Минобрнауки России

Федеральное государственное автономное образовательное учреждение высшего образования   
Национальный исследовательский университет   
«Московский институт электронной техники»

Кафедра информатики и программного обеспечения вычислительных систем

Василиадис Янис

Бакалаврская работа   
по направлению 09.03.04 «Программная инженерия»

«Разработка программного модуля для проведения финансовых операций на POS-терминале»  
(шифр ПМ ФО)

Студент Василиадис Я.

Руководитель,   
доцент, к.т.н. Федотов А.А.

Москва 2018

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 4](#_Toc515894367)

[1. ИССЛЕДОВАТЕЛЬСКИЙ РАЗДЕЛ 6](#_Toc515894368)

[1.1. Предварительные исследования 6](#_Toc515894369)

[1.2. Виды банковских карт и технологий оплаты 7](#_Toc515894370)

[1.3. Анализ существующих решений 13](#_Toc515894371)

[Выводы по исследовательскому разделу 18](#_Toc515894372)

[2. КОНСТРУКТОРСКАЯ ЧАСТЬ 19](#_Toc515894373)

[2.1. Анализ языков программирования 19](#_Toc515894374)

[2.2. Выбор среды разработки 21](#_Toc515894375)

[2.3. Объектно-ориентированное программирование 25](#_Toc515894376)

[2.4. Паттерны программирования 30](#_Toc515894377)

[2.5. Структура входных и выходных данных 39](#_Toc515894378)

[2.6. Программная реализация ПМ ФО 50](#_Toc515894379)

[2.7. Разработка интерфейса 59](#_Toc515894380)

[Выводы по конструкторскому разделу 63](#_Toc515894381)

[3. Тестирование и отладка 63](#_Toc515894382)

[3.1. Виды тестирования 65](#_Toc515894383)

[3.2. Выбор средств отладки и тестирования ПМ ФО 69](#_Toc515894384)

[3.3. Тестирование интерфейса. 72](#_Toc515894385)

[3.4. Профайлинг ПМ ФО 76](#_Toc515894386)

[Выводы по технологическому разделу 77](#_Toc515894387)

[СПИСОК ИСТОЧНИКОВ 77](#_Toc515894388)

СОКРАЩЕНИЯ

ККТ – контрольно-кассовая техника

POS терминал - электронное программно-техническое устройство для приема к оплате платежных карт

AID – Application ID

EMV – Europay, MasterCard и VISA

IDE – Integrated development environment

ISO – International Organization for Standardization

NFC – Near Field Communication

POS – Point of sale

XML – Extensible Markup Language

TLV – Tag Length Value

# ВВЕДЕНИЕ

В наше время невозможно представить какие-либо финансовые операции без работы с безналичными расчетами. Появляются все новые виды POS-терминалов, а с ними и новые способы оплаты, от магнитных полос до смартфона.

На сегодняшний день предлагаются решения от различных компаний, от обычных терминалов, которые мы можем увидеть в любом магазине, до подключаемых к телефону. Все решения являются специализированным устройством, с собственной операционной системой, либо с сильно модифицированным Embedded Linux [2], а программное обеспечение пишется под каждую модель отдельно. Это означает, что для разработки программного обеспечения, необходимо изучать структуру системы, драйвера и многое другое, что очень замедляет разработку.

Таким образом, разработка является актуальной, так как позволяет повысить эффективность разработки и проведения финансовых операций.

Цель выполнения ВКР: повышение эффективности проведения финансовых операций.

Задачи ВКР:

* исследование предметной области;
* сравнительный анализ существующих решений;
* выбор языка и среды программирования;
* разработка структуры ПМ ФО;
* разработка пользовательского интерфейса;
* программная реализация ПМ ФО;
* отладка и тестирование ПМ ФО;
* разработка документации.

Выполнение выпускной квалификационной работы проходило на предприятии ООО “Терминальные Технологии”, которое занимается разработкой программного обеспечения для POS-терминалов и кассового оборудования. Разрабатываемый ПМ ФО будет использоваться на терминалах нового поколения, которые крайне важны для развития компании и сферы в целом.

Программный модуль для проведения финансовых операций должен обеспечивать следующие возможности:

* сбор информации, необходимой для проведения транзакции в зависимости от используемого типа карта-приемника;
* анализ корректности входной информации, при необходимости сигнализированные о некорректности введенных данных;
* обеспечение шифрования данных между картой и терминалом, а также между незащищенными модулями устройства;
* сохранение нужной информации в базу данных;
* верификация ПИН - кода
* анализ полученных данных;
* подготовка решения по транзакции;
* формирование чека.

Пояснительная записка состоит из введения, трех разделов, заключения, списка литературы и трех приложений:

* Исследовательский - анализ проведенных предварительных научно-исследовательских работ, описание постановки целей и задач, описание входных и выходных данных для разрабатываемого программного модуля, детальные схемы основных алгоритмов работы ПМ ФО, перечисление требований к ПМ ФО, а также сведения об инструментах разработки;
* Конструкторский - содержит информацию о методике создания данного программного модуля, специализированном инструментарии, эскизы пользовательского интерфейса.
* Технологический - описание технологий программирования, а также сведения о приемах тестирования и отладки разрабатываемого программного комплекса.

# ИССЛЕДОВАТЕЛЬСКИЙ РАЗДЕЛ

## Предварительные исследования

Согласно статистике, доля операций по оплате гражданами России товаров и услуг картами в 2016 году составила порядка 80% от общего количества транзакций. Это стало рекордным показателем за последние шесть лет (рис 1.1 и рис 1.2) [2].





Рис. 1.1 Число операций с банковскими картами

Рис. 1.2 График стран по переходу к безналичным расчетам за 2016 год

Как можно увидеть на рисунке 1.2 страны с сильнейшими экономиками находятся на первых позициях к полному переходу на безналичные расчеты.

Так же с 1 февраля 2017 года контрольно-кассовая техника должна отправлять электронные версии чеков оператору фискальных данных — новые правила установлены в 54-ФЗ ст.2 п.2., что в свою очередь увеличит спрос на POS терминалы и кассовую технику.

Проанализировав статистические данные, можно с уверенностью сказать, что будущее за безналичными расчетами.

## Виды банковских карт и технологий оплаты

Банковская карта [4] - пластиковая карта, привязанная к расчётному счету в банке. Используется для оплаты товаров и услуг, в том числе через Интернет. Карты бывают двух видов дебетовые и кредитные. Кредитные карты используется для распоряжения деньгами банка, которые при совершении платежа автоматически берутся у банка в кредит (их требуется вернуть банку). Дебетовые карты используются для распоряжения собственными деньгами, находящимися на расчетном счете в банке.

Виды банковских карт по технологии:

* карты с магнитной полосой [5];
* карты с чипом [6] (микропроцессором), которые в зависимости от способа связи выделяют:
  + бесконтактные карты [7];
  + контактные карты (ISO/IEC 7810, ISO/IEC 7816...);

Виды персонализации карт:

Штрих-код — нанесение на карту цифробуквенной информации, закодированной в виде штрихов;

Тиснение [8] — нанесение на поверхность пластиковой карты цифробуквенной информации в виде рельефных знаков с возможным последующим окрашиванием. Обеспечивает возможность механического копирования данных (например, с помощью импринтера). Тиснение знаков возможно только при горизонтальной ориентации карточки. Тиснение осуществляется двумя видами шрифтов: высотой 4,5 мм — большой (только цифры); высотой 3 мм — малый (цифры и буквы).

Индент-печать — нанесение на поверхность пластиковой карты цифробуквенной информации в виде плоских знаков с возможным последующим типированием (окрашиванием). Характерно для карт, предназначенных только для «электронного» использования;

Магнитная полоса — нанесение на карту магнитного носителя информации с последующей записью на него информации. Имеет три трека для записи: один для цифробуквенной информации и два трека для цифр;

Подписная панель — специальный слой, нанесенный на поверхность карты, позволяющий делать надписи;

Scratch (скретч)-панель — непрозрачный защитный слой, наносимый на поверхность карты поверх защищаемой информации (пин-код, выигрышное слово, код пополнения счета и т. д.);

Чип — микропроцессорный носитель информации, который вмонтирован в карту. Либо имеет контактную площадку, либо использует радиосвязь (RFID).



Рис. 1.3 Магнитная карта

Большинство платёжных карт имеют стандарт ISO 7810 ID-1 формат — 85,595 × 53,98 мм — и используют в качестве носителя данных магнитную полосу.

В большинстве этого типа карт магнитная полоса (рис. 1.1) содержит похожую на пластик плёнку. Магнитная полоса располагается на расстоянии 5,01 мм от края карты и имеет 11,1 мм в ширину. В полосе содержатся три дорожки по 2,79 мм шириной каждая. На первой и третьей в среднем записано 210 бит на дюйм длины, в то время как на второй дорожке плотность составляет 75 бит на дюйм. Каждая дорожка содержит 7-битные буквенно-цифровые символы и 5-битные цифровые символы. Стандарт первой дорожки был разработан Международной ассоциация воздушного транспорта. Стандарт второй дорожки разработала банковская индустрия - American Bankers Association. Стандарты третьей дорожки сформированы ссудо-сберегательной ассоциацией.

Лицевая сторона банковской карты отображает следующую информацию:



Рис. 1.4 Лицевая часть банковской карты

1. В левом верхнем углу располагается название типа карточки по набору возможностей.

2. В правом верхнем углу располагается логотип банка-эмитент

3. Чуть выше середины слева находится чип.

4. Чуть ниже середины располагается эмбоссирован (нанесен в виде рельефных знаков) номер банковской карты.

У платежных систем Visa и MasterCard номер состоит из 16 цифр разделенных на 4 блока по 4 цифры (4-4-4-4).

У платёжной системы American Express номер карты состоит из 15 цифр, разбитых на 3 блока по 4,6,5 цифр в каждом (4-6-5).

Иногда номер карты может иметь 18 или 19 цифр.

18 цифр имеют карты, содержащие последние 2 цифры, как дополнительные, означающие регион отделения банка, в котором был осуществлен выпуск карты (или непосредственный эмбоссинг).

19 цифр имеют карты, которые выпущены дополнительно к основной карте.

Номер банковской карты юридического лица состоит из 20 цифр.

Первая цифра номера банковской карты означает принадлежность к определенной платежной системе, например:

MasterCard – 5,

VISA – 4,

American Express — 3.

Оставшиеся три цифры (вторая, третья и четвертая) – это сформированный номер банковской организации, которая предоставила банковскую карту.

Дополнительно идентифицируют банковское учреждение пятая и шестая цифры. Именно поэтому первоначальные 6 знаков на банковской карте носят название банковского идентификатора (БИН, BIN).

Следующие две цифры номера (седьмая и восьмая) уточняют программу банковской организации, в пределах которой выпущена карта.

Непосредственно номер кредитной карты определяют цифры с девятой по пятнадцатую.

На обратной стороне карты находится магнитная полоса, бумажная полоса с подписью владельца, а на некоторых — CVV2 (англ. Card Verification Value 2 — трёхзначный код проверки подлинности карты) код или его аналог.

В конце 1990-х в банковские карты стали интегрироваться чипы. Чип-карты (рис. 1.2) содержат микропроцессор и собственную операционную систему, контролирующую устройство и доступ к объектам в его памяти. Кроме того, как правило, обладают возможностью проводить криптографические вычисления.

EMV [6] (Europay, MasterCard и VISA) — международный стандарт для операций по банковским картам с чипом. Этот стандарт разработан совместными усилиями компаниями Europay, MasterCard и Visa, чтобы повысить уровень безопасности финансовых операций. Основное отличие для пользователя карты стандарта EMV — это требование ввода ПИН-кода при проведении любого платежа через терминал (например, в магазинах, ресторанах).

Стандарт EMV определяет физическое, электронное и информационное взаимодействие между банковской картой и платёжным терминалом для финансовых операций.

Основные преимущества — повышенный уровень безопасности транзакций и повышение функциональности карт (например, платежная карта с электронным проездным и пропуском).

Повышенный уровень безопасности обеспечивается за счёт ухода от визуального контроля (проверка голограммы, подписи, сверка имени с удостоверением личности) к использованию ПИН-кода и криптографических алгоритмов, таких как DES, Triple DES и приходящим им на замену RSA и SHA для аутентификации карты.

Чип имеет более высокую степень защиты по сравнению с магнитной полосой. Ключ чипа, идентифицирующий карту в банковских операциях, хранится в специальной защищенной памяти, записывается на стадии изготовления и его невозможно оттуда извлечь с помощью внешних устройств. ПИН-код чипа проверяется самим чипом, в отличие от ПИН-кода магнитной полосы, который проверяется компьютером банка. Так же чип в отличие от магнитной полосы не подвержен воздействию магнитных полей.

С 2003 года, такие компании как MasterCard и VISA, начали тестировать возможность бесконтактной оплаты. В 2005 технология начала набирать популярность на западе. В Россию она попала в 2008.

Visa PayWave [9] — технология бесконтактных платежей, предоставляющая бесконтактный способ проведения оплаты на суммы до определённой величины, без подтверждения подписью или PIN-кодом, путём поднесения карты к платёжному терминалу вместо проведения ею для считывания или вставки её в терминал. Данная технология основана на стандарте ISO/IEC 14443 и технологии компании VISA, а также совместима с международным стандартом EMV.

MasterCard PayPass [10] — это совместимая с EMV бесконтактная технология проведения платежа, основанная на стандарте ISO/IEC 14443, предоставляющая держателям карт MasterCard и Maestro способ совершения оплаты путём близкого поднесения или прикосновения платёжной картой или иным платёжным инструментом, таким как телефон или брелок для ключей, к считывающему платёжному терминалу вместо проведения ею для считывания или вставки её в терминал.

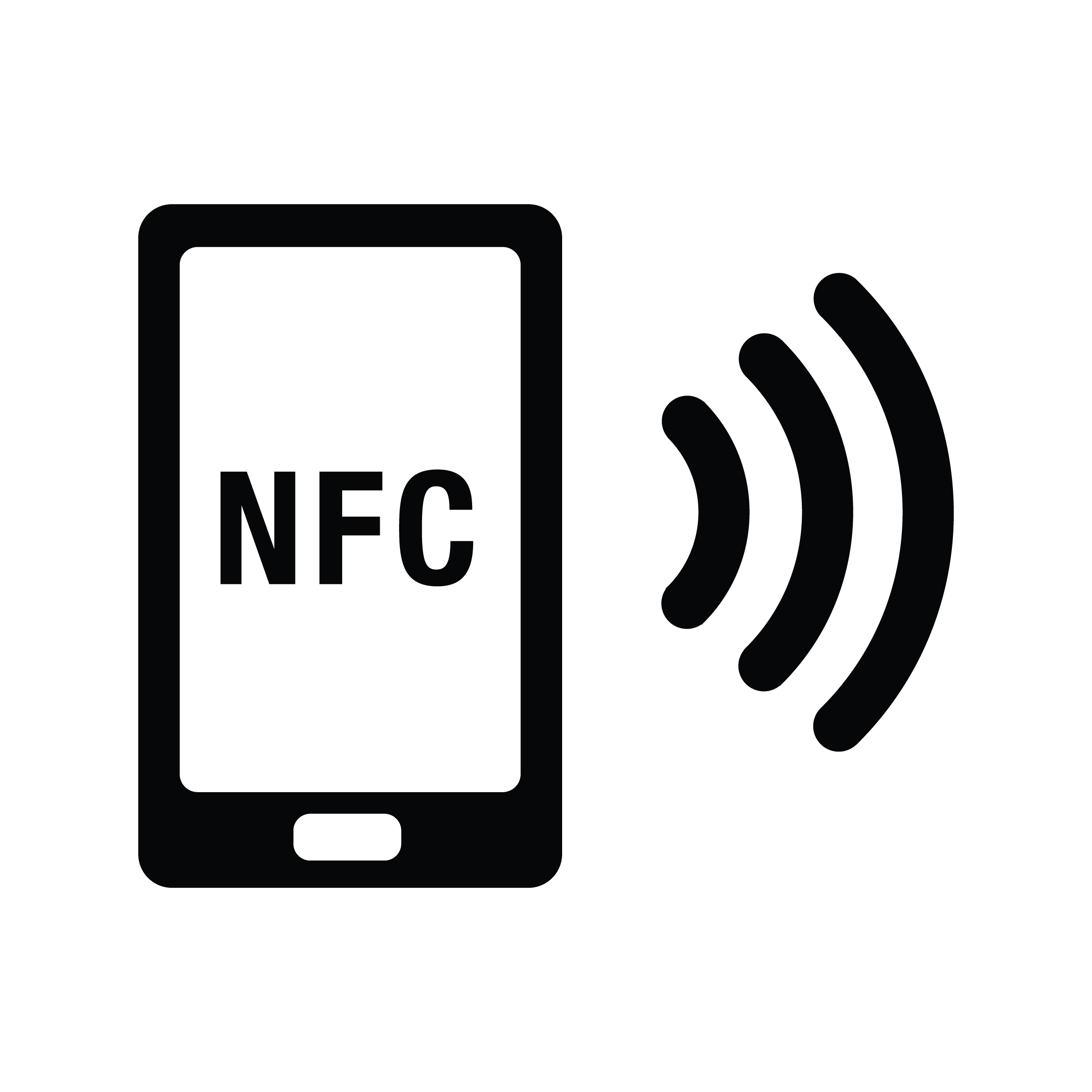


Рис. 1.5 NFC в смартфонах

После успешного внедрения технологии бесконтактной оплаты, появилась идея использовать смартфоны и технологию беспроводной передачи данных малого радиуса (NFC) для передачи данных карты продавцу. Он заменяет чип с ПИН-кодом, либо магнитную полосу на кредитной и дебетовой картах, позволяя пользователю загрузить эти данные в устройство. Пользователь может добавить платежную карту в сервис, сделав фото карты, либо введя информацию о карте вручную. Для совершения платежа необходимо поднести платежное устройство к терминалу оплаты и задержать его до завершения транзакции. Подобный платеж аналогичен повсеместно используемой бесконтактной оплате, но, в отличие от последней, требует двухфакторную аутентификацию для повышения уровня безопасности.

Первыми запустившими эту возможность была компания Apple:

Apple Pay [11] — система мобильных платежей от корпорации Apple. Была представлена 9 сентября 2014 года. С помощью программ Apple Pay пользователи iPhone 6/6+, 6s/6s+, SE, 7/7+, 8/8+, X, Apple Watch могут оплачивать покупки по технологии NFС в сочетании с программой Wallet и Touch ID (Face ID для iPhone X 2017 года).

Позже почти одновременно, были запущены Samsung Pay и Google Pay:

Google Pay [12] (до 20 февраля 2018 года — Android Pay) — разработанная компанией Google система электронных платежей с мобильных устройств (смартфонов, планшетов и умных часов), работающих под операционной системой Android.

Samsung Pay [13] — это служба мобильных платежей, созданная Samsung Electronics, которая позволяет пользователям осуществлять платежи, используя для этого поддерживаемые телефоны и прочую технику компании. Сервис не только поддерживает технологию бесконтактной оплаты с использованием технологии NFC [14], но также и поддерживает оплату с применением технологии электромагнитной передачи (MST), которая позволяет производить оплату с помощью терминалов, поддерживающих только карты с магнитной полосой.

## Анализ существующих решений

Был проведен анализ конечных решений, устройств, так как на данный момент не существует программного обеспечения для работы с картами, написанного под ОС Android, будем рассматривать POS-терминалы различных компаний.

Можно, с некоторыми упрощениями назвать аналогами существующие решения на OC Linux Embedded.

Таблица будет состоять не из конкретных программных решений компаний, так как большая часть информации о них сокрыта. Отождествим программное обеспечение с устройствами, так как каждая компания использует свою операционную систему и программное обеспечение. Рассмотрены лучшие решения компаний на рынке (Табл. 1.1).

Таблица 1.1

Сравнение с аналогами

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Название** | Ingenico iCT220 Dual [1] | Verifone VX 680 [2] | S920 PAX [3] | Azur 01-Ф  (ПМ ФО) [4] |
| Поддержка Java/Kotlin | - | - | - | + |
| Поддержка С/С++ | + | + | + | + |
| Цветной экран | - | + | - | + |
| Контактный считыватель | + | + | + | + |
| Бесконтактный считыватель | + | + | - | + |
| Магнитный считыватель | - | + | - | + |
| Касса и POS-терминал в одном | - | - | - | + |
| Поддержка карт Mifare | - | - | - | + |
| Полноценный графический интерфейс | - | - | - | + |
| Поддержка протокола ТТК | - | - | - | + |

Источники информации:

[1] https://ingenico.us/smart-terminals/telium2/payment-terminals/ict-series/ict220-cl.html

[2] https://www.verifone.com/en/us/devices/portables-transportables/vx-680

[3] http://www.pax.us/portfolio\_page/s920-mobile-payment-terminal/

[4] http://azurpos.ru/



Рис. 1.6 Ingenico iCT220 DUAL

Ingenico iCT220 Dual (рис. 1.6) – одно из самых бюджетных решений, повсеместно применяется, стоимость около 10 000р. Поддерживает контактные карты, Visa, MasterCard, МИР. Так же есть поддержка бесконтактного интерфейса. Магнитный считыватель отсутствует. ОС собственной разработки Telium TETRA OS[15], основана на Linux Embedded.



Рис. 1.7 Verifone VX 680

Verifone VX 680 (рис. 1.7) – также очень популярное решение, но уже другой ценовой категории, стоимость около 30 000р. Поддерживает контактные карты Visa, MasterCard, Maestro, American Express, UnionPay, МИР. Так же поддерживает все виды бесконтактных платежей. Есть считыватель магнитных лент. ОС собственной разработки, основана на Linux Embedded, обычно используется браузер Opera и FST FancyPants[16].



Рис. 1.8 S920 PAX

S920 PAX – данное решение в ценовой категории между Ingenico iCT220 Dual и Verifone VX 680 ~ 20 000р. Контактные карты, Visa, MasterCard, Maestro, МИР. Так же есть поддержка бесконтактного интерфейса. ОС собственной разработки, основана на Linux Embedded.



Рис. 1.9 Azur 01-Ф

Azur 01-Ф [18] – решение компании Azur, для данного устройства проектируется программный модуль ПМ ФО. Устройство работает под управлением ОС Android 5.1. Присутствуют все виды считывателей, контактный, бесконтактный и магнитный. Полная поддержка EMV и L2Wave, а также есть возможность принимать карты MiFare. Поддержка карт Visa, MasterCard, Maestro, American Express, UnionPay, МИР. Средняя стоимость 22 000р.

Требования к ПМ ФО.

1. Поддержка контактных карт:

- поддержка EMV карт

1. Поддержка бесконтактных карт:

- поддержка EMV

- поддержка оплаты смартфоном

- поддержка Google Pay

- поддержка Samsung Pay

- поддержка Apple Pay

1. Поддержка магнитных карт:

- поддержка магнитных банковских карт

- поддержка карт лояльности

- поддержка технологии Samsung MST

1. Поддержка финансовых операций: продажа, возврат, отмена, авто-отмена, сверка итогов, выгрузка журнала транзакций.
2. Поддержка следующих алгоритмов шифрования: RSA, AES, DES, 3DES.
3. Поддержка базы данных для хранения транзакций.
4. Поддержка CVM (Card Verification Method).
5. Поддержка печати чеков.

Выводы по исследовательскому разделу

В ходе решения задач выпускной квалификационной работы в исследовательском разделе проведены следующие работы:

* исследована область использования POS-терминалов;
* проведен обзор существующих программных решений;
* сформулированы задачи ВКР.

# КОНСТРУКТОРСКАЯ ЧАСТЬ

## Анализ языков программирования

Платформа Android поддерживает множество языков. Какие-то языки лучше интегрированы и является более удобными в разработке, какие-то нет. Ниже представлена таблица с возможными языками, их плюсами и минусами (Табл. 1.2).

Таблица 1.2

Сравнение языков программирования

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Критерий | С[21] | С++[21] | C#[20] | Java[19] | Kotlin[22] |
| Опыт разработчика | + | + | - | + | + |
| Безопасность кода | - | - | + | + | + |
| Скорость выполнения кода | + | + | - | +(JIT) | +(JIT) |
| Совместимость с ОС Android | - | - | - | + | + |
| Объектно-ориентированный язык | - | + | + | + | + |
| Функциональный язык | - | - | - | + | + |
| Обобщенное программирование | - | - | + | + | + |
| Динамическая типизация | - | - | - | - | + |
| Параметрический полиморфизм | - | - | + | + | + |
| Сборка мусора | - | - | + | + | + |
| Шаблоны/Generics | - | - | + | + | + |

С – один из старейших языков, обладает высокой скоростью выполнения, за счет того, что является компилируемым языком, и исполняется нативно. Предоставляет полный контроль над памятью и множество низкоуровневых инструментов. Из минусов, нет полной совместимости с Android, использовать можно лишь в виде небольших библиотек и обертками интерфейсов. Поэтому он не подходит для разработки.

С++ - так же огромным плюсом является высокая скорость исполнения и поддержка объектно-ориентированного программирования. Так же, как и С дает полный контроль над памятью. Большим плюсом является наличие STL – Standard Template Library. Но также, как и С, полной интеграции с Android нет, обычно используется в виде небольших библиотек для выполнения быстрых операций. Поэтому он так же не подходит для разработки.

С# – язык более высокого уровня абстракции чем С или С++. Имеет множество встроенных в язык библиотек. В данный момент интеграция с Android осуществляется с помощью Xamarin [23], это фреймворк для кроссплатформенной разработки мобильных приложений. Пока что С# слишком не удобен, чтобы использовать его как основной язык для написания приложений Android.

Java – самым большим плюсом является то, что Java работает под JVM [24] (в случае Android это ART) и это официальный язык ОС Android, на нем написана большая часть системы. Огромное количество встроенных в язык функций и библиотек, а также большее количество документации и примеров кода, что значительно облегчает разработку. Так же большим плюсом является сборщик мусора (Garbage Collector) который освобождает память и удаляет ненужные объекты. Данный язык был выбран как основной для разработки.

Kotlin – сравнительно новый язык, работает под JVM (ART) и так же является официальным языком ОС Android. По сравнению с Java, он элегантнее и многие конструкции в нем в разы короче по объёму кода. Одной из ключевых особенностей языка является Null-Safety, что означает безопасность при работе с объектами, так как в Java можно допустить множество ошибок, связанных с обращением к несуществующему объекту. Скорость компиляции быстрее в 3-5 раз. Более расширенное взаимодействие с IDE. Лучше приспособлен к функциональному программированию. Данный язык так же использовался для разработки.

## Выбор среды разработки

Под платформу Android можно разрабатывать с помощью нескольких сред разработки. Ниже представлена таблица с IDE которые в той или иной мере поддерживают разработку под Android (Табл. 1.3).

Таблица 1.3

Сравнение сред разработки для Android

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Критерий | Eclipse [1] | Microsoft Visual Studio [2] | Android Studio [3] | IntelliJ IDEA [4] |
| Поддержка различных языков | - | + | + | + |
| Поддержка разработки под Android | - | +\- | + | + |
| Поддержка со стороны Google | - | - | + | - |
| Глубокая интеграция с Android (в том числе, частичная сборка и запуск сборки без установки) | - | - | + | - |
| Автодополнение | - | - | + | + |
| Статический анализатор | - | - | + | + |
| Рефакторинг[29] | - | - | + | + |
| Подсветка | + | + | + | + |
| Встроенный отладчик | + | + | + | + |
| Интеграция SVN / Git [30] | - | + | + | + |
| Проверка соблюдения стандартов (форматирования кода, включая именование) | - | - | + | + |
| Компилятор | + | + | + | + |
| Переводчик | - | - | + | + |
| Управление и редактирование баз данных | - | + | + | + |
| Профайлинг | - | + | + | + |

Источники информации:

[1] https://www.eclipse.org/

[2] https://www.visualstudio.com/vs/android/

[3] https://developer.android.com/

[4] https://www.jetbrains.com/idea/

Microsoft Visual Studio - один из старейших программных продуктов для создания как консольных приложений, так и обладающие графическим интерфейсом. Добавление сторонних плагинов позволяет серьёзно расширить функциональность среды, в том числе до кроссплатформенного состояния. Есть поддержка разработки под Android, в том числе и Xamarin, но по функциональности значительно уступает Android Studio. Из минусов, новичку будет просто невозможно самостоятельно разобраться с Visual Studio без прохождения специальных курсов и чтения литературы. Это продукт скорее для опытных разработчиков, обращающих внимание на качество редактора и функции тестирования. Данная среда разработки доступна только для Windows.

IntelliJ IDEA - IDE, разработанная компанией JetBrains, позволяющая создавать программы на множестве популярных языков, среди которых Java, JavaScript, Python, Ruby, Groovy, Scala, PHP, C, C++. Поддержка Android с помощью дополнений. По функционалу уступает Android Studio. Данная среда разработки доступна для Windows, OS X и Linux.

Eclipse - среда разработки, изначально ориентированная на работу с Java, прославилась большим количеством внешних модулей, существенно расширяющих её функциональность (в том числе, это касается количества поддерживаемых языков). Из минусов - существенная нехватка документации, нет единого сообщества разработчиков, на данный момент серьезно уступает аналогичным решениям.

Android Studio - относительно молодая и стремительно развивающаяся IDE, ориентированная на разработчиков приложений для Android. Основанная на программном обеспечении нашей любимой IntelliJ IDEA от компании JetBrains, официальное средство разработки Android приложений. В данный момент является основной IDE для разработки под Android. Так же плюсом является поддержка Google, что означает, что в данной IDE всегда будут самый новый и полный функционал. Очень много различных дополнений, облегчающих разработку, от статических анализаторов, которые в отличии от обычных, более глубоко исследуют код, помогая найти ошибки еще во время написания кода. Данная среда разработки доступна для Windows, OS X и Linux. Данной IDE мы и воспользуемся для разработки ПМ ФО.

Так же для разработки ПМ ФО, а точнее библиотек обработки карт, необходимы инструменты работы с C\C++. Ниже представлена таблица сравнения IDE (Табл. 1.4).

Таблица 1.4

Сравнение сред разработки для C\C++

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Критерий | Eclipse | Microsoft Visual Studio | NetBeans | IntelliJ IDEA / CLion | Code Blocks |
| Поддержка различных языков | - | + | + | + | + |
| Автодополнение | - | - | + | + | + |
| Статический анализатор | - | - | - | + | - |
| Рефакторинг | - | - |  | + | - |
| Подсветка | + | + | + | + | + |
| Встроенный отладчик | + | + | + | + | + |
| Интеграция SVN / Git | - | + | - | + | - |
| Проверка соблюдения стандартов (форматирования кода, включая именование) | - | - | - | + | - |
| Компилятор | + | + | + | + | + |
| Переводчик | - | - | + | + | - |
| Управление и редактирование баз данных | - | + | + | + | - |
| Профайлинг | - | + | + | + | - |

Источники информации:

[1] https://www.eclipse.org/

[2] https://www.visualstudio.com/vs/android/

[3] https://netbeans.org/

[4] https://www.jetbrains.com/clion/

[5] http://www.codeblocks.org/

Так как Eclipse и Microsoft Visual Studio были рассмотрены нами ранее, мы их пропустим.

NetBeans — бесплатная интегрированная среда разработки. Позволяет создавать приложения на языках программирования Java, Python, PHP, JavaScript, C, C++ и ряде других. NetBeans IDE доступна в виде готовых дистрибутивов для платформ Microsoft Windows, Linux, FreeBSD, Mac OS X, OpenSolaris и Solaris, а для всех остальных платформ есть возможность собрать NetBeans самостоятельно из исходников.

Code Blocks - позволит писать не только на С/С++, но и обеспечит поддержкой таких языков программирования, как Fortran и D (с некоторыми ограничениями). Можно расширить набор стандартных инструментов за счет установки расширений. Продукт разрабатывается под Windows, Mac OS X и Linux, однако есть возможность установить среду на любую Unix-подобную систему при помощи сборки исходник

## Объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП) - это парадигма программирования, основанная на концепции «объектов», которая может содержать данные, в виде полей, часто называемых атрибутами; и код, в виде процедур, часто известных как методы. Особенностью объектов является то, что процедуры объекта могут получить доступ и часто изменять поля данных объекта, с которым они связаны (объекты имеют понятие «это» или «я»). В ООП компьютерные программы разрабатываются путем создания их из объектов, которые взаимодействуют друг с другом. Существует значительное разнообразие языков ООП, но самые популярные из них основаны на классах, что означает, что объекты являются экземплярами классов, которые обычно также определяют их тип. Ключевые черты ООП хорошо известны:

Совместно с не-ООП-предшественниками:

Переменные, которые могут хранить информацию, отформатированную в небольшом количестве встроенных типов данных, таких как целые и буквенно-цифровые символы. Это может включать структуры данных, такие как строки, списки и хеш-таблицы, которые либо встроены, либо являются результатом объединения переменных с помощью указателей памяти

Процедуры - также известные как функции, методы, подпрограммы или подпрограммы - которые принимают входные данные, генерируют выходные данные и управляют данными. Современные языки включают структурированные конструкции программирования, такие как циклы и условные обозначения.

Модульная поддержка программирования обеспечивает возможность группировать процедуры в файлы и модули в организационных целях. Модули имеют имена, поэтому идентификаторы в одном модуле не будут случайно путаться с процедурой или переменной, разделяющей одно и то же имя в другом файле или модуле.

Объекты и классы.

Языки, поддерживающие объектно-ориентированное программирование, обычно используют наследование для повторного использования и расширяемости кода в виде классов или прототипов. Те, кто использует классы, поддерживают две основные концепции:

Классы - определения формата данных и доступных процедур для данного типа или класса объекта; могут также содержать данные и процедуры (известные как методы класса), то есть классы содержат члены данных и функции-члены

Объекты - экземпляры классов

Объекты иногда соответствуют вещам, найденным в реальном мире. Например, графическая программа может иметь такие объекты, как «круг», «квадрат», «меню». В системе онлайн-покупок могут быть такие объекты, как «корзина покупок», «клиент» и «продукт». Иногда объекты представляют собой более абстрактные объекты, такие как объект, представляющий открытый файл, или объект, который предоставляет услугу перевода измерений из США в обычную метрику.

Объектно-ориентированное программирование - это не просто классы и объекты; это целая парадигма программирования, основанная на объектах (структурах данных), которые содержат поля данных и методы. Это важно понимать; использование классов для организации совокупности несвязанных методов вместе не является объектной ориентацией.

Каждый объект считается экземпляром определенного класса (например, объект с его полем имени, установленным в «Мэри», может быть экземпляром класса Employee). Процедуры в объектно-ориентированном программировании известны как методы; переменные также известны как поля, члены, атрибуты или свойства. Это приводит к следующим условиям

Переменные класса - относятся к классу в целом; есть только одна копия каждого из них. Переменные или атрибуты экземпляра - данные, принадлежащие отдельным объектам; каждый объект имеет свою собственную копию каждого. Переменные-члены - ссылаются как на переменные класса, так и на экземпляр, определенные определенным классом. Методы класса - относятся к классу в целом и имеют доступ только к переменным класса и входам вызова процедуры. Методы экземпляра - относятся к отдельным объектам и имеют доступ к переменным экземпляра для определенного объекта, на который они вызывают, входов и переменных класса

Объекты доступны как переменные со сложной внутренней структурой, а на многих языках - эффективные указатели, служащие фактическими ссылками на один экземпляр указанного объекта в памяти в куче или стеке. Они обеспечивают слой абстракции, который можно использовать для отделения внутреннего от внешнего кода. Внешний код может использовать объект путем вызова метода конкретного экземпляра с определенным набором входных параметров, чтения переменной экземпляра или записи в переменную экземпляра. Объекты создаются путем вызова специального типа метода в классе, известном как конструктор. Программа может создавать множество экземпляров того же класса, что и при работе, которые работают независимо. Это простой способ использования одних и тех же процедур для разных наборов данных.

Объектно-ориентированное программирование, использующее классы, иногда называется программированием на основе классов, в то время как прототипное программирование обычно не использует классы. В результате для определения понятий объекта и экземпляра используется существенно другая, но аналогичная терминология.

В некоторых языках классы и объекты могут быть составлены с использованием других понятий, таких как черты и миксины.

В классах языки классы определяются заранее, и объекты создаются на основе классов. Если два объекта apple и orange создаются из класса Fruit, они по своей сути являются плодами, и гарантируется, что вы можете обращаться с ними одинаково; например, программист может ожидать существования таких же атрибутов, как цвет или содержание сахара, или созрел.

В языках на основе прототипов объекты являются первичными объектами. Никаких классов не существует. Прототипом объекта является еще один объект, к которому привязан объект. Каждый объект имеет одну прототипную ссылку (и только одну). Новые объекты могут быть созданы на основе уже существующих объектов, выбранных в качестве их прототипа. Вы можете назвать два разных предмета яблочным и оранжевым фруктами, если существует плод объекта, и оба яблока и апельсина имеют плод в качестве своего прототипа. Идея класса фруктов не существует явно, а как класс эквивалентности объектов, имеющих один и тот же прототип. Атрибуты и методы прототипа делегируются всем объектам класса эквивалентности, определенным этим прототипом. Атрибуты и методы, принадлежащие отдельному объекту, могут не разделяться другими объектами одного и того же класса эквивалентности; например, атрибуты содержания сахара могут неожиданно не присутствовать в яблоке. Через прототип может быть реализовано только одно наследование.

Пользователь, а не внешний код, несет ответственность за выбор процедурного кода для выполнения в ответ на вызов метода, как правило, путем поиска метода во время выполнения в таблице, связанной с объектом. Эта функция называется динамической отправкой и отличает объект от абстрактного типа данных (или модуля), который имеет фиксированную (статическую) реализацию операций для всех экземпляров. Если изменчивость вызова основана на более чем одном типе объекта, на который он вызван (например, в выборе метода задействован хотя бы один другой объект параметра), говорят о множественной отправке.

Вызов метода также известен как передача сообщений. Он концептуализируется как сообщение (имя метода и его входные параметры), передаваемое объекту для отправки.

Инкапсуляция представляет собой концепцию объектно-ориентированного программирования, которая связывает данные и функции, которые манипулируют данными, и которая защищает от внешних помех и неправильного использования. Инкапсуляция данных привела к важной концепции скрытия данных ООП.

Если класс не позволяет вызывающему коду получать доступ к внутренним данным объекта и разрешает доступ только с помощью методов, это сильная форма абстракции или скрытия информации, известная как инкапсуляция. Некоторые языки (например, Java) позволяют классам явно вводить ограничения доступа, например, обозначать внутренние данные с ключевым словом private и назначать методы, предназначенные для использования кодом вне класса с открытым ключевым словом. Методы также могут быть разработаны для публичных, частных или промежуточных уровней, таких как protected (что позволяет получить доступ из того же класса и его подклассов, но не для объектов другого класса). В других языках (например, Python) это выполняется только по соглашению (например, частные методы могут иметь имена, начинающиеся с подчеркивания). Инкапсуляция не позволяет внешнему коду воздействовать на внутреннюю работу объекта. Это облегчает рефакторинг кода, например, позволяет автору класса изменять, как объекты этого класса представляют свои данные внутренне, без изменения внешнего кода (до тех пор, пока вызовы метода «public» работают одинаково). Он также поощряет программистов размещать весь код, который связан с определенным набором данных в одном классе, который организует его для легкого понимания другими программистами. Инкапсуляция - это метод, который поощряет развязку.

Объекты могут содержать другие объекты в своих переменных экземпляра; это известно, как композиция объекта. Например, объект класса Employee может содержать (прямо или через указатель) объект в классе Address в дополнение к своим собственным переменным экземпляра типа «first\_name» и «position». Композиция объекта используется для представления отношений «has-a»: каждый сотрудник имеет адрес, поэтому каждый объект Employee имеет доступ к месту хранения объекта Address (либо непосредственно встроенному в себя, либо в отдельном адресе, адресованном через указатель).

Языки, поддерживающие классы, почти всегда поддерживают наследование. Это позволяет организовать классы в иерархии, которая представляет отношения «есть-тип». Например, класс Employee может наследовать от класса Person. Все данные и методы, доступные родительскому классу, также отображаются в дочернем классе с одинаковыми именами. Например, класс Person может определять переменные «first\_name» и «last\_name» с помощью метода «make\_full\_name ()». Они также будут доступны в классе Employee, что может добавить переменные «позиция» и «зарплата». Этот метод позволяет легко повторить использование тех же процедур и определений данных, в дополнение к потенциальному зеркальному отражению реальных отношений интуитивным способом. Вместо использования таблиц базы данных и подпрограмм программирования разработчик использует объекты, которые пользователь может лучше знать: объекты из своего домена приложения.

Подклассы могут переопределять методы, определенные суперклассами. Множественное наследование разрешено на некоторых языках, хотя это может сделать усложняющие переопределения сложными. Некоторые языки имеют специальную поддержку для mixins, хотя на любом языке с множественным наследованием mixin - это просто класс, который не представляет отношения типа a-a-type. Микшины обычно используются для добавления одних и тех же методов к нескольким классам. Например, класс UnicodeConversionMixin может предоставить метод unicode\_to\_ascii (), если он включен в класс FileReader и класс WebPageScraper, которые не имеют общего родителя.

Абстрактные классы не могут быть созданы в объекты; они существуют только с целью наследования в другие «конкретные» классы, которые могут быть созданы. В Java ключевое слово final может использоваться для предотвращения подкласса класса.

Доктрина композиции над наследованием выступает за реализацию - отношения, использующие композицию вместо наследования. Например, вместо наследования класса Person класс Employee может предоставить каждому объекту Employee внутренний объект Person, который затем имеет возможность скрыться от внешнего кода, даже если у класса Person есть много общедоступных атрибутов или методов. Некоторые языки, такие как Go, вообще не поддерживают наследование.

«Принцип открытого / закрытого» защищает, что классы и функции «должны быть открыты для расширения, но закрыты для модификации».

Делегирование - это еще одна языковая функция, которая может использоваться как альтернатива наследованию.

Подтипирование, форма полиморфизма, заключается в том, что вызов кода может быть агностиком относительно того, принадлежит ли объект родительскому классу или одному из его потомков. Например, функция может вызывать «make\_full\_name ()» на объекте, который будет работать, если объект принадлежит классу Person или классу Employee. Это еще один тип абстракции, который упрощает код, внешний по отношению к иерархии классов, и обеспечивает сильное разделение проблем.

В языках, поддерживающих открытую рекурсию, методы объекта могут вызывать другие методы на одном и том же объекте (включая самих себя), обычно используя специальную переменную или ключевое слово, называемое этим или самостоятельно. Эта переменная связана с поздней задержкой; он позволяет методу, определенному в одном классе, вызывать другой метод, который определен позже, в некотором подклассе.

## Паттерны программирования

Для проектирования ПМ ФО, были применены паттерны объектно-ориентированного программирования.

Паттерн проектирования — это часто встречающееся решение определённой проблемы при проектировании архитектуры программ.

В разработке программного обеспечения шаблон разработки программного обеспечения является общим, многоразовым решением общей проблемы в рамках данного контекста при разработке программного обеспечения. Это не готовый дизайн, который можно преобразовать непосредственно в исходный или машинный код. Это описание или шаблон для решения проблемы, которая может использоваться во многих разных ситуациях. Шаблоны проектирования - это формализованные передовые методы, которые программист может использовать для решения общих проблем при разработке приложения или системы.

Объектно-ориентированные шаблоны проектирования обычно показывают отношения и взаимодействия между классами или объектами без указания конечных классов приложений или объектов, которые задействованы. Шаблоны, которые подразумевают изменчивое состояние, могут быть непригодными для функциональных языков программирования, некоторые шаблоны могут быть ненужными на языках, которые имеют встроенную поддержку для решения проблемы, которую они пытаются решить, а объектно-ориентированные шаблоны не обязательно подходят для не-объектов ориентированные языки.

Шаблоны проектирования могут рассматриваться как структурированный подход к компьютерному программированию, промежуточный между уровнями парадигмы программирования и конкретным алгоритмом.

Шаблоны возникли как архитектурная концепция Кристофера Александра (1977/79). В 1987 году Кент Бек и Уорд Каннингем начали экспериментировать с идеей применения шаблонов к программированию, в частности языками шаблонов, и представили свои результаты на конференции OOPSLA в этом году. В последующие годы Бек, Каннингем и другие следили за этой работой.

Модели дизайна приобрели популярность в информатике после книги «Шаблоны дизайна: элементы многоразового объектно-ориентированного программного обеспечения» были опубликованы в 1994 году так называемой «Бандой четырех» (Gamma и др.), Которая часто сокращается как «GoF». В том же году была проведена первая конференция по языкам программирования на языке шрифтов, а в следующем году был создан репозиторий образцов Портленда для документирования шаблонов проектирования. Сфера действия этого термина остается предметом спора.

Шаблоны проектирования могут ускорить процесс разработки, предоставив проверенные, проверенные парадигмы развития. Эффективный дизайн программного обеспечения требует рассмотрения проблем, которые могут не стать видимыми до тех пор, пока они не будут реализованы. Новый код часто может содержать скрытые тонкие проблемы, требующие времени для обнаружения, которые иногда могут вызвать серьезные проблемы в будущем. Повторное использование шаблонов проектирования помогает предотвратить такие тонкие проблемы, а также улучшает читаемость кода для кодеров и архитекторов, знакомых с шаблонами.

Чтобы достичь гибкости, шаблоны проектирования обычно вводят дополнительные уровни косвенности, что в некоторых случаях может усложнять полученные проекты и ухудшать производительность приложений.

По определению шаблон должен быть запрограммирован заново в каждом приложении, которое его использует. Поскольку некоторые авторы рассматривают это как шаг назад от повторного использования программного обеспечения, предоставляемого компонентами, исследователи работали над превращением шаблонов в компоненты. Мейер и Арноу смогли обеспечить полную или частичную компоненту двух третей образцов, которые они пытались сделать.

Модели проектирования предоставляют общие решения, документированные в формате, который не требует специфики, связанной с конкретной проблемой.

В ПМ ФО были использованы следующие паттерны:

* Фабричный метод
* Абстрактная фабрика
* Адаптер
* Строитель
* Одиночка
* Заместитель
* Наблюдатель

**Паттерн Фабричный метод (Factory Method)**

Также известный как Virtual Constructor, Factory Method связан с идеей работы библиотек: библиотека использует абстрактные классы для определения и поддержания отношений между объектами. Одним из видов ответственности является создание таких объектов. Библиотека знает, когда нужно создать объект, но не тот объект, который он должен создать, это специфично для приложения, использующего библиотеку.

Метод Factory работает точно так же: он определяет интерфейс для создания объекта, но оставляет его выбор в подклассах, при этом создание откладывается во время выполнения.



Рис. 2.1 UML диаграмма паттерна Фабричный метод

Фабричный метод является одним из наиболее используемых и одним из наиболее надежных шаблонов проектирования. При внедрении фабричного метода необходимо учитывать только несколько моментов.

**Абстрактная фабрика (Abstract Fabric)**

Модуляция - большая проблема в сегодняшнем программировании. Программисты во всем мире пытаются избежать идеи добавления кода в существующие классы, чтобы заставить их поддерживать инкапсуляцию более общей информации. Возьмите случай менеджера информации, который управляет номером телефона. Номера телефонов имеют определенное правило, по которому они генерируются в зависимости от областей и стран. Если в какой-то момент приложение должно быть изменено, чтобы поддерживать добавление чисел в новую страну, код приложения должен быть изменен, и это будет становиться все более и более сложным.

Чтобы предотвратить его, используется шаблон шаблона Abstract Factory. Используя этот шаблон, определяется структура, которая создает объекты, которые следуют общей схеме, и во время выполнения эта фабрика сопрягается с любой конкретной фабрикой для создания объектов, которые следуют образцу определенной страны. Другими словами, Абстрактная фабрика является суперзаводом, который создает другие заводы (фабрики фабрик).

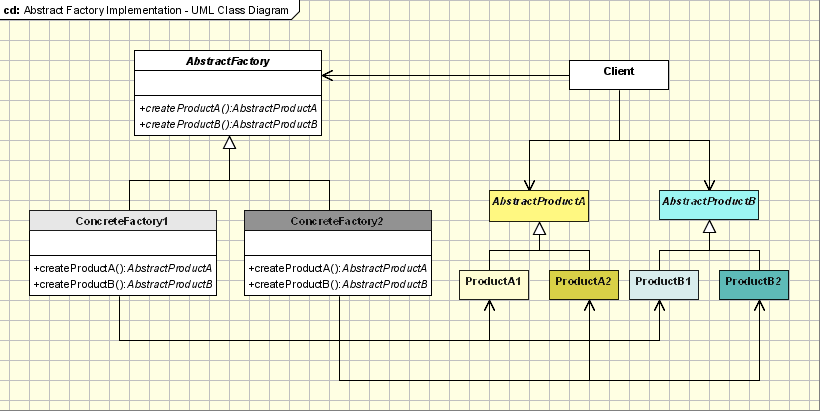


Рис. 2.2 UML диаграмма паттерна Абстрактная фабрика

Класс AbstractFactory объявляет только интерфейс для создания продуктов. Фактическое создание - это задача классов ConcreteProduct, где хороший подход использует шаблон проектирования Factory Method для каждого продукта семейства.

Расширение фабрик можно сделать, используя один метод Create для всех продуктов и содержащий информацию о типе необходимого продукта.

**Одиночка(Singleton)**

Иногда важно иметь только один экземпляр для класса. Например, в системе должен быть только один диспетчер окон (или только файловая система или только диспетчер очереди печати). Обычно синглтоны используются для централизованного управления внутренними или внешними ресурсами и обеспечивают глобальную точку доступа к себе.  
 Шаблон singleton является одним из простейших шаблонов проектирования: он включает только один класс, который отвечает за создание экземпляра, чтобы убедиться, что он создает не более одного экземпляра; в то же время он предоставляет глобальную точку доступа к этому экземпляру. В этом случае один и тот же экземпляр можно использовать повсюду, поскольку невозможно каждый раз ссылаться на конструктор.

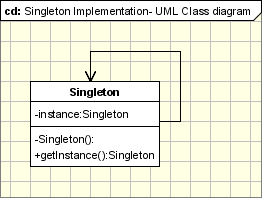


Рис. 2.3 UML диаграмма паттерна Одиночка

Многопоточность. Особое внимание следует уделять использованию одноточечного устройства в многопоточном приложении.

Сериализация. Когда Singletons реализуют интерфейс Serializable, им необходимо реализовать метод readResolve, чтобы избежать наличия двух разных объектов.

Загрузчики классов - eсли класс Singleton загружен двумя разными загрузчиками классов, у нас будет 2 разных класса, по одному для каждого загрузчика классов.

Глобальная точка доступа, представленная именем класса. Экземпляр singleton получается с использованием имени класса. На первом представлении это простой способ доступа к нему, но он не очень гибкий. Если нужно заменить класс Sigleton, все ссылки в коде должны быть изменены соответственно.

**Паттерн строитель (Builder).**

Чем сложнее приложение, тем сложнее увеличиваются классы и объекты. Сложные объекты изготовлены из деталей, изготовленных другими объектами, которые при особом уходе нуждаются в особой заботе. Для приложения может потребоваться механизм для создания сложных объектов, которые не зависят от тех, которые составляют объект. Если это проблема, с которой вы сталкиваетесь, вы можете попробовать использовать шаблон проектирования Builder (или Adaptive Builder).

Этот шаблон позволяет объекту клиента создавать сложный объект, указывая только его тип и контент, защищаясь от деталей, связанных с представлением объекта. Таким образом, процесс построения может быть использован для создания различных представлений. Логика этого процесса изолирована от фактических шагов, используемых при создании сложного объекта, поэтому процесс может быть использован снова для создания другого объекта с тем же набором простых объектов, что и первый.

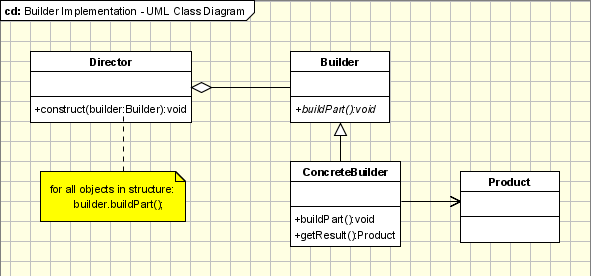


Рис. 2.4 UML диаграмма паттерна Строитель

**Паттерн адаптер (Adapter).**

Шаблон адаптера работает как мост между двумя несовместимыми интерфейсами. Этот тип шаблона проектирования имеет структурную структуру, поскольку этот шаблон объединяет возможности двух независимых интерфейсов.

Этот шаблон включает в себя один класс, который отвечает за объединение функциональных возможностей независимых или несовместимых интерфейсов. Примером реальной жизни может служить кард-ридер, который выступает в качестве адаптера между картой памяти и ноутбуком. Вы подключаете карту памяти к считывателю карт и считывателю карт в ноутбук, чтобы можно было считывать карту памяти с помощью ноутбука.

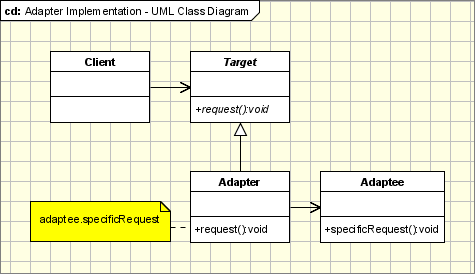


Рисунок 2.5 UML диаграмма паттерна Адаптер

**Паттерн заместитель (Proxy).**

Иногда нужна возможность контролировать доступ к объекту. Например, если нам нужно использовать только несколько методов для некоторых дорогостоящих объектов, будем инициализировать эти объекты, когда они понадобятся полностью. До этого момента можно использовать некоторые световые объекты, подвергая тот же интерфейс, что и тяжелые объекты. Эти легкие объекты называются прокси-серверами, и они будут создавать экземпляры этих тяжелых объектов, когда они действительно нужны, и к тому времени мы будем использовать некоторые легкие объекты.

Эта способность контролировать доступ к объекту может потребоваться по целому ряду причин: контроль, когда необходимо создать и инициализировать дорогостоящий объект, предоставить разные права доступа к объекту, а также предоставить сложные средства доступа и ссылки на объекты работает в других процессах, на других машинах.

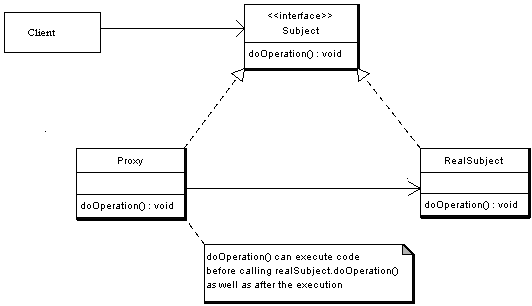


Рис. 2.6 UML диаграмма паттерна Заместитель

**Паттерн наблюдатель (Observer).**

Шаблон наблюдателя представляет собой шаблон разработки программного обеспечения, в котором объект, называемый субъектом, поддерживает список своих иждивенцев, называемых наблюдателями, и автоматически уведомляет их о любых изменениях состояния, как правило, путем вызова одного из своих методов.

Он в основном используется для реализации распределенных систем обработки событий в программном обеспечении, управляемом событиями. Большинство современных языков, таких как Java и Kotlin, построены в конструкциях «событий», которые реализуют компоненты шаблона наблюдателя, для легкого программирования и короткого кода.

Шаблон наблюдателя также является ключевым элементом в привычном архитектурном шаблоне модели-представления-контроллера (MVC). Шаблон наблюдателя реализован в многочисленных библиотеках и системах программирования, включая почти все инструменты GUI.

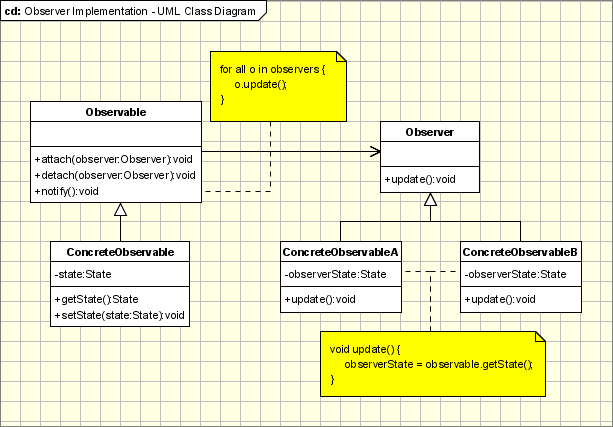


Рис. 2.7 UML диаграмма паттерна Наблюдатель

## Структура входных и выходных данных

Общая схема входных и выходных данных приведена на рис. 2.8.

Входные данные в программный модуль поступают:

* файлы конфигураций в формате XML [31], содержащие общие настройки терминала, такие как адрес и порт сервера процессинга, код валюты, номер категории мерчанта и типа терминала, а также специальные настройки для работы с картами PayPass, PayWave, МИР, American Express, Union Pay;
* от сервера банковского процессинга запросы в формате по стандарту ISO8583[32];
* запросы от кассы в виде сообщений протокола ТТК[33];
* непосредственно данные с банковских карт считанные с помощью одного из считывателей: контактный, бесконтактный, магнитный.
* команды от TMS (Terminal Management System)

К выходным данным ПМ ФО:

* сообщения кассе в формате протокола ТТК, содержащие информацию о банковской карте и ее ответы на некоторые команды;
* сообщения серверу банковского процессинга, содержащие информацию о банковской карте и ее ответы на некоторые команды;
* сохраненные данные о карте в базе данных, для проведения таких операций как отмена или возврат.

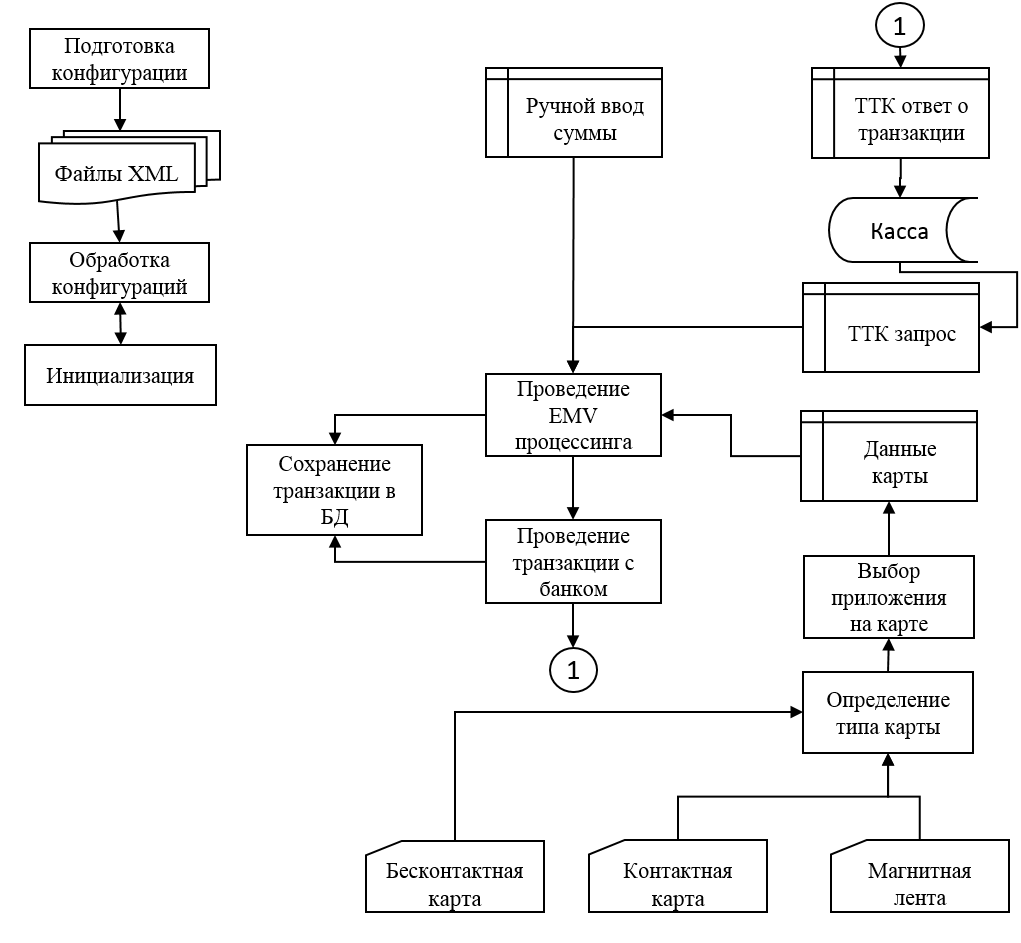


Рис. 2.8 Схема данных ПМ ФО

В терминал должны быть загружены следующие файлы:

Специальный файл, в котором указаны пути на другие файлы, которые необходимо загрузить:

|  |
| --- |
| <Package>  <DroidConfig target=**"ca\_keys"** origin=**"../ca\_keys.xml"**/>  <DroidConfig target=**"contact\_config"** origin=**"../train/contact\_config.xml"**/>  <DroidConfig target=**"ctless\_config"** origin=**"../train/ctless\_config.xml"**/>  <DroidConfig target=**"terminal\_config"** origin=**"../train/terminal\_config.xml"**/>  </Package> |

Общая конфигурация терминала:

|  |
| --- |
| <terminal\_config>  <terminal  add\_app\_caps=**"C0D0A0F0FCF8"**  additional\_caps=**"FF80F0A001"**  application\_caps=**"040FC8000"**  country\_code=**"643"**  currency\_code=**"643"**  currency\_exponent=**"02"**  currency\_name=**"RUR"**  emv\_type=**"22"**  allow\_refund\_no\_card=**"0"**  allow\_refund\_no\_pin=**"0"**  magstripe\_pin\_mode=**"off"**  magstripe\_amount\_limit=**"1000000**  **……**/>  <host  name=**"TRAINING"**  address=**"127.0.0.1"**  main\_timeout=**"60"**  port=**"443"**  ssl=**"0"**/>  <kld  address=**"192.168.0.1"**  port=**"55555"**/>  </terminal\_config> |

В данном файле находятся основные параметры для терминала: битовые маски для конфигурации работы терминала, код страны и валюты, экспонента и имя валюты, идентификационный номер продавца и терминала, имя продавца, параметры банковского сервера, параметры Key Loading Device (устройства для безопасной загрузки ключей) и другие параметры.

Конфигурация контактной части:

|  |
| --- |
| <contact\_cards>  <card\_groups>  <row  AIDs=**"A0000000041010"**  CVM\_caps\_if\_not\_req=**"08"**  CVM\_caps\_if\_req=**"B0"**  CVM\_req\_limit=**"0"**  EMV\_floor\_limit=**"0"**  TAC\_default=**"FE50BСA0"**  TAC\_denial=**"00000000"**  TAC\_online=**"FE50F8"**  issuer\_name=**"MASTERCARD"**/>  <row  AIDs=**"A0000000043060"**  CVM\_caps\_if\_not\_req=**"90"**  CVM\_caps\_if\_req=**"90"**  max\_target\_percent=**"00"**  compare\_AID\_length=**"0"**  default\_DDOL=**"9F3704"**  default\_TDOL=**"9F0209A039C0195059F3704"**  issuer\_name=**"MAESTRO"**/>  <row AIDs=**"A0000000031010,A0000000032010"**  CVM\_caps\_if\_not\_req=**"08"**  CVM\_caps\_if\_req=**"B0"**  CVM\_req\_limit=**"0"**  compare\_AID\_length=**"0"**  versions=**"008C,008D,0096"**  issuer\_name=**"VISA"**/>  <row  AIDs=**"A0000006581010,A0000006582010,A0000006581099,A0000006582099"**  CVM\_caps\_if\_not\_req=**"08"**  CVM\_caps\_if\_req=**"B0"**  CVM\_req\_limit=**"0"**  EMV\_floor\_limit=**"0"**  TAC\_default=**"BC40800"**  TAC\_denial=**"0010000"**  TAC\_online=**"BC4000"**  mandatory\_data=**"5A48C8D"**  threshold\_value=**"00000000"**  issuer\_name=**"MIR"**/>  </card\_groups>  </contact\_cards> |

В данном файле находится конфигурация библиотеки EMV – ответственной за прием контактных карт. В основном настройка поведения библиотеки в различных ситуациях и различных карт.

Конфигурация бесконтактных карт:

|  |
| --- |
| <Config>  <!-- Main configuration with id=0 must be present -->  <Group group\_id=**"0"**>  <Terminal\_Floor\_Limit>**000000000**</Terminal\_Floor\_Limit>  <Terminal\_Contactless\_Floor\_Limit>**000000000**</Terminal\_Contactless\_Floor\_Limit>  <Terminal\_Contactless\_Transaction\_Limit>**001000000**  </Terminal\_Contactless\_Transaction\_Limit>  <Terminal\_CVM\_Required\_Limit>**000100000**</Terminal\_CVM\_Required\_Limit>  <Flags>**00**</Flags>  <!-- Application capability -->  <Tag number=**"600007"** type=**"b"**>**0400000**</Tag>  <!-- Additional Terminal Capabilities -->  <Tag number=**"9F40"** type=**"b"**>**FF80F0A001**</Tag>  </Group>  <!-- Custom group for MIR-->  <Group group\_id=**"3"**>  <Terminal\_Floor\_Limit>**999999999**</Terminal\_Floor\_Limit>  <Terminal\_Contactless\_Floor\_Limit>**9999999999**</Terminal\_Contactless\_Floor\_Limit>  <Terminal\_Contactless\_Transaction\_Limit>**000001000**</Terminal\_Contactless\_Transaction\_Limit>    <!-- Visa Electron PayWave -->  <Application AID=**"A0000000032010"** group\_id=**"2"** kernel=**"3"** issuer\_name=**"VISA"**>  </Application>  <!-- MIR -->  <Application AID=**"A000006581010"** kernel=**"643"** group\_id=**"3"**>  <Flags>**00**</Flags>  <Tag number=**"DF811B"** type=**"b"**>**A0**</Tag>  </Application>  <Application AID=**"A0000006581099"** kernel=**"643"** group\_id=**"3"**>  <Flags>**00**</Flags>  <Tag number=**"DF811B"** type=**"b"**>**A0**</Tag>  </Application>  </Config> |

Данная конфигурация нужна для настройки бесконтактной библиотеки L2WAVE - которая ответствен за прием бесконтактных карт и смартфонов с NFC. В ней так же, как и в EMV, находятся параметры для настройки поведения библиотеки в зависимости от ситуаций и конкретных карт.

Полное описание тегов EMV/L2WAVE находятся в официальных спецификациях [6,35,36].

Файл с публичными ключами шифрования:

|  |
| --- |
| <CAKeys>  <RID value=**"A000000004"**>  <Key index=**"04"**>  <Modulus>**A6DA428387A502D7DDFB7A74D3F412BE762627197B25435B7A81716A700157DDD06F7CC99D6CA28C2470527E2C03616B9C59217357C2674F583B3BA5C7DCF2838692D023E3562420B4615C439CA97C44DC9A249CFCE7B3BFB22F68228C3AF13329AA4A613CF8DD853502373D62E49AB256D2BC17120E54AEDCED6D96A4287ACC5C04677D4A5A320DB8BEE2F775E5FEC5**</Modulus>  <Exponent>**03**</Exponent>  <Hash>**381A035DA58B482EE2AF75F4C3F2CA469BA4AA6C**</Hash>  <Expire>**171231**</Expire>  </Key>  </Key> |

В данном файле находятся публичные части RSA ключей шифрования, составные CA ключа для EMV процессинга. Каждый ключ имеет индекс, модуль с экспонентой ключа, а также дату окончания действия.

Для работы с картой, необходимо было поддержать формат сообщения BER TLV, формат которого определен стандартом ISO 8825.

В протоколах передачи данных TLV (тип длины или значение длины тега) представляет собой схему кодирования, используемую для необязательного информационного элемента в определенном протоколе.

Тип и длина фиксируются по размеру (обычно 1-4 байта), а поле значения имеет размер переменной. Эти поля используются следующим образом:

Тип - двоичный код, часто просто буквенно-цифровой, который указывает вид поля, который представляет эта часть сообщения.

Размер поля значений (обычно в байтах).

Длина должна быть либо 1, 2, 4, либо 8. В зависимости от ее длины, она кодируется следующим образом:

* если длина равна 1 (то есть длина значения 1 октет), она кодируется в одном октете;
* если длина равна 2 (= длина значения - 2 октета), она кодируется в 2 октетах в байтовом порядке сети;
* если длина равна 4 (длина значения равна 4 октетам), она кодируется в 4 октетах, в байтовом порядке сети;
* если длина равна 8 (= длина значения 8 октетов), она кодируется в 8 октетов в байтовом порядке сети.

Значение - набор байтов с переменным размером, которые содержат данные для этой части сообщения.

Некоторые преимущества использования решения системы представления данных TLV:

Последовательности TLV легко просматриваются с использованием обобщенных функций синтаксического анализа;

Новые элементы сообщения, которые принимаются на более раннем узле, могут быть безопасно пропущены, а остальная часть сообщения может быть проанализирована. Это похоже на то, что неизвестные теги XML могут быть безопасно пропущены;

Элементы TLV могут быть размещены в любом порядке внутри тела сообщения;

Элементы TLV обычно используются в двоичном формате, который делает разборчивость быстрее, а данные меньше;

Легче генерировать XML из TLV, чтобы сделать возможной проверку человеком данных.

2.6 Программная архитектура и алгоритм работы ПМ ФО

Схема алгоритма представлена на рис. 2.10 и 2.11.

Для приведения терминала в рабочее состояние, его необходимо инициализировать. Примеры конфигурации были представлены в предыдущем разделе. Так же после конфигурации обычно необходимо ввести вручную идентификатор терминала, имя продавца, и идентификатор продавца.

Для старта финансовой операции, необходимо выбрать тип операции и получить информацию о сумме платежа. Старт операции возможен как по средствам интерфейса и ввода суммы, так и по средствам протокола ТТК.

Так же операция может быть выполнена с использованием данных предыдущих транзакций, к примеру отмена транзакции или возврат. В этом случае данные о карте и транзакции берутся из базы данных.

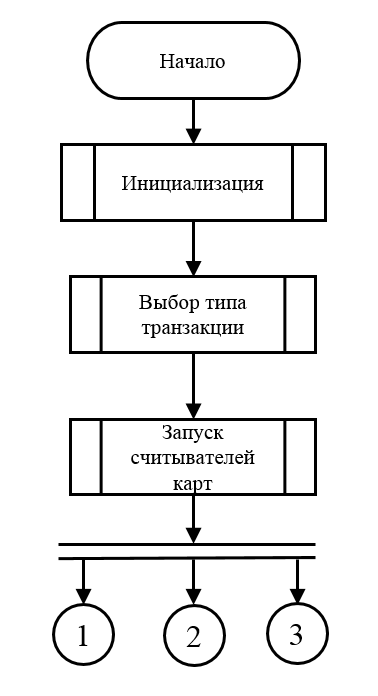


Рис. 2.10 Общая схема алгоритма инициализации ПМ ФО

После введения суммы, три считывателя: контактный, бесконтактный и магнитный, ждут банковскую карту. Считыватель имеет таймаут, по истечению которого, операция будет прервана. После поднесения карты терминал проверяет карту на исправность, в случае чипа и бесконтакта, проверяется, что карта отвечает на первые команды, а в случае магнитной карты, проверяется наличие трека 2. Если карта в порядке, то работа продолжается, если нет, пользователю сообщается, что карта неисправна и предлагается использовать другую.

После проверки карты, происходит обмен командами, которые выбирают нужное приложение и проверяют, поддерживает ли терминал данное приложение. Если приложение поддерживается, терминал начинает основную часть процессинга EMV.

Во время процессинга происходит проверка карты на ее срок действия и подлинность. Подлинность проверяется с помощью публичных ключей CA.

Далее в зависимости от суммы и еще нескольких параметров, которые определяются картой и настройкой библиотеки, терминал предлагает пользователю провести CVM (Cardholder Verification Method).

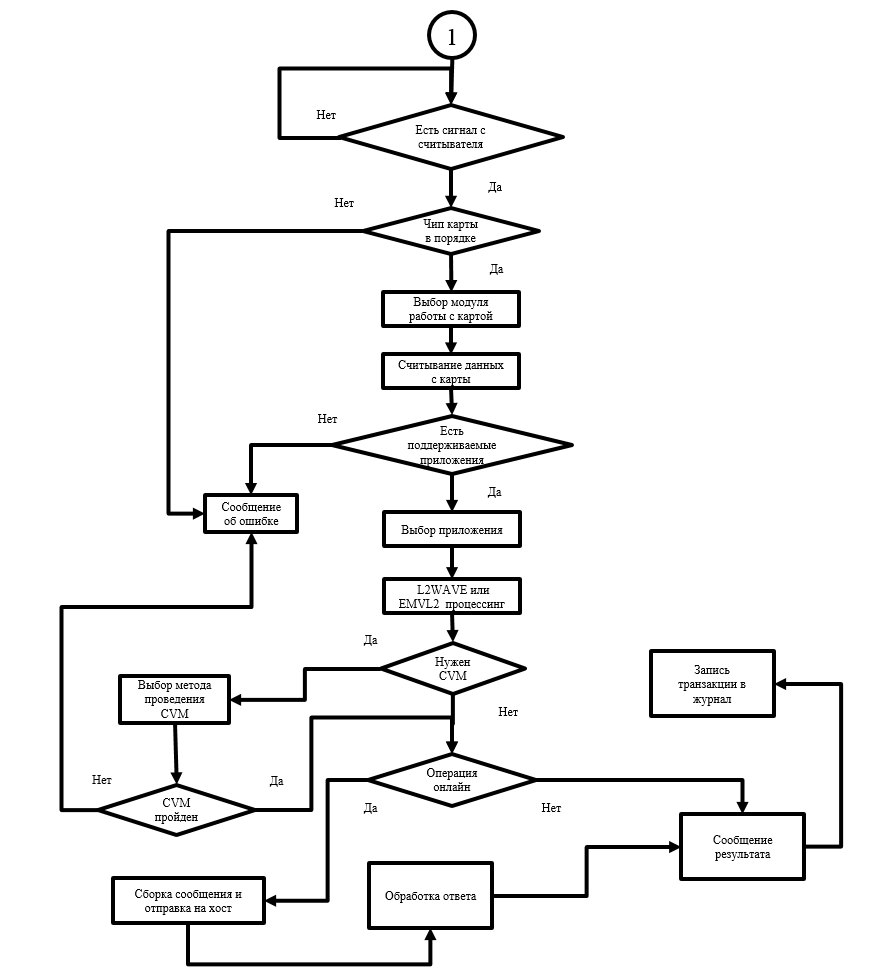


Рис. 2.11 Общая схема алгоритма работы ПМ ФО

CVM бывает нескольких видов:

* Online PIN – введенный пин-код шифруется на ключе выданном сервером работы с ключами банка, помещается в тело транзакции для отправки на сервер процессинга банка. С сервера в зависимости от правильности пин-кода, приходит ответ, транзакция одобрена или нет.
* Offline PIN Plain Text – введенный пин-код без шифрования отправляется на карту. Карта отвечает, верен пин-код или нет и соответственно принимает решение о транзакции.
* Offline Enciphered PIN – введенный пин-код шифруется на ключе, сгенерированном на карте. И так же в зависимости от правильности принимается решение о транзакции.
* NoCVM – данный вид верификации выбирается в случае когда верификация не нужна, чаще всего это бесконтактный платеж менее CVM Limit, в Росии этот лимит является - 1000р.
* Signature – данный вид верификации обычно выбирается при не возможности ввода пина. На чеке владелец карты оставляет подпись. Данный метод возможен лишь тогда, когда на тыльной стороне карточки есть роспись владельца карты.

Так же необходимо учитывать, что для каждого вида считывания карты, существуют ограничения по CVM. К примеру:

* контакт – теоретически поддерживает все виды CVM, но на практике, карта может и не поддерживать все виды, а лишь некоторую часть, это зависит от самого чипа и его конфигурации.
* бесконтакт – для такого вида использования карты, возможен лишь Online PIN, Signature и NoCVM. Причиной является то, что существует необходимость делать транзакции бесконтактом крайне быстрыми, что бы владельцу не нужно было держать карту несколько секунд, тогда все плюсы бесконтактной оплаты исчезнут.
* магнитная полоса – такие же правила, как и для бесконтакта, но на практике обычно это Signature.

Далее в зависимости от обстоятельств, есть у терминала интернет или нет и настроек библиотеки происходит выбор между отправления транзакции онлайн на сервер и оффлайн транзакции. В первом случае решение о транзакции принимает сервер, а в другом библиотека и карта. Есть случай, когда даже при одобрении с сервера, карта все же решает отклонить транзакцию, в этом случае на сервер отправляется еще одна транзакция которая отменяет предыдущую.

Так же, существует защита от проблемных ситуаций, связанных с плохой связью терминала. До отправки транзакции на сервер, она сохраняется в базу данных и, если ответ сервера не доходит до терминала, перед следующей транзакцией будет отправлена отмена предыдущей.

После получения ответа, транзакция, сохраненная ранее в базе данных, обновляется и печатается чек.

## Программная реализация ПМ ФО

Применение SWIG и JNI

Используя Java Native Interface (далее JNI), можно вызывать код С/С++ из Java и, наоборот, из программы на С\С++, можно создавать Java объекты и вызывать Java методы. Использование JNI обычно ведет к потере платформо-независимости Java-кода. Использование JNI позволяет совместить объектно-ориентированный подход Java и существующий системно-зависимый код на С\С++. Это является важным и необходимым условием к использованию Java при разработке компонентов, которые должны работать быстро и безотказно, к примеру компоненты серверов.

Есть несколько причин совместного использования С\С++ и Java: стандартные библиотеки Java не всегда поддерживают низкоуровневые системно-зависимые возможности, такие как работа напрямую с памятью или устройствами. Так же существует необходимость использования уже написанного кода на других языках или желание эффективно реализовать участок кода, для которого критично время исполнения. Эти причины, возможно, не очень существенны при разработке клиентских приложений, однако в случае серверных они становятся основными.

**Java Native Interface**

Взаимодействие кодов Java и С\С++ может осуществляться несколькими способами:

* С\С++ код получает управление непосредственно из Java путем вызова native метода.
* С\С++ код динамически загружает JVM с помощью Invocation API.

Для того чтобы передать управление С\С++ - коду из Java, необходимо создать собственный Java\Kotlin метод, сгенерировать с помощью Javah заголовочный файл для С\С++, разработать сами функции, в которые будет передаваться управление и данные, и поместить их в файл библиотеки.

Для того, чтобы создать собственный метод, нужно дописать специальный спецификатор native. Этот спецификатор показывает компилятору, что реализация данного метода будет представлена в библиотеке на C\C++.

Кроме описания собственного метода, в Java\Kotlin необходимо динамически загрузить библиотеку, содержащую С\С++ код реализации данного метода. Для этого в классе Java.Lang.System существует метод public static void loadLibrary**(**String LibName**)**, загружающий указанную библиотеку. Следующий пример демонстрирует описание собственного метода [3].

Создание native-метода в Java\Kotlin:

|  |
| --- |
| class MyClass **{**  static **{**  System**.**loadLibrary**(**"test"**);**  **}**  public native int nativeMethod**();**  **}** |

В приведенном примере nativeMethod**() –** собственный метод, и его С\С++ реализация находится в библиотеке test. Метод loadLibrary**()** вызывается в статическом блоке, что обеспечивает вызов этого метода после загрузки класса MyClass. Метод loadLibrary**()** преобразует передаваемый параметр, имя, в соответствии с тем, на какой платформе будет запущен код. В данном примере test преобразуется в libtest.so для Linux\Unix и test.dll для Windows. Для Windows DLL должна находиться либо в текущем каталоге процесса, либо в каталоге, содержащем исполняемый exe файл, либо в системном каталоге Windows или в каталогах, указанных в переменной окружения PATH. Для UNIX библиотечный файл должен находиться либо в текущем каталоге процесса, либо в подкаталоге Lib основного каталога Java\Kotlin, либо в каталогах, перечисленных в переменной окружения LD\_LIBRARY\_PATH. Если указанную библиотеку найти не удается, метод loadLibrary**()** генерирует исключение Java.Lang.UnsatisfitedLinkError.

Для более надежной работы с собственными методами можно использовать, к примеру, следующий код:

|  |
| --- |
| pulibc class App **{**  pulibc static void main**(**String args**)** **{**  MyClass myClassInstance **=** **new** MyClass**();**  **try** **{**  myClassInstance**.**nativeMethod**();**  **}**  **catch** **(**UnsatisfiedLinkError e**)** **{**  System**.**out**.**println**(**"Method not found (" **+** e **+** ")"**);**  **}**  **}**  **}**    class MyClass **{**  static **{**  **try** **{**  System**.**loadLibrary**(**"test"**);**  **}**  **catch** **(**UnsatisfiedLinkError e**)** **{**  System**.**out**.**println**(**"Library not found (" **+** e **+** ")"**);**  **}**  **}**    native void nativeMethod**();**  **}** |

Компиляция программ, содержащих собственные методы, ничем не отличается от компиляции обычных программ. Например, если записать предыдущий пример в файл с именем MyClass.java, то для его компиляции необходимо выполнить следующую команду:

|  |
| --- |
| # javac MyClass.java |

**Создание заголовочного файла**

Создание С\С++ кода необходимо начинать с создания заголовочного файла. Его можно написать вручную или воспользоваться утилитой Javah. Второй путь предпочтительней, так как допускает меньшее количество ошибок. При обращении к утилите Javah указывается имя класса и параметр -JNI. Без него Javah будет генерировать файл в формате JDK 1.0 NI. Имя класса представляет собой полное квалифицированное имя класса.

Пример:

|  |
| --- |
| #javah -JNI Java.Lang.Runtime |

Перед использованием утилиты Javah соответствующий Java\Kotlin класс должен быть скомпилирован в class-файл. Утилита Javah анализирует class-файл и строит заголовочный файл, в котором перечислены объявления С\С++ функций, представляющих реализации соответствующих собственных методов.

Например, если выполнить следующие команды:

|  |
| --- |
| # javac MyClass.java  # javah -jni MyClass |

то javah сгенерирует следующий файл MyClass.h:

|  |
| --- |
| #include <jni.h>  #ifndef \_included\_MyClass  #define \_included\_MyClass  #ifdef \_ \_cplusplus  extern "c" **{**  #endif  /\*  \* class: MyClass  \* method: nativeMethod  \* signature: ()  \*/  jniexport void jnicall Java\_MyClass\_nativeMethod**(**jnienv **\*,** jobject**);**  #ifdef \_ \_cplusplus  **}**  #endif  #endif |

Данный файл можно создать вручную или с помощью утилиты Javah, далее будет продемонстрирован более удобный способ генерации заголовочного файла.

Директива препроцессора #include <jni.h> включает файл jni.h в котором находятся все необходимые объявления типов и функций для реализации собственного метода.

Макросы jniexport и jnicall необходимы для платформы Windows, где они превращаются соответственно в \_\_declspec (dllexport) и \_\_stdcall и позволяют более эффективно строить DLL.

Имя С\С++ функции значительно отличается от имени собственного Java\Kotlin метода. Важным понятием при построении имени С\С++ функции и использовании JNI-функций является сигнатура метода (signature или method arguments signature).

За основу связывания библиотек, написанных на C и C++, и проекта на Java\Kotlin был взят SWIG (англ. simplified wrapper and interface generator), свободный инструмент для связывания программ и библиотек, написанных на языках C и C++, с интерпретируемыми или компилируемыми языками: Perl, Python, Ruby, PHP, Java, C#. Для использования SWIG необходимо создать файл \*.i с описанием экспортируемых функций; SWIG генерирует С\С++ код для связывания с нужным языком.

Интерфейсный файл использованный в ПМ ФО (emv.i):

|  |
| --- |
| **%**module**(**directors**=**"1"**)** emv  **%**javaconst**(**1**);**  **%**include "enumtypeunsafe.swg"  **%**include "std\_vector.i"  **%**include "std\_string.i"  **%**include "stdint.i"  **%**include "cpointer.i"  **%{**  #include "CData.h"  #include "CTypes.h"  #include "CCAKeys.h"  #include "CCryptoLayer.h"  #include "CErrors.h"  #include "CEMVTxn.h"  #include "CRevocationInfo.h"  #include "CTransport.h"  #include "CUserAction.h"  #include "CVerifyPIN.h"  #include "CCustomTags.h"  **%}**  **%**feature**(**"director"**)** CCryptoLayer**;**  **%**feature**(**"director"**)** CTransport**;**  **%**feature**(**"director"**)** CUserAction**;**  **%**feature**(**"director"**)** CVerifyPIN**;**  **namespace** std  **{**  **%**template**(**vector\_ca\_key**)** vector**<**emv**::**CCAKeys**::**ca\_key**>;**  **%**template**(**vector\_char**)** vector**<**char**>;**  **%**template**(**vector\_appinfo**)** vector**<**emv**::**CEMVTxn**::**SApplInfo**>;**  **%**template**(**vector\_revocation**)** vector**<**emv**::**CRevocationInfo**>;**  **%**template**(**vector\_string**)** vector**<**std**::**string**>;**  **};**  **%**include "CData.h"  **%**include "CCustomTags.h"  **%**pointer\_class**(**emv**::**CEMVTxn**::**CRYPTOGRAM\_TYPE**,** CryptogramType**);** |

В интерфейсном файле необходимо указать все заголовочные .h из которых будут извлечены методы и объекты для последующей обработки SWIG-ом. Так же для работы с некоторыми объектами языков С\С++, с которыми нельзя работать напрямую, необходимо сделать обертку в виде template**(output type\_name)** vector**<**input type\class**>**. Есть возможность создания указателей на класс pointer\_class**(**input class\type**,** output class\type**).**

Далее после написания интерфейсного файла необходимо выполнить следующую команду:

|  |
| --- |
| swig -java test.i |

После выполнения данной команды сгенерируются файлы \*.cxx и \*.java.

Далее необходимо скомпилировать исходный код библиотеки:

|  |
| --- |
| g++ -fPIC -c \*.cpp |

После этого скомпилировать файлы \*.cxx созданные SWIG с указанием, в нашем случае, путь к файлам Java - Java Development Kit.

|  |
| --- |
| g++ -fPIC -c test\_wrap.c –I …/jdk\*\*\*/include  -I …/ jdk\*\*\*/include/linux |

После выполнения данных команд, будут собраны объектных файла \*.o.

Последним шагом, слинкуем объектные файлы:

|  |
| --- |
| g++ -shared test\_wrap.o test.o -o libtest.so |

Итогом данных действий, будет являться общая библиотека (shared library) libtest.so.

Для того, чтобы использовать собранную библиотеку, нужно ее загрузить.

Как видно из примера, использование SWIG позволяет автоматизировать создание интерфейсных, связующих файлов, для последующего использования с помощью JNI.

Для автоматизации компиляции дополнительных библиотек и их конвертацию в необходимый вид, был написан скрипт на языке BASH.

|  |
| --- |
| #!/bin/bash -e  NAME**=**emv  PKI**=/**opt**/**pki-development**/**tt  GOAL**=**${1:-all}  CFG**=**${2:-release}  source **/**opt**/**devtools-1.0**/**devtools-env.sh  **export** make+**=**"make -j4"  **function** sign  **{**  **declare** -r ARCH**=**"$1"  **declare** -r RELEASE**=**"${RELEASES}/${ARCH}"  **declare** -r XML\_FILES**=(**  "${RELEASE}/candidates.xml"**)**  **declare** -A pki  pki**[**target**]=**"/opt/pki-development/tt"  pki**[**maxim**]=**"/opt/pki-development\_maxim/tt"  **declare** -A sign\_script  sign\_script**[**target**]=**"MAKE\_SIGN"  sign\_script**[**maxim**]=**"MAKE\_SIGN256"  CP ${SOURCES}**/**xml**/**candidates.xml ${RELEASE}**/**candidates.xml  FILL\_VERSION ${XML\_FILES[\*]}  **echo** ""  **echo** "Signing [${ARCH}] software..."  ${sign\_script[${ARCH}]} ${XML\_FILES[\*]} ${pki[${ARCH}]}  **}**  **function** build  **{**  **local** ARCH**=$1**  **local** RELEASE**=**${RELEASES}**/**${ARCH}  source arch-env.sh  source **/**opt**/**arm-linux-androideabi-26**/**setenv.sh  source **/**opt**/**x86\_64-linux-android-26**/**setenv.sh  COMPILE ${ARCH} ${CFG} ${GOAL}  **echo** ""  **echo** "Copying [${ARCH}] libraries..."  # copy libraries  MKDIR ${RELEASE}**/**lib  **find** ${SOURCES}**/**obj**/**${ARCH} **-**name '\*.so\*' **-**exec cp -d **{}** ${RELEASE}**/**lib \;  **find** ${SOURCES}**/**obj**/**${ARCH} **-**name '\*.a' **-**exec cp -d **{}** ${RELEASE}**/**lib \;  **}**  copy\_headers  copy\_jar  **cp** ${SOURCES}**/**version.mk ${RELEASES}**/**  MESSAGE "Build finished successfully" |

Данный скрипт использовал утилиту CMake для сборки библиотек EMV. Так же был написан необходимый для данной утилиты файл Makefile:

|  |
| --- |
| target maxim**:** $(OUTDIR)/$(OUTPUT).a $(OUTDIR)/$(OUTPUT).so.$(MAJOR)  android-arm android-x86 linux-i686 linux-x86\_64**:** $(JAVA\_JNI\_LIB)  $(OUTDIR)/$(OUTPUT).a**:** $(LIBFILES)  $(AR) cr $@ $(LIBFILES)  $(OUTDIR)/$(OUTPUT).so.$(MAJOR)**:** $(LIBFILES)  $(LD) -fpic -shared $(LDFLAGS) -o $@ $(LIBFILES)  $(RM) $(OUTDIR)/$(OUTPUT).so  @ln -s $(OUTPUT).so.$(MAJOR) $(OUTDIR)/$(OUTPUT).so  $(OUTDIR)/emv\_wrap.o**:** $(JAVA\_OUTDIR)/emv\_wrap.cxx Makefile  $(call COMPILE)  $(JAVA\_JNI\_LIB)**:** $(LIBFILES)  $(ECHO) "Compiling JAVA JNI dynamic library..."  $(MD) "`dirname $@`"  $(CXX) -fpic -shared $(LDFLAGS) -o "$@" $(LIBFILES) -lstdc++  $(JAVA\_OUTDIR)/emv\_wrap.cxx**:** Makefile  $(ECHO) "Generating JAVA files..."  $(MD) $(JAVA\_OUTDIR) $(JAVA\_GEN\_DIR) $(JAVA\_CLASS\_DIR)  @/opt/tt-swig-4.0/bin/swig -c++ -java -package $(JAVA\_PACKAGE) \  -MD -MP \  -directors \  -outdir $(JAVA\_GEN\_DIR) \  -Iinclude \  -o $(JAVA\_OUTDIR)/emv\_wrap.cxx \  swig/emv.i  $(MD) $(JAVA\_CLASS\_DIR)  $(ECHO) "Compiling JAVA files..."  @javac -d $(JAVA\_CLASS\_DIR) $(JAVA\_GEN\_DIR)/\*.java  $(ECHO) "Packing JAVA files to [$@]..."  @cd $(JAVA\_CLASS\_DIR) && jar cf $(SOURCES)/$(JAVA\_JAR\_FILE) ./com  $(OUTDIR)**:**  $(MD) $@  clean**:**  $(RM) -r $(OUTDIR)  $(RM) -r $(JAVA\_OUTDIR)  rebuild**:**  $(MAKE) clean  $(MAKE) all  .PHONY**:** all clean rebuild  -include $(LIBFILES**:**.o=.d)  -include $(JAVA\_OUTDIR)/emv\_wrap.d |

## Разработка интерфейса

Пользовательский интерфейс был разработан в среде Android Studio.

Экранные формы сделаны с учетом требований унификации:

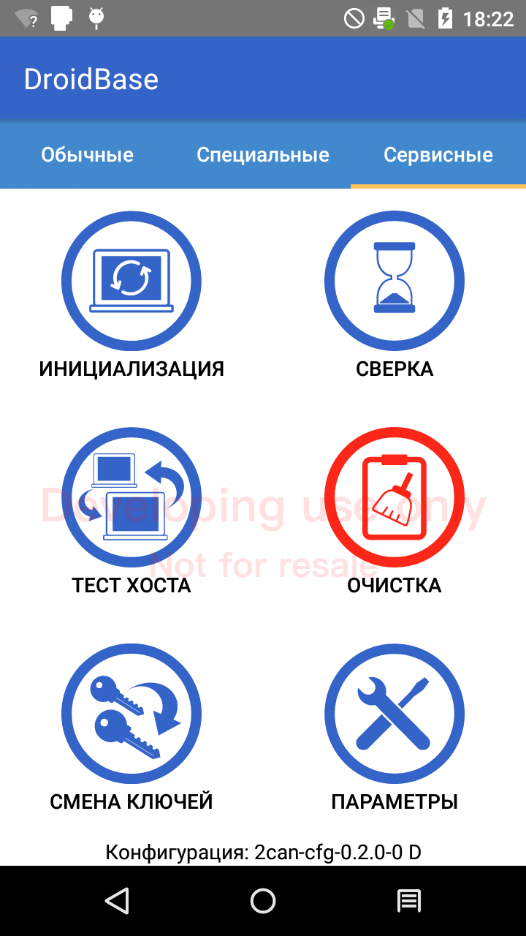
* экранные формы выполнены в едином графическом стиле, с одинаковым расположением основных элементов управления и навигации;
* для обозначения сходных операций используются сходные графические навигационные элементы;
* внешнее поведение сходных элементов интерфейса (нажатие на пункты меню) реализованы одинаково.

В основном формы состоят из 3 частей:

- значки операций

- навигационный бар

- нижний бар для управления переходами

****На рис 2.12 представлен главный экран. С данной экранной формы совершается выполнение транзакций.

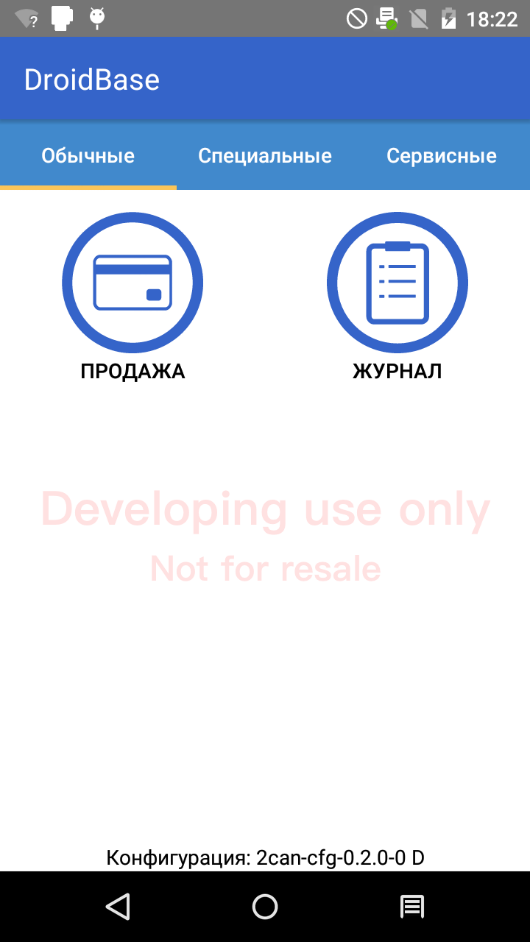
****

Рис. 2.12 Экранная форма Рис. 2.13 Экранная форма «Обычные операции» «Сервисные операции»

На рис. 2.13 представлена сервисная экранная форма. С данной формы проводятся различные сервисные операции, такие как очистка журнала, конфигурирование, проверка сервера банка, смена ключей безопасности терминала, сверка, а также переход в экранную форму параметров.

На рис. 2.14 представлена экранная форма конфигурации. В данной форме можно ввести необходимые данные для начала работы, сменить язык интерфейса, запустить сервис ТТК, а также выполнить синхронизацию ключей.

На рис. 2.15 можно посмотреть идентификационные параметры терминала:

* Идентификационный номер терминала
* Идентификационный номер продавца
* Имя продавца

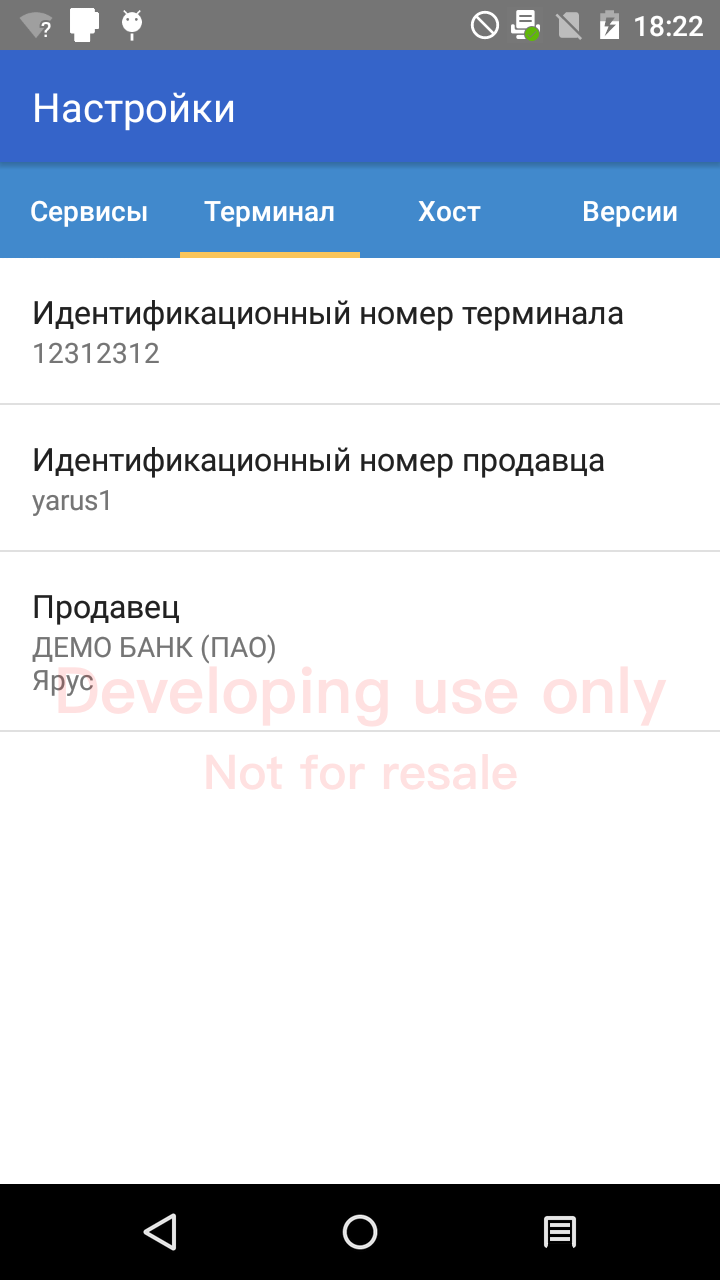
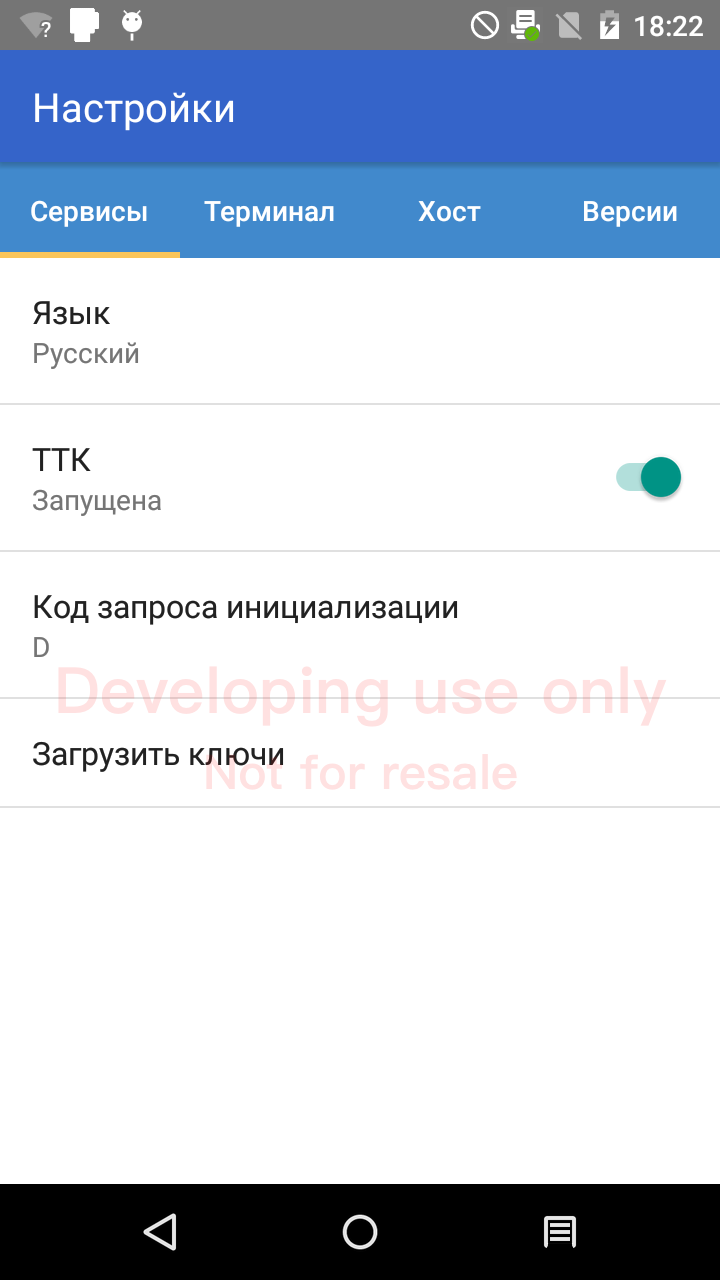


Рис. 2.14 Экранная форма «Сервисы» Рис. 2.15 Экранная форма «Терминал»

На рис. 2.16 представлена экранная форма настроек хостовой части. В данной экранной форме можно просмотреть настройки хостового модуля, его адрес, порт, название и включен или не включен SSL. Так же для некоторых видов хостов необходимо ввести логин с паролем.

На рис. 2.17 представлена экранная форма с информацией о версиях приложения, конфигурации, прошивки, а также серийного номера устройства и IMEI.

На рис. 2.18 представлена экранная форма журнала. Данная экранная форма, нужна для просмотра проведённых транзакций и при желании их возврата, отмены или печати копии чека.

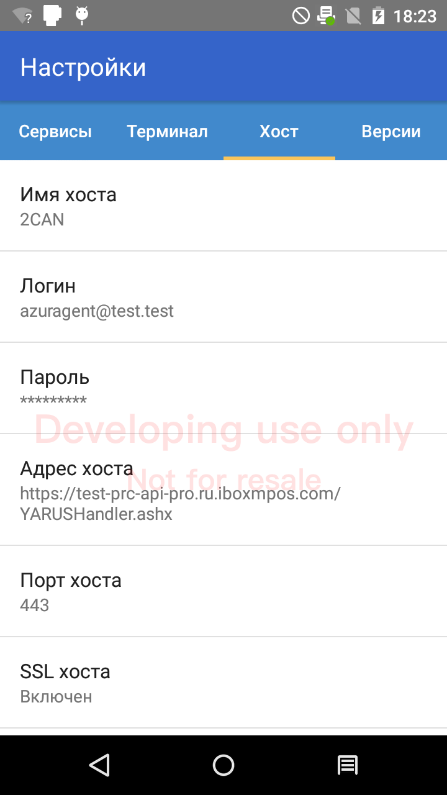
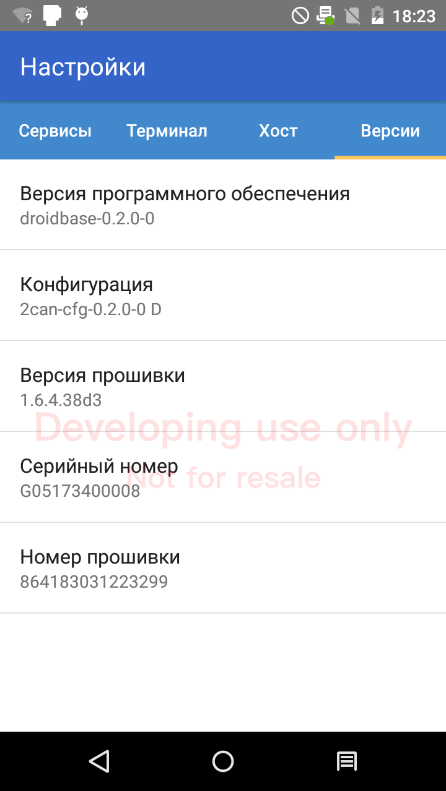
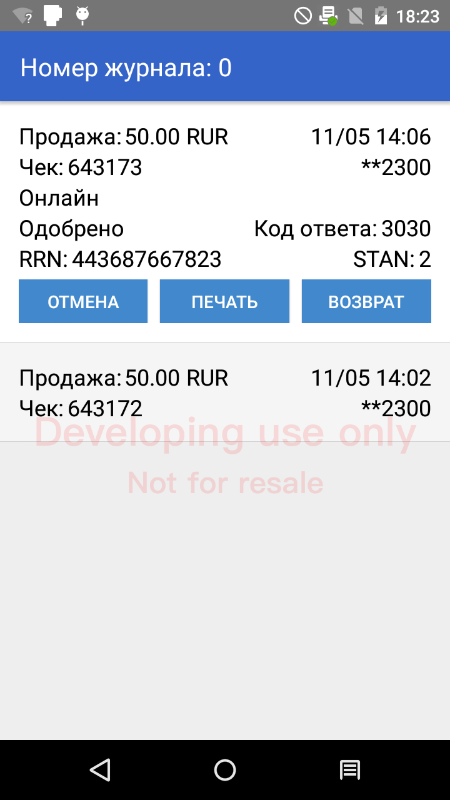


Рис. 2.16 Экранная форма Рис. 2.17 Экранная форма

«Хост» «Версии»

 На рис. 2.19 представлена экранная форма введения пин-кода. Она сделана специальным образом, чтобы нельзя было перехватит значения касаний и тем самым попытаться скомпрометировать пин-код.

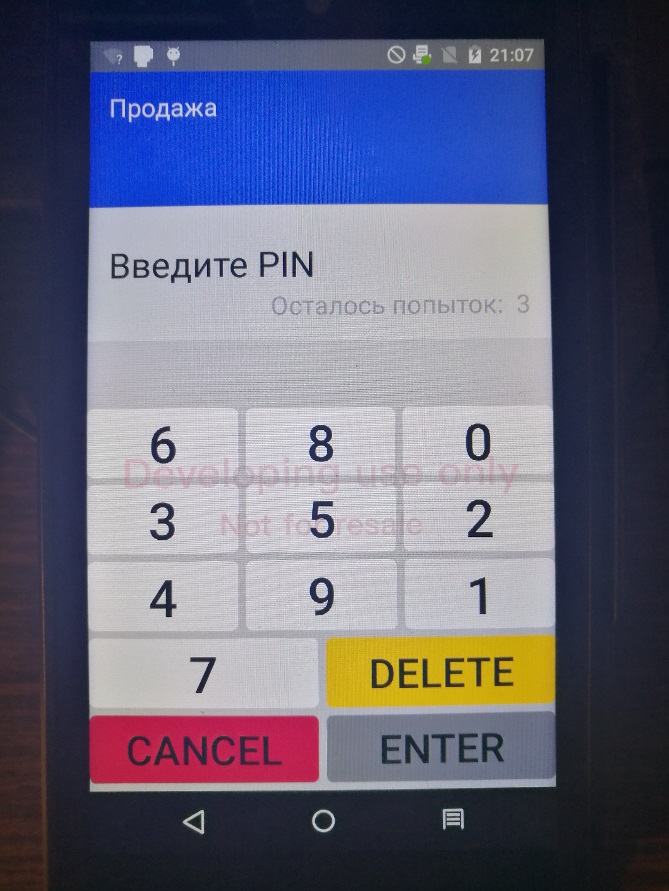


Рис. 2.18 Экранная форма «Журнал» Рис. 2.19 Экранная форма «Пин»

Интерфейс разрабатывался по средствам написания XML-разметки и последующих применений стилей.

Выводы по конструкторскому разделу

В ходе решения задач выпускной квалификационной работы в исследовательском разделе проведены следующие работы:

* выбран язык программирования;
* выбрана среда разработки для Java\Kotlin и C\C++;
* разработан алгоритм функционирования ПМ ФО;
* выполнена программная реализация алгоритма в виде программного модуля;
* произведена разработка пользовательского интерфейса ПМ ФО.

# Тестирование и отладка

## Отладка

Отладка - это процесс поиска и устранения дефектов или проблем в компьютерной программе, которые предотвращают правильную работу программного обеспечения или системы.

Тактика отладки может включать интерактивную отладку, анализ потока управления, модульное тестирование, интеграционное тестирование, анализ файлов журналов, мониторинг на уровне приложений или системы, дампы памяти и профилирование.

Обычно первым шагом в отладке является попытка воспроизвести проблему. Это может быть нетривиальная задача, например, как с параллельными процессами или с некоторыми необычными ошибками программного обеспечения. Кроме того, конкретная пользовательская среда и история использования могут затруднить воспроизведение проблемы.

После того, как ошибка будет воспроизведена, возможно, потребуется упростить ввод программы, чтобы упростить ее отладку. Например, ошибка в компиляторе может привести к сбою при анализе некоторого большого исходного файла. Однако, после упрощения тестового примера, только нескольких строк из исходного исходного файла может быть достаточным для воспроизведения одного и того же сбоя. Такое упрощение можно сделать вручную, используя подход «разделяй и властвуй». Программист попытается удалить некоторые части исходного тестового примера и проверить, существует ли проблема. При отладке проблемы в графическом интерфейсе программист может попытаться пропустить некоторое взаимодействие пользователя с исходным описанием проблемы и проверить, достаточны ли оставшиеся действия для появления ошибок.

После того, как тестовый пример достаточно упрощен, программист может использовать инструмент отладчика для проверки состояния программы (значения переменных, плюс стек вызовов) и отслеживать происхождение проблемы. В качестве альтернативы можно использовать трассировку. В простых случаях трассировка - это всего лишь несколько операторов печати, которые выводят значения переменных в определенных точках выполнения программы.

## Инструменты отладки

Android Studio предоставляет отладчик, который позволяет делать следующее и многое другое:

* Выбрать устройство для отладки приложения;
* Установить контрольные точки в коде Java, Kotli;
* Изучить переменные и проверить выражения во время выполнения.

Android Debug Bridge (adb) - это универсальный инструмент командной строки, который позволяет общаться с устройством. Команда adb облегчает различные действия устройства, такие как установка и отладка приложений, и обеспечивает доступ к оболочке Unix, которую можно использовать для запуска различных команд на устройстве. Это клиент-серверная программа, которая включает в себя три компонента:

* Клиент, который отправляет команды. Клиент работает на компьютере разработчика.
* Демон (adbd), который запускает команды на устройстве. Демон запускается как фоновый процесс на каждом устройстве.
* Сервер, который управляет связью между клиентом и демоном. Сервер работает как фоновый процесс на вашей машине разработки.

## Виды тестирования

Тестирование — необходимая деятельность для улучшения качества и стабильности программного обеспечения. Тестирование основывается на проведении тестов с конкретными входными данными, ожидаемыми выходными данными и начальными условиями.

Отладка — этап разработки, необходимый для выявления и устранения ошибок в программе, существование которых заранее известно. Суть подхода в том, чтобы вовремя исполнения программы следить за его поведением как с алгоритмической точки зрения, так и со стороны правильного значения переменных.

Для тестирования ПМ ФО использовались несколько подходов:

* модульное тестирование (Unit)
* интеграционного
* системного
* тестирование интерфейса

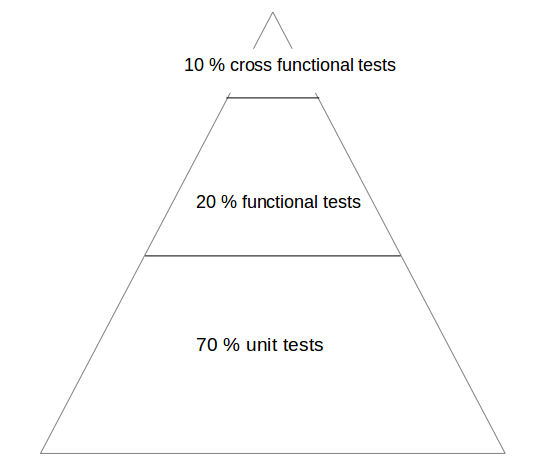
Сначала выполнялось тестирование модульное (Unit), затем выполнялось интеграционное и системное тестирование и последним этапом тестирование интерфейса.

Полное покрытие тестами, практически невозможно. Поэтому обычно выделяют те модули и сценарии, которые являются наиболее важными. Для разработки тест кейсов необходимо проделать следующие шаги:

1. Оценку требований;
2. Составление набора тестовых данных, как правильных, так и не правильных, для выявления проблем обработки таких ситуаций;
3. Составление точной последовательности действий;
4. Написание тест-кейса.

Тест должен быть сосредоточен на бизнес-логике приложения. Хорошим правилом является следующее распределение тестов:

* 70-80% модульных тестов для обеспечения стабильности вашего кода
* 20-30% функциональных тестов, чтобы убедиться, что приложение действительно работает
* некоторые кросс-функциональные тесты, если приложение интенсивно интегрируется с другими компонентами приложения.



**Метод “черного ящика”**

Метод “черного ящика” – метод тестирования, когда нам не доступен исходный код, а доступен только конечный интерфейс. Основная задача данного метода тестирования состоит в том, чтобы выявить несовпадения работы программы относительно ее технического задания.

Для данного типа тестирования используют следующие методы:

1. Разбиение на классы эквивалентности. Данное разбиение направлено на уменьшения количества одинаковых тестов, разбиение помогает выявить схожие части кода, и сделать для них как можно меньшее кол-во тестов.
2. Анализ граничных значений входных данных. Считается что граничные значения входных данных позволят быстрее найти неисправности в работе программы. Подходящими по данные условия считаются значения, находящиеся у верхней и нижней границы, определяющих границы класса эквивалентности.
3. Анализ поведения путем проверки выходных данных программы в зависимости от входных.

**Метод “белого ящика”**

Метод “белого ящика” – метод тестирования, когда доступен весь исходный код программы, и в связи с этим тестирование направлено на проверку работы как программы, так и библиотек, которые она использует. Данный метод позволяет проверить даже самые мелкие детали выполнения программы, за счет этого можно досконально проверить работоспособность программы.

Для данного метода выделяются следующие приемы тестирования:

1. Покрытие ветвлений (проверяется поведение при истинных и ложных значениях).
2. Покрытие условий (при наличии не покрытых операторов тестовые наборы дополняются таким образом, чтобы каждый оператор выполнялся хотя бы один раз).
3. Комбинаторное покрытие условий и решений (проверяются все комбинации условий и решений, причем каждый из них должен быть выполнен несколько раз).

**Тестирование производительности**

Тестирование производительности производится для определения пределов возможностей работы программы, а также ошибок, связанных с высокой нагрузкой. Так же тестирование производительности позволяет выявить низко производительные части программы, которые следует переделать.

Данный вид тестирования рассматривает такие вопросы как:

* уровень потребляемых ресурсов центральным и в нашем случае вспомогательным процессором;
* потребление оперативной памяти;
* время необходимое для выполнения/обработки запросов;
* работа с постоянной файловой системой (процессы записи и чтения).

**Unit тестирование**

Unit тестирование - это уровень тестирования программного обеспечения, в котором тестируются отдельные компоненты программного обеспечения. Цель состоит в том, чтобы подтвердить, что каждый компонент программного обеспечения выполняет свою роль, как было задумано. Компонент является самой маленькой тестируемой частью любого программного обеспечения. Обычно он имеет один или несколько входов и один выход. В процедурном программировании единица может быть индивидуальной программой, функцией, процедурой и т.д.

В объектно-ориентированном программировании наименьшая единица - это метод, который может принадлежать базовому/суперклассу, абстрактному классу или производному/дочернему классу. Рамочные модули тестирования, драйверы, заглушки и макеты / поддельные объекты используются для поддержки модульного тестирования.

Достоинства Unit-тестирования:

1. Процесс тестирования полностью автоматизирован. На выходе мы получаем полноценный отчет о пройденных и не пройденных тестах.
2. Высокий процент покрытия кода unit тестами обеспечивает высокое качество кода. Что означает, что, если код будет на 100% покрыт тестами, значит он на 100% правилен.
3. Возможность композиции тестов, для проверки сложных модулей, состоящих из нескольких модулей. Другими словами, можно проверять большие сложные модуля, по частям.
4. Простой рефакторинг кода. Unit тесты за счет своей простоты, позволяют проверять быстро части измененного кода.

## Выбор средств отладки и тестирования ПМ ФО

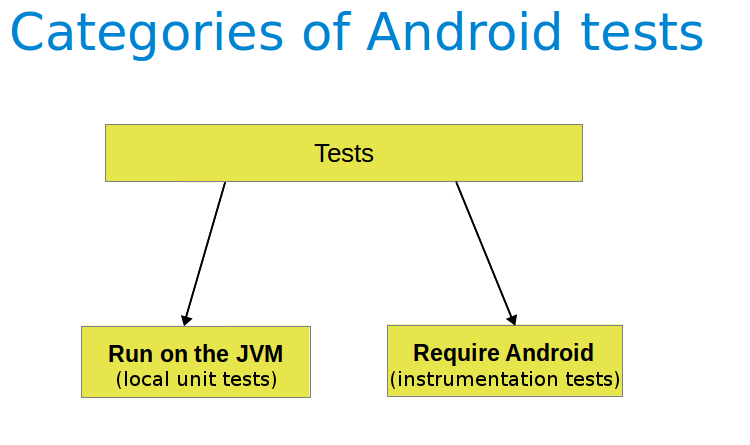
Модульные тесты - это фундаментальные тесты в стратегии тестирования приложений. Создавая и запуская модульные тесты против кода, можно легко проверить правильность логики отдельных блоков. Запуск модульных тестов после каждой сборки помогает быстро поймать и исправить программные регрессии, внесенные с помощью изменений кода в приложение.

Единичный тест обычно выполняет функциональность наименьшего возможного блока кода (который может быть методом, классом или компонентом) повторяемым образом. Необходимо построить модульные тесты, когда нужно проверить логику конкретного кода в вашем приложении. Например, если тестируется модуль класса, тест может проверить, что класс находится в правильном состоянии. Как правило, блок кода тестируется изолированно; тест влияет и контролирует изменения только этого устройства. Издевательская структура может использоваться для изоляции вашего устройства от его зависимостей.

Для тестирования приложений Android обычно создаются эти типы автоматизированных модульных тестов:

Локальные тесты: единичные тесты, которые выполняются только на локальной машине. Эти тесты скомпилированы для локального запуска на виртуальной машине Java (JVM) для минимизации времени выполнения. Данный метод используется для запуска модульных тестов, которые не имеют зависимости от платформы Android, или имеют зависимости, которые могут быть заполнены с помощью макетных объектов.

Инструментальные тесты: модульные тесты, которые запускаются на устройстве Android или эмуляторе. Эти тесты имеют доступ к инструментальной информации, такой как контекст для тестируемого приложения. Этот подход используется для запуска модульных тестов с зависимостями Android, которые не могут быть легко заполнены с помощью макированных объектов.



**Локальные тесты.**

Если модульный тест не имеет зависимостей или имеет только простые зависимости от Android, можно запустить свой тест на локальной машине разработки. Этот метод тестирования эффективен, поскольку он помогает избежать накладных расходов на загрузку целевого приложения и кода модульного теста на физическое устройство или эмулятор каждый раз, когда выполняется тест. Следовательно, время выполнения для тестирования вашего устройства значительно сокращается. При таком подходе обычно используется так называемое макирование, Mockito хороший инструмент, для выполнения любых замещений необходимых ресурсов, таких как к примеру работа с сетью.

В проекте Android Studio необходимо хранить исходные файлы для локальных тестов в модуле /src/test/java/. Этот каталог уже существует при создании нового проекта. Также необходимо настроить зависимости тестирования для проекта на использование стандартных API-интерфейсов, предоставляемых инфраструктурой JUnit. Если тест должен взаимодействовать с зависимостями Android, включите библиотеку Mockito, чтобы упростить локальные тесты.

Примеры тестов:

|  |
| --- |
| @Test  public void hex2bin\_vector\_char\_Test**()**  **{**  //input data  vector\_char in\_vchar **=** **new** vector\_char**();**  in\_vchar**.**add**(** 'A'**);**  in\_vchar**.**add**(** '0'**);**  in\_vchar**.**add**(** '0'**);**  in\_vchar**.**add**(** 'A'**);**  in\_vchar**.**add**(** '0'**);**  in\_vchar**.**add**(** 'A'**);**  in\_vchar**.**add**(** 'A'**);**  in\_vchar**.**add**(** '0'**);**  in\_vchar**.**add**(** 'A'**);**  in\_vchar**.**add**(** '0'**);**  in\_vchar**.**add**(** 'A'**);**  in\_vchar**.**add**(** 'A'**);**  in\_vchar**.**add**(** '0'**);**  //expected out  vector\_char check\_vchar **=** **new** vector\_char**();**  check\_vchar**.**add**((**char**)** 160**);**  check\_vchar**.**add**((**char**)** 10**);**  check\_vchar**.**add**((**char**)** 10**);**  check\_vchar**.**add**((**char**)** 160**);**  //out data  vector\_char out\_vchar **=** conv**.**hexvchar2bcdvchar**(**in\_vchar**);**  **for** **(**int i **=** 0**;** i**<**out\_vchar**.**size**();++**i**)**  **{**  assertEquals**(**check\_vchar**.**get**(**i**),** out\_vchar**.**get**(**i**));**  **}**  **}** |

В данном примере, создается объект vector\_char, специальный объект для использования строк C\C++ в Java, заполняется тестовыми данными. Далее создается еще один экземпляр вектора и заполняется проверяемыми (ожидаемыми) данными. После этого происходит вызов необходимой функции и проверка правильности возвращаемых данных.

Если запустить набор тестов, то Android Studio позволяет сделать отчеты о пройденных и не пройденных тестах с вставками выводов в виде HTML и XML. На рис. 3.1 представлен режим тестирования в Android Studio.

**Инструментальные тесты.**

API тестирования Android обеспечивает перехваты компонентов Android в жизненном цикле приложения. Эти перехваты называются API-интерфейсом инструментальных тестов и позволяют им контролировать жизненный цикл и события взаимодействия с пользователем.

В нормальных условиях приложение не может контролировать события жизненного цикла, и пользователь управляет приложением. Например, если Android создает активность, вызывается метод onCreate (). Или, если пользователь нажимает кнопку, вызывается соответствующий код. С помощью инструментария можно управлять этими событиями с помощью тестового кода. Например, контрольно-измерительный тест может начать работу. После этого он может вызвать финиш () и перезапустить действие, чтобы проверить, правильно ли восстановлено состояние экземпляра активности.

Инструментальные модульные тесты - это единичные тесты, которые запускаются на устройствах и эмуляторах Android, а не на виртуальной машине Java. Эти тесты имеют доступ к реальному устройству и его ресурсам и полезны для модульной тестовой функциональности, которую нелегко высмеивать из-за насмешливых фреймворков. Пример - это тест, который проверяет реализацию Parcelable.

Контрольный класс, основанный на инструментах, позволяет отправлять ключевые события (или события касания) в тестируемое приложение.

Пример инструментального теста:

|  |
| --- |
| @Test  public void doSettlementTest**()** **throws** Exception  **{**  LogToFile**.**writeLogCat**(new** Object**(){}.**getClass**().**getEnclosingMethod**().**getName**());**  UiDevice mDevice **=** UiDevice**.**getInstance**(**InstrumentationRegistry**.**getInstrumentation**());**  openActionBarOverflowOrOptionsMenu**(**getInstrumentation**().**getTargetContext**());**  onView**(**withText**(**"Service"**)).**perform**(**click**());**  Thread**.**sleep**(**1000**);**  onView**(**withText**(**"Settle"**)).**perform**(**click**());**  Thread**.**sleep**(**1000**);**  onView**(**withText**(**"OK"**)).**perform**(**click**());**  UiObject button\_ok\_amount **=** mDevice**.**findObject**(new** UiSelector**().**text**(**"OK"**));**  button\_ok\_amount**.**click**();**  Espresso**.**closeSoftKeyboard**();**  **}** |

## Тестирование интерфейса.

Для тестирования интерфейса были использованы следующие инструменты Espresso и UIAutomator. Данные инструменты позволяют описывать порядок действий и работу с данными, такую как ввод и проверка вывода в графическом интерфейсе.

Espresso - это фреймворк, который предоставляет простой API для тестирования пользовательского интерфейса программы. Начиная с версии 2.0, Espresso является частью Android Support Repository, что делает добавление Espresso в проект более легким.

От других фреймворков Espresso отличается следующим:

* API тестов Espresso выглядит как обычный английский текст, что позволяет быстро научиться работать с ним.
* Имеет маленький API
* Espresso быстро запускается
* Поддержка Gradle+Android Studio

Пример теста Espresso:

|  |
| --- |
| @Test  public void doTransaction10Test**()** **throws** Exception  **{**  LogToFile**.**writeLogCat**(new** Object**(){}.**getClass**().**getEnclosingMethod**().**getName**());**  mDevice **=** UiDevice**.**getInstance**(**getInstrumentation**());**  UiObject edit\_text\_amount **=** mDevice**.**findObject**(new** UiSelector**().**className**(**"android.widget.EditText"**));**  UiObject text\_view\_card **=** mDevice**.**findObject**(new** UiSelector**().**className**(**"android.widget.TextView"**));**  UiObject button\_ok\_amount **=** mDevice**.**findObject**(new** UiSelector**().**text**(**"OK"**));**  mDevice**.**click**(**300**,**300**);**  edit\_text\_amount**.**setText**(**"1000"**);**  assertTrue**(**button\_ok\_amount**.**click**());**  Thread**.**sleep**(**8000**);**  String text **=** text\_view\_card**.**getText**();**  Thread**.**sleep**(**5000**);**  assertArrayEquals**();**  onView**(**withText**().**perform**());**  Espresso**.**closeSoftKeyboard**();**  System**.**out**.**print**(**text**);**  assertEquals**(**"\uF05E"**,**text**);**  **}** |

UI Automator - это платформа для тестирования пользовательского интерфейса, подходящая для тестирования пользовательского интерфейса для разных приложений через установленные системы и установленные приложения. Более продвинутый инструмент для тестирования чем Expresso.

UI Automator предоставляет набор API для создания тестов пользовательского интерфейса, которые взаимодействуют с пользовательскими приложениями и системными приложениями. API-интерфейсы UI Automator позволяют выполнять такие операции, как открытие меню настроек или запуска приложения в тестовом устройстве. UI Automator хорошо подходит для написания автоматизированных тестов в черном ящике, где тестовый код не полагается на внутренние детали реализации целевого приложения.

К основным функциям платформы тестирования UI Automator относятся следующие:

* Средство просмотра для проверки иерархии макетов. Дополнительные сведения см. В разделе «Средство просмотра пользовательского интерфейса».
* API для извлечения информации о состоянии и выполнения операций на целевом устройстве. Дополнительные сведения см. В разделе Доступ к состоянию устройства.
* API, которые поддерживают тестирование пользовательского интерфейса кросс-приложения. Для получения дополнительной информации см. API-интерфейсы пользовательского интерфейса.

Структура тестирования UI Automator предоставляет класс UiDevice для доступа и выполнения операций на устройстве, на котором запущено целевое приложение. Можно вызвать его методы для доступа к свойствам устройства, таким как текущая ориентация или размер экрана. Класс UiDevice также позволяет выполнять такие действия, как:

* Изменить ориентацию устройства;
* Нажать кнопку приложения;
* Нажать кнопки «Назад», «Дом» или «Меню»;
* Открыть бар уведомлений;
* Сделать снимок экрана.

Например, чтобы имитировать нажатие кнопки «Домой», необходимо вызвать метод UiDevice.pressHome ().

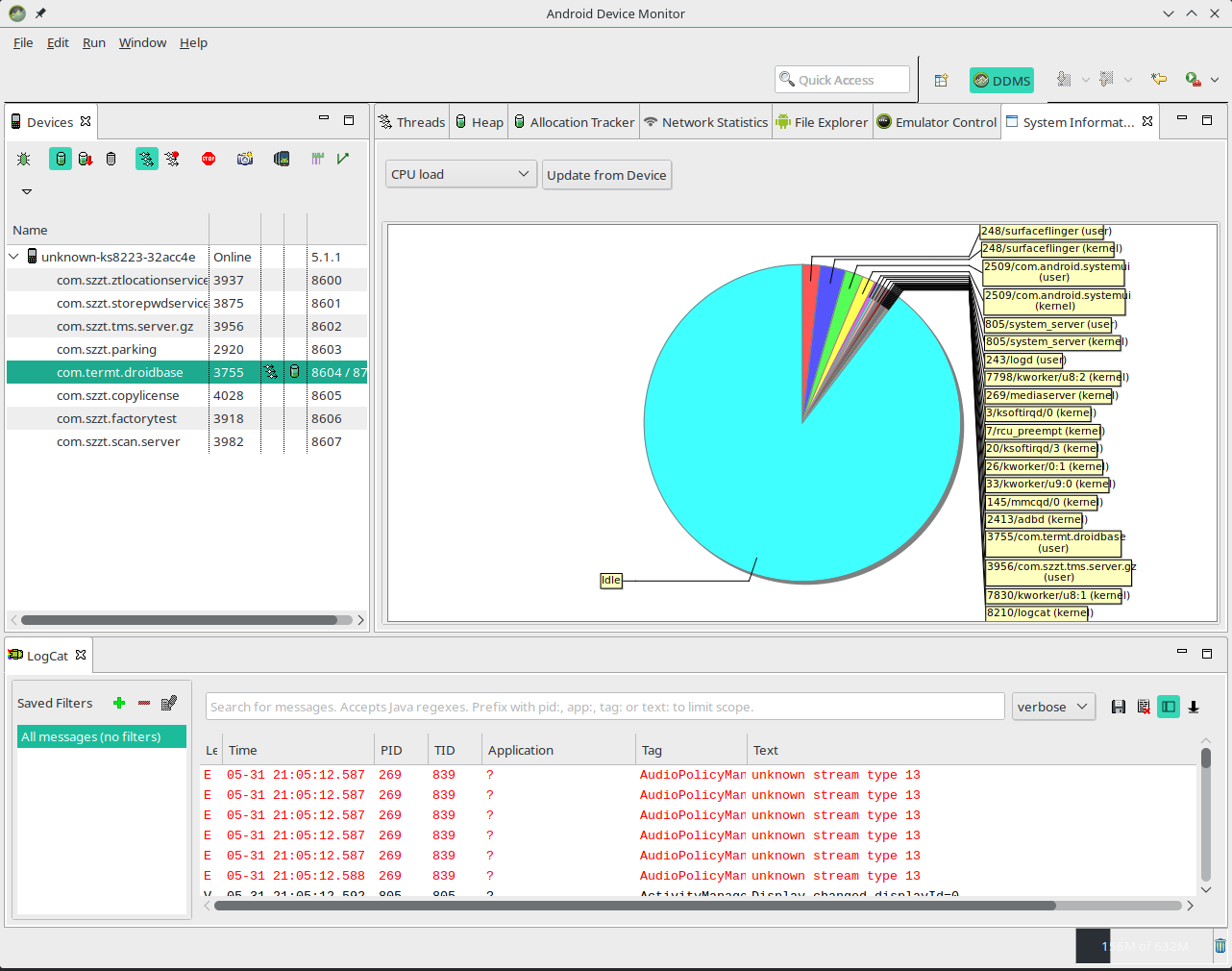
Пример теста UIAutomator:

|  |
| --- |
| static class PressAndHoldAction **implements** ViewAction  **{**  @Override  public Matcher**<**View**>** getConstraints**()** **{**  **return** isDisplayingAtLeast**(**90**);** // Like GeneralClickAction  **}**  @Override  public String getDescription**()** **{**  **return** "Press and hold action"**;**  **}**  @Override  public void perform**(**final UiController uiController**,** final View view**)** **{**  **if** **(**sMotionEventDownHeldView **!=** **null)** **{**  **throw** **new** AssertionError**(**"Only one view can be held at a time"**);**  **}**  float**[]** precision **=** Press**.**FINGER**.**describePrecision**();**  float**[]** coords **=** GeneralLocation**.**CENTER**.**calculateCoordinates**(**view**);**  sMotionEventDownHeldView **=** MotionEvents**.**sendDown**(**uiController**,** coords**,** precision**).**down**;**  // TODO: save view information and make sure release() is on same view  **}**  **}**  static class ReleaseAction **implements** ViewAction **{**  @Override  public Matcher**<**View**>** getConstraints**()** **{**  **return** isDisplayingAtLeast**(**90**);** // Like GeneralClickAction  **}**  @Override  public String getDescription**()** **{**  **return** "Release action"**;**  **}**  @Override  public void perform**(**final UiController uiController**,** final View view**)** **{**  **if** **(**sMotionEventDownHeldView **==** **null)** **{**  **throw** **new** AssertionError**(**"Before calling release(), you must call pressAndHold() on a view"**);**  **}**  float**[]** coords **=** GeneralLocation**.**CENTER**.**calculateCoordinates**(**view**);**  MotionEvents**.**sendUp**(**uiController**,** sMotionEventDownHeldView**,** coords**);**  **}**  **}** |

## Профайлинг ПМ ФО

Для получения дополнительной информации о работе программного модуля, был использован Dalvik Debug Monitor Server (рис. 3.1), монитор ресурсов входящий в состав Android Studio.

Данный монитор ресурсов, помог выявить утечки памяти и проблемы с обработкой работы с сетью и файлами.



Android Profiler в Android Studio 3.0 и выше заменяет инструменты Android Monitor. Эти новые инструменты профилирования предоставляют данные в реальном времени для процессора, памяти и сетевой активности вашего приложения. Можно выполнить трассировку метода на основе образца до момента выполнения кода, захватить дамп кучи(heap), просмотреть распределения памяти и просмотреть сведения о сетевой активности.

Выводы по технологическому разделу

В ходе решения задач выпускной квалификационной работы в исследовательском разделе проведены следующие работы:

* произведена отладка ПМ ФО;
* описаны методы тестирования;
* выбраны методы тестирования ПМ ФО;
* описаны инструменты для модульного тестирования ПМ ФО;
* описаны инструменты профайлинга ПМ ФО;
* произведено тестирование и корректирование кода ПМ ФО.

### СПИСОК ИСТОЧНИКОВ

1. Гагарина Л.Г., Касимов Р.А., Коваленко Д.Г., Федотова Е.Л., Чжо Зо Е, Черников Б.В., Методические указания по выполнению выпускной квалификационной работы по направлению подготовки бакалавров 09.03.04 «Программная инженерия»/ Под редакцией д.т.н. Черникова Б.В. МИЭТ, 2016.
2. Building Embedded Linux Systems, Karim Yaghour, Jon Masters, Gilad Ben-Yossef 2008
3. Число операций с банковскими картами за 2010 – 2016 год [Электронный ресурс]. URL: https://bcs-express.ru/novosti-i-analitika/dolia-beznalichnykh-raschetov-rastet
4. Credit Cards & Debit Cards: Your Guide to the Bankcard Industry and Best Practices 2010
5. ISO Magnetic Stripe Card Standards [Электронный ресурс]. URL: https://www.q-card.com/about-us/iso-magnetic-stripe-card-standards/page.aspx?id=1457
6. EMV Integrated Circuit Card Specifications for Payment Systems Book 1 2011
7. EMV Contactless Specifications for Payment Systems Book B 2016
8. Теснение банковских карт [Электронный ресурс]. URL: https://en.wikipedia.org/wiki/Payment\_card#Embossing
9. EMV Contactless Specifications for Payment Systems Book C-2 Kernel 2 Specification 2016
10. EMV Contactless Specifications for Payment Systems Book C-3 Kernel 3 Specification 2017
11. Apple Pay [Электронный ресурс]. URL: https://ru.wikipedia.org/wiki/Apple\_Pay
12. Google Pay [Электронный ресурс]. URL: https://en.wikipedia.org/wiki/Google\_Pay
13. Samsung Pay [Электронный ресурс]. URL:https://en.wikipedia.org/wiki/Samsung\_Pay
14. Mobile NFC Infrastructure Version 1.0 July 2012
15. Ingenico Telium TETRA OS [Электронный ресурс]. URL: https://apac.ingenico.com/smart-terminals/operating-system/telium-tetra
16. Verix V Operating System Programmer's Manual 2015
17. Azur Pos 01-Ф [Электронный ресурс]. URL: http://azurpos.ru/
18. Java [Электронный ресурс]. URL: http://www.oracle.com/technetwork/java/index.html
19. C# [Электронный ресурс]. URL: http://msdn.microsoft.com/ru-ru/vcsharp/default.aspx
20. C\C++ [Электронный ресурс]. URL: https://gcc.gnu.org/
21. Kotlin [Электронный ресурс]. URL: https://kotlinlang.org/
22. Xamarin [Электронный ресурс]. URL: https://www.xamarin.com/
23. The Java Virtual Machine Specification Java SE 8 Edition Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley 2015
24. Eclipse IDE [Электронный ресурс]. URL: https://www.eclipse.org/
25. Microsost Visual Studio IDE [Электронный ресурс]. URL: https://www.visualstudio.com/vs/android/
26. Android Studio IDE [Электронный ресурс]. URL: https://developer.android.com/
27. IntelliJ IDEA IDE [Электронный ресурс]. URL: https://www.jetbrains.com/idea/
28. Refactoring: Improving the Design of Existing Code 1-st edition Martin Fowler, Kent Back, John Brant
29. Pro Git 1st Edition Scot Chacoon 2017
30. Learning XML, 2nd Edition Creating Self-Describing Data By Erik Ray 2009
31. ISO 8583 [Электронный ресурс]. URL: https://www.iso.org/obp/ui/#iso:std:iso:8583:-1:ed-1:v1:en
32. Спецификация протокола ТТК. Терминальные Технологии.
33. EMV Contactless Specifications for Payment Systems Book 2 Security and Key Management 2011
34. EMV Contactless Specifications for Payment Systems Book 3 Application Specifiacation 2011
35. Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER)
36. SWIG documentation [Электронный ресурс]. URL: http://www.swig.org/Doc3.0/SWIGDocumentation.html
37. BSD License [Электронный ресурс]. URL: https://directory.fsf.org/wiki/License:BSD\_4Clause
38. The Java Native Interface Programmer’s Guide and SpecificationSheng Liang
39. ГОСТ 19.201-78. ЕСПД. Техническое задание. Требования к содержанию и оформлению. Москва, 1980.
40. ГОСТ 19.701-90. ЕСПД. Схемы алгоритмов, программ, данных и систем. Условные Обозначения и правила выполнения. Москва, 1992.
41. ГОСТ 19.505-79. ЕСПД. Руководство оператора. Требования к содержанию и оформлению. Москва, 1980.
42. ГОСТ 19.102-77. ЕСПД. Стадии разработки. Москва, 1980.
43. ГОСТ 19.002-80. ЕСПД. Схемы алгоритмов и программ. Правила выполнения. Москва, 1980.
44. ГОСТ 19.003-80. ЕСПД. Схемы алгоритмов и программ. Обозначения условные графические. Москва, 1980.
45. ГОСТ 19.106-78. ЕСПД. Общие требования к программным документам, выполненным печатным способом. Москва, 1980.
46. ГОСТ 19.101-77. ЕСПД. Виды программ и программных документов. Москва, 1980.
47. ГОСТ 19.404-79 Пояснительная записка. Требования к содержанию и оформлению. Москва, 1980.
48. ГОСТ 19.105-78 Общие требования к программным документам. Москва, 1980.
49. ГОСТ 19.401-78 Текст программы. Требования к содержанию и оформлению. Москва, 1980.
50. ГОСТ 19.104-78 Единая система программной документации. Основные надписи. Москва, 1980.
51. ГОСТ 19.103-77 Обозначения программ и программных документов. Москва, 1980.
52. ГОСТ 19.506-79 Описание языка. Требования к содержанию и оформлению. Москва, 1980.
53. ГОСТ 19.301-79 Программа и методика испытаний. Требования к содержанию и оформлению. Москва, 1980.
54. Ю.И. Галушкина, А.Н. Марьямов. Конспект лекций по дискретной математике - 2-е изд., испр. - М.: Айрис-пресс, 2008. – 176 с.
55. Колдаев В.Д. Численные методы и программирование, 2007.
56. Колдаев В.Д., С.А.Лупин Архитектура ЭВМ: Издательство Инфра-M, 2008.
57. Колдаев В.Д. Основы алгоритмизации и программирования: Учебное пособие / В.Д. Колдаев. Под ред. Гагариной Л.Г… – М.: ИД «ФОРУМ» – ИНФРА-М, 2006, 2009. – 416 с.
58. Гагарина Л.Г., Колдаев В.Д. Алгоритмы и структуры данных: Издательство Финансы и статистика, 2009г.
59. Гагарина Л.Г., Киселев Д.В., Федотова Е.Л. Разработка и эксплуатация автоматизированных информационных систем Москва 2007, 384 с.
60. Колдаев В.Д. Основы логического проектирования: Издательство Форум, Инфра-М,2011
61. Б.С. Лунин, В.А. Матвеев, М.А. Басараб “Волновой твердотельный гироскоп”, место издания Издательство "Радиотехника" Москва, ISBN 978-5-88070-381-4, 174 с, 2014г.