Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение  
высшего профессионального образования  
«Пермский государственный национальный исследовательский университет»

Кафедра математического обеспечения  
вычислительных систем

УДК 004.02

**Инструментальное окружение сборки готовых приложений для мобильных платформ**

Отчет по «Преддипломной практике»

Работу выполнил студент группы ПМИ-1,2-2013 4 курса механико-математического факультета   
 А. В. Тюрнин

Научный руководитель: старший преподаватель кафедры МОВС  
 К. А. Юрков

Пермь 2017

# Аннотация

В данной работе решается проблема сборки готовых приложений для мобильных платформ. Приводится описание проблемы формулирование основных принципов работы инструментального окружения, в первую очередь понятие «слабой зависимости». Приводится обзор существующих решений, используемых для сборки мобильных приложений, их особенности и недостатки для решения задачи. Приводится проектирование архитектуры инструментального окружения, описание моделей, описание используемых алгоритмов и технологических процессов. Приводится описание процесса и особенности реализации инструментального окружения окружения для сборки мобильных приложений.

Содержание

[Введение 4](#_Toc484076628)

[Обзор существующих решений 8](#_Toc484076629)

[Конструкторы приложений 8](#_Toc484076630)

[Менеджеры пакетов 10](#_Toc484076631)

[Рассмотрение менеджеров пакетов будем проводить по следующим основным пунктам: 10](#_Toc484076632)

[1. CocoaPods 10](#_Toc484076633)

[2. Swift Package Manager 15](#_Toc484076634)

[3. NuGet 18](#_Toc484076635)

[4. Maven 21](#_Toc484076636)

[Выводы 23](#_Toc484076637)

[Архитектура и модели инструментального окружения 25](#_Toc484076638)

[Модели 25](#_Toc484076639)

[Архитектура 25](#_Toc484076640)

[Алгоритмы 25](#_Toc484076641)

[Технологические процессы 25](#_Toc484076642)

[Реализация системы 25](#_Toc484076643)

[Список литературы 26](#_Toc484076644)

# Введение

Мобильные устройства стали для нас чем то само собой разумеющимся. Этот рынок растет невероятными темпами и вот лишь некоторая статистика, которая это подтверждает. В четвёртом квартале 2015 года совместная доля мобильных ОС от Google и Apple на мировом рынке составляла 98,4%. Наиболее распространённой мобильной платформой остаётся Android с долей 80,7%, в то время как iOS владеет 17,7% рынка — это глобальные цифры. [1][2]. В октябре 2016 года мобильные устройства опередили ПК по использованию интернета. Если в 2009 году распределение составляло приблизительно 100% против 0% в пользу ПК, то в прошлом месяце две линии наконец пересеклись и мобильные устройства вышли вперед [3] [4]. В 2017 году мировой рынок мобильных приложений достигнет $77 млрд [5].

Тенденции, диктуемые рынком, позволяют заглянуть за грань текущей границы развития рынка и понять, что будет важно в следующие годы. Создания специализированных приложений с небольшим функционалом сильно влияет на рынок. Пользователям не нужны громоздкие приложения, а простые эффективные решения. В то же время корпоративный сектор все больше внедряет мобильные решения для оптимизации и автоматизации внутренних процессов компании. [6] Именно эти тенденции диктуют разработчикам наращивать скорость разработки и не терять, а то и повышать качество приложений.

Часто приходится создавать приложения из схожих высокоуровневых модулей. Как пример таких приложений, фирменные приложения "умного дома" для жилых комплексов. Их главная особенность в том, что у разных заказчиков свои требования к набору функционала, но при этом любые два набора пересекаются в виде некоторого базового функционала [7]. Эта особенность предполагает, что каждый уникальный функционал реализуется в виде отдельного модуля. Кроме того, для ускорения сборки готовых приложений, необходимо наложить на модуль и его описание такие ограничения, которые позволили бы не писать какого либо дополнительного исходного кода, а генерировать исходный код сборки. Такие ограничения позволяют полностью изолировать пользователя от финальной сборки приложения.

Но каким именно ограничениям необходимо следовать при написании таких модулей. Так как модули являются функционально обособленными фрагментами исходного кода, то для их функционирования может потребоваться использования функционала других модулей. Например, модулю авторизации необходим функционал модули, реализующего HTTP-запросы. Но в то же время, если в приложении отсутствует HTTP-модуль, то модуль авторизации может авторизовать пользователя локально, например запрашивая пин-код, сохраненный на телефоне. Первый тип взаимодействия между модулями назовем "зависимостью". Тогда второй тип будет называться "слабой зависимостью". Одна из главных целей данной работы показать, что при наличии определенных ограничений на модуль и его описания и реализации концепции "слабой зависимости", можно ускорить процесс сборки готовых приложений для мобильных платформ из таких модулей, полностью автоматизировав данный процесс. Цель данной работы является проверить возможность реализации такого инструментального окружения, которое позволит реализовывать выше перечисленные принципы.

В главе "Обзор существующих решений" рассмотрены возможные готовые решения, доступные для использования в мобильной разработке - конструкторы приложений, case-решения для мобильной разработки, менеджеры пакетов. Это отражено в главе "Обзоры существующих решений". Кроме того, в данной главе будет описаны причины по которым данные решения не подходят для данной задачи. Именно по этим причинам было принято решение разработать собственное инструментальное окружение, которое позволило бы наиболее эффективно решать поставленные задачи. Так же будет обосновано решение о выборе для текущей реализации инструментального окружения CocoaPods.

Для реализации такой системы требуется решить множество задач по проектированию и формализации:

* + описать основные понятия инструментального окружения.
  + спроектировать используемые модели
  + спроектировать архитектуру системы
  + описать компоненты и способы их взаимодействия
  + описать используемые алгоритмы
  + подробно прописать технологические процессы использования инструментального окружения

Решение данных задач описано в главе "Архитектура и модели инструментального окружения".

В рамках данной выпускной квалификационной работы будет реализовано данной инструментальное окружение для сборки приложений на платформе iOS. Особенности платформы iOS и подробности реализации информационной системы будут описаны в последней главе. В заключении приводится описание как и процесса разработки самого инструментального окружения, так и описание разработки готовых приложений на базе данного окружения. На основании данного опыта реализации и сформированных моделей в главе 2 будет принято решение о расширении данной информационной системы для сборки приложений и для других платформ и описаны возможные перспективы развития данного окружения.

# Обзор существующих решений

## Конструкторы приложений

**Конструкторы приложений** это сервисы, которые позволяют создавать приложения для определенных нужд (например, доставка еды или такси) вообще не используя программирование, на основе заданных шаблонов с добавлением нужных виджетов и вариантов оформления. Как правило **конструкторы** делают кросс-платформенные приложения и размещают приложения в магазинах от своего имени.

Рассмотрим некоторые популярные конструкторы приложений, доступные на рынке.

AppGyver Composer [16] – drag-and-drop HTML5 конструктор, который может работать с данными из REST API и умеет взаимодействовать с популярными бэкендами, вроде Facebook Parse и Built.io.

Компания ориентирует его на разработчиков, желающих быстро выпустить простое мобильное гибридное приложение.

В конструкторе вы можете построить быстрый прототип и экспортировать его код для тонкой настройки вручную. Приложения работают на AngularJS, фреймворке Ionic и все это оборачивается PhoneGap.

AppMachine [15] – конструктор, позволяющий создавать «родные» приложения для iOS, Android, Windows Phone вообще без знания программирования.

Платформа предлагает более 30 «блоков», из которых можно собирать приложения, шаблоны дизайна, шаблоны навигации, связь с данными или API, импорт данных, публикацию приложений в магазинах, продвижение и аналитику.

GameSalad [17] — специальное приложение, разработанное компанией Gendai Games для создания простых игр для iPhone и iPod Touch. Приложение основано на принципе Drag-and-drop специально для людей не знающих программирование. Приложение имеет полностью интуитивный, понятный, визуальный интерфейс. Приложение запускается на Mac OS X с помощью данного приложения возможно создание, весьма красочных и интересных игр. Благодаря простоте использования, приложением могут пользоваться, как профессиональные игровые дизайнеры, так и простые люди.

Данные решения не подходят для решения задачи автоматической сборки мобильных приложений по нескольким причинам:

1. Приложение построенные на базе web-технологий, таких как PhoneGap и Cordova, предоставляют ограниченную функциональность доступа к части системных возможностей. Кроме того рендеринг web-страниц сильно влияет на скорость работы приложения, что является большим минусов для мобильных приложений.
2. Приложения построенные на базе платформ, работающих с «нативным» кодом приложений отличаются не очень большой возможностью расширения. Кроме того, на другой стороне сервиса сидят такие же программисты, которые предоставляют поддержку при работе в данных конструкторам
3. Большинство из существующих конструкторов приложений являются платными, что является большим минусов при продолжительной поддержке приложений.

## Менеджеры пакетов

## Рассмотрение менеджеров пакетов будем проводить по следующим основным пунктам:

1. Описание.
2. Что из себя представляют пакеты (модули).
3. Как пакеты интегрируются в текущее приложение.
4. Как реализована работа со слабыми зависимостями.

### CocoaPods

#### Описание

CocoaPods [8] - это менеджер зависимостей уровня приложений для Objective-С, Swift и любых других языков, которые могут работать в Objective-C Runtime, такие как C++, C, RubyMotion и другие. Он был разработан Eloy Durán и Fabio Pelosin, которые до сих пор продолжают управлять проектом не без помощи сообщества. Его разработка началась в августе 2011 году и первый публичный релиз был выпущен уже в сентябре того же года. В мае 2016 года проект дошел до релизной версии 1.0. Так же было выпущено десктопное приложение. Проект CocoaPods был вдохновлен комбинецией менеджера пакетов для Ruby RubyGems и проектом Bundler.

CocoaPods сфокусирован на дистрибуции проектов с открытым исходным кодом и интеграции их в Xcode проекты.

С CocoaPods можно работать несколькими способами:

- Напрямую из командной строки

- В IDE таких как XCode или JetBrains AppCode в виде плагинов

- В отдельном десктопном приложении

#### Что из себя представляют пакеты?

Описание любого пакета для CocoaPods находится в специальном файле-спецификации \*.podspec. Спецификация описывает версию Pod библиотеки, Она включает в себя данные о том, откуда нужно подгружать исходный код, какие файлы использовать, какие параметры сборки устанавливать и другие метаданные такие как имя библиотеки, ее версия и описание.

В файле спецификации много возможных полей, но не все из них одинаково полезны. Рассмотрим только некоторые из них:

* version - поле, определяющее номер версии Pod библиотеки. Пример: '3.1.0'
* source - содержит в себе информацию о пути к репозиторию, а так же номеру версии библиотеки в репозитории. Пример: { :git => 'https://github.com/tonymillion/Reachability.git', :tag => 'v3.1.0' }
* source\_files - регулярное выражение, описывающее все необходимые файлы для библиотеки. Пример: 'Reachability/common/\*.swift'
* {OS}.source\_files - регулярное выражение, описывающее все необходимые файлы для библиотеки для определенной платформы. {OS} может быть ios, osx. Пример: 'Reachability/ios/\*.swift', 'Reachability/extensions/\*.swift'
* {OS}.framework - Список системных фреймворков необходимых для этой библиотеки. {OS} может быть ios, osx. Пример: 'UIKit', 'Foundation'
* dependency - В каждом таком поле указываются зависимости текущей библиотеки от других. Например: 'RestKit/CoreData', '~> 0.20.0'
* weak\_framework - Список "слабых" системных фрейморков. Например 'Twitter'. Этот фреймфорк появился впервые в iOS в версии 5.0. Если бы вы пытались собрать проект Cocoapods для версии 4.2, то компилятор будет ругаться, что не может найти такой фреймворк. Поэтому можно его указать как weak\_dependency. И если в текущей версии iOS/MacOsX нет этого фрейворка, то он просто не будет указан в заголовочных файлах. А код уже должен сам обработать его отсутствие.
  + compiler\_flags = '-Wno-incomplete-implementation -Wno-missing-prototypes'
* subspec - Тоже самое описание спецификации, но только для "под"-библиотеки. Все поля, которые есть в спецификации, будут и здесь. Пример:

s.subspec 'Core' do |cs|

cs.dependency 'RestKit/ObjectMapping'

cs.dependency 'RestKit/Network'

cs.dependency 'RestKit/CoreData'

end

#### Как пакеты интегрируются в текущее приложение?

Podfile - это спецификация, которая пописывает зависимости target'ов одного или более проекта Xcode. Target определяет продукт сборки, который содержит в себе иснструкции по компиляции из набора файлов проекта(project) или workspace'а. Target определяет один продукт; он организовывает входные данные для системы сборки - исходные файлы и инструкции для обработки этих файлов, необходимые для для сборки продукта. Проекты могут содержать один или более target'ов, который соответствует одному продукту. (Как же отвратительно это звучит на русском)

Podfile должен располагаться радом с файлом проект \*.xcodeproj . Далее, например в терминале, вызывается команда “pod install”. Она формирует так называемую “рабочую область” или Workspace. В ней будет располагаться основной проект со всем присоединенными проектами и таргетами. Также в нем будет таргет, который будет называться Pods. В нем будут находиться все зависимости, необходимые для проекта. Pods target будет компилироваться в одну единственную библиотеку, поэтому в последующие разы не придется дожидаться компиляции всех зависимостей проекта.

Podfile, как и файл Podspec, имеет достаточно много опций для конфигурации различных сборок, но рассмотрим здесь только некоторые основные:

* target – в нем указывается имя Target’а, для которого далее будут прописаны зависимости. Пример: «target 'MyApp'»
* pod – описание зависимости и ее некоторых параметров.
  + Имя зависимости в виде строки. Имя зависимости уникально и берется из БД платформы CocoaPods, если не указать путь до репозитория, откуда ее брать.
  + Путь до репозитория локального или удаленного. Если репозиторий локальный, то путь указывается в виде «:path => ‘~/temp’». Если же репозиторий удаленный, то используется «:git => ‘https://github.com/gowalla/AFNetworking.git’».

#### Работа со слабыми зависимостями

В CocoaPods никак не реализована возможность работы со «слабыми зависимостями». Любой пакет, который указывается в файле спецификации podspec или в podfile обязательно внедряется в приложение.

### Swift Package Manager

#### Описание

Swift Package Manager [9][10] – это инструмент для управления и распространения Swift кода. Он интегрирован в build-систему Swift для автоматической загрузки, компиляции и линкования зависимостей. Package Manager был включен в build-систему начиная с Swift 3.

#### Модули

Swift организует код в модули. Каждый модуль определяет namespace и регламентирует контроль доступа для тех участков кода, которые могут использоваться вне модуля.

Программа может иметь весь свой код в одном модуле или импортировать другие модули как зависимости. В отличие от небольшого количества системных модулей, таких как Darwin в macOS или Glibc в Linux, для большинства зависимостей требуется загрузка и компиляция для дальнейшего использования.

#### Пакеты

Пакет состоит из файлов исходного кода и (см. рис. 1) файла манифеста, который называется Package.swift. Он определяет название пакета и его описание в поле PackageDescription. Package может иметь один или несколько target’ов. Каждый target определяет продукт и может описывать одну или несколько зависимостей. Каждый пакет может иметь несколько подмодулей, спецификация которых описывается в поле targets.



рис. 1 Пример манифеста Swift Package Manager

#### Product

Target может быть собрана как в библиотеку так и в исполняемый файл. Библиотека содержит модуль, который может быть импортирован в код. Исходный файл может быть запущен ОС.

#### Зависимости

Зависимости target’а это модули, которые используются в коде. Зависимость содержит в себе относительный или абсолютный URL к исходному коду пакета и набор требований для версии пакета, который должен использоваться. Роль Package Manager’а это уменьшить затраты на координацию автоматизированием процесса загрузки и сборки всех зависимостей в проекте. Это рекурсивный процесс: зависимость может иметь собственные зависимости, каждая из которых может иметь свои формируя некоторый граф зависимостей. Package Manager загружает и собирает все необходимое, что бы «удовлетворить» требования графа зависимостей.

#### Как пакеты интегрируются в текущее приложение?

Есть возможность из Package-файла и файлов с исходным кодом создать проект для Xcode, но все дело в том, что Package использует только open-source библиотеки и не может быть использован для полноценной разработки для iOS, т.к. основная библиотека для разработки под iOS UIKit является проприетарной.

#### Работа со слабыми зависимостями

В Swift Package Manager никак не реализована возможность работы со «слабыми зависимостями».

### NuGet

#### Описание

NuGet [11][12] – пакетный менеджер для разработки на платформе Microsoft включая .NET. NuGet client tools предоставляют возможность создавать и использовать кастомные пакеты. NuGet Gallery это центральный репозиторий пакетов используемый всеми, кто использует или создает пакеты.

#### Что из себя представляют пакеты?

Пакеты в NuGet представляют из себя пару файлов: .nuspec и .nupkg.

Nuspec это xml-манифест файл (см. рис. 2), который описывает содержание пакета и процесс создания NuGet пакета. Как минимум, манифест включает в себя идентификатор пакета, номер версии, название, которое отображается в Галерее, автор и владелец информации и длинное описание. Он также может содержать описание релиза, информация о копирайте, короткое описание для Менеджера Пакетов в Visual Studio, локальный идентификатор, адрес домашней страницы и адрес лицензии, ссылка на иконку, список зависимостей и ссылок, тэги, которые помогаю поиску в Галерее и другие.

Начиная с NuGet 3.5, пакеты могут быть отмечены специфическим типом для идентификации использования пакета. Пакеты не отмеченные никаким типом, включая все пакеты более ранних версий, отмечаются как пакеты “**Зависимости”.**

* Пакеты типа **“Зависимость”** добавляют некоторые возможности на этапе компиляции или во время работы приложения или библиотеки и может быть установлен в проект любого типа (учитывая то, что они совместимы). Пакеты **Зависимости** после установки в проект помещаются в папку *dependencies*.
* **“ DotnetCliTool” –** расширения для.NET CLI и вызываются из командной строки. Такие пакеты могут быть установлены только в .NET Core проекты и никак не влияют на операции восстановления. Когда пакет устанавливается, он помещается в отдельную папку в проекте *tools*.
* Тип **“Custom”** использует произвольный идентификатор типа, который поддерживает те же правила форматирования, что и id пакета.

Любой другой тип, кроме **Зависимости** и **DotNetCliTool** не распознаетсяя автоматически пакетным менеджеров NuGet в Visual Studio.

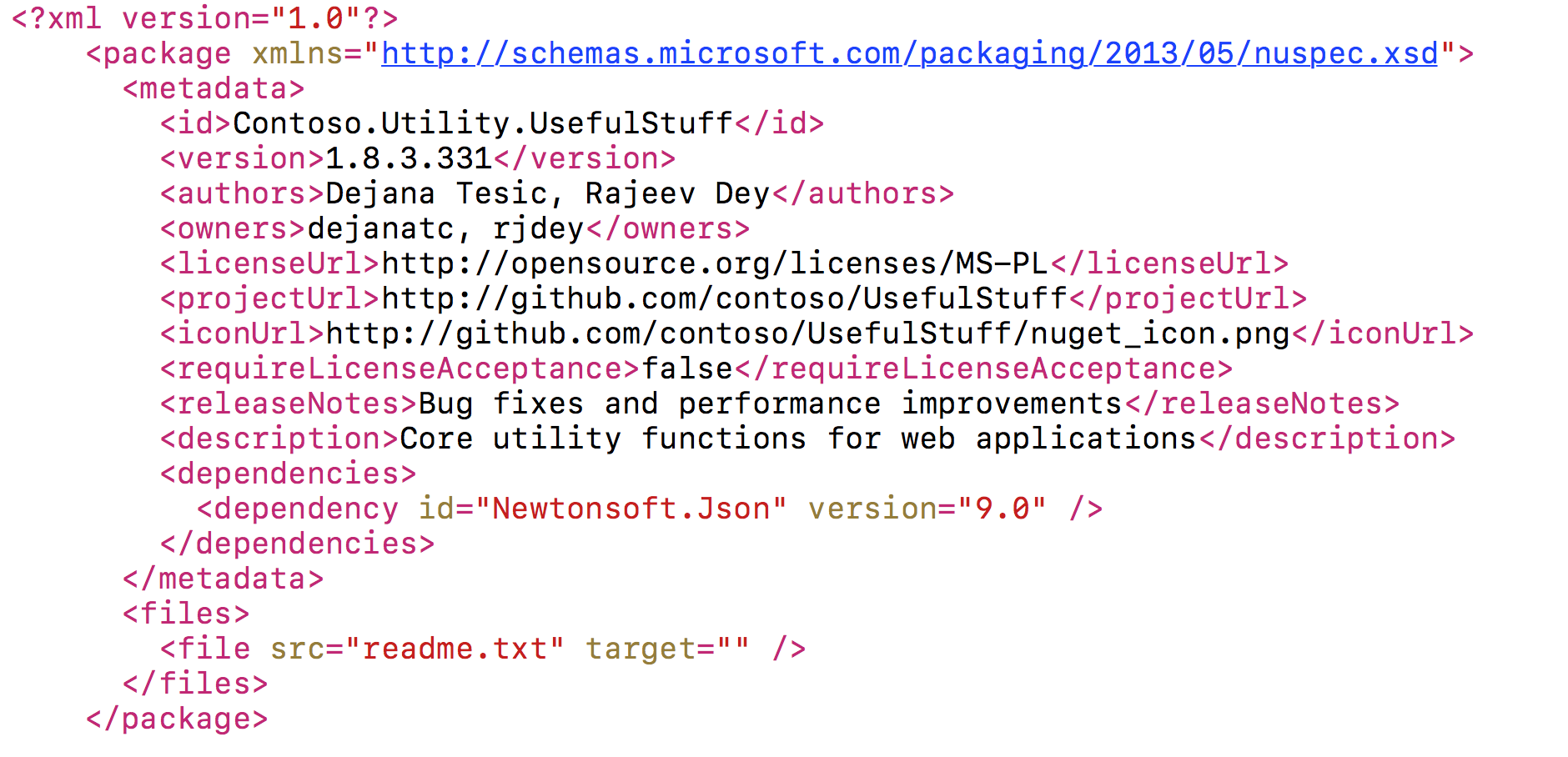


рис. 2 Пример манифеста NuGet

Nupkg представляет из себя архив, который содержит в себе с nuspec файл и скомпилированные файлы пакета в виде dll файлов.

#### Как пакеты интегрируются в текущее приложение?

NuGet как расширение для Visual Studio в 2010 году, а начиная с Visual Studio 2012 распространяается вместе с ней. Так что любой проект, созданный в Visual Studio начиная с 2011 года имеет возможность работы с NuGet «из коробки».

#### Работа со слабыми зависимостями

В NuGet никак не реализована возможность работы со «слабыми зависимостями».

### Maven

#### Описание

Maven [13][14] – фреймворк для автоматизации сборки Java-проектов на основе описания их структуры в файлах на языке POM (Project Object Model), являющимся подмножеством языка XML. Проект Maven издается сообществом Apache Software Foundation, где формально является частью проекта Jakarta Project.

#### Что из себя представляют пакеты?

Ключевым понятием Maven является **артефакт** – это, по сути, любая библиотека, хранящаяся в репозитории.

**Зависимость** – артефакт, который непосредственно используется в проекте.

Вся структура проекта описывается в файле *pom.xml*, который должен находиться в корневой папке проекта.

Тип проекта – некоторая стандартная компоновка файлов и каталогов в проектах различной направленности. (Например web-сервер, Andriod-приложение)

**POM** – описание модели проекта. В ней описываются такие общие характеристики как имя, версия, авторы и их контактная информация, VCS проекта и вообще связанные с ним сетевые ресурсы, тип проекта, связи с другими проектами, используемые при сборке плагины и описания способа их задействования. Мне кажутся особенно важными два компонента этой модели. Каждый POM имеет как минимум следующие три поля:

* groupId – наименование организации или подразделения. Для этого поля действую такие же правила составление, как и для любого проекта Java – записывают доменное имя, имя организации или сайт проекта.
* artifactId – название проекта.
* version – версия проекта.

Такой тройкой можно однозначно идентифицировать артефакт. Если состояние кода проекта не зафиксировано, то в конце к версии добавляется постфикс “-SNAPSHOT” что обозначает, что версия не является стабильной.



рис. 3 Пример описания зависимостей в Maven

Рассмотрим несколько основных составляющих POM-файла:

Dependencies – список зависимостей, необходимых для проекта. Как и сам проект, любая зависимость описывается такой же тройкой groupId, artifactId и version (см. рис. 3).

#### Как пакеты интегрируются в текущее приложение?

Для интеграции Maven в уже существующий проект необходимо совершить следующие шаги:

1. Поместить в корневую папку pom-файл
2. Описать в pom-файле проект через тройку groupId, artifactId, version и описать все необходимые для него зависимости.
3. Вызвать команду в mvn complile в терминале. Эта команда скомпилирует все исходные файлы проекта, подтянет все необходимые зависимости и создаст необходимую инфраструктуру для них.

#### Работа со слабыми зависимостями

В Maven никак не реализована возможность работы со «слабыми зависимостями».

## Выводы

Рассмотренные выше существующие решения не подходят для решения задачи. Конструкторы приложений могут позволить создавать приложения не написав ни строчки исходного кода, но они страдают либо от низкой скорости работы из-за использования PhoneGap и Cordova, либо используют «нативный» код, но абсолютно закрыты для расширения и модификации. Кроме того большинство из них платные, а их поддержкой этих конструкторов занимаются другие разработчики, которые в итог пишут код за нас. Менеджеры пакетов в свою очередь предоставляют возможности практически неограниченного расширения приложения, но при этом не предоставляют никакого механизма управления «слабыми» зависимостями, что не подходит для решаемой задачи. При этом менеджеры пакетов хорошо решают задачи связанные с интеграцией пакетов в готовое приложение и данных их функционал можно уверенно использовать при реализации нашего инструментального окружения. Как было сказано ранее, в рамках данной выпускной работы будет реализовано инструментальное окружение для сборки мобильных приложений для платформы iOS. Как было показано выше, CocoaPods имеет широкий функционал для управления пакетами, их версионностью. Данные преимущества CocoaPods позволят ускорить процесс разработки инструментального окружения.

Выводы из данного обзора и выбор CocoaPods для управления зависимостями позволят начать важный этап проектирования используемых моделей и архитектуры инструментального окружения. На базе спроектированной архитектуры и моделей станет возможно описать компоненты системы и все технологические процессы, используемые при работе с инструментальным окружением. Данные этапы будут рассмотрены в следующей главе.

# Архитектура и модели инструментального окружения

В первую очередь представим некоторые основные понятия инструментального окружения.

**Приложение** – ПО, предназначенное для работы на смартфонах, планшетах и других мобильных устройствах, состоящее из модулей, связанных между собой в рамках некоторой конфигурации.

**Модуль** - логическая часть приложения с четко определенной функциональностью, является самодостаточной частью приложения, выполняющий четко определенную задачу,

**Модуль-экран** - модуль, который имеет графическую составляющую, позволяющий пользователю взаимодействовать с системой и управлять навигацией между другими экранами

При помощи описания модуля через специальные файлы-манифесты абстрагироваться от реализации конкретного модуля и обработать его отсутствие в конечной сборке приложения. Манифест модуля содержит в себе информацию о уникальном имени модуля, его «сильных» и «слабых» зависимостях, а так же дополнительная мета информация, которая позволяет на сборки генерации готового приложения генерировать код сборки для конкретного модуля.

У приложение так же есть свой файл-манифест, который содержит в себе информацию, необходимую для сборки. Манифест приложения содержит в себе название приложения, список его модулей, модуль–«точка входа», а так же пути к файлам конфигурации и набору ресурсов.

**Сильная зависимость** – взаимосвязь между двумя модулями, когда для работы одного из них необходимы вызовы процедур и/или функций или доступ к данным из другого модуля.

**Слабая зависимость** – зависимость между двумя модулями, когда один из них может использовать фунционал второго модуля, но при всем этом присутствие второго модуля не является обязательными, первый модуль имеет возможность обрабатывать его отсутствие в приложении не вызывая ошибок компиляции или времени выполнения а также может корректировать логику своей работы в зависимости от этого .

## Технологические процессы и архитектура

## Технологический процесс написания модуля выглядит следующим образом:

1. Создание Git-репозитория модуля
2. Написание модуля. Описание интерфейса доступа к модулю
3. Написание манифеста модуля со всей необходимой для сборки информацией информации об интерфейсе.
4. Отправка исходного кода модуля на сервер при помощи команды git «push» и обработка последнего коммитов специальным хуком.

Технологический процесс сборки приложения будет выглядит следующим образом:

1. Создание манифеста приложения. Указание все необходимой информации: имени приложения, пути к файлам конфигурации и ресурсам приложения, список всех необходимых модулей и корневого модуля.
2. Передача манифест приложения программе-сборщику.
3. Программа сборщик отправляет данные из манифеста на сервер.
4. На стороне сервера строится граф зависимостей и генерируется файл сборки, который отправляется обратно на клиент.
5. Программа сборщик на основании файла сборки создает проект приложения и загружает в него все необходимые зависимости.
6. Запускается генератор сборки. Это программа, которая, используя мета-данные каждого модуля, генерирует исходный код для инициализации каждого модуля и «протягивания» всех необходимых и доступных ему зависимостей.
7. Запускается процесс компиляции. Если во время этого процесса происходит ошибка, то по каждой ошибке в определенном модуле генерируется отчет и отправляется на сервер. На сервере автор модуля оповещается об ошибке письмом, с прикрепленным к нему файлом с кодом и описанием ошибки, манифестом приложения и файлом сборки. Если ошибок нет, то приложение является завершенным и отправляется отчет о сборке на сервер

Граф зависимостей [20] — ориентированный граф, отображающий соотношение множества элементов некоторой совокупности в соответствии с выбранным транзитивным отношением над ней.

Технологические процессы, описанный выше, подразумевают использование определенный паттернов проектирования как самом приложения, так и модуля в частности. Кроме того, желательно наличие централизованного сервера, на котором будут храниться модули, мета-информация о них и, в том числе, должен быть реализован механизм разрешения зависимостей, который, при наличии большого количества модулей и связей между ними, может требовать больших вычислительных мощностей. И не мало важен доступ к этому функционалу на только на определенной машине, но и удалённо. При этом сама сборка не требует высокой вычислительной мощности и может выполняться на локальной машине. Кроме того, в следствии некоторых ограничений мобильных платформ [21], сборка iOS-приложения, написанного на Obj-C или Swift, не может производиться не на базе операционной системы macOS, но в то же время на хранение и обработку такого исходного кода ограничений нет.

Все это приводит к тому, что оптимальная архитектура для такой системы является архитектура клиент-сервер, представленную на Рис. 1.

# Untitled%20Diagram.png

Рис. 4 Архитектура инструментального окружения

Подробнее опишем компоненты данной архитектуры:

Git-репозиторий – хранилище системы контроля версий, в котором хранится исходный код всех модулей и мета-данные о них. К каждом репозиторию «прикреплен» специальный хук. Хук – это скрипт, обрабатывающий события Git-репозитория, например «push». С помощью данного скрипта модуль валидируется на основании его манифеста и, если модуль валиден, то мета-данные о нем добавляются в БД модулей. Тем самым на этапе разработки существует возможность поддерживать версионность модулей на каждом этапе разработки. И, в теории, не будет происходить ситуации, когда незаконченный модуль попадет в сборку приложения. Кроме того, использование интерфейсов для доступа к функциональности модуля позволяют эффективно использовать Mock-объекты для упрощения и ускорения разработки. Mock-объект представляет собой конкретную фиктивную реализацию [интерфейса](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), предназначенную исключительно для тестирования взаимодействия и относительно которого высказывается утверждение [22].

В базе данных хранятся мета-данные о доступных для сборки модулях. Это позволяет поддерживать версионность модулей, а так же оперативно предоставлять информацию о модуле, по запросу к API сервера.

API позволяет клиенту без необходимости работы напрямую с БД получать достоверную информацию о состоянии доступных в системе модулей. Кроме того, API на основе манифеста приложения генерирует Podfile со всеми необходимыми для сборки модулями, а так же файл сборки.

Сборщик – программа, которая на основе манифеста приложения формирует готовое к сборке приложение. У сборщика следующий алгоритм работы:

* На входе: манифест приложения
* Манифест приложения отправляется на сервер через API.
* С сервера приходит Podfile и файл сборки.
* На базе Podfile’а формируется проект приложения.
* Далее в в проекта создается файл сборки приложения и в него помещается автоматически сгенерированный код, на базе файла сборки.
* Генерируется код для инициализации сборки в приложении
* Запускается процесс компиляции проекта
* На выходе: готовое приложение

# Реализация системы

# Список литературы

1. 80% всех мобильных устройств работает на Andriod [Электронный ресурс] URL: <https://therunet.com/articles/5948> (дата обращения 31.05.2017)
2. Статистика мобильных операционных систем за декабрь 2016 год [Электронный ресурс] URL: <http://wp-seven.ru/stat-i/novosti/statistika-mobil-ny-h-os-za-dekabr-2016.html> (дата обращения 31.05.2017)
3. Мобильные устройства впервые превзошли по количеству ПК в Интернете [Электронный ресурс] URL: <http://itc.ua/blogs/mobilnyie-ustroystva-vpervyie-prevzoshli-po-kolichestvu-pk-v-internete/> (дата обращения 31.05.2017)
4. Mobile and tablet internet usage exceeds desktop for the first time worldwide[Электронный ресурс] URL: <http://gs.statcounter.com/press/mobile-and-tablet-internet-usage-exceeds-desktop-for-first-time-worldwide> (дата обращения 31.05.2017)
5. Топ-5 трендов мобильной разработки 2017 года: предположения [Электронный ресурс] URL: <https://mobile.simbirsoft.ru/blog/top-5-trendov-mobilnoy-razrabotki-2017-goda-predpolozheniya/> (дата обращения 31.05.2017)
6. Тренды разработки мобильных приложений в 2017 году [Электронный ресурс] URL: <https://rb.ru/opinion/mobile-2017/> (дата обращения 31.05.2017)
7. Etrok – система умного жилого комплекса [Электронный ресурс] URL: <https://www.etrok.ru> (дата обращения 31.05.2017)
8. CocoaPods Official Site [Электронный ресурс] URL: <https://cocoapods.org/> (дата обращения 31.05.2017)
9. Swift. Основы разработки приложений на iOS / Усов В.А. / Питер / 201 г. – 304 стр.
10. Swift.org [Электронный ресурс] URL: <https://swift.org/> (дата обращения 31.05.2017)
11. Nuget 2 Essentials/ Ozon / 2013 г. – 253 стр.
12. NuGet Documentaion. Microsoft Docs [Электронный ресурс] URL: <https://docs.microsoft.com/en-us/nuget/> (дата обращения 31.05.2017)
13. Maven Official Site [Электронный ресурс] URL: <https://maven.apache.org/> (дата обращения 31.05.2017)
14. Руководство по Maven [Электронный ресурс] URL: <http://www.apache-maven.ru/> (дата обращения 31.05.2017)
15. AppMachine: make an app within hours [Электронный ресурс] URL:  <http://www.appmachine.com> (дата обращения 31.05.2017)
16. AppGyver: Low-Code Enterprise-Grade App Creation [Электронный ресурс] URL: <https://www.appgyver.eu> (дата обращения 31.05.2017)
17. GameSalad – Drag&Drop programming [Электронный ресурс] URL: <https://gamesalad.com> (дата обращения 31.05.2017)
18. Эффективная реализация графа потока зависимостей (Дроздов А. Ю., Тютюник О. М., Шилов В.В..)
19. Developing native iOS Apps on Windows [электронный ресурс] URL: <http://stackoverflow.com/questions/22358/how-can-i-develop-for-iphone-using-a-windows-development-machine> (дата обращения 31.05.2017)
20. Unit Testing with Mock-objects [Электронный ресурс] URL: <https://msdn.microsoft.com/en-us/library/ff650441.aspx> (дата обращения 31.05.2017)