с разрешениями

В качестве приложения Android для социальных сетей Facebook имеет разумный доступ к этим системным ресурсам и пользовательским данным. Однако

на скриншоте справа на рисунке 1.12 видно, что UNO хочет получить доступ к идентификационным данным, местоположению и файлам пользователей, которые не нужны для карточной игры. Однако этот механизм обычно не обеспечивает безопасность Android, как ожидалось. Многие пользователи просто принимают каждый запрос. Несомненно, этот механизм приводит к нескольким проблемам тестирования и уязвимостям безопасности, которые можно использовать для эксплуатации устройств Android.

Предполагается, что механизм разрешений обеспечивает безопасность пользователей, в то время как некоторые разработчики используют его для выполнения вредоносных действий, которые обычно не обнаруживаются традиционными методами тестирования.

### 1.2.7 Разнообразные сетевые подключения

Обычно смартфон Android оснащен несколькими видами сетевых подключений, чаще всего сотовой передачей данных и WiFi. По умолчанию, когда доступно соединение WiFi, система Android сначала пытается передать данные через WiFi, поскольку соединения WiFi используют относительно меньше энергии, работают с более высокой скоростью загрузки и выгрузки и стоят дешевле. Если WiFi недоступен, если сотовые данные не отключены вручную, система Android автоматически переключится на сотовое соединение для передачи данных, которое имеет высокую скорость сети, дороже и потребляет больше энергии. Это переключение может вызвать проблемы в различных сценариях. Например, IP-адреса различаются, когда устройство подключено к различным сетям.

Очень часто сеть, к которой подключается смартфон, имеет затруднения при возобновлении соединения и не может продолжить выполнение предыдущих незавершенных задач. Прерванные установки, приводящие к частичным и нестабильным установкам, а также неполные или дублирующие мобильные заказы на покупку, очень распространены. Многие разработчики и тестировщики игнорируют эту ситуацию, в результате чего пользователи должны помнить о том,

чтобы оставаться подключенным к одной сети, не перемещаясь и не переключаясь, прежде чем важное приложение завершит выполнение своих задач.

### 1.2.8 Ограниченное время автономной работы

В отличие от ПК, которые имеют постоянный источник питания, мобильные устройства должны полагаться на ограниченный заряд батареи. Никакое выполнение не может произойти, когда батарея разряжена. Поэтому разработчики должны учитывать использование батареи при реализации своих приложений.

Некоторые производители предусматривают с своих девайсах энергосберегающий режим, они ограничивают функциональность мобильного устройства, тем самым ослабляя нагрузку на процессор и уменьшая расход энергии. Таких режимов может быть несколько, например простой, более экономный, суперэкономный.

Режим энергосбережения полезен тем тем, что не нужно отключать каждое приложение, это не удобно, проще зайти в настройки и включить режим, а дальше система все сделает сама. Например, смартфоны Sony Xperia, которые имеют режим Stamina. А у телефонов Philips с большим объемом аккумуляторов, рассчитанными на тех, кто желает более длительную работы смартфона от батареи, на корпусе может быть специальная кнопка: достаточно нажать на нее, и даже не нужно входить в меню и искать нужную настройку.

Однако многие разработчики упускают из виду использование батареи. Некоторые исследователи обнаружили, что некоторые неправильные методы программирования могут увеличить потребление энергии приложениями Android [14]. Некоторые исследовательские проекты маркируют их как энергетических жуков [15, 16, 17].

Даже если энергетические ошибки не ухудшают функциональность приложения, они могут серьезно повлиять на систему Android и сократить

доступность всей системы. На рисунке 1.13 сравниваются две диаграммы энергопотребления, полученные из одного и того же приложения.

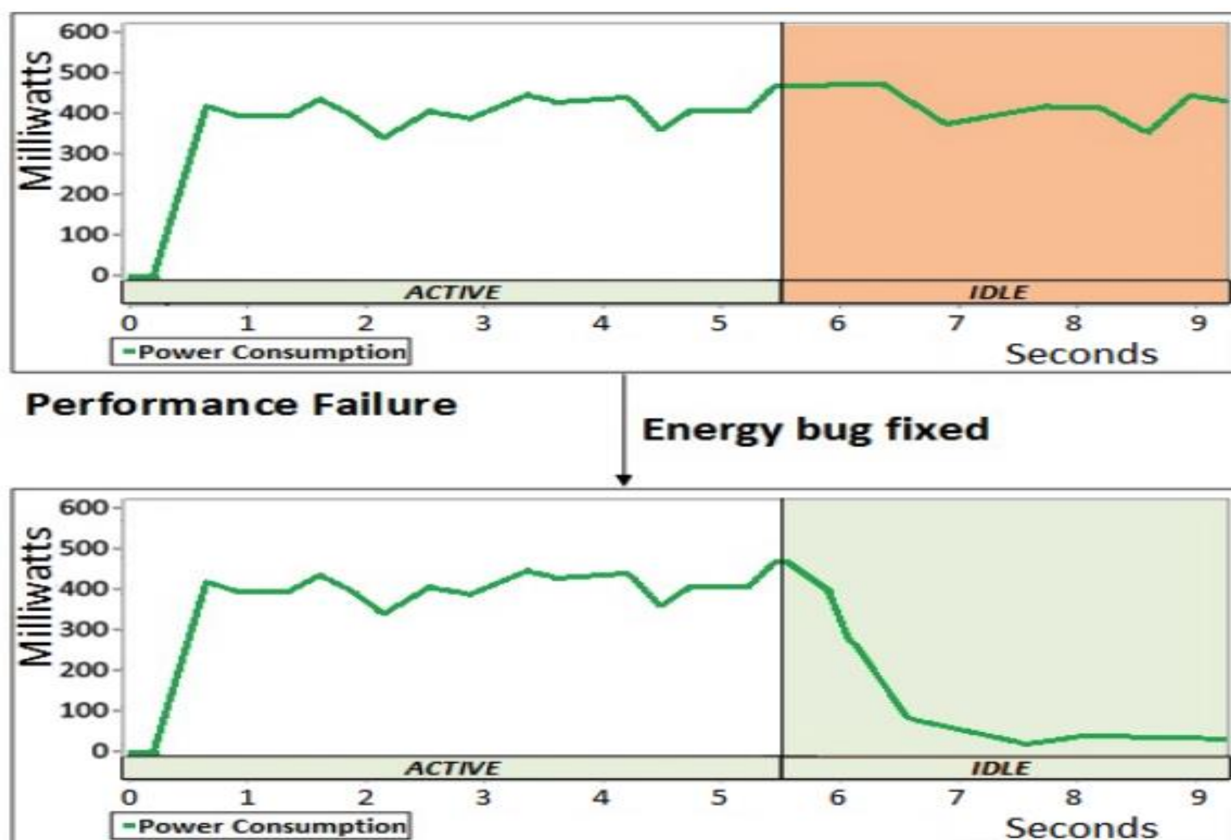


Рисунок 1.13 - Пример ошибки энергопотреблении

Первоначально в приложении имеется ошибка энергопотребления (вверху), поэтому после перехода из активного состояния в состояние ожидания оно не освобождает запрошенные системные ресурсы и продолжает потреблять энергию батареи. В идеале приложение должно получать системные ресурсы в активном состоянии и освобождать их в режиме ожидания. Эти состояния отличаются от фона и состояния переднего плана компонентов Android, поскольку приложение может быть активным в фоновом режиме, например services. Диаграмма внизу рисунка 1.13 показывает ожидаемое потребление энергии после исправления энергетической ошибки.

## Выводы по разделу один

В данной главе была исследована архитектура операционной системы Android, проведен анализ компонентов и функций Android-приложений, выяснилось, что Android использует новые функции программирования, которые никогда ранее не использовались традиционным программным обеспечением. Эти уникальные характеристики приложений Android приводят к новым типам ошибок, которые обычно не выявляются существующими методами тестирования программного обеспечения. Следовательно, нужны специализированные инструменты и подходы для тестирования Android-приложений.

# 2 ИССЛЕДОВАНИЕ ИНСТРУМЕНТОВ И МЕТОДОЛОГИЙ ТЕСТИРОВАНИЯ ANDROID-ПРИЛОЖЕНИЙ

## 2.1 Введение в тестирование программного обеспечения

Тестирование программного обеспечения – анализ программного продукта и различной связанной с ним документации для обнаружения ошибок и дефектов, и улучшения качества продукта.

Раннее обнаружение ошибок экономит огромное количество ресурсов проекта и снижает затраты на обслуживание программного обеспечения. Это самая известная причина для написания тестов для проекта по разработке программного обеспечения. Увеличение производительности станет очевидным. Кроме того, написание тестов даст более глубокое понимание требований и решаемой проблемы. Невозможно писать тесты для программного обеспечения, которое не понятно.

Чем больше кода охватывается тестами, тем выше вероятность обнаружения скрытых ошибок. Если во время этого анализа покрытия обнаружится, что некоторые области кода не выполняются, следует добавить дополнительные тесты, чтобы охватить и этот код.

## 2.2 Жизненный цикл программного обеспечения

Жизненный цикл программного обеспечения обычно включает в себя следующее: анализ требований, проектирование, кодирование, тестирование, установка и обслуживание. В промежутке между ними может возникнуть необходимость в выполнении различных операций и поддержки продукта.

1. Анализ требований. Организации по разработке программного обеспечения предоставляют решения для требований клиентов, разрабатывая соответствующее программное обеспечение, которое наилучшим образом соответствует их спецификациям. Таким образом,

жизнь программного обеспечения начинается с возникновения требований. Очень часто эти требования являются расплывчатыми, возникающими и всегда подвержены изменениям. Анализ выполняется для: - Провести углубленный анализ предлагаемого проекта, оценить техническую осуществимость, выяснить, как разделить систему, определить, какие области требований необходимо разработать от клиента, чтобы определить влияние изменений. к требованиям, чтобы определить, какие требования должны быть выделены для каких компонентов.

2. Проектирование и технические характеристики. Результатом анализа требований является спецификация требований. Используя это, разрабатывается общий дизайн для предполагаемого программного обеспечения. Действия на этом этапе - Выполнение архитектурного проектирования для программного обеспечения, Разработка базы данных (если применимо), Разработка пользовательских интерфейсов, Выбор или разработка алгоритмов (если применимо), Выполнение детального проектирования.

3. Программирование. Процесс разработки имеет тенденцию проходить итеративно, а не линейно; для описания этого процесса было предложено несколько моделей (спираль, водопад и т. д.). Действия на этом этапе - создание тестовых данных, создание исходного кода, генерация объектного кода, создание рабочей документации, планирование интеграции, выполнение интеграции.

4. Тестирование. Процесс использования разработанной системы с целью поиска ошибок. Дефекты / недостатки / ошибки, обнаруженные на этом этапе, будут отправлены разработчику для исправления и должны быть повторно протестированы. Эта фаза повторяется пока ошибки исправляются в соответствии с требованиями. Действия на этом этапе -

планирование и проверка плана, выполнение задач проверки и проверки, сбор и анализ метрических данных, планирование тестирования, разработка требований к испытаниям, установка выполнения испытаний. Разработанное и протестированное программное обеспечение должно быть наконец установлено на клиентском компьютере.

5. Установка ПО. Тщательное планирование должно быть сделано, чтобы избежать проблем для пользователя после завершения установки. Действия на этом этапе – планирование установки, популяризация программы, установка программы, делегация программного продукта в операционную систему.
6. Поддержка. Дальнейшая поддержка осуществляется компанией, которая разработала программное обеспечение. Обе стороны обычно принимают решение об этих действиях до разработки системы. Действия на этом этапе - эксплуатация системы, оказание технической помощи и консультации, ведение журнала запросов на поддержку.
7. Техническое обслуживание. Процесс не останавливается, когда он полностью внедрен и установлен у пользователя; на этом этапе осуществляется разработка новых функций, улучшений и т. д. Действия на этом этапе - повторное применение жизненного цикла программного обеспечения.

### 2.3 Жизненный цикл тестирования программного обеспечения

Жизненный цикл тестирования программного обеспечения состоит из шести (общих) этапов:

1. Планирование;
2. Анализ;

3. Проектирование;
4. Разработка;
5. Циклы тестирования;
6. Окончательное тестирование;
7. Внедрение.

Каждая фаза в жизненном цикле описана с соответствующими действиями:

1. Планирование. Планирование плана тестирования высокого уровня, плана обеспечения качества (цели качества), определение - процедуры отчетности, классификация проблем, критерии приемлемости, базы данных для тестирования, критерии измерения (количество дефектов / уровень серьезности и происхождение дефектов), показатели проекта и, наконец, начало графика проекта тестирование. Кроме того, планируем сохранить все контрольные примеры (ручные или автоматизированные) в базе данных;
2. Анализ. Включает действия, которые: - разработка функциональной проверки на основе требований бизнеса (написание тестовых примеров на основе этих деталей), разработка формата тестовых примеров (оценки времени и приоритетов), разработка циклов тестирования (матриц и временных шкал), определение тестовых случаев для автоматизации (если ), определить область стресс-тестирования и тестирования производительности, спланировать циклы тестирования, необходимые для проекта и регрессионного тестирования, определить процедуры для обслуживания данных (резервное копирование, восстановление, проверка), просмотреть документацию;
3. Проектирование. Действия на этапе проектирования - пересмотреть план тестирования на основе изменений, пересмотреть матрицы и сроки цикла тестирования, убедиться, что план и случаи тестирования

находятся в базе данных или реквизите, продолжать писать тестовые примеры и добавлять новые на основе изменений, разрабатывать критерии оценки риска формализовать детали для стресс-тестирования и тестирования производительности, завершить циклы тестирования (количество тестовых наборов за цикл на основе оценок времени для тестового набора и приоритета), завершить план тестирования (оценить ресурсы для поддержки разработки в модульном тестировании);

4. Построение (фаза модульных испытаний). Заполните все планы, заполните матрицы и сроки цикла тестирования, выполните все контрольные примеры (вручную), начните стресс-тестирование и тестирование производительности, протестируйте автоматизированную систему тестирования и исправьте ошибки (поддержка разработки в модульном тестировании), запустите пакет приемочных испытаний QA для сертификации программного обеспечения. готов сдать в QA;
5. Тестовый цикл(ы) / Исправление ошибок (фаза повторного тестирования и тестирования системы). Запустите тестовые случаи (внешний и внутренний), отчеты об ошибках, проверку и пересмотрите / добавьте тестовые примеры по мере необходимости;
6. Окончательное тестирование и внедрение (этап замораживания кода). Выполнение всех предварительных тестовых случаев - ручное и автоматическое, выполнение всех внутренних тестовых примеров - ручное и автоматическое, выполнение всех стресс-тестов и тестов производительности, обеспечение текущих показателей отслеживания дефектов, обеспечение текущих показателей сложности и проектирования, оценка обновлений для контрольные примеры и планы тестирования, документирование циклов тестирования, регрессионное тестирование и обновление соответственно;

7. Пост реализации. Совещание по оценке после реализации может быть проведено для обзора всего проекта. Действия на этом этапе - подготовка окончательного отчета о дефектах и связанных с ним метрик, определение стратегий для предотвращения подобных проблем в будущем проекте, группа автоматизации - обзор контрольных примеров для оценки других случаев, подлежащих автоматизации для регрессионного тестирования, очистка автоматизированных контрольных примеров переменные и рассмотреть процесс объединения результатов автоматического тестирования с результатами ручного тестирования.

### 2.2 Типы тестов Android-приложений

Тестирование приложений доступно в различных фреймворках с различными уровнями поддержки от Android SDK и выбранной IDE. На данный момент мы собираемся сосредоточиться на том, как тестировать приложения Android с помощью инструментальной среды тестирования Android, которая имеет полную поддержку SDK. Тестирование может быть осуществлено в любое время в процессе разработки, в зависимости от используемого метода тестирования.

Существует несколько типов тестов в зависимости от тестируемого кода. Независимо от его типа, тест должен проверять условие и возвращать результат этой оценки в виде одного логического значения, которое указывает на его успех или неудачу.

#### 2.2.1 Unit-тесты

Unit-тесты – это тесты, написанные программистами для других программистов, и они должны изолировать тестируемый компонент и иметь возможность его повторного тестирования. Вот почему юнит-тесты и mock-объекты обычно размещаются вместе. Mock-объекты используются чтобы изолировать устройство от его зависимостей, контролировать взаимодействия, а также чтобы иметь возможность повторить тест любое количество раз. Например,

если тест удаляет некоторые данные из базы данных, вероятно, не нужно, чтобы данные были фактически удалены и, следовательно, не найдены при следующем запуске теста. JUnit является стандартом де-факто для модульных тестов на Android. Это простая среда с открытым исходным кодом для автоматизации модульного тестирования, изначально написанная Эрихом Гаммой и Кентом Беком. Тестовые случаи Android используют JUnit 4.

Есть много причин, по которым unit-тесты должны быть приняты для любого проекта разработки программного обеспечения:

1. При внедрении unit-тестов уходит меньше времени на отладку программы, так как многие компоненты с уверенностью работают, даже после изменений. Это позволяет проводить более безопасный рефакторинг кода и добавлять новый функции. Без внедрения unit-тестов можно легко поломать системы при рефакторинге или при добавлении новых функций, потому что неизвестно, что может сломать в итоге;

2. Отладка становится быстрее. Без unit-тестирования, время для отладки неудачного функционального теста значительно увеличивается. Но с unit-тестами, сам тест становится минимальным, и причина сбоя может быть найдена быстрее;

3. Более качественное проектирование и документация. Написание тестов требует того, чтобы программа была спроектирована качественно. Набор тестов может помочь составить документацию для неявного поведения методов, тем самым давая возможность разработчикам узнать больше о предполагаемом поведении класса. Идеальное решение для обратной связи, когда все unit-тесты для программного обеспечения выполняются вместе, как единое целое, чтобы в целом оценить состоянии системы. Благодаря unit-тестам другие разработчики получают механизм оценки кодовой базы, то есть проверить насколько

отдельные части системы соответствуют их требованиям или находятся в стадии разработки. Изменения тестов порой могут вызвать проблемы в работе программного кода, подробные и частые отчеты о работе unit-тестов могут помочь найти проблему и указать состоянии системы.;

4. Набор инструментов unit-тестирования позволяет с большой уверенностью проводить реструктуризацию программного кода. Выполнение unit-тестов гарантирует разработчика, что их рефакторинг кода не повлиял на работу программного продукта и функциональность осталась прежней;
5. Уменьшение затрат в будущем. Многочисленные исследования доказали тот факт, что исправление ошибки, которую обнаружили после релиза программного продукта, значительно выше, чем в начале разработки. Качественное unit-тестирование позволит избежать появления ошибок в работе программы на ранних этапах разработки, и тем самым позволит уменьшить вероятность появления других неисправностей и снизить затраты на дальнейшее обслуживание.

2.2.2 Интеграционные тесты

Интеграционные тесты предназначены для тестирования совместной работы отдельных компонентов. Модули, которые были протестированы модульно независимо, теперь объединяются для проверки интеграции. Обычно для работы Android требуется некоторая интеграция с системной инфраструктурой. Им нужен жизненный цикл Activity, предоставляемый ActivityManager, и доступ к ресурсам, файловой системе и базам данных. Те же критерии применяются к другим компонентам Android, таким как Services или ContentProviders, которые должны взаимодействовать с другими частями системы для выполнения своих обязанностей. Во всех этих случаях существуют специальные тестовые классы, предоставляемые платформой тестирования Android, которая облегчает создание тестов для этих компонентов.

### 2.2.3 Тесты пользовательского интерфейса

Тесты пользовательского интерфейса проверяют визуальное представление приложения, например, как выглядит диалоговое окно или какие изменения в пользовательском интерфейсе, вносятся при закрытии диалогового окна. Особые соображения следует учитывать, если тесты программного обеспечения включают компоненты пользовательского интерфейса.

Только основной поток может изменять пользовательский интерфейс в Android. Таким образом, специальная аннотация @UIThreadTest используется для указания того, что конкретный тест должен быть запущен в этом потоке, и он будет иметь возможность изменять пользовательский интерфейс. С другой стороны, если необходимо запускать только части своего теста в потоке пользовательского интерфейса, можно использовать метод Activity.runOnUiThread (Runnable r), который предоставляет соответствующий Runnable, содержащий инструкции по тестированию.

Вспомогательный класс TouchUtils также предоставляется для помощи в создании теста пользовательского интерфейса, позволяя генерировать следующие события для отправки в представления, такие как:

1. Клик;
2. Перетаскивание;
3. Длинный клик;
4. Прокручивание списка;
5. Касание.

Таким образом можно удаленно управлять своим приложением из тестов. Кроме того, Android представила Espresso для инструментальных тестов пользовательского интерфейса. Unit - тесты, интеграционные тесты и тесты пользовательского интерфейса можно вынести с схему на рисунке 2.1.

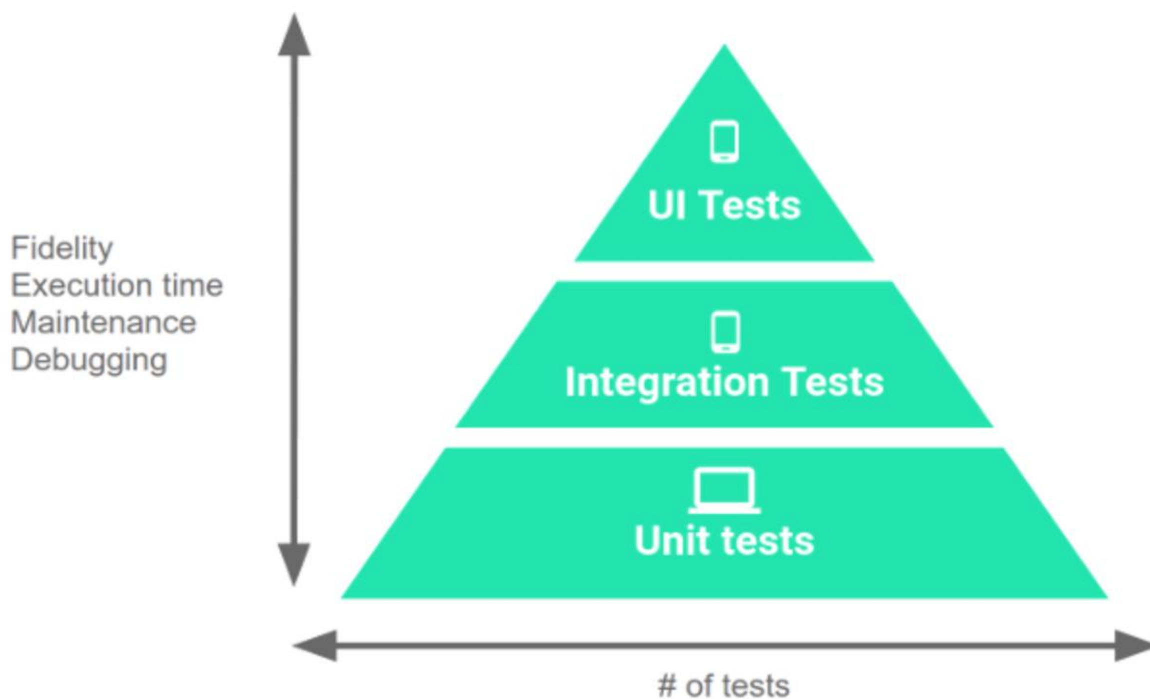


Рисунок 2.1 – Пирамида тестирования

По мере движения по пирамиде, от небольших тестов до больших тестов, каждый тест увеличивает точность, но также увеличивает время выполнения и усилия по обслуживанию и отладке. Поэтому необходимо написать больше unit-тестов, чем интеграционных тестов, и больше интеграционных тестов, чем тестов интерфейса. Хотя доля тестов для каждой категории может меняться зависимо от вариантов использования вашего приложения, мы обычно рекомендуем следующее разделение по категориям: 70 процентов – unit-тесты, 20 процентов – интеграционных и 10 процентов – тестов интерфейса.

## 2.3 JUnit-фреймфорк

JUnit – это набор инструментов для тестирования, разработанный для языка программирования Java с целью обеспечения удобного и простого способа для создания unit-тестов. Он играет решающую роль в разработке на основе тестирования и представляет собой семейство платформ unit – тестирования, известных под общим названием xUnit. JUnit помогает реализовать разработку через тестирование, для этого сначала настраиваются все тестовые данные для фрагмента кода, его сначала тестируют, потом реализовывают. Это помогает значительно повысить производительность разработчика и качество программного кода, это помогает сократить время на отладку кода и снижает загруженность программиста.

Преимущества JUnit:

1. JUnit – это набор инструментов для модульного тестирования. Все преимущества модульного тестирования применимы, потому что JUnit – отлично подходит для модульного тестирования;
2. JUnit достаточно прост: разработчик может за короткий промежуток времени научиться писать модульные тесты, и сама структура JUnit допускает разработчика написать свои собственные модульные тесты в течение пары минут;
3. Создание тестов с использованием JUnit обходится дешево: время обучения не занимает много времени и объем кода, который нужно написать для создания теста невелик;
4. Не нужно много разработчиков для написания теста, это позволяет снизить цену на каждый написанный тест;
5. JUnit тесты написаны на языке программирования Java, поэтому разработчика не нужно изучать дополнительный язык

программирования для написания модульных тестов, так же можно использовать язык программирования Kotlin;

6. JUnit является фреймворком с открытым исходным кодом, поэтому не требуется затрат на получение лицензии для его использования. Так же, открытый исходный код позволяет другим разработчикам вносить свои изменения во фреймворк и делать еще лучше. Имея доступ к исходному коду, разработчика предоставляется возможность настраивать платформу под свои нужны и специфичные задачи, и требования, которые могут потребоваться при разработке программного продукта.;
7. JUnit тестами не сложно управлять. JUnit предоставляет удобную работу с модульными тестами, разработчики могут сформировать иерархию в своих тестах, это позволяет всем разработчикам выполнять тесты в любом количестве и в любое время.

### 2.4 Espresso-фреймворк

В целом, автоматизация тестирования является сложной задачей. Наличие Android для различных устройств и платформ становится затруднительным для автоматизации тестирования. Чтобы сделать это проще, Google взял на себя задачу и разработал каркас Espresso. Он предоставляет очень простой, согласованный и гибкий API для создания автоматизированных тестов интефейса мобильного приложения. Эспрессо-тесты могут быть написаны как на Java, так и на Kotlin, современном языке программирования для разработки приложений для Android. API Espresso прост и легок в освоении. Можно легко выполнять тестирование пользовательского интерфейса Android без сложного многопоточного тестирования. Google Drive, Карты и некоторые другие приложения в настоящее время используют Espresso.

Вот некоторые характерные особенности, поддерживаемые Espresso:

1. Очень простой API;

2. Высоко масштабируемый и гибкий инструмент;
3. Предоставляет отдельный модуль для тестирования компонента Android WebView.
4. Предоставляет отдельный модуль для проверки, а также макет Android-содержимого.
5. Обеспечивает автоматическую синхронизацию между приложением и тестами.

Ниже приведены некоторые преимущества Espresso:

1. Обратная совместимость;
2. Прост в настройке;
3. Высокостабильный цикл тестирования;
4. Поддерживает тестирование вне приложения;
5. Поддерживает JUnit4;
6. Автоматизация пользовательского интерфейса подходит для написания тестов черного ящика.

Изначально среда тестирования эспрессо предоставляется как часть библиотеки поддержки Android. Позже, команда Android предоставляет новую библиотеку – AndroidX и переносит в библиотеку новейшую разработку фреймворка для Espresso. Последние разработки (Android 9.0, API уровня 28 или выше) фреймворка для Espresso будут выполнены в библиотеке AndroidX. Включить инфраструктуру тестирования эспрессо в проект так же просто, как установить среду тестирования эспрессо, как зависимость в файле gradle приложения app / build.gradle.

## 2.5 Разработка через тестирование

Разработка через тестирование TDD (Test driven development) является одной из распространенных практик разработки ядра Agile. Он основан на принципах Agile манифеста и экстремального программирования. В экстремальном программированияи есть два важных метода тестирования: разработка через тестирование (TDD) и приемочное тестирование. Это обеспечивает стабильность, чтобы повысить доверие разработчика и достичь высокого уровня охвата тестированием для слабо и сильно связанных систем. Это также мотивирует ясность относительно области развертывания. TDD является одним из методов разработки программного обеспечения, которое объединяет разработку программы и тестирование в серии микро-итераций. Поэтому он рассматривается как объединение тестовой первой разработки, в которой модульные тесты проводятся перед развертыванием кода. Рефакторинг также связан с этим процессом, и он играет важную роль в реструктуризации фрагмента кода. Кроме того, тесты должны быть успешными, чтобы уменьшить сложность и улучшить ее понятность, ремонтопригодность и ясность.

Разработка через тестирование предполагает, что разработчик напишет модульные тесты автоматизированные тесты со всеми требованиями к коду перед написанием самого программного кода. В тест входят проверки тех или иных условий, которые могут выполниться или могу не выполниться. Если они выполнились, значит тест удачно прошел. Прохождение теста означает, что поведение программного кода совпадает с ожиданием программиста. Для разработки таких тестов, используются различные библиотеки, некоторые из них приведены выше в данном разделе. В основном, модульные тесты покрывают самые важные и необходимые фрагменты кода. Например, код, который часто изменяется, или код, который обеспечивает работу очень важного компонента, или код с многочисленными зависимостями или код, от работы которого зависит значительная часть системы или другого кода.

IDE (интегрированная среда разработки) должна быстро дать обратную связь на небольшие изменения кода. Программа должна быть спроектирована таким образом, что компоненты должны иметь высокую внутреннюю связность и должны быть слабо связаны друг с другом, это намного упрощает написание тестов и сам процесс тестирования.

В TDD подразумевается проверять не только правильность и корректность программного кода, но и само проектирование архитектуры. Взглянув на сами тесты, разработчик может понять и догадаться какие функции и способы необходимо добавить еще в программный продукт.

Написание программного кода для тестов подразумевает использование всех стандартов, как и для программного кода.

Разработку через тестирование можно разделить на следующие этапы:

1. Добавление самого теста. Подход разработка через тестирование предполагает, что добавление какой-либо новой фичи в программный продукт всегда начнется с написанием для этой функциональности нового теста. Разумеется, сразу этот тест провалится так как код еще не был написан. А если тест прошел сразу, то может быть несолько причин, либо есть уже такая функция, либо тест неисправен. Для написание корректного теста разработчик должен четко понимать какие требование выставлены к новой функциональности. Можно рассмотреть некоторые пользовательские сценарии использования программы. Добавление новых функциональных возможностей может потребовать изменить тест. В этом и заключается различие стандартных подходов к тестированию от тестирования через разработку, заставляя программиста заранее задумать о всех требованиях.
2. Запуск тестов. Изначально нужно быть уверенным, что написанный тест не проходит. На этом этапе и некоторая проверка для самих тестов, если они все проходят, значит с ними не все в порядке и нужно их исправить.

Каждый написанный новый тест, должен обоснованно не проходить. Это внушает доверие, хоть и не на сто процентов, что написанный новый тест действительно работает так как нужно по определенный требованиям.

3. Написание программного кода. Необходимо написать программный код таким образом, чтобы тест прошел успешно. Он может быть сразу не качественно написать, пока это не важно. Главное в том, что он должен пройти тест, пусть и не очень красивым способом. В этом нет ничего страшного, в дальнейшем этот код измениться. Необходимо сосредоточиться на прохождении теста, не нужно писать код не причастный к тестированию.
4. Запуск тестов. Нужно удостовериться, что всех написанные тесты прошли. Если этот так, то разработчику гарантируется, что весь код работает, как и планировалась изначально, а потом можно начать следующий этап.
5. Рефакторинг. Если программный продукт выполняет всех требуемые функциональные возможности, можно отрефакторить сам код, то есть изменить сам код, но при этом никак не затронуть и не изменить функциональность программного продукта, так же это помогает сделать код более читабельным и сделать дальнейшее сопровождение более простым.
6. Заново выполнить весь цикл. При каждом добавлении новых возможностей и функций описанный цикл повторяется. Нужно делать небольшие итерации. Нужно постоянно держать новые и старые тесты в рабочем состоянии, в противном случае обратиться к отладке кода. Когда применяются различные сторонние библиотеки, не нужно тестировать саму библиотеку, тест должен относиться к вашему программному продукту.

На рисунке 2.2 изображено графическое представление цикла разработки, в виде блок-схемы.

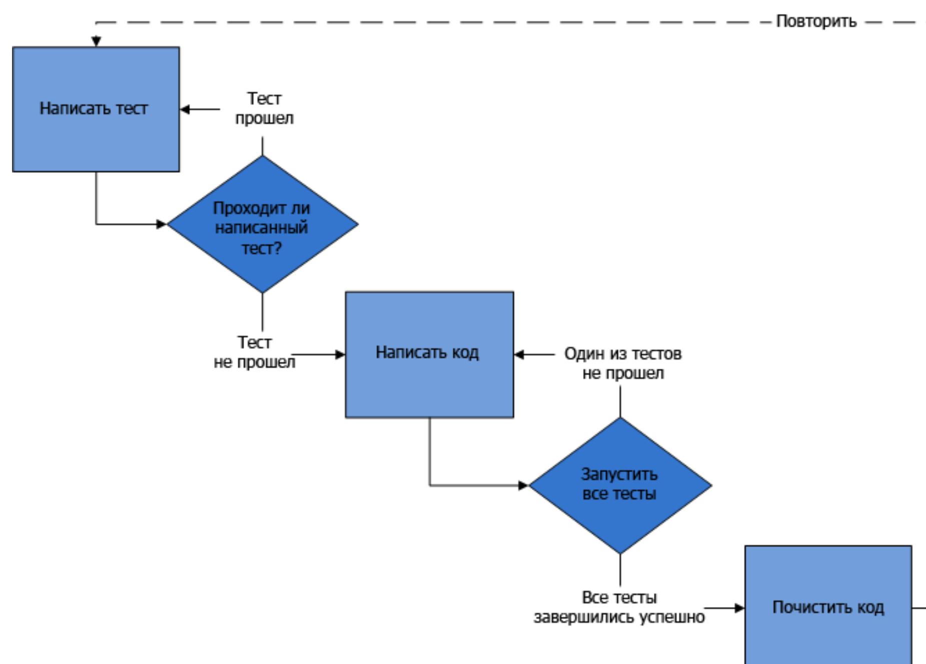


Рисунок 2.2 – Цикл разработки в виде блок-схемы

Все модульные тесты предназначены для тестирования отдельных компонентов системы и модулей. Количество написанных тестов не имеет значения. При написании тестов перед разработкой не нужно использовать возможность сетевого взаимодействия или низкоуровневые операции. Это может повлечь за собой большую трату времени, и программисты будет все чаще игнорировать запуск тестов. Включение зависимостей может превратить модельное тестирование в интеграционное, но, если какой-то модуль работает неисправно, может потребовать врем для поиска неисправного модуля.

Выводы по разделу два

В данном разделе были исследованы инструменты и методологии тестирования Android-приложений, рассмотрены самые популярные и рекомендуемые подходы к тестированию.

# 3 РАЗРАБОТКА И АВТОМАТИЗАЦИЯ ТЕСТОВЫХ СЦЕНАРИЕВ

## 3.1 Автоматизированное тестирование

Автоматизированное тестирование программного обеспечения – процесс, при котором проверяется работоспособность программы, а основные этапы тестирования, такие как запуск, проверка, выполнение, отчет о работе выполняются без участия человека, то есть автоматизировано. Тестировать программу можно на разных уровнях: код (юнит-тесты и интеграционное тестирование), API и GUI. Разные виды тестов лучше подходят в разных ситуациях. Рассмотрим подробнее виды автоматизированного тестирования.

Виды автоматизированного тестирования делятся на:

1. Unit-тестирование (или модульное):
   a. Что тестируется: правильность работы классов и методов;
   b. Кто тестирует: разработчик, написавший код;
   c. Зачем: чтобы отловить самые очевидные ошибки на стадии написания кода.
2. Тестирование API:
   a. Что тестируется: REST или SOAP сервисы;
   b. Кто тестирует: разработчики сервисов или специалисты по автоматизации тестирования;
   c. Зачем: для обнаружения ошибок на стадии тестирования компонентов.
3. UI-тестирование:
   a. Что тестируется: функционал графического интерфейса;

b. Кто тестирует: специалисты по автоматизации тестирования;

c. Зачем: для облегчения работы ручных тестировщиков и наиболее быстрого обнаружения ошибок на стадии приемочного тестирования.

Юнит-тесты предназначены для тестирования одного модуля кода в изолированном окружении. Если в коде используются другие сторонние классы, то на их место, ка правило, вставляют моки и стабы, код для тестрования не должен использовать сеть, файламы, базы данных.

Стабы – классы-заглушки, они просто возвращают какие-либо данные не выполняя действий. Например, стаб класса работы с базой данных может вместо реального обращения к базе данных возвращать, что запрос успешно выполнен. А при попытке прочитать что-то из нее возвращает заранее подготовленный массив с данными.

Моки – это классы-заглушки, которые используются чтобы проверить, что определенная функция была вызвана.

Обычно юнит-тест передает функции разные входные данные и проверяет, что она вернет ожидаемый результат. Например, если у нас есть функция проверки правильности номера телефона, мы даем ей заранее подготовленные номера, и проверяем что она определит их правильно. Если у нас есть функция решения квадратного уравнения, мы проверяем, что она возвращает правильные корни (для этого мы заранее делаем список уравнений с ответами).

Юнит-тесты хорошо тестируют такой код, который содержит какую-то логику. Если в коде мало логики, а в основном содержатся обращения к другим классам, то юнит-тесты написать может быть сложно.

API – это набор функций, которые можно вызывать, чтобы получить какие-то данные. Например, у сервиса Яндекс.Карты есть свой API геокодера. Отправив к нему запрос с географическим адресом, ты можешь получить координаты точки

(и наоборот), а у Центробанка есть API, которое возвращает официальный курс валют в заданный день. Если у любого приложения есть API, то можно тестировать его, посылая заранее подготовленные запросы и сравнивая пришедший ответ с ожидаемым. GUI – это графический интерфейс, то есть это то, что пользователь видит на экране. Это самый сложный для тестирования вид. Если речь идет о проверке работы приложения, то мы должны эмулировать работу браузера, который довольно сложно устроен, анализировать информацию, которая выводится на странице. Но этот вид тестирования очень важен, так как он взаимодействует с приложением так же, как и пользователь. GUI тесты еще называют End-toEnd (E2E) или приемочные (aceptance) тесты.

Сделаем промежуточный вывод о том, что необходимо автоматизировать:

1. Труднодоступные места в системе (backend-процессы, логирование файлов, запись в БД);
2. Часто используемая функциональность, риски от ошибок, в которой достаточно высоки. Автоматизировав проверку критической функциональности, можно гарантировать быстрое нахождение ошибок, а значит и быстрое их решение;
3. Рутинные операции, такие как переборы данных (формы с большим количеством вводимых полей;
4. Автоматизировать заполнение полей различными данными и их проверку после сохранения);
5. Валидационные сообщения (Автоматизировать заполнение полей некорректными данными и проверку на появление той или иной валидации);
6. Длинные End-to-End сценарии;
7. Проверка данных, требующих точных математических расчетов;
8. Проверка правильности поиска данных.

При тестировании не стоит впадать в крайности. Например, нельзя говорить, что 100% Android-приложения должно быть покрыто автотестами. Автоматизированные тесты должны прежде всего повышать качество кода, а также проверять работу наиболее критического функционала. На отладку и написание тестов уходит приличное количество времени. Стоит проанализировать все затраты и выяснить, на написание уйдет больше затрат, чем ожидаемая выгода, то стоит отказаться от тестов. При разработки небольшого приложения, которое потом не нужно поддерживать, то проще всего проверить его вручную, чем тратить время на написание автоматизированные скрипты. С другой стороны, если большая команда работает над сложным Android-приложением, то автотесты необходимы, иначе большую часть времени будет уходить на поиск с помощью ручного (функционального) тестирования и исправление сломанного функционала. Далеко не все разработчики, при выполнении и разработки сложный функций системы используют автоматизированное тестирование. Где-то программа проверяется людьми. Нельзя откидывать человеческий фактор. Люди могут устать, могут быть ленивы или невнимательны, в то время как автоматизированные тесты могут выполняться сколько угодно в любое время.

Необходимо помнить:

1. Автотесты должны быть повторяемыми.
2. Автотесты должны выполняться в контролируемом окружении;
3. Автоматизированные тесты не должны повторять алгоритм исходного кода, это может привести к одинаковой ошибке. Ошибочные результаты в этом случае совпадут.
4. Нужно тестить правильные и неправильные сценарии. Тестирую форму регистрации нужно проверить работоспособность не только с верными и правильными данными, но также вводить искаженные данные. При негативном сценарии должно появляться сообщение об ошибке;
5. При выполнении автотестов надо отслеживать возникающие ошибки. Если при возникновении ошибки просто выводится сообщение, которое

автотест не обрабатывает, и программа продолжает выполняться, то это бесполезный тест.

6. Автотесты должны быть легко запускаемые, в идеале одной командой. Если для его запуска необходимо выполнить много действий, то людям будет лень это делать. В компаниях обычно настраивают CI сервер, который сам забирает обновления с репозитория, запускает тесты и рассылает разработчикам сообщения при ошибках.

## 3.2 Создание тестовых случаев Android-приложения для ведения и управления задач

Тестовый случай (тест-кейс) – это описание проверки работы системы, которое может выполнить любой человек из команды, будь это специалист по тестированию, разработчик, аналитик или даже бизнес-заказчик. Тест-кейсы составляются специалистами по тестированию, если быть конкретнее, то тест-аналитиками или тест-дизайнерами. В современном мире нет однозначного мнения о необходимости выделения тест-аналитика в отдельную проектную единицу. Далеко не для всех очевидны различия функций тест-аналитика, тест-дизайнера и тестировщика. И если с обязанностями тестировщика все более ли менее понятно, то тест-дизайном и тест-анализом чаще всего занимаются одни и те же люди. Лишь в некоторых организациях эти роли четко разделены. Тест-аналитик (тест-дизайнер) – это член команды тестирования, основная задача которого определить, что и как нужно тестировать. Для этого необходимо выполнить следующие действия:

1. Исследовать продукт. Необходимо исследовать всю документацию, стараясь понять, какие бизнес-цели выполняет данный продукт и что хочет увидеть заказчик;
2. Составить логическую карту продукта при помощи майнд-карты – техники представления любого процесса, события в систематизированной визуальной форме;

3. Разбить программный продукт на составные части. Процесс построения майнд-карты неразрывно связан с выполнение декомпозиции продукта. Глубина декомпозии определяется удобством восприятия получаемой иерархической структуры. При декомпозиции следует сохранять баланс между полнатой и простотатой описания системы;
4. Описать тестовые случаи (тест-кейсы) для проверки работы функционала используя техники тест-дизайна (классы эквивалентности, граничные значения и др.);
5. Расставить приоритеты тестирования. Приоритеты необходимо выставлять, учитывая требования клиента, степень риска, сложность системы и временные ограничения;
6. Согласовать тест-кейсы с бизнес-заказчиком и системынми аналитиками.

Рассмотрим подробнее тестовые случаи. Тест-кейсы деляется по ожидаемому результату на:

1. Позитивные тесты. Кейсы используют только корректные данные и проверяют, что программное обеспечение правильно выполнило вызываемую функцию;
2. Негативные тесты. Кейсы оперируют как корректными, так и некорректными данными (хотябы 1 некорректный параметр) и ставит целью проверку исключительных ситуаций (срабатывание валидаторов), а также проверяет, что вызываемая приложением функция не выполняется при срабатывании валидатора.

Каждый тест-кейс может иметь 3 части:

1. Предусловие. Список действий, при выполнении которых система приводится к пригодному состоянию для проведения основных шагов проверки. Или это может быть список условий, при выполнении

которых система находится в состоянии, пригодном для проведения основного теста;

2. Шаги теста. Список действий, который переводит систему из одного состояния в другое, с целью для получения результата. На основании результата можно сделать вывод об удовлетворении реализации поставленным требованиям;
3. Постусловия. Список действий, который возвращает тестируемую систему в исходное состояние до проведения теста. Постусловия не является обязательной частью, но считается правилом хорошего тона. Данная часть может быть актуальна при автоматизированном тестировании, когда за один автотест можно наполнить базу данных большим количеством некорректных документов.

В рамках разработки Android-приложения для управления и ведения задач было необходимо создать тестовые сценарии и реализовать их. С учетом функциональности мобильного приложения было разработано множество тестовых сценариев, некоторые из них приведены в таблице 1.

Таблица 1 – Список некоторых тестовых сценариев

| Название теста | Номер шага | Шаги | Ожидаемый результат |
|---|---|---|---|
| Создание задачи | 1 | Открыть вкладку «Создание задачи» | Открывается вкладка «Создание задачи» с пустыми полями для заполнения |

Продолжение таблицы 1

| | | | |
|---|---|---|---|
| | 2 | Заполнить все поля | Поля заполнены без деформации данных |
| | 3 | Нажать кнопку «Сохранить» | Задача создана |
| | 4 | Открыть созданную задачу | Все данные отображаются корректно |
| Удаление задачи | | | |
| | 1 | Перейти в окно подробной информации о задаче | Отображается информация о задаче |
| | 2 | В меню найти кнопку «удалить» и нажать на нее | Задача полностью удаляется |
| | 3 | Перезапустить приложение и посмотреть список задач | Удаленной задачи нет в списке |

Таким образом были созданы сценарии для проверки всего функционала приложения, по этим сценариям были написаны автоматизированные тест-кейсы, подробнее о них будет рассказано в следующем разделе.

## 3.3 Создание автоматизированных сценариев тестирования

Для достижения наиболее качественно тестирования необходимо автоматизировать все созданные тестовые сценарии при помощи инструментов, описанных во втором разделе и при помощи среды разработки Android-приложений Android Studio. Android Studio – это официальная интегрированная среда разработки (IDE) для операционной системы Google Android [3], созданная на основе программного обеспечения JetBrains IntelliJ IDEA и разработанная специально для разработки под Android. Он доступен для загрузки в операционных системах Windows, macOS и Linux. Он заменяет Eclipse Android Development Tools (ADT) в качестве основной IDE для разработки собственных приложений Android. Android Studio была анонсирована 16 мая 2013 года на Google I / Oconference. Он находился на ранней стадии предварительного просмотра, начиная с версии 0.1 в мае 2013 года, затем вступил в бета-версию, начиная с версии 0.8, выпущенной в июне 2014 года. Первая стабильная сборка была выпущена в декабре 2014 года, начиная с версии 1.0.

Текущая версия предоставляет следующие функции:

1. Поддержка сборки на основе Gradle;
2. Рефакторинг;
3. Инструменты Lint для отслеживания производительности, удобства использования, совместимости версий и других проблем;
4. Возможности интеграции ProGuard и приложения для создания подписей на основе шаблонов для создания общих дизайнов и компонентов Android;
5. Богатый редактор макетов, который позволяет пользователям перетаскивать компоненты пользовательского интерфейса, возможность

предварительного просмотра макетов на нескольких конфигурациях экрана;

6. Поддержка создания приложений Android Wear;
7. Встроенная поддержка Google Cloud Platform для интеграции с Firebase Cloud Messaging (ранее Google Cloud Messaging) и Google App Engine;
8. Android Virtual Device (эмулятор) для запуска и отладки приложений в студии Android.

Android Studio поддерживает все те же языки программирования IntelliJ (и CLion), например Java, C ++ и другие с расширениями, такими как Go; и Android Studio 3.0 или более поздней версии поддерживают Kotlin и «все функции языка Java 7 и подмножество функций языка Java 8, которые зависят от платформы». Внешние проекты поддерживают некоторые функции Java 9. Хотя IntelliJ утверждает, что Android Studio поддерживает все выпущенные версии Java и Java 12, неясно, на каком уровне Android Studio поддерживает версии Java вплоть до Java 12 (в документации упоминается частичная поддержка Java 8). По крайней мере, некоторые новые языковые функции до Java 12 могут быть использованы на Android. На рисунке 3.1 представлен интерфейс Android Studio.

Некоторые особенности Android Studio 4.0 включают в себя новый редактор Motion Editor, Build Analyzer для расследования причин более медленных сборок и языковые API-интерфейсы Java 8, которые вы можете использовать независимо от минимального уровня API вашего приложения. Так же был переработан пользовательский интерфейс CPU Profiler, чтобы обеспечить более интуитивный рабочий процесс и упростить параллельный анализ активности потоков. А улучшенный Layout Inspector теперь предоставляет оперативные данные пользовательского интерфейса приложения, поэтому можно легко отлаживать именно то, что отображается на устройстве.

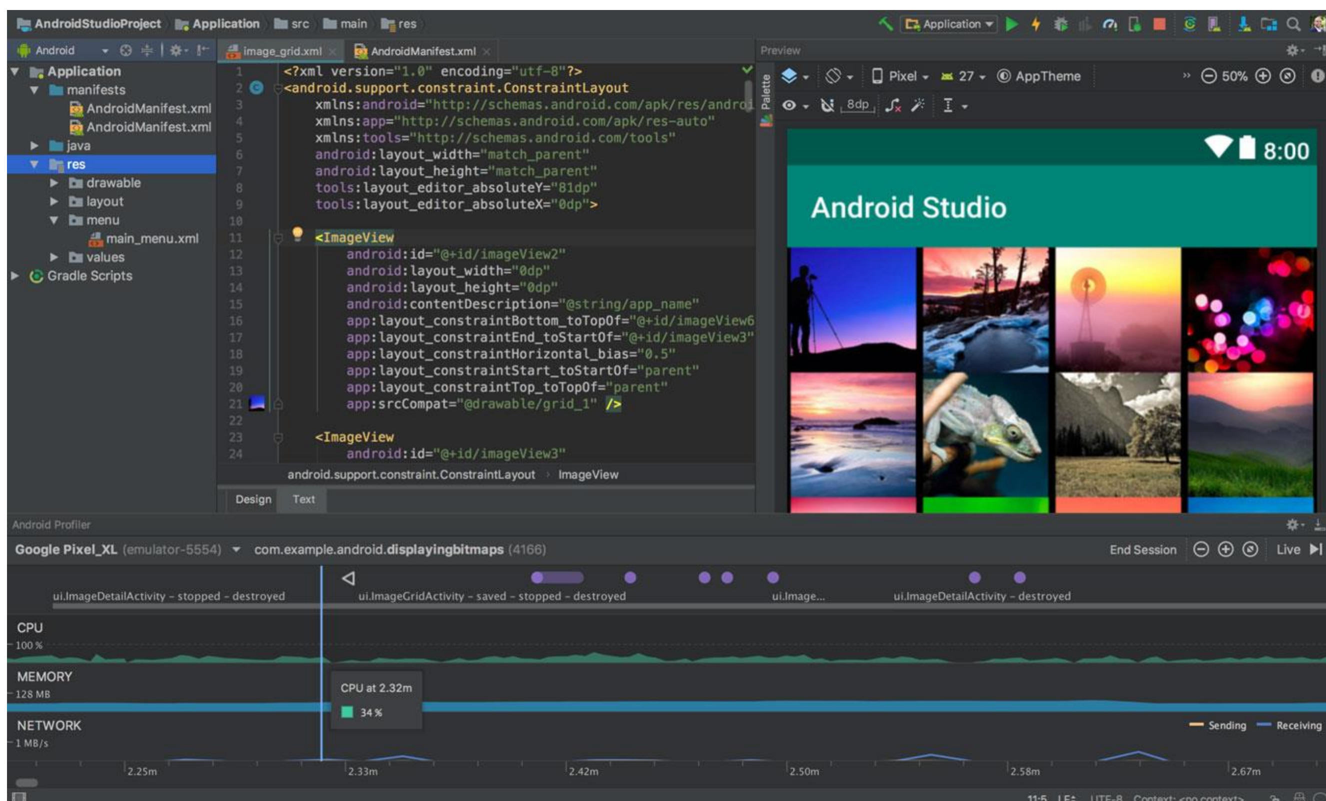


Рисунок 3.1 – Интерфейс Android Studio

Помимо IDE в современной разработке, а также при написании автоматизированных сценариев, используются системы контроля версий. Применяются различные системы, для контроля версий программного продукта. Такие системы обеспечивают контроль за изменением файлов программной системы. Системы хранит в себе различные версии всех документов программного продукта и если понадобиться вернуться к какой-либо версии документа или программы в целом, то это без труда можно сделать и можно посмотреть какой разработчик внес те или иные изменения в документ.

Данные системы наиболее широко распространены в разработке программного обеспечения для хранения исходных кодов разрабатываемой программы. Однако они могут с успехом применяться и в других областях, в которых ведётся работа с большим количеством непрерывно изменяющихся электронных документов. В частности, системы контроля версиями применяются в САПР, обычно в составе систем управления данными об изделии (PDM). Управление версиями используется в инструментах конфигурационного

управления (Software Configuration Management Tools). Одними из самых популярных систем контроля версиями являются Git. Программа является свободно распространяемой и выпущена под лицензией GNU GPLv2. Система спроектирована как набор программ, специально разработанных с учётом их использования в сценариях. Это позволяет удобно создавать специализированные системы контроля версий на базе Git или пользовательские интерфейсы. Git поддерживает быстрое разделение и слияние версий, включает инструменты для визуализации и навигации по нелинейной истории разработки. Git предоставляет каждому разработчику или специалисту по автоматизированному тестированию локальную копию всей истории разработки, изменения копируются из одного репозитория в другой. Удалённый доступ к репозиториям Git обеспечивается git-daemon, SSH- или HTTP-сервером. TCP-сервис git-daemon входит в дистрибутив Git и является наряду с SSH наиболее распространённым и надёжным методом доступа. Метод доступа по HTTP, несмотря на ряд ограничений, очень популярен в контролируемых сетях, потому что позволяет использовать существующие конфигурации сетевых фильтров. Пример визуализации git-веток представлен на рисунке 3.2.

Основной репозиторий системы – GitHub. Git и GitHub используют более 1.8 миллионов предприятий и организаций (по данным GitHub на март 2018 года). Среди них самые известные: IBM, Google, PayPal, Facebook, Spotify, Bloomberg и другие компании из различных социальных сфер. С января 2019 года GitHub предлагает неограниченное количество частных репозиториев для всех планов, включая бесплатные аккаунты. По состоянию на январь 2020 года GitHub сообщает, что у него более 40 миллионов пользователей и более 100 миллионов репозиториев (в том числе не менее 28 миллионов общедоступных репозиториев), что делает его крупнейшим хостом исходного кода в мире.

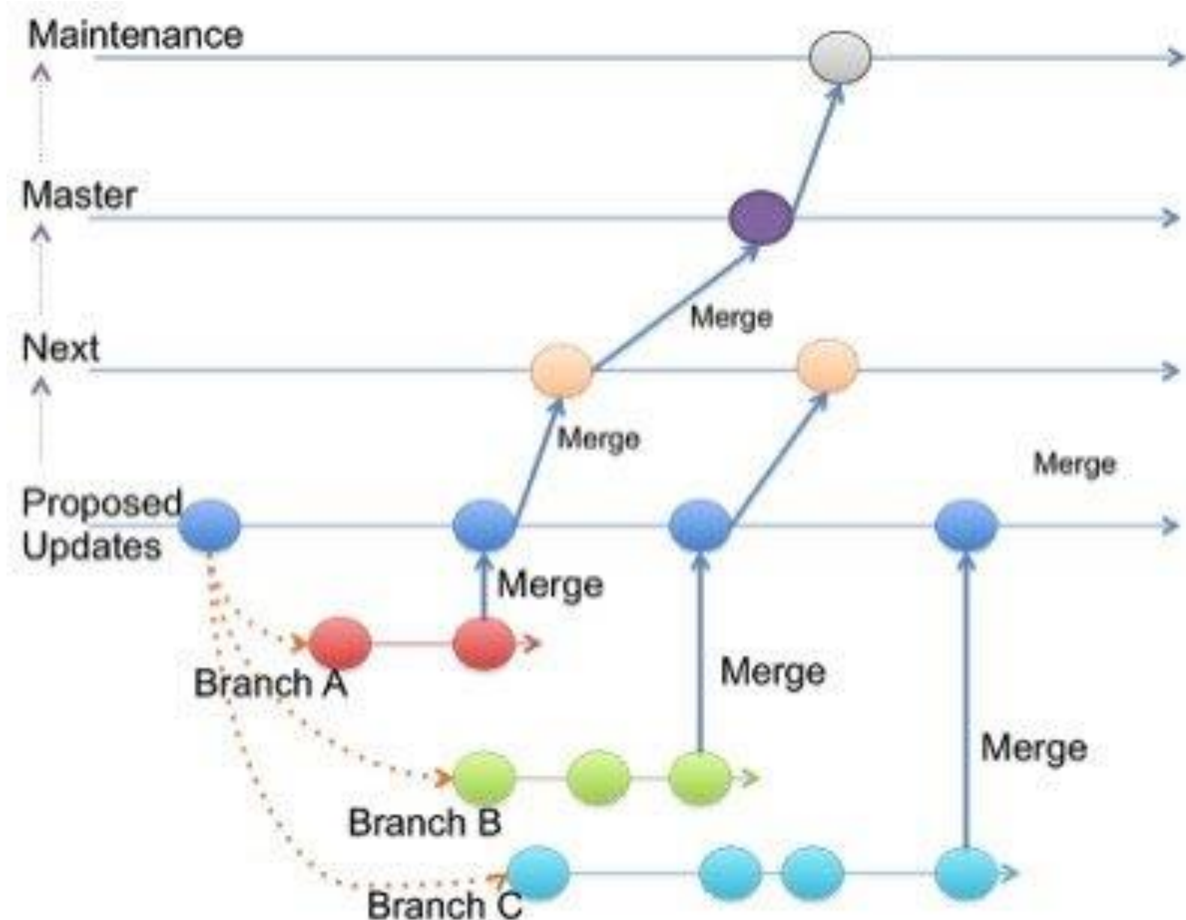


Рисунок 3.2 – Пример визуализации git-веток

Незаменимой компонентой для модульного тестирования программы является библиотека JUnit. JUnit помогает в тестировании отдельных участков кода, например, методов или классов. Опыт, полученный при работе с JUnit, важен в разработке концепций тестирования программного обеспечения. JUnit позволяет в любой момент быстро убедиться в работоспособности кода. Если программа не является совсем простой и включает множество классов и методов, то для её проверки может потребоваться значительное время. Естественно, что данный процесс лучше автоматизировать. Использование JUnit позволяет проверить код программы без значительных усилий и не занимает много времени.

Юнит тесты классов и функций являются своего рода документацией к тому, что ожидается в результате их выполнения. И не просто документацией, а

документацией, которая может автоматически проверять код на соответствие предъявленным функциям. На рисунке 3.3 представлен пример юнит-теста.

```
// For each unit test you write,
// answer these questions:
@Test public void whatComponentAspectAreYouTesting_whatShouldTheFeatureDo()
    throws Exception {
  final Object actual = "What is the actual output?";
  final Object expected = "What is the expected output?";

  assertThat(actual).isEqualTo(expected);
}
```

Рисунок 3.3 – Пример юнит-теста в Android Studio

Espresso – это среда тестирования для Android, которая позволяет легко создавать надежные тесты пользовательского интерфейса непосредственно в интегрированной среде разработки (IDE) Android Studio. Espresso ориентирован на разработчиков, которые понимают, что автоматизированное тестирование является неотъемлемой частью жизненного цикла разработки. Понимание того, как использовать Espresso, быстро становится необходимым навыком для разработчиков, особенно для тех, кто занимается разработкой мобильных приложений.

Одной из самых полезных функций тестирования Android-приложений является Espresso Test Recorder. Espresso Test Recorder позволяет разработчикам создавать тесты Espresso, автоматически записывая взаимодействия, выполняемые на эмуляторе устройства. Например, разработчики могут выбирать элементы пользовательского интерфейса, такие как текстовые поля и раскрывающиеся списки, а также опции и флажки, в которые вводятся данные с помощью касания экрана клавиатуры. Вся активность записывается регистратором тестов. После записи разработчик может использовать сценарий, являющийся результатом записи, в качестве шаблона для построения более комплексных тестов.

На рисунке 3.4 приведен фрагмент кода теста пользовательского интерфейса, созданного с помощью Espresso Test Recorder. Этот тестовый код облегчает нажатие кнопки и подтверждает, что текстовое поле отображает текст.

```
@Test
public void simpleButtonClickTest() {
    ViewInteraction appCompatButton = onView(
            allOf(withId(R.id.button), withText("Get Saying"), isDisplayed()));
    appCompatButton.perform(click());

    ViewInteraction viewGroup = onView(
            allOf(childAtPosition(childAtPosition(withId(android.R.id.content),
                    0),1), isDisplayed()));
    viewGroup.check(matches(isDisplayed()));

}
```

Рисунок 3.4 – Фрагмент кода теста пользовательского интерфейса

Получить доступ к Recorder можно из пункта меню Run в меню Android Studio, как показано ниже на рисунке 3.5.

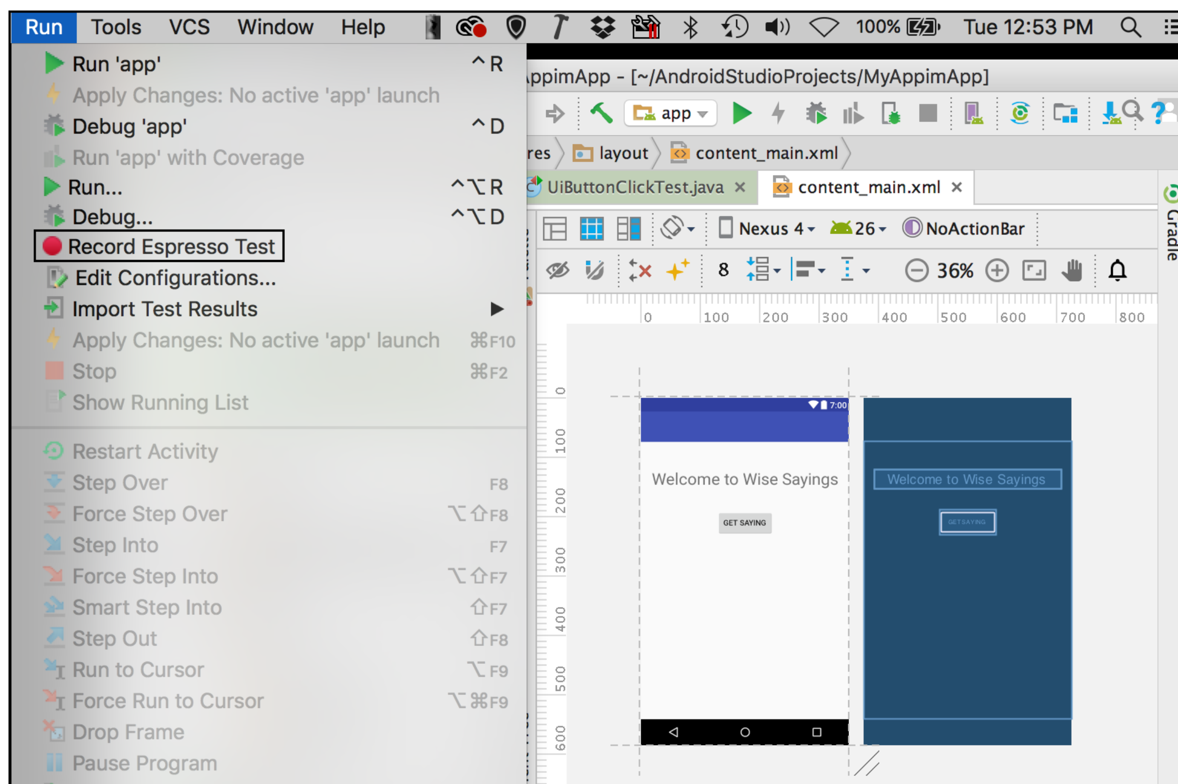


Рисунок 3.5 – Расположение Espresso Test Recorder в Android Studio

Test Recorder запустит диалоговое окно, в котором можно выбрать эмулятор устройства для использования. Например, Test Recorder запустит диалоговое окно, в котором можно выбрать эмулятор устройства для использования. Можно выбрать общий эмулятор x86 или ARM. Или можно выбрать более специфичный эмулятор, такой как устройство Galaxy или Pixel. Конечно, разработчик всегда может подключить настоящее мобильное устройство к своей машине для разработки и протестировать его на реальном оборудовании. Затем разработчик приступает к созданию жестов и вводу данных, которые имеют отношение к объему проводимого тестирования. Регистратор позволяет разработчику добавлять утверждение после каждого события ввода данных. Возможность добавлять утверждения как часть сеансов записи экономит значительное время на создание теста.

После запуска теста Android Studio отображает результаты в панели «Выполнить» в левом нижнем углу среды IDE, как показано ниже на рисунке 3.5. Кроме того, если вы запустили тесты с покрытием, отчет о покрытии будет доступен на панели в верхний правый из IDE.

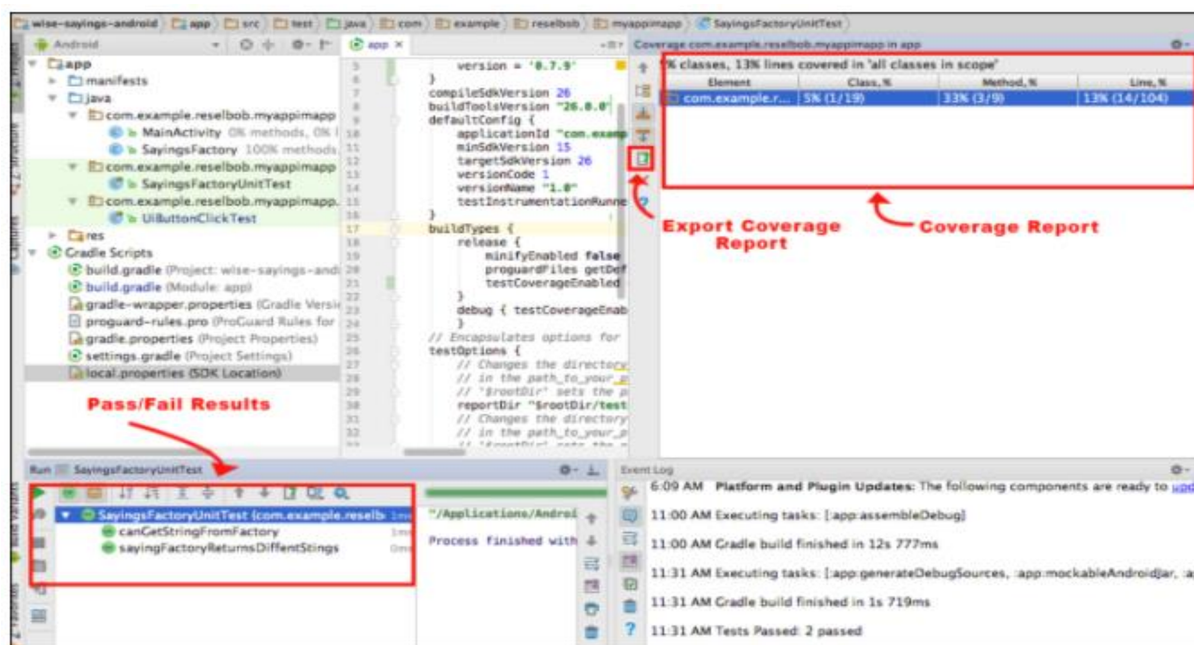


Рисунок 3.5 – Информацию о пройденных/неудачных тестах и покрытиях в среде IDE

Помимо возможности просмотра отчета о покрытии из среды IDE, вы можете экспортировать отчет о покрытии в набор HTML-документов, которые позволяют детализировать проверку. Вы экспортируете отчет о покрытии, нажимая кнопку «Экспорт» в левой части панели покрытия, как показано на рисунке 3.6. Вам будет предложено указать место для сохранения набора файлов HTML. После сохранения отчета о покрытии вы можете просмотреть его в своем браузере.

[ all classes ] [ com.example.reselbob.myappimapp ]

Coverage Summary for Class: SayingsFactory (com.example.reselbob.myappimapp)

| Class | Class, % | Method, % | Line, % |
|---|---|---|---|
| SayingsFactory | 100% (1/ 1) | 100% (3/ 3) | 100% (14/ 14) |

```
 1  package com.example.reselbob.myappimapp;
 2
 3  import java.util.ArrayList;
 4  import java.util.List;
 5
 6  public class SayingsFactory {
 7      private List<String> list = new ArrayList<String>();
 8      private static SayingsFactory ref;
 9      private int currentIndex = 0;
10      private SayingsFactory()
11      {
12          list.add("Be Kind To Strangers");
13          list.add("Always Be Honest");
14          list.add("The Truth is the Best");
15          list.add("Tip Well Always");
16      }
17
18      public static SayingsFactory getInstance()
19      {
20          if (ref == null) ref = new SayingsFactory();
21          return ref;
22      }
23
24      public String getNextSaying(){
25          if(currentIndex == list.size()) currentIndex = 0;
26          String rtn = list.get(currentIndex);
27          currentIndex++;
28          return rtn;
29      }
30  }
```

Рисунок 3.6 – Отчет о тестирование в виде HTML – документа

Отчет об охвате, показанный на рисунке 3.6, показывает, что все строки кода, написанные в классе, SayingsFactory, были выполнены модульным тестом в Android Studio. 100% тестовое покрытие для класса отлично. Однако из-за того, что один класс имеет 100-процентное покрытие, это не обязательно означает, что вся кодовая база хорошо реализована. Цель состоит в том, чтобы иметь высокое число покрытия, максимально близкое к 100% для всей кодовой базы. Таким образом,

разработчикам и тестировщикам необходимо спроектировать реализацию тестирования, которая обеспечивает высокий процент охвата всей базы кода.

### 3.4 Анализ результатов тестирования

Ошибки и сбои, полученные во время тестирования, чаще всего появляются из-за некоторых дефектов и проблем в рассматриваемом программного коде и системе. На появление сбоев может повлиять и тестовое окружение или работа операционной системы. Все такие дефекты должны тщательно анализироваться, что бы можно было с легкостью обнаружить причину и место появление данного дефекта. Это может быть плохое формирование требований или технического задания к разрабатываемой системе, ошибочный дизайн или проблемы на низком уровне. Все результаты тестирования и аналитические данные могут быть использованы для улучшения процесса тестирования и на сколько такие изменения и улучшения важны.

Оценка результатов тестирования формируется на основании сбора статистики по конкретным метрикам. Выбор метрик осуществляется после определения целей проекта применительно к ожиданиям от тестирования и выявления главной проблемы проекта на текущий момент. Несомненно, ожидания формируются каждым участником цикла разработки: менеджером проекта, разработчиками, аналитиками, специалистами службы поддержки. Например, если есть жалобы на пропускаемые дефекты, то необходимо их фиксировать для анализа причин пропуска. Допустим, после пары итераций становится понятно, что причина в недостаточном покрытии функционала тестами. В этом случае необходимо начать фиксировать процент покрытия функционала тестами. Если оказывается, что у нас недостаточно тестов, то необходимо разобраться, почему так произошло. Причины могут быть разными: от халатности специалистов тестирования до того, что сроки на подготовку и тестирование слишком сжатые, и специалисты не успевают писать тесты в требуемом для полного покрытия объеме. Оценка сроков может быть адекватная, только вот передача версии в тестирование

происходит позже запланированной даты, а сроки релиза отодвигать нельзя. Возможен вариант, что разработчики не успевают – их оценка на разработку всегда оказывается на 2-3 дня меньше фактического времени на реализацию. Возможно, менеджер даёт разработчикам незапланированные задачи в середине итерации. Само собой, решение этой проблемы не ляжет на плечи отдела тестирования, а вот менеджеру проекта будет гораздо проще объяснить клиенту важность стабилизации задач версии, имея красивые таблицы, статистики и графики влияния незапланированных задач на качество продукта на выходе. Заказчик и менеджер в этом случае могут прийти к решению: либо стоит развивать продукт итеративно, стараясь соблюдать поставленные сроки и тогда придется умерить желания заказчика, либо любое изменение в составе версии будет сопровождаться повторной оценкой и откладыванием срока релиза.

Самое важное правило внедрения улучшений: сначала формулируется цель и выявляются проблемы в ее достижении, а затем решаем, что нужно замерять для оценки изменений. И, конечно, без сплоченности всех членов команды и без общей заинтересованности в успехе проекта сбор метрик совершенно бесполезен – собранные результаты не получится оценить, чтобы использовать их в дальнейшем.

Проанализируем преимущества и недостатки автоматизированного тестирования по сравнению с ручным (функциональным):

1. Скорость выполнения тест-кейсов может в разы и на порядки превосходить возможности человека. Если представить, что человеку придётся вручную сверять несколько файлов размером в несколько десятков мегабайт каждый, оценка времени ручного выполнения становится пугающей: месяцы или даже годы. При этом 36 проверок, реализуемых в рамках дымового тестирования командными скриптами, выполняются менее чем за пять секунд и требуют от специалиста по

автоматизированному тестированию только одного действия – запустить скрипт;

2. Отсутствует влияние человеческого фактора в процессе выполнения тест-кейсов (усталости, невнимательности и т.д.) Какова вероятность, что человек ошибётся, сравнивая посимвольно даже два обычных текста размером в 100 страниц каждый? А если таких текстов 20? И проверки нужно повторять раз за разом? Можно смело утверждать, что человек ошибётся гарантированно. Автоматика не ошибётся;
3. Средства автоматизации способны выполнить тест-кейсы, в принципе непосильные для человека в силу своей сложности, скорости или иных факторов;
4. Средства автоматизации способны собирать, сохранять, анализировать, агрегировать и представлять в удобной для восприятия человеком форме колоссальные объёмы данных. Журналы работы систем автоматизированного тестирования могут занимать десятки гигабайт по каждой итерации. Логично, что человек не в состоянии вручную проанализировать такие объёмы данных, но правильно настроенная среда автоматизации сделает это сама, предоставив на выход аккуратные отчёты в 2-3 страницы, удобные графики и таблицы, а также возможность погружаться в детали, переходя от агрегированных данных к подробностям, если в этом возникнет необходимость;
5. Средства автоматизации способны выполнять низкоуровневые действия с приложением, операционной системой, каналами передачи данных и т.д. Задача сбора информации об используемых приложением ресурсах является классическим примером. Однако средства автоматизации могут не только собирать подобную информацию, но и воздействовать на среду исполнения приложения или само приложение, эмулируя типичные события (например, нехватку оперативной памяти или процессорного времени) и фиксируя реакцию приложения.

Итак, с использованием автоматизации мы получаем возможность увеличить тестовое покрытие за счёт:

1. Выполнения тест-кейсов, о которых раньше не стоило и думать;
2. Многократного повторения тест-кейсов с разными входными данными;
3. Высвобождения времени на создание новых тест-кейсов.

Однако, с внедрением автоматизации тестирования связана серия серьёзных недостатков и рисков:

1. Необходимость наличия высококвалифицированного персонала в силу того факта, что автоматизация – это проект со своими требованиями, планами, кодом и т.д. Техническая квалификация сотрудников, занимающихся автоматизацией, как правило, должна быть ощутимо выше, чем у их коллег, занимающихся ручным (функциональным) тестированием;
2. Разработка и сопровождение как самих автоматизированных тесткейсов, так и всей необходимой инфраструктуры занимает очень много времени. Ситуация усугубляется тем, что в некоторых случаях (при серьёзных изменениях в проекте или в случае ошибок в стратегии) всю соответствующую работу приходится выполнять заново с нуля: в случае ощутимого изменения требований, смены технологического домена, переработки интерфейсов (как пользовательских, так и программных) многие тест-кейсы становятся безнадёжно устаревшими и требуют создания заново;
3. Автоматизация требует более тщательного планирования и управления рисками, т.к. в противном случае проекту может быть нанесён серьёзный ущерб;
4. Коммерческие средства автоматизации стоят ощутимо дорого, а имеющиеся бесплатные аналоги не всегда позволяют эффективно решать поставленные задачи. И здесь снова необходимо вернуться к вопросу ошибок в планировании: если изначально набор технологий и

средств автоматизации был выбран неверно, придётся не только переделывать всю работу, но и покупать новые средства автоматизации.

5. Средств автоматизации крайне много, что усложняет проблему выбора того или иного средства, затрудняет планирование и определение стратегии тестирования, может повлечь за собой дополнительные временные и финансовые затраты, а также необходимость обучения персонала или найма соответствующих специалистов.

Автоматизированное тестирование требует больших инвестиций и может сильно повысить проектные риски, поэтому существуют специальные подходы по оценке эффективности и применимости к проекту автоматизированного тестирования. Если выразить всю их суть очень кратко, то в первую очередь следует учесть:

1. Затраты времени на выполнение тест-кейсов вручную и на выполнение этих же самых проверок, только уже автоматизированных. Чем больше разница, тем более выгодной представляется автоматизация;
2. Количество повторений выполнения одних и тех же тест-кейсов. Чем больше повторений, тем больше есть возможность сэкономить ресурсов за счёт средств автоматизации;
3. Затраты времени на обновление, отладку, а также поддержку автоматизированных сценариев. Данный параметр сложнее всего оценить. Именно он может представить наибольшую угрозу успеху автоматизации. Потому для проведения оценки следует привлекать наиболее опытных специалистов;
4. Наличие в команде соответствующих специалистов и их рабочую загрузку. Обычно написанием автоматизированных сценариев занимаются более квалифицированные сотрудники, которые в это время не могут решать иные задачи.

Необходимо понимать, что положительный эффект от автоматизации наступает не всегда и не сразу. Автоматизация, как и любой дорогостоящий инструмент, при правильном применении может принести ощутимую выгоду, однако при неверном применении принесёт только большие затраты.

Выводы по разделу три

В данном разделе были описаны основные понятия, связанные с автоматизированным тестированием. Одновременно с этим, были описаны подходы к написанию автоматизированных тестовых сценариев и рассмотрены инструменты, посредством которых были решены практические задачи. Оценка результатов тестирования формируется на основании ожидаемых результатов работы системы и сбора статистики по заведённым ошибкам. На продуктовую среду не должны попасть блокирующие и критические ошибки в системе, иначе компании рискуют потерять клиентов и прибыль. Целесообразность применения автоматизированного тестирования оценивается заранее, так как в некоторых случаях эффект от функционального (ручного) тестирования может быть выше.

# ЗАКЛЮЧЕНИЕ

Мобильные приложения в современном мире доминируют на мировом рынке с точки зрения пользователей, разработчиков, выпусков приложений и загрузок, соответственно качество приложений является растущей и значительной проблемой.

В данной выпускной квалификационной работе была исследована архитектура операционной системы Android, проведен анализ компонентов и функций Android-приложений, выяснилось, что Android использует новые функции программирования, которые никогда ранее не использовались традиционным программным обеспечением. Эти уникальные характеристики приложений Android приводят к новым типам ошибок, которые обычно не выявляются существующими методами тестирования программного обеспечения. Следовательно, нужны специализированные инструменты и подходы для тестирования Android-приложений.

Также были исследованы инструменты и методологии тестирования Android-приложений, рассмотрены самые популярные и рекомендуемые подходы к тестированию.

Кроме того, были описаны основные понятия, связанные с автоматизированным тестированием. Одновременно с этим, были описаны подходы к написанию автоматизированных тестовых сценариев и рассмотрены инструменты, посредством которых были решены практические задачи.

В результате выполнения всех поставленных задач было улучшено качество Android-приложения, были своевременно обнаружены и устранены ошибки в работе программы, что позволило сэкономить затраты на дальнейшее развитие программного продукта.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1 Android Repository [Электронный ресурс]. – Режим доступа: http://android.git.kernel.org

2 Open Handset Alliance website [Электронный ресурс]. – Режим доступа: http://www.openhandsetalliance.com

3 Android developers guide [Электронный ресурс]. – Режим доступа: https://developer.android.com/

4 Джонсон С, Настройка производительности для серверов Linux. – М.: IBM Press, 2005. – 576 с.

5 Лианг В, Системная интеграция для операционной системы Android. – National Taipei University, 2010. – 16 с.

6 Android Software Development Kit [Электронный ресурс]. – Режим доступа: http://developer.android.com/sdk

7 Виртуальная машина Davlik [Электронный ресурс]. – Режим доступа: https://sites.google.com/site/io/

8 Хегер Д, Количественная оценка стабильности. Publisher, 2010. – 222 с.

9 ZDNet [Электронный ресурс]. – Режим доступа: www.zdnet.com .

10 Р. Минелли и М. Ланца. Аналитика программного обеспечения для мобильных приложений - понимание и извлеченные уроки. – CSMR, 2013. – 153 с.

11 Д. Амальфитано, А. Р. Фасолино, П. Трамонтана, Б. Д. Та и А. М. Мемон, Автоматизированное тестирование мобильных приложений на основе моделей. IEEE Software, 2015 г. – 53 с.

12 Уиттакер, Д. Как тестируют в Google / Д. Уиттакер, Д. Арбон, Д. Каролло. – СПб: Питер, 2014. – 320 с.

13 Устранение распространенных проблем жизненного цикла Android в играх [Электронный ресурс]. – Режим доступа: https://developer.nvidia.com/fixing-common-android-lifecycle-questions-games

14 Android ошибка, вызывающая сбой при изменении ориентации обходной путь [Электронный ресурс]. – Режим доступа: http://www.jayway.com/2015/02/03

15 Рис, Э. Бизнес с нуля: Метод Lean Startup для быстрого тестирования идей и выбора бизнес-модели / Эрик Рис: пер. с англ. – Альпина Паблишер, 2013. – 269 с.

16 Джез, Хамбл Непрерывное развертывание ПО. Автоматизация процессов сборки, тестирования и внедрения новых версий программ / Хамбл Джез. - М.: Диалектика / Вильямс, 2016. – 933 с.

17 Кристиан, Д. Бакли ClearCase. Искусство внедрения. Секреты успешной реализации / Кристиан Д. Бакли, Даррен Палсифер. - Москва: ИЛ, 2008. - 288 с.

18 Савин, Р. Тестирование Дот Ком / Роман Савин – 2-е изд.; Дело, 2007. – 312 с.

19 Фаулер, Мартин Рефакторинг. Улучшение существующего кода / Мартин Фаулер. - М.: Символ-плюс, 2008. – 432 с.

20 Гленфорд, Майерс Искусство тестирования программ / Майерс Гленфорд. - М.: Диалектика / Вильямс, 2015. – 618 с.

21 Кон, Майк Пользовательские истории. Гибкая разработка программного обеспечения / Майк Кон. - Москва: Машиностроение, 2012. - 256 с.

22 Кристиан, Д. Бакли. Искусство внедрения. Секреты успешной реализации / Даррен Палсифер. - Москва: Наука, 2008. – 288 с.

23 Томас, Д. Программист-прагматик. Путь от подмастерья к мастеру / Д. Томас. - М.: ЛОРИ, 2014. – 101 с.

24 Git [Электронный ресурс]. – Режим доступа: https://git-scm.com/

25 JUnit [Электронный ресурс]. – Режим доступа: https://junit.org/junit5/

26 Герберт Java 2 v5.0 (Tiger). Новые возможности / Герберт, Шилдт. - М.: СПб: БХВ-Петербург, 2015. - 208 с.

27 Нотон Java. Справочное руководство. Все, что необходимо для программирования на Java / Нотон, Патрик. - М.: Бином, 2006. - 448 с.

28 Герберт Java 2 v5.0 (Tiger). Новые возможности / Герберт, Шилдт. - М.: СПб: БХВ-Петербург, 2005. - 208 с.

29 TDD [Электронный ресурс]. – Режим доступа: https://docs.microsoft.com/en-us/previous-versions/bb985498