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

Кафедра Информатики и программного обеспечения вычислительных систем

Утверждаю

Зав. Кафедрой ИПОВС,

д.т.н., проф.\_\_\_\_\_\_\_Гагарина Л.Г.

«\_\_»\_\_\_\_\_\_\_\_\_2018 г.

Пояснительная записка на ВКР

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

Направление подготовки – 231000.62  
Квалификация – бакалавр

Руководитель выпускной работы:

доцент, канд. техн. наук \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/Федотов А.А./

Исполнитель:

студент гр. МП-45 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/Василиадис Я./

СОГЛАСОВАНО:  
Консультант от предприятия: ООО

“Терминальные Технологии” \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/Капранов А.В./

Москва, 2018

СОДЕРЖАНИЕ

СОДЕРЖАНИЕ 2  
СОКРАЩЕНИЯ 3  
ВВЕДЕНИЕ 4  
1. ИССЛЕДОВАТЕЛЬСКИЙ РАЗДЕЛ 6  
 1.1. Предварительные исследования 8  
 1.2. Виды банковских карт и технологий оплаты 7  
 1.3. Анализ существующих решений 13

2. КОНСТРУКТОРСКИЙ РАЗДЕЛ 16  
 2.1. Анализ языков программирования 12  
 2.2. Выбор среды разработки 18  
 2.3. Структура входных и выходных данных 22

2.4. Программная архитектура и алгоритм работы ПМ ФО 30

2.5. Программная реализация ПМ ФО 33  
 2.5.1 Применение SWIG и JNI …… 33

3. ТЕХНОЛОГИЧЕСКИЙ РАЗДЕЛ 16  
 3.1. Анализ языков программирования 12  
 3.2. Выбор среды разработки 18

СПИСОК ИСТОЧНИКОВ 43

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

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]. 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 она удовлетворяет потребность программного обеспечения для устройств нового класса, обеспечивающие быстрое расширение функциональности и увеличение скорости разработки.

Цель выполнения ВКР: разработка программного модуля для терминалов нового поколения, обеспечивающая оптимизацию процесса работы с банковской картой.

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:

* удаленное конфигурирование;
* поддержка магнитного, контактного и бесконтактного интерфейса;
* оплата при помощи телефона;
* база данных транзакций;

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.

1. ИССЛЕДОВАТЕЛЬСКИЙ РАЗДЕЛ

1.1 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) [2].

Рисунок 1.1 Число операций с банковскими картами



Рисунок 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.

1.2 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 цифро-буквенной информации, закодированной в виде штрихов;

Тиснение[8] — нанесение на поверхность пластиковой карты цифро-буквенной 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 цифро-буквенной 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).



Рисунок 1.1 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:



Рисунок 1.2 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.

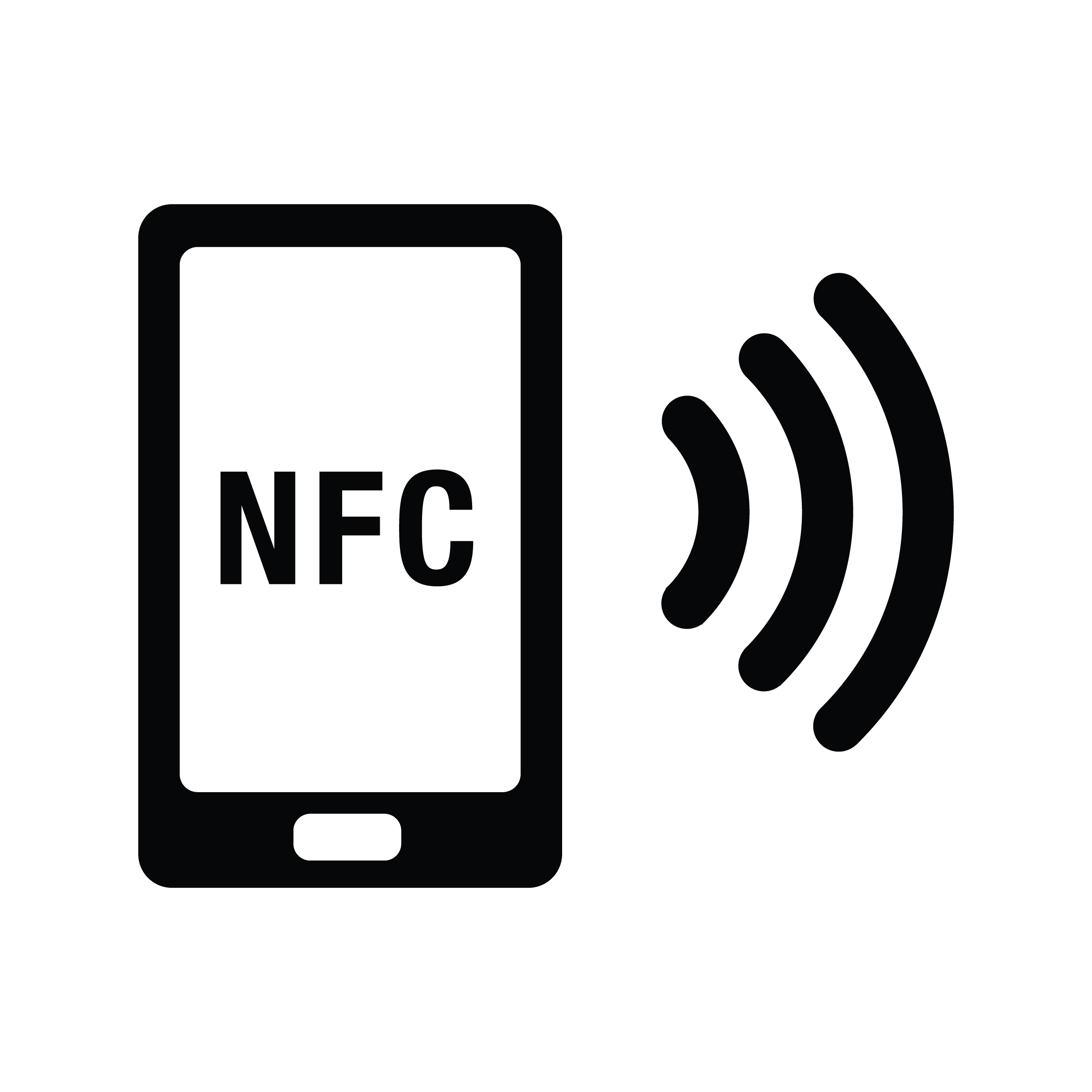


Рисунок 1.3 NFC смартфонах

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.

1.3 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 рис. 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 решениями компаний на рынке.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Название** | Ingenico iCT220 Dual | Verifone VX 680 | S920 PAX | Эвотор 5  (Кассовое решение) | Azur 01-Ф  (ПМ ФО) |
| 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 | - | - | - | - | + |

Ingenico iCT220 Dual – 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.

Verifone VX 680 – так же 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].

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.

Эвотор 5[17] – кассовое решение, рассматривается исключительно из-за того, что является устройством на ОС Android 5.1, стоимость порядка 18 000р. Для приема карт, подключается внешний POS-терминал с помощью COM порта.

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 контактный карт:

- поддержка EMV карт

1. Поддержка бесконтактных карт:

- поддержка EMV

- поддержка оплаты смартфоном

- поддержка Google Pay

- поддержка Samsung Pay

- поддержка Apple Pay

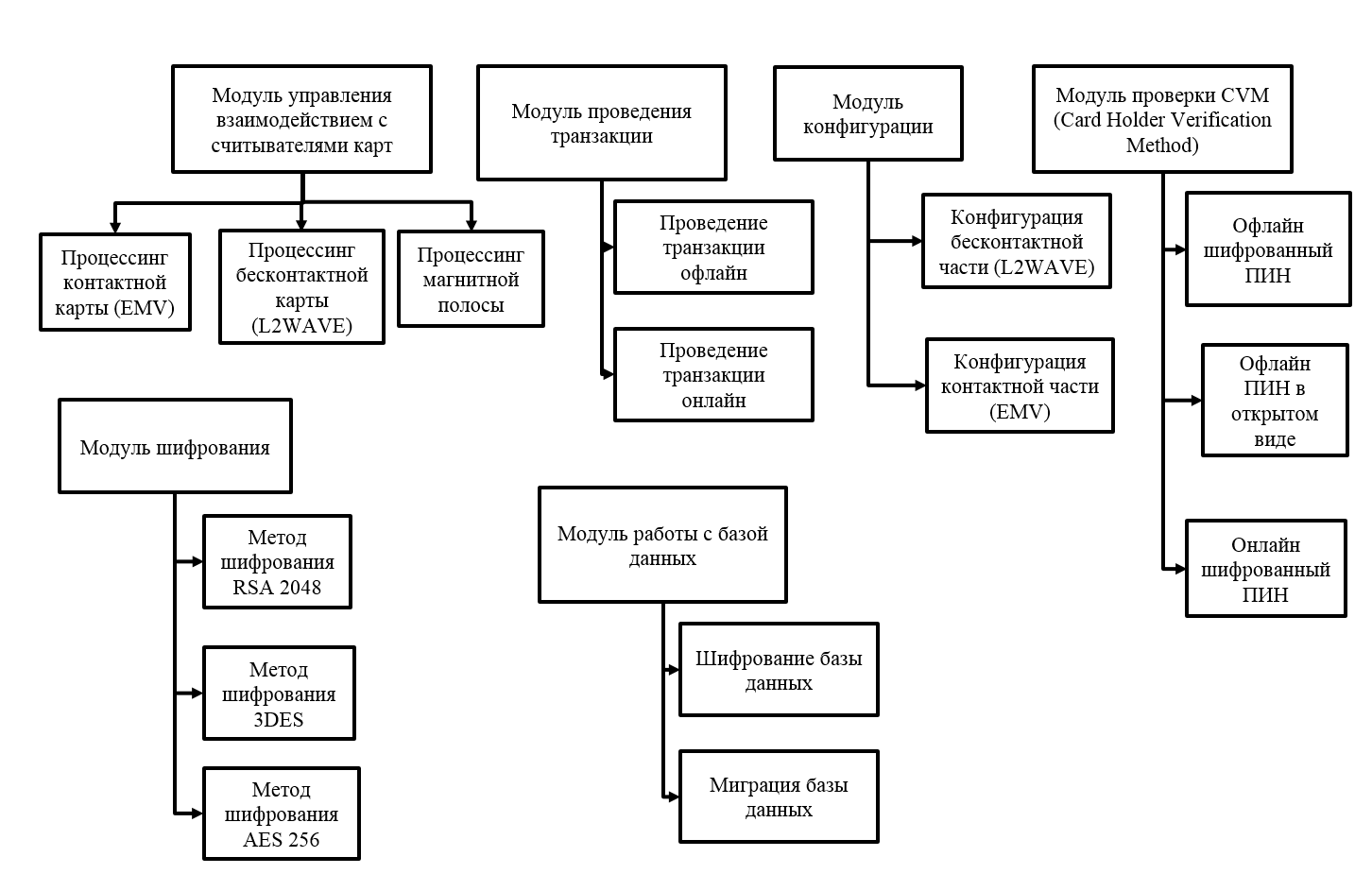
1. Поддержка магнитных карт:

- 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.



Конструкторская часть

2.1 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.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 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.

2.2 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.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Критерий | Eclipse | Microsoft Visual Studio | Android Studio | IntelliJ IDEA |
| 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 | - | + | + | + |

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.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 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 | - | + | + | + | - |

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

2.3 Объектно-ориентированное программирование

При разработке ПК ФО использовался объектно-ориентированный подход. Объектно-ориентированная парадигма программирования не нова. ООП (Объектно-ориентированное программирование) приобрело популярность во второй половине 80-х вместе с такими языками, как С++, Objective C (другое расширение C), Object Pascal и Turbo Pascal, CLOS (объектно-ориентированное расширение Lisp'a), Eiffel, Ada (в её последних воплощениях) и недавно — в Java. В этой статье внимание сосредоточено на C++, Object Pascal и Java, иногда упоминаются и другие языки.

Ключевые черты ООП хорошо известны:

1. Первая — **инкапсуляция** — это определение классов — пользовательских типов данных, объединяющих своё содержимое в единый тип и реализующих некоторые операции или методы над ним. Классы обычно являются основой модульности, инкапсуляции и абстракции данных в языках ООП.
2. Вторая ключевая черта, — наследование — способ определения нового типа, когда новый тип наследует элементы (свойства и методы) существующего, модифицируя или расширяя их. Это способствует выражению специализации и генерализации.
3. Третья черта, известная как полиморфизм, позволяет единообразно ссылаться на объекты различных классов (обычно внутри некоторой иерархии). Это делает классы ещё удобнее и облегчает расширение и поддержку программ, основанных на них.

Инкапсуляция, наследование и полиморфизм — фундаментальные свойства, которыми должен обладать язык, претендующий называться объектно-ориентированным (языки, не имеющие наследования и полиморфизма, но имеющие только классы, обычно называются основанными на классах). Различные ОО языки используют совершенно разные подходы. Мы можем различать ОО языки, сравнивая механизм контроля типов, способность поддерживать различные программные модели и то, какие объектные модели они поддерживают.

Алан Кей в свое время вывел пять основных черт объектно-ориентированного языка:

1. Все является объектом. Объект как хранит информацию, так и способен ее преобразовывать. В принципе любой элемент решаемой задачи (дом, собака, услуга, химическая реакция, город, космический корабль и т. д.) может представлять собой объект. Объект можно представить себе, как швейцарский нож: он является набором различных ножей и «открывашек» (хранение), но в то же самое время им мы можем резать или открывать что-либо (преобразование).
2. Программа — совокупность объектов, указывающих друг другу что делать. Для обращения к одному объекту другой объект «посылает ему сообщение». Как вариант возможно и «ответное сообщение». Программу можно представить себе, как совокупность к примеру 3 объектов: писателя, ручки и листа бумаги. Писатель «посылает сообщение» ручке, которая в свою очередь «посылает сообщение» листу бумаги — в результате мы видим текст (посыл сообщения от листа к писателю).
3. Каждый объект имеет свою собственную «память», состоящую из других объектов. Таким образом программист может скрыть сложность программы за довольно простыми объектами. К примеру, дом (достаточно сложный объект) состоит из дверей, комнат, окон, проводки и отопления. Дверь, в свою очередь, может состоять из собственно полотна, ручки, замка и петель. Проводка тоже состоит из проводов, розеток и, к примеру, щитка.
4. У каждого объекта есть тип. Иногда тип называют еще и классом. Класс (тип) определяет какие сообщения объекты могут посылать друг другу. Например, аккумуляторная батарея может передавать электролампе ток, а вот импульс или физическое усилие - нет.
5. Все объекты одного типа могут получать одинаковые сообщения. К примеру, у нас есть 2 объекта: синяя и красная кружки. Обе разные по форме и материалу. Но из обеих мы можем пить (или не пить, если они пустые). В данном случае кружка — это тип объекта.

Самое лаконичное описание объекта предложил Буч: «Объект обладает состоянием, поведением и индивидуальностью».

Языки программирования можно оценить по тому, насколько они строги к типам. Контроль типов включает проверку существования вызываемых методов, видов их параметров, проверку границ массивов и подобное.

C++, Java, и Object Pascal предпочитают более или менее тщательный контроль типов во время компиляции. С++, возможно, наименее точен в этом отношении (на что указывает, к примеру, возможность присвоения double к float), тогда как Java использует проверку типов наиболее широко. Это оттого, что C++ обеспечивает совместимость с Си, который не очень строго проверяет типы во время компиляции. Например, C и C++ считают, что все арифметические типы совместимы (хотя присвоение float целой переменной вызовет предупреждение компилятора). В Object Pascal и Java логическое значение не целое, а символ - еще один отличный и несовместимый тип.

Тот факт, что виртуальная машина Java интерпретирует байтовый код во время выполнения, не означает, что этот язык отказывается от проверки типов во время компиляции. Наоборот, в этом языке проверка наиболее тщательна. Другие ОО языки, такие как Smalltalk и CLOS, наоборот, склонны большинство проверок типов (если не все) осуществлять во время исполнения.

Как уже говорилось выше, в современных объектно-ориентированных языках программирования каждый объект является значением, относящимся к определённому классу. Класс представляет собой объявленный программистом составной тип данных, имеющий в составе:

Поля данных

Параметры объекта (конечно, не все, а только необходимые в программе), задающие его состояние (свойства объекта предметной области). Иногда поля данных объекта называют свойствами объекта, из-за чего возможна путаница. Фактически поля представляют собой значения (переменные, константы), объявленные как принадлежащие классу.

Методы

Процедуры и функции, связанные с классом. Они определяют действия, которые можно выполнять над объектом такого типа, и которые сам объект может выполнять.

Классы могут наследоваться друг от друга. Класс-потомок получает все поля и методы класса-родителя, но может дополнять их собственными либо переопределять уже имеющиеся. Большинство языков программирования поддерживает только единичное наследование (класс может иметь только один класс-родитель), лишь в некоторых допускается множественное наследование — порождение класса от двух или более классов-родителей. Множественное наследование создаёт целый ряд проблем, как логических, так и чисто реализационных, поэтому в полном объёме его поддержка не распространена. Вместо этого в 1990-е годы появилось и стало активно вводиться в объектно-ориентированные языки понятие интерфейса. Интерфейс — это класс без полей и без реализации, включающий только заголовки методов. Если некий класс наследует (или, как говорят, реализует) интерфейс, он должен реализовать все входящие в него методы. Использование интерфейсов предоставляет относительно дешёвую альтернативу множественному наследованию.

Взаимодействие объектов в абсолютном большинстве случаев обеспечивается вызовом ими методов друг друга.

Инкапсуляция обеспечивается следующими средствами:

Контроль доступа

Поскольку методы класса могут быть как чисто внутренними, обеспечивающими логику функционирования объекта, так и внешними, с помощью которых взаимодействуют объекты, необходимо обеспечить скрытость первых при доступности извне вторых. Для этого в языки вводятся специальные синтаксические конструкции, явно задающие область видимости каждого члена класса. Традиционно это модификаторы public, protected и private, обозначающие, соответственно, открытые члены класса, члены класса, доступные внутри класса и из классов-потомков, и скрытые, доступные только внутри класса. Конкретная номенклатура модификаторов и их точный смысл различаются в разных языках.

Методы доступа

Поля класса в общем случае не должны быть доступны извне, поскольку такой доступ позволил бы произвольным образом менять внутреннее состояние объектов. Поэтому поля обычно объявляются скрытыми (либо язык в принципе не позволяет обращаться к полям класса извне), а для доступа к находящимся в полях данным используются специальные методы, называемые методами доступа. Такие методы либо возвращают значение того или иного поля, либо производят запись в это поле нового значения. При записи метод доступа может проконтролировать допустимость записываемого значения и, при необходимости, произвести другие манипуляции с данными объекта, чтобы они остались корректными (внутренне согласованными). Методы доступа называют ещё аксессорами (от англ. access — доступ), а по отдельности — геттерами (англ. get — чтение) и сеттерами (set — запись).

Свойства объекта

Псевдополя, доступные для чтения и/или записи. Свойства внешне выглядят как поля и используются аналогично доступным полям (с некоторыми исключениями), однако фактически при обращении к ним происходит вызов методов доступа. Таким образом, свойства можно рассматривать как «умные» поля данных, сопровождающие доступ к внутренним данным объекта какими-либо дополнительными действиями (например, когда изменение координаты объекта сопровождается его перерисовкой на новом месте). Свойства, по сути, не более чем синтаксический сахар, поскольку никаких новых возможностей они не добавляют, а лишь скрывают вызов методов доступа. Конкретная языковая реализация свойств может быть разной. Например, в C# объявление свойства непосредственно содержит код методов доступа, который вызывается только при работе со свойствами, то есть не требует отдельных методов доступа, доступных для непосредственного вызова. В Delphi объявление свойства содержит лишь имена методов доступа, которые должны вызываться при обращении к полю. Сами методы доступа представляют собой обычные методы с некоторыми дополнительными требованиями к сигнатуре.

Полиморфизм реализуется путём введения в язык правил, согласно которым переменной типа «класс» может быть присвоен объект любого класса-потомка её класса.

2.4 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 отличие от готовых функций или библиотек, паттерн нельзя просто взять и скопировать в программу. Паттерн представляет собой не какой-то конкретный код, а общую концепцию решения той или иной проблемы, которую нужно будет ещё подстроить под нужды вашей программы.

Паттерны часто путают с алгоритмами, ведь оба понятия описывают типовые решения каких-то известных проблем. Но если алгоритм — это чёткий набор действий, то паттерн — это высокоуровневое описание решения, реализация которого может отличаться в двух разных программах.

Если привести аналогии, то алгоритм — это кулинарный рецепт с чёткими шагами, а паттерн — инженерный чертёж, на котором нарисовано решение, но не конкретные шаги его реализации.

Описания паттернов обычно очень формальны и чаще всего состоят из таких пунктов:

* проблема, которую решает паттерн;
* мотивации к решению проблемы способом, который предлагает паттерн;
* структуры классов, составляющих решение;
* примера на одном из языков программирования;
* особенностей реализации в различных контекстах;
* связей с другими паттернами.

Такой формализм в описании позволил создать обширный каталог паттернов, проверив каждый из них на состоятельность.

Паттерны отличаются по уровню сложности, детализации и охвата проектируемой системы. Проводя аналогию со строительством, вы можете повысить безопасность перекрёстка, поставив светофор, а можете заменить перекрёсток целой автомобильной развязкой с подземными переходами.

Самые низкоуровневые и простые паттерны — *идиомы*. Они не универсальны, поскольку применимы только в рамках одного языка программирования.

Самые универсальные — *архитектурные паттерны*, которые можно реализовать практически на любом языке. Они нужны для проектирования всей программы, а не отдельных её элементов.

Кроме того, паттерны отличаются и предназначением. В этой книге будут рассмотрены три основные группы паттернов:

* Порождающие паттерны беспокоятся о гибком создании объектов без внесения в программу лишних зависимостей.
* Структурные паттерны показывают различные способы построения связей между объектами.
* Поведенческие паттерны заботятся об эффективной коммуникации между объектами.

Концепцию паттернов впервые описал Кристофер Александер в книге «Язык шаблонов. Города. Здания. Строительство». В книге описан «язык» для проектирования окружающей среды, единицы которого — шаблоны (или паттерны, что ближе к оригинальному термину patterns) — отвечают на архитектурные вопросы: какой высоты сделать окна, сколько этажей должно быть в здании, какую площадь в микрорайоне отвести под деревья и газоны.

Идея показалась заманчивой четвёрке авторов: Эриху Гамме, Ричарду Хелму, Ральфу Джонсону, Джону Влиссидесу. В 1995 году они написали книгу «Приемы объектно-ориентированного проектирования. Паттерны проектирования», в которую вошли 23 паттерна, решающие различные проблемы объектно-ориентированного дизайна. Название книги было слишком длинным, чтобы кто-то смог всерьёз его запомнить. Поэтому вскоре все стали называть её «book by the gang of four», то есть «книга от банды четырёх», а затем и вовсе «GOF book».

С тех пор были найдены десятки других объектных паттернов. «Паттерновый» подход стал популярен и в других областях программирования, поэтому сейчас можно встретить всевозможные паттерны и за пределами объектного

В ПМ ФО были использованы следующие паттерны:

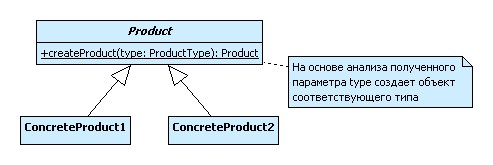
* Фабричный метод
* Абстрактная фабрика
* Адаптер
* Строитель
* Одиночка
* Заместитель
* Наблюдатель

Паттерн Фабричный метод (Factory Method)

Это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.

В системе часто требуется создавать объекты самых разных типов. Паттерн Factory Method может быть полезным в решении следующих задач:

* Система должна оставаться расширяемой путем добавления объектов новых типов. Непосредственное использование выражения new является нежелательным, так как в этом случае код создания объектов с указанием конкретных типов может получиться разбросанным по всему приложению. Тогда такие операции как добавление в систему объектов новых типов или замена объектов одного типа на другой будут затруднительными (подробнее в разделе Порождающие паттерны). Паттерн Factory Method позволяет системе оставаться независимой как от самого процесса порождения объектов, так и от их типов.
* Заранее известно, когда нужно создавать объект, но неизвестен его тип.

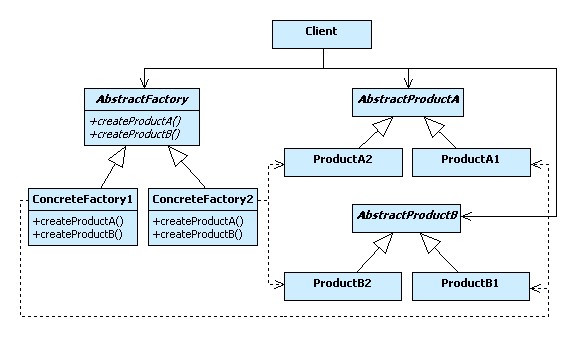


Абстрактная фабрика (Abstract Fabric)

В ПМ ФО данный паттерн был необходим так как:

Система должна оставаться независимой как от процесса создания новых объектов, так и от типов порождаемых объектов. Непосредственное использование выражения new в коде приложения нежелательно (подробнее об этом в разделе Порождающие паттерны).

Необходимо создавать группы или семейства взаимосвязанных объектов, исключая возможность одновременного использования объектов из разных семейств в одном контексте.



Одиночка(Singleton)

Назначение паттерна Singleton

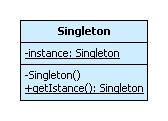
Часто в системе могут существовать сущности только в единственном экземпляре, например, система ведения системного журнала сообщений или драйвер дисплея. В таких случаях необходимо уметь создавать единственный экземпляр некоторого типа, предоставлять к нему доступ извне и запрещать создание нескольких экземпляров того же типа.

Паттерн Singleton предоставляет такие возможности.

Описание паттерна Singleton

Архитектура паттерна Singleton основана на идее использования глобальной переменной, имеющей следующие важные свойства:

1. Такая переменная доступна всегда. Время жизни глобальной переменной - от запуска программы до ее завершения.
2. Предоставляет глобальный доступ, то есть, такая переменная может быть доступна из любой части программы.



Паттерн строитель.

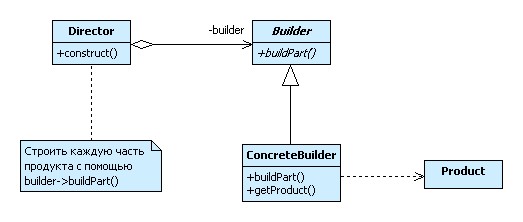
Назначение паттерна Builder

Паттерн Builder может помочь в решении следующих задач:

* В системе могут существовать сложные объекты, создание которых за одну операцию затруднительно или невозможно. Требуется поэтапное построение объектов с контролем результатов выполнения каждого этапа.
* Данные должны иметь несколько представлений. Приведем классический пример. Пусть есть некоторый исходный документ в формате RTF (Rich Text Format), в общем случае содержащий текст, графические изображения и служебную информацию о форматировании (размер и тип шрифтов, отступы и др.). Если этот документ в формате RTF преобразовать в другие форматы (например, Microsoft Word или простой ASCII-текст), то полученные документы и будут представлениями исходных данных.

Описание паттерна Builder

Паттерн Builder отделяет алгоритм поэтапного конструирования сложного продукта (объекта) от его внешнего представления так, что с помощью одного и того же алгоритма можно получать разные представления этого продукта.

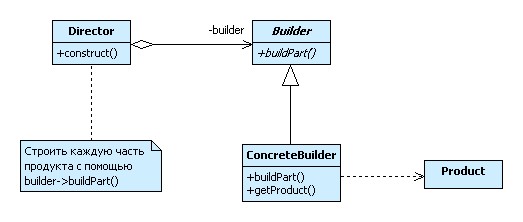


Паттерн адаптер.

Паттерн Adapter, представляет собой программную обертку над существующими классами, преобразует их интерфейсы к виду, пригодному для последующего использования.

Пусть класс, интерфейс которого нужно адаптировать к нужному виду, имеет имя Adaptee. Для решения задачи преобразования его интерфейса паттерн Adapter вводит следующую иерархию классов:

* Виртуальный базовый класс Target. Здесь объявляется пользовательский интерфейс подходящего вида. Только этот интерфейс доступен для пользователя.
* Производный класс Adapter, реализующий интерфейс Target. В этом классе также имеется указатель или ссылка на экземпляр Adaptee. Паттерн Adapter использует этот указатель для перенаправления клиентских вызовов в Adaptee. Так как интерфейсы Adaptee и Target несовместимы между собой, то эти вызовы обычно требуют преобразования.



Паттерн заместитель.

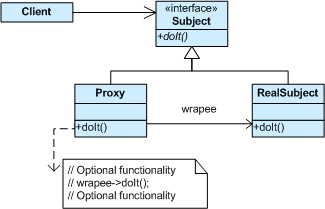
Назначение паттерна Proxy

* Паттерн Proxy является суррогатом или замеcтителем другого объекта и контролирует доступ к нему.
* Предоставляя дополнительный уровень косвенности при доступе к объекту, может применяться для поддержки распределенного, управляемого или интеллектуального доступа.
* Являясь "оберткой" реального компонента, защищает его от излишней сложности.

Решаемая проблема

Вам нужно управлять ресурсоемкими объектами. Вы не хотите создавать экземпляры таких объектов до момента их реального использования.

Заместитель Proxy и реальный объект RealSubject имеют одинаковые интерфейсы класса Subject, поэтому заместитель может использоваться "прозрачно" для клиента вместо реального объекта.



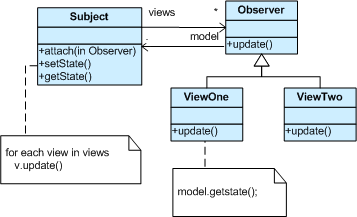
Паттерн наблюдатель.

Назначение паттерна Observer

* Паттерн Observer определяет зависимость "один-ко-многим" между объектами так, что при изменении состояния одного объекта все зависящие от него объекты уведомляются и обновляются автоматически.
* Паттерн Observer инкапсулирует главный (независимый) компонент в абстракцию Subject и изменяемые (зависимые) компоненты в иерархию Observer.
* Паттерн Observer определяет часть "View" в модели Model-View-Controller (MVC) .

Решаемая проблема

Имеется система, состоящая из множества взаимодействующих классов. При этом взаимодействующие объекты должны находиться в согласованных состояниях. Вы хотите избежать монолитности такой системы, сделав классы слабо связанными (или повторно используемыми).



2.3 Структура входных и выходных данных

Общая схема входных и выходных данных приведена на рис. 1.4.

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.

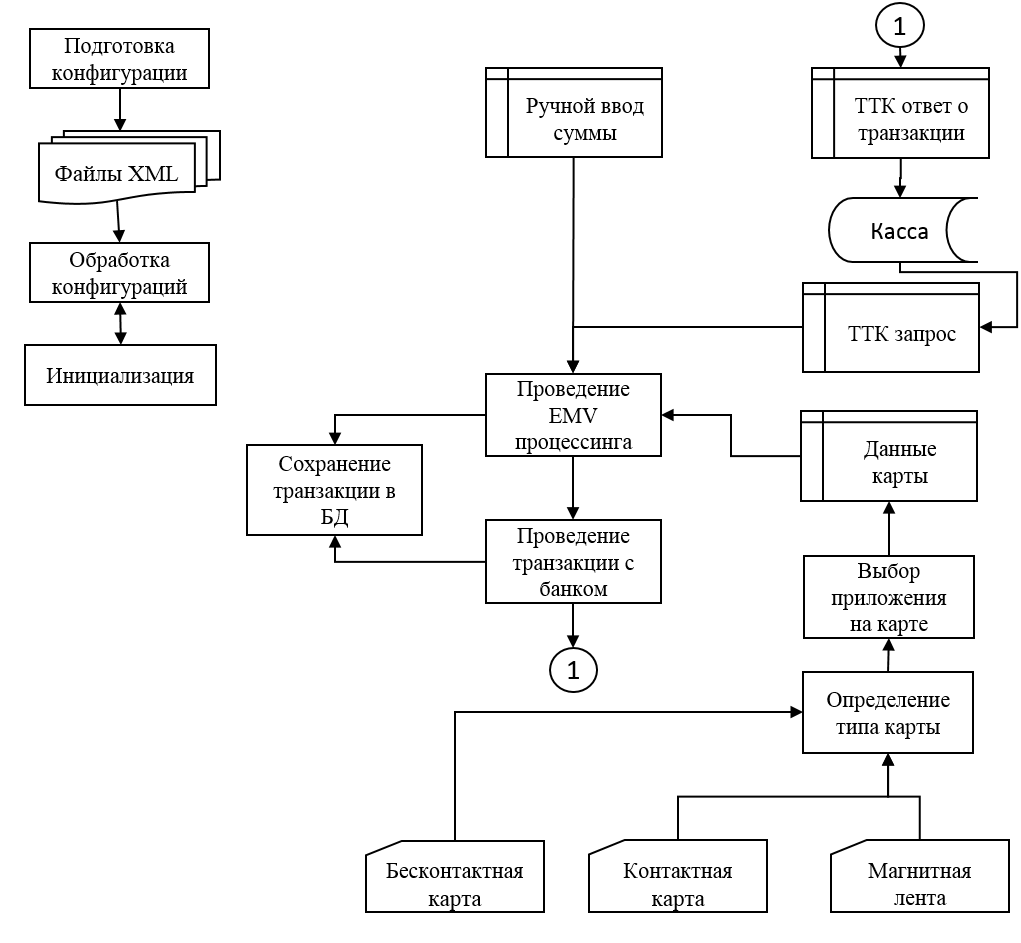


Рисунок 1.4 Общая схема данных ПМ ФО

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"

final\_screen\_timeout="4"

input\_caps="E0"

merchant\_id="XXXXXXX"

merchant\_name = "XXXXXXX"

remove\_timeout="10"

merchant\_category="5499"

wait\_timeout="25"

allow\_expired\_cards="0"

allow\_zero\_amount="0"

ttk\_enabled="0"

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 KeyLoadingDevice(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="DD50BСA000"

TAC\_denial="00BBAB0000"

TAC\_online="FE50BСF800"

mandatory\_data="5A5AAAAC8D"

threshold\_value="000000000000"

target\_percent="00"

max\_target\_percent="00"

compare\_AID\_length="0"

default\_DDOL="9F3704"

default\_TDOL="9F0AC56629A039C0195059F3704"

offline\_PIN\_success\_required="0"

security\_caps="C8"

send\_arqc\_to\_host="0"

treat\_Z3\_as\_approve="0"

versions="0002"

issuer\_name="MASTERCARD"/>

……..

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 (часть конфигурации относящаяся к картам МИР):

<!-- Custom group for MIR-->

<Group group\_id="3">

<Terminal\_Floor\_Limit>999999999999</Terminal\_Floor\_Limit>

<Terminal\_Contactless\_Floor\_Limit>999999999999</Terminal\_Contactless\_Floor\_Limit>

<Terminal\_Contactless\_Transaction\_Limit>000001000000</Terminal\_Contactless\_Transaction\_Limit>

<Terminal\_CVM\_Required\_Limit>999999999999</Terminal\_CVM\_Required\_Limit>

<Tag number="9F09" type="b">0101</Tag>

<Tag number="9F71" type="b">E3E0</Tag>

<Tag number="DF51" type="n">000000100000</Tag> <!-- Terminal Floor Limit-->

<Tag number="DF52" type="n">000000100000</Tag> <!-- Terminal No CVM Limit-->

<Tag number="DF53" type="n">000001000000</Tag> <!-- Terminal Contactless Limit (Non CD-CVM)-->

<Tag number="DF54" type="n">000001000000</Tag> <!-- Terminal Contactless Limit (CD-CVM)-->

<Tag number="DF55" type="b">6800</Tag> <!-- Terminal TPM Capabilities-->

<Tag number="DF56" type="b">03</Tag> <!-- Transaction Recovery Limit-->

<Tag number="FF01" type="b">B640000000</Tag> <!-- TAC - denial-->

<Tag number="FF02" type="b">B640808000</Tag> <!-- TAC - online-->

<Tag number="FF03" type="b">FFFFFFFFFF</Tag> <!-- TAC - default-->

<Tag number="FF04" type="b">5A</Tag> <!-- Data Exchange Tag List-->

</Group>

<!-- AID CONFIGURATIONS -->

<!-- Mastercard paypass -->

<Application AID="A0000000041010" group\_id="1" kernel="2" issuer\_name="MASTERCARD">

<Flags>20</Flags>

<!-- Kernel Configuration -->

<Tag number="DF811B" type="b">20</Tag>

<Tag number="9F1D" type="b">2C3A000000000000</Tag>

</Application>

<!-- Maestro paypass -->

<Application AID="A0000000043060" group\_id="4" kernel="2" issuer\_name="MAESTRO">

<Flags>00</Flags>

<!-- Kernel Configuration -->

<Tag number="DF811B" type="b">A0</Tag>

<Tag number="9F1D" type="b">0C12800000000000</Tag>

</Application>

<!-- Visa Debit/Credit PayWave -->

<Application AID="A0000000031010" group\_id="2" kernel="3" issuer\_name="VISA">

</Application>

<!-- Visa Electron PayWave -->

<Application AID="A0000000032010" group\_id="2" kernel="3" issuer\_name="VISA">

</Application>

<!-- MIR -->

<Application AID="A0000006581010" kernel="643" group\_id="3">

<Flags>00</Flags>

<Tag number="DF811B" type="b">A0</Tag>

</Application>

<Application AID="A0000006582010" 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>

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 index="05">

<Modulus>B8048ABC30C90D976336543E3FD7091C8FE4800DF820ED55E7E94813ED00555B573FECA3D84AF6131A651D66CFF4284FB13B635EDD0EE40176D8BF04B7FD1C7BACF9AC7327DFAA8AA72D10DB3B8E70B2DDD811CB4196525EA386ACC33C0D9D4575916469C4E4F53E8E1C912CC618CB22DDE7C3568E90022E6BBA770202E4522A2DD623D180E215BD1D1507FE3DC90CA310D27B3EFCCD8F83DE3052CAD1E48938C68D095AAC91B5F37E28BB49EC7ED597</Modulus>

<Exponent>03</Exponent>

<Hash>EBFA0D5D06D8CE702DA3EAE890701D45E274C845</Hash>

<Expire>241231</Expire>

</Key>

<Key index="06">

<Modulusodulus>

<Exponent>03</Exponent>

<Hash>F910A1504D5FFB793D94F3B500765E1ABCAD72D9</Hash>

<!-- expiration date YYMMDD -->

<Expire>371215</Expire>

</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.

TLV (Tag-length-value) — широко распространённый метод записи коротких данных в компьютерных файлах и телекоммуникационных протоколах.

Метод определяет простую двоичную структуру из трёх полей: тег, длина данных и собственно данные. Первые два поля имеют фиксированную длину (обычно один или два октета на поле), длина третьего поля определяется значением второго поля (значение указывается в байтах). Тег является идентификатором данных, определяя их назначение.

BER - набор правил кодирования бинарных данных

BER (basic encoding rules)[36] - формат на принципе TLV, используется для кодирования и сериализации бинарных данных. Формат предлагается в качестве примера реализации принципа TLV.

Типы. Типы, в частности поле "тип", кодируются одним или несколькими байтами. В первом байте три старшие бита выделены для классификации идентификатора типа: базовый, прикладной, контекстно-зависимый, личный (private); тип может быть составным или примитивным. Для представления собственно идентификатора типа используется пять младших бит в байте. Если требуется закодировать идентификатор со значением более 30, то первый байт в поле идентификатора содержит число 31, а значение кодируется в последующих байтах.

Длина. Поле кодируется одним или несколькими байтами. Одним байтом кодируется длина от 0 до 127. Большая длина кодируется несколькими байтами. При разборе формата, если в старшем бите первого байта длины указана 1, то длина кодируется несколькими байтами, в битах 7-1 первого байта содержится количество последующих байт, в которых закодирована длина.

Строки. Строки могут быть символьные или бинарные. в поле длины записывается длина строки в байтах. А сама строка в формате UTF-8 записывается в поле значения.

Числа. Целые числа кодируются минимальным количеством байт. Старшие байты со значением 0хFF или 0х00 могут опускаться.

Правила кодирования.

* При кодировании данных в формате BER (TLV), первым записывается номер тега длиной один или несколько байт, за ним следует один или несколько байт длины данных и сами данные, если длина не нулевая.

Кодирование номера тега:

* если биты 1-5 (счёт от 1 до 8) первого байта номера тега равны «1», то следующий байт также относится к номеру тега. Для каждого следующего байта: если бит 8 равен «1», то далее следует еще один байт номера. Иначе — это последний байт.

Кодирование длины данных:

* если бит 8 в первом байте длины равен 0, то биты 1-7 кодируют количество байт данных.
* если бит 8 в первом байте длины равен 1, то биты 1-7 кодируют количество байт длины. Последующие байты длины идут в порядке big-endian.

2.4 Программная архитектура и алгоритм работы ПМ ФО

Общая схема алгоритма представлена на рис. 1.5 и 1.6

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.

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).

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.

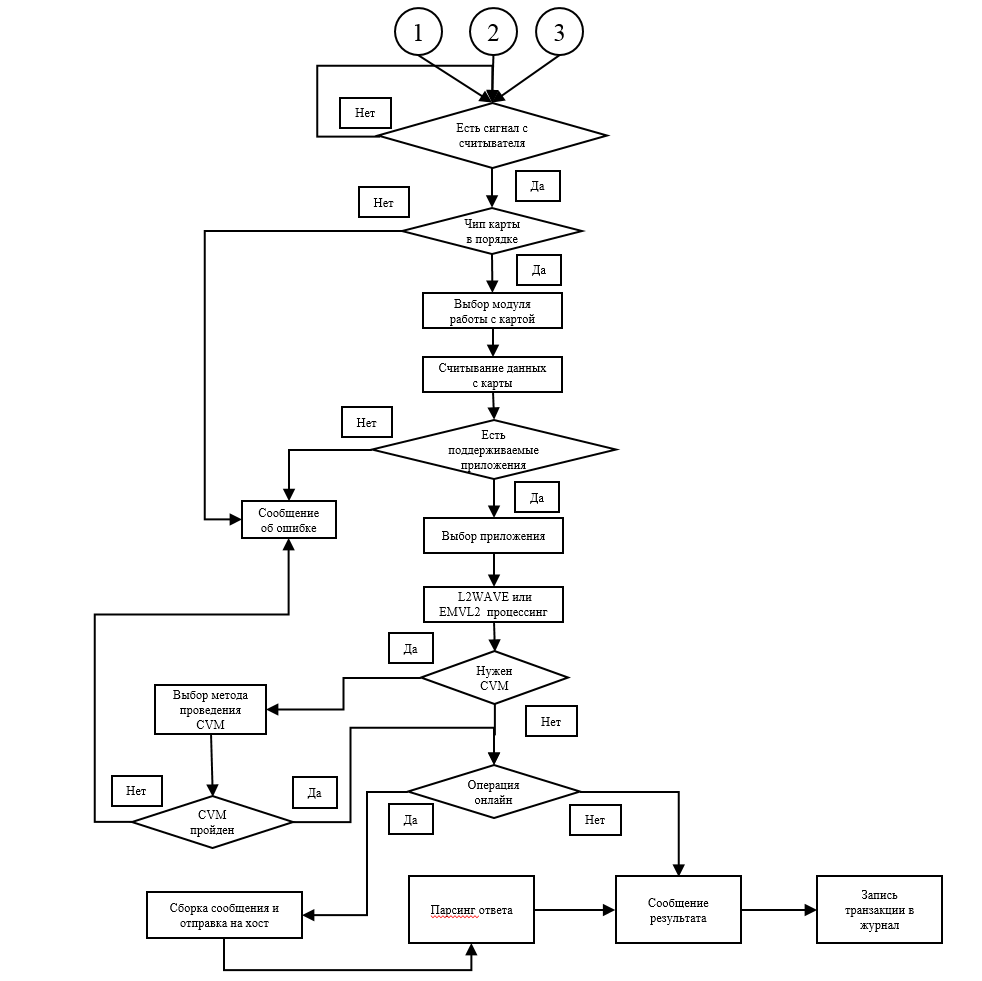


Рисунок 1.5 Общая схема алгоритма работы ПМ ФО

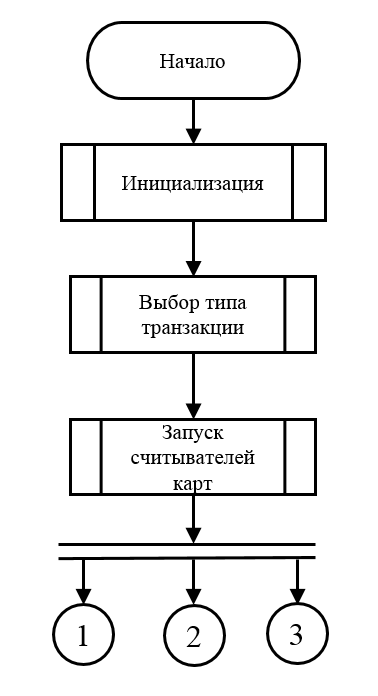


Рисунок 1.6 Общая схема начала работы ПМ ФО

2.5 Программная реализация ПМ ФО

2.5.1 Применение SWIG и JNI

Взаимодействие кодов JAVA и С/С++ может осуществляться двумя способами: С/С++ - код получает управление непосредственно из Java-программы путем вызова собственного (native) метода; С/С++ - код динамически загружает JVM с помощью Invocation API. Во втором случае, по сути, реализуется специализированная JVM, так как разработчик С/С++ - кода сам решает, в какой последовательности выполнять Java-код (когда и какие Java-объекты создавать, какие методы вызывать и т. д.).

Рассмотрим первую из указанных возможностей.

Для того чтобы передать управление С/С++ - коду из Java-программы, необходимо создать собственный Java-метод, сгенерировать с помощью утилиты Javah заголовочный файл для С/С++ - функций, разработать сами функции, в которые будет передаваться управление, и оттранслировать их, поместив в библиотечный файл. После создания библиотеки ее можно загружать из Java-программы для последующего вызова собственных методов.

Например, создание native-метода в Java выглядит так:

class MyClass **{**

static **{**

System**.**loadLibrary**(**"test"**);**

**}**

public native int nativeMethod**();**

**}**

Собственный метод создается путем добавления к его описанию спецификатора native, при этом он не должен иметь реализации (также, как и методы в описании интерфейса). Спецификатор native сообщает компилятору, что реализация данного метода будет представлена в виде откомпилированного С/С++-кода, помещенного в библиотечный файл. Когда JVM встречает обращение к собственному методу, происходит вызов соответствующей С/С++-функции. Помимо описания собственного метода, JAVA-код должен динамически загрузить библиотеку, содержащую С/С++-функцию с реализацией 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.

В приведенном примере метод nativeMethod**()**является собственным, и его С/С++-реализация находится в библиотеке test. Метод loadLibrary**()** вызывается в статическом инициализаторе, что обеспечивает единственный вызов этого метода после загрузки класса MyClass загрузчиком классов . В принципе, loadLibrary**()** можно вызывать более одного раза (например, в конструкторе), однако загрузка библиотеки будет происходить только при первом обращении к loadLibrary**()**, поскольку при последующих вызовах этого метода определяется, что библиотека уже загружена и будет просто возвращаться управление.

Метод loadLibrary**()** преобразует свой параметр в соответствии с тем, как именуются библиотечные файлы на конкретной платформе. В данном примере test преобразуется в libtest.so и test.dll для Win32 и UNIX соответственно. Метод loadLibrary**()** использует стандартный алгоритм поиска библиотеки для данной платформы. Для Win32 DLL должна находиться либо в текущем каталоге процесса, либо в каталоге, содержащем EXE-файл, то есть исполняемый модуль JVM, находящийся в подкаталоге BIN основного каталога JAVA, либо в системном каталоге Win32, либо каталоге Windows или в каталогах, указанных в переменные окружения PATH. Для UNIX библиотечный файл должен находиться либо в текущем каталоге процесса, либо в подкаталоге LIB основного каталога JAVA, либо в каталогах, перечисленных в переменной окружения LD\_LIBRARY\_PATH. Если указанную библиотеку найти не удается, метод loadLibrary**()** генерирует исключительную ситуацию Java.Lang.UnsatisfitedLinkError. Однако данная ситуация возникает не только в этом случае. Когда интерпретатор встречает вызов собственного метода, он ищет его (точнее его полную сигнатуру) в списке методов загруженных библиотек. Если метод не найден, то генерируется указанная исключительная ситуация.

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. Например:

Javah -JNI Java.Lang.Runtime

Перед использованием утилиты Javah соответствующий Java-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. В качестве имен создаваемых заголовочных файлов используются полные квалифицированные имена классов, которые описаны в указанном файле и содержат собственные методы.

Например, если выполнить следующие команды:

# javac MyClass.java

# javah -jni MyClass

то javah сгенерирует следующий файл MyClass.h:

/\* Do not edit this file - it is machine generated \*/

#include <jni.h>

/\* Header for class MyClass \*/

#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

Как указывалось, выше, данный файл можно создать вручную или с помощью утилиты Javah. В последнем случае не рекомендуется вносить в него какие-либо изменения, так как при последующем применении Javah к данному классу все внесенные изменения будут потеряны.

Директива препроцессора #include <jni.h> включает файл jni.h (из подкаталога include основного каталога Java), в котором находятся все необходимые объявления типов и функций для реализации собственного метода.

Макросы jniexport и jnicall необходимы только для платформы Win32, где они раскрываются соответственно в \_\_declspec (dllexport) и \_\_stdcall и позволяют более эффективно строить DLL. Платформа UNIX использует для этих целей обычные С-соглашения, поэтому указанные макросы раскрываются в пустые строки.

Как видно из примера, имя С/С++ - функции значительно отличается от имени собственного Java-метода. Важным понятием при построении имени С/С++ - функции и использовании JNI-функций является сигнатура метода (signature или method arguments signature).

**SWIG**

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 (Tcl, Perl, Python, Ruby, PHP) или компилируемыми (Java, C#, Scheme, OCaml) языками. Основная цель: обеспечение возможности вызова функций, написанных на одних языках, из кода на других языках. Программист создаёт файл \*.i с описанием экспортируемых функций; SWIG генерирует исходный код для связывания C/C++ и нужного языка.

История использования SWIG начинается с 1996 года. Автором и главным разработчиком был Дэвид Бэйнсли, на тот момент аспирант Лос-Аламосской национальной лаборатории. В настоящее время поддержку SWIG осуществляют свободная группа разработчиков, во главе которых Уильям Фултон.

SWIG написан на языках С и C++, распространяется по лицензии, похожей на BSD, с февраля 1996 года. Данная лицензия позволяет использовать, распространять и модифицировать код SWIG для коммерческих и некоммерческих целей без ограничений.

Вид исполняемого файла зависит от выбранного языка:

* исполняемый файл со встроенным интерпретатором скриптового языка;
* библиотека, функции C/C++ из которой автоматически становятся доступны из другого языка;
* библиотека функций C/C++ и библиотека функций - обёрток над функциями C/C++ (например, JNI для Java).

Пример создания JNI интерфейса к библиотекам, написанным на С\С++:

Для начала необходимо составить интерфейсный файл, в котором будут специальные директивы для обработки SWIG.

Пример интерфейсного файла (test.i):

**%**module**(**directors**=**"1"**)** test

**%**javaconst**(**1**);**

**%**include "enumtypeunsafe.swg"

**%**include "std\_vector.i"

**%**include "std\_string.i"

**%**include "stdint.i"

**%**include "cpointer.i"

**%{**

#include "Data.h"

#include "Types.h"

#include "Keys.h"

………

**%}**

**%**feature**(**"director"**)** CryptoLayer**;**

**%**feature**(**"director"**)** Transport**;**

**namespace** std

**{**

**%**template**(**vector\_ca\_key**)** vector**<**Keys**::**ca\_key**>;**

**%**template**(**vector\_char**)** vector**<**char**>;**

**%**template**(**vector\_appinfo**)** vector**<**Txn**::**ApplInfo**>;**

**%**template**(**vector\_revocation**)** vector**<**RevocationInfo**>;**

**%**template**(**vector\_string**)** vector**<**std**::**string**>;**

**};**

**%**include "Data.h"

**%**include "Types.h"

**%**include "Keys.h"

**%**include "CryptoLayer.h"

**%**include "Errors.h"

**%**pointer\_class**(**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.

Данный файл необходимо компилировать вместе с исходным кодом библиотеки. Ниже пример части MAKE-файла для сборки библиотеки.

JNI\_BUILD=

ifeq ($(ARCH),android-arm)

JNI\_BUILD = yes

endif

ifeq ($(ARCH),android-x86)

JNI\_BUILD = yes

endif

ifeq ($(ARCH),linux-i686)

JNI\_BUILD = yes

endif

ifeq ($(ARCH),linux-x86\_64)

JNI\_BUILD = yes

endif

# definitions

OUTPUT=libEMVL2

OUTDIR=obj/$(ARCH)

# JNI flags

ifeq ($(JNI\_BUILD), yes)

CCFLAGS = -fPIC -fpermissive -O2

CCFLAGS += -DSWIG

LDFLAGS = -s

ifeq ($(CFG), trace)

CCFLAGS += -DMAKE\_TRACE

ifneq ($(findstring android, $(ARCH)),)

LDFLAGS += -llog

CCFLAGS += -DFAKEVERSION=$(FAKEVERSION) -DCRC=$(CRC)

CCFLAGS += -fvisibility=hidden

# libraries

LDFLAGS+=-lstdc++ -lc

LDFLAGS+=-Wl,-soname,$(OUTPUT).so.$(MAJOR)

ifeq ($(ARCH), linux-i686)

LDFLAGS+= -m32

endif

# files

LIBFILES = \

$(OUTDIR)/EMV\_Library/CEMVTxn.o \

$(OUTDIR)/EMV\_Library/scr\_command.o \

$(OUTDIR)/EMV\_Library/EMV\_Context.o \

………………

ifeq ($(JNI\_BUILD), yes)

LIBFILES += \

$(OUTDIR)/emv\_wrap.o

endif

# targets

all: $(ARCH)

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)

$(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

-include $(LIBFILES:.o=.d)

-include $(JAVA\_OUTDIR)/emv\_wrap.d

История использования SWIG начинается с 1996 года. Автором и главным разработчиком был Дэвид Бэйнсли, на тот момент аспирант Лос-Аламосской национальной лаборатории. В настоящее время поддержку SWIG осуществляют свободная группа разработчиков, во главе которых Уильям Фултон.

2.6 Разработка интерфейса

Пользовательский интерфейс был разработан в среде Android Studio.

2.6.1 Экранные формы

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 1.7 представлен главный экран. С данной экранной формы совершается выполнение транзакций.

**ТУТ НУЖНА КАРТИНКА**

На рис. 1.8 представлена сервисная экранная форма. С данной формы проводятся различные сервисные операции, такие как очистка журнала, конфигурирование, проверка сервера банка и другие.

**ТУТ НУЖНА КАРТИНКА**

На рис. 1.9 представлена экранная форма конфигурации. В данной форме можно ввести необходимые данные для начала работы, сменить язык интерфейса, а также посмотреть различные значения, такие как версии приложения, прошивки терминала, серийный номер терминала, его IMEI и другую полезную информацию.

**ТУТ НУЖНА КАРТИНКА**

На рис. 1.10 представлена экранная форма введения пин-кода. Она сделана специальным образом, чтобы нельзя было перехватит значения касаний и тем самым попытаться скомпрометировать пин-код.

**ТУТ НУЖНА КАРТИНКА**

На рис. 1.11 представлена экранная форма журнала. Данная экранная форма, нужна для просмотра проведённых транзакций и при желании их возврата, отмены или печати копии чека.

**ТУТ НУЖНА КАРТИНКА**

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

2.7 Тестирование и отладка

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.
   * 1. 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.
   * 1. 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).
   * 1. 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).

* + 1. Выбор средств отладки и тестирования ПМ ФО

Модульное и интеграционное тестирование выполнялось с использованием инструмента JUnit, который входит в состав Android Studio.

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.

**Пример тест кейса (КОД И ОПИСАНИЕ)**

* + 1. 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, эспрессо 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**

UIAutomator - это аналог инструмента UIAutomation компании Apple для тестирования Android приложений. Android SDK предоставляет следующие инструменты для поддержки автоматизированного функционального тестирования пользовательского интерфейса:

* UIAutomatorviewer — графический инструмент для распознавания компонентов пользовательского интерфейса в Android приложении;
* UIAutomator — библиотеки Java API, содержащие методы для создания тестов пользовательского интерфейса.

Преимущества UIAutomator, для тестирования приложений:

* Отсутствие зависимости от разрешения экрана;
* Действия привязываются к Android UI компонентам. Это позволяет работать напрямую с элементами пользовательского интерфейса. Например, если необходимо нажать кнопку «ОК», можно средствами UIAutomator API отправить скрипту команду: нажми кнопку с надписью - «ОК», и он её нажимает. Таким образом не приходится привязываться к координатам;
* Можно воспроизводить сложные последовательности действий пользователя, и всегда эта последовательность будет одинаковой;
* Тесты могут быть запущены необходимое количество раз на различных устройствах без необходимости изменения Java кода;
* Можно использовать внешние кнопки на устройстве (кнопка «назад», «выключить», «громкость» и т.д.).

Недостатки:

* Тяжело использовать для приложений, написанных на HTML 5 и OpenGL, так как в этих приложениях нет Android UI элементов. В связи с этим, необходимо либо привязываться к координатам, либо искать альтернативные варианты тестирования;
* Необходимо проверять, и в случае необходимости, обновлять Java скрипты при обновлении Android приложения.

Тестирование приложения с помощью UIAutomator состоит из следующих шагов:

* Подготовка к тесту: установка приложения на устройство, анализ его UI компонент;
* Создание автоматизированного теста для приложения;
* Компиляция теста в JAR файл и копирование его на устройство;
* Запуск теста и анализ результатов;
* Исправление различных ошибок, найденных в процессе тестирования.

**Пример теста UIAUTOMATOR**

2.4.7 Профайлинг ПМ ФО

Для получения дополнительной информации о работе программного модуля, был использован профайлер Dalvik Debug Monitor Server, входящий в состав .

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. Эвотор [Электронный ресурс]. URL: http://wiki.evotor.ru/
18. Azur Pos 01-Ф [Электронный ресурс]. URL: http://azurpos.ru/
19. Java [Электронный ресурс]. URL: http://www.oracle.com/technetwork/java/index.html
20. C# [Электронный ресурс]. URL: http://msdn.microsoft.com/ru-ru/vcsharp/default.aspx
21. C\C++ [Электронный ресурс]. URL: https://gcc.gnu.org/
22. Kotlin [Электронный ресурс]. URL: https://kotlinlang.org/
23. Xamarin [Электронный ресурс]. URL: https://www.xamarin.com/
24. The Java Virtual Machine Specification Java SE 8 Edition Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley 2015
25. Eclipse IDE [Электронный ресурс]. URL: https://www.eclipse.org/
26. Microsost Visual Studio IDE [Электронный ресурс]. URL: https://www.visualstudio.com/vs/android/
27. Android Studio IDE [Электронный ресурс]. URL: https://developer.android.com/
28. IntelliJ IDEA IDE [Электронный ресурс]. URL: https://www.jetbrains.com/idea/
29. Refactoring: Improving the Design of Existing Code 1-st edition Martin Fowler, Kent Back, John Brant
30. Pro Git 1st Edition Scot Chacoon 2017
31. Learning XML, 2nd Edition Creating Self-Describing Data By Erik Ray 2009
32. ISO 8583 [Электронный ресурс]. URL: https://www.iso.org/obp/ui/#iso:std:iso:8583:-1:ed-1:v1:en
33. Спецификация протокола ТТК. Терминальные Технологии.
34. EMV Contactless Specifications for Payment Systems Book 2 Security and Key Management 2011
35. EMV Contactless Specifications for Payment Systems Book 3 Application Specifiacation 2011
36. Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER)
37. SWIG documentation [Электронный ресурс]. URL: http://www.swig.org/Doc3.0/SWIGDocumentation.html
38. BSD License [Электронный ресурс]. URL: https://directory.fsf.org/wiki/License:BSD\_4Clause
39. The Java Native Interface Programmer’s Guide and SpecificationSheng Liang
40. ГОСТ 19.201-78. ЕСПД. Техническое задание. Требования к содержанию и оформлению. Москва, 1980.
41. ГОСТ 19.701-90. ЕСПД. Схемы алгоритмов, программ, данных и систем. Условные Обозначения и правила выполнения. Москва, 1992.
42. ГОСТ 19.505-79. ЕСПД. Руководство оператора. Требования к содержанию и оформлению. Москва, 1980.
43. ГОСТ 19.102-77. ЕСПД. Стадии разработки. Москва, 1980.
44. ГОСТ 19.002-80. ЕСПД. Схемы алгоритмов и программ. Правила выполнения. Москва, 1980.
45. ГОСТ 19.003-80. ЕСПД. Схемы алгоритмов и программ. Обозначения условные графические. Москва, 1980.
46. ГОСТ 19.106-78. ЕСПД. Общие требования к программным документам, выполненным печатным способом. Москва, 1980.
47. ГОСТ 19.101-77. ЕСПД. Виды программ и программных документов. Москва, 1980.
48. ГОСТ 19.404-79 Пояснительная записка. Требования к содержанию и оформлению. Москва, 1980.
49. ГОСТ 19.105-78 Общие требования к программным документам. Москва, 1980.
50. ГОСТ 19.401-78 Текст программы. Требования к содержанию и оформлению. Москва, 1980.
51. ГОСТ 19.104-78 Единая система программной документации. Основные надписи. Москва, 1980.
52. ГОСТ 19.103-77 Обозначения программ и программных документов. Москва, 1980.
53. ГОСТ 19.506-79 Описание языка. Требования к содержанию и оформлению. Москва, 1980.
54. ГОСТ 19.301-79 Программа и методика испытаний. Требования к содержанию и оформлению. Москва, 1980.
55. Ю.И. Галушкина, А.Н. Марьямов. Конспект лекций по дискретной математике - 2-е изд., испр. - М.: Айрис-пресс, 2008. – 176 с.
56. Колдаев В.Д. Численные методы и программирование, 2007.
57. Колдаев В.Д., С.А.Лупин Архитектура ЭВМ: Издательство Инфра-M, 2008.
58. 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.
59. Гагарина Л.Г., Колдаев В.Д. Алгоритмы и структуры данных: Издательство Финансы и статистика, 2009г.
60. Гагарина Л.Г., Киселев Д.В., Федотова Е.Л. Разработка и эксплуатация автоматизированных информационных систем Москва 2007, 384 с.
61. Колдаев В.Д. Основы логического проектирования: Издательство Форум, Инфра-М,2011
62. 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.