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

Научно-исследовательская работа

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

Оглавление

Аннотация

В данной работе будут рассмотрены некоторые проблемы современной мобильной разработки. В частности проблема схожих по функционалу приложений и переиспользование кода между ними. Сформирвано понятие «слабой зависимости». Проведено исследование существующих решений для разработки под мобильные платформ, их спецификации и наличие решенной проблемы «слабых зависимостей». Сформированы спецификации для будущей реализации инструментального окружения, а так же составлен план на реализацию функционала инструментального окружения для реализации ее в рамках выпускной работы.

Введение

Мир большими шагами вступил в эпоху мобильных и носимых устройств. В 2007 году бы представлен iPhone, которых "взорвал" рынок смартфонов. Спустя почти 10 лет мы уже не представляем свою жизнь без смартфона в кармане с кучей разнообразных приложений от редактирования фотографий до управления квадрокоптерами, от игр до приложений для управления проектами, от мессенджеров до видео-конференций и стриминговых платформ. За эти 10 лет “взлетели” и “умерли” такие хиты как “Vine”, “Flappy bird”, “Pokémon go” и многие другие. За эти годоы технические возможности смартфонов возросли многократно. Например, на текущий момент последний выпущеный iPhone 7s в 120 раз производительнее iPhone 2007 года. Кроме всего прочего появились умные гаджеты, которыми можно управлять с телефона и интегрировать с ним. Чего только стоит растущий рынок смарт-часов и фитнес-браслетов.

На данный момент конкуренция на рынке мобильных приложений очень высока. Раньше приложению достаточно было предоставить пользователью какой нибудь новый пользоватеьский опыт или возможность, аналога которой нет сейчас на рынке, и приложение появлялось почти на каджом смартфоне и расширяло свою аудиторию тысячекратно. Самыми яркими примерами являются Instagram и Uber. Они являются неоспоримыми (по крайней мере пока что) лидерами в своём сегменте во многом из-за того, что несколько лет назад “так как они никто не делал”. Instagram позволил отредактировать фотографию и поделиться ей с сотнями тысяч пользователей буквально в несколько кликов. Uber - вызвать личный автомобиль в пару кликов. Сейчас у них есть сотни конкурентов по всему

миру, но тем не менее они о сих пор держат лидерство на рынке. Сейчас же гораздо сложнее просто сделать что то новое. Пользователи уже привыкли в эргономичность дизайну и откликам за доли секунды. Поэтому сейчас как никогда важно найти равновесие между скоростью разработки, оптимизацией приложения и хорошим UI/UX.

Формулировка проблемы

На идею данной работы меня натолкнул опыт компании “Рамблер и Со” и один из проектов на работе. Представим себе, например, приложение по типу “Афиша” (см. рис. 1). Например, мы его пишем для какого нибудь театра. В нем требуется возможность просмотра ближайших спектаклей, их описание, отзывы о них, заказ билетов и просмотр информации о ближайших фестивалях. И оно будет иметь какой ниубдь станадртный табличный интерфейс и в нем будет использован, конечно же, фирменный стиль приложения: иконки, картинки, цветовая палитра бренда данного театра. И есть еще один заказ, но уже от кинотеатра. Им тоже требуется просмотр ближайших сеансов, описание фильмов в прокате, заказ билетов, возможность авторизации пользователя в приложении, личный кабинет, настройки приложения, привязка банковской карты к пользователю и раздел акций. И в это приложении так же будет табличный интерфейс и свой фирменный стиль. Как вы видите, есть некоторое пересечение возможностей этих двух приложений, но и есть возможности, которые уникальны для каждого приложения. И никому не хочется ни писать приложение с нуля для каждого заказчика, ни использовать банальный “копипаст” кода из одного проекта в другой.

Для этого, конечно, адекватнее всего использовать некоторый менеджер зависимостей. Но хотелось бы сделать так, что бы можно было только указать необходимые для приложения модули в некоторой спецификации и автоматически сгенерировать по ней проект, готовый к сборке. Но при этом, если каких то необязательных модулей нет в конфигурации, то модули, использующие их, могли обработать их отсутствие

и работать с имеющимся функционалом.

рис. 1 Пример двух схожих по функционалу приложений

В данной работе будет рассмотрены некоторые существующие менеджеры зависимостей, их возможности. Так же будут сформированы модели для инструментального окружения. Разрабатываться инструментальное окружение будет для платформы iOS. В дальнейшем возможно так же расширение функциональности и для сборки для платформы Android.

Перед началом обзора дадим несколько основных определений:

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 UI составляющую.

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

Слабая зависимость – зависимость меджу двумя модулями, когда один из них может использовать фунционал второго модуля, но при всем этом присутствие второго модуля не является обязательным и первый модуль имеет возможность обрабатывать его отсутствие не вызывая ошибок компиляции или runtime.

Существующие решения

Рассмотрение существующих решений будем проводить по следующим основным пунктам:

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

CocoaPods

Описание

CocoaPods - это менеджер зависимостей уровня приложений для Objective-C, Swift и любый 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 софкусирован на дистрибюции 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 толкьо 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" сиситемных 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'a. 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 опиций для конфигурации различных сборок, но рассмотрим здесь только некоторые осноные:

- target – в нем указывается имя Target’a, 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 – 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. 2) 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.

```
import PackageDescription

let package = Package(
    name: "DeckOfPlayingCards",
    targets: [],
    dependencies: [
        .Package(url: "https://github.com/apple/example-package-fisheryates.git",
                 majorVersion: 1),
        .Package(url: "https://github.com/apple/example-package-playingcard.git",
                 majorVersion: 1),
    ]
)
```

рис. 2 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 – 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. 3), 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.

```
<?xml version="1.0"?>
    <package xmlns="http://schemas.microsoft.com/packaging/2013/05/nuspec.xsd">
      <metadata>
        <id>Contoso.Utility.UsefulStuff</id>
        <version>1.8.3.331</version>
        <authors>Dejana Tesic, Rajeev Dey</authors>
        <owners>dejanatc, rjdey</owners>
        <licenseUrl>http://opensource.org/licenses/MS-PL</licenseUrl>
        <projectUrl>http://github.com/contoso/UsefulStuff</projectUrl>
        <iconUrl>http://github.com/contoso/UsefulStuff/nuget_icon.png</iconUrl>
        <requireLicenseAcceptance>false</requireLicenseAcceptance>
        <releaseNotes>Bug fixes and performance improvements</releaseNotes>
        <description>Core utility functions for web applications</description>
        <dependencies>
          <dependency id="Newtonsoft.Json" version="9.0" />
        </dependencies>
      </metadata>
      <files>
        <file src="readme.txt" target="" />
      </files>
    </package>
```

рис. 3 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 распространяется 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 – 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.

```
<dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.4</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.powermock</groupId>
            <artifactId>powermock-reflect</artifactId>
            <version>${version}</version>
        </dependency>
        <dependency>
            <groupId>org.javassist</groupId>
            <artifactId>javassist</artifactId>
            <version>3.13.0-GA</version>
            <scope>compile</scope>
        </dependency>
    </dependencies>
```

рис. 4 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. 4).

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 зависимоти.
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».

Решение

Как было показано в предыдущей главе, ни одно из существующих решений не поддерживает «слабые зависимости». Но тем не менее нет никакого смысле писать заново какой либо из существующих менеджеров пакетов. Так что было решено создать некоторую надстройку над CocoaPods, которая могла бы разрешать проблемы слабых зависимостей.

Каджый модуль будет иметь обязательное описание в виде json-файла со следующими полям:

- Name – уникальное имя пакета. Требуется для его идентификации в системе и именно он будет указываться в конфигурационном файле сборки. В соответствие с именем модуля будет ставиться конфигурация pod-пакета.
- Description – некоторое описание модуля
- Type – тип модуля. В зависимости от типа модуля, на него будут накладываться некоторые необходимые для сборки ограничения. Вот список возможных типов:
    - sys – системный модуль. Содержит в себе какую-то релаизованную функциональность.
    - view – UI-модуль. Это модуль экрана, в котором реализована некоторая пользовательская логика и некоторое количество экранов приложения (более одного). Например модуль покупки билета будет иметь: экран выбора привязанной карты, экран ввода данных карты, экран подтверждения оплаты.
- rootClass – здесь указывается имя класса, в котором инициализируется объект модуля. В нем могуть быть поля зависимостей и во время

генерации сборки будет сгенерирован код, который протягивает все необходимые зависимости модуля

- rootInitialize – название метода инициализации модуля. Код его вызова будет добавлен сразу после протягивания ссылок ко всем необходимым зависимостям модуля. В это методе инициализируются все внутренности модуля с использованием присутствующих в модуле зависимостями.
- protocolName – имя протокола, по которому получается доступ к модулю извне. rootClass обязан наследоваться от этого протокола, иначе во время сборки будет выдаваться ошибка. Код протокола должен лежать в отдельном файле.
- protocolFile – локальный путь до файла, в котором описан протокол.
- strongDependencies – массив модулей-зависимостей для текущего модуля. В массиве перечислены имена модулей.
- weakDependencies – массив «слабых зависимостей». В момент генерации шаблона проекта специальным флагом можно будет указать подгрузить слабые зависимости или нет. Если будет стоять флаг «нет», то в проект добавиться только протокол доступа к слабой зависимости, но не исходный код. Так что ее функциональность будет отсутствовать. Это необходимо для тестирования когда на наличие ошибок компиляции и runtime.

Вся сборка приложения будет тоже иметь вид json’a. В ней будет два основных поля:

- modules – список имен модулей, которые будут встроены в проект

- rootModule - название UI-модуля, который будет корневым. Это означает, что при инициализации приложения он будет инициализирован первым и будет сразу отображен на экране.

По этим конфигурационным файлам будет происходить поиск модулей, установка их в проект при помощи CocoaPods, генерация кода для их инициализации и управления их зависимостями.

Заключение

В данной работа была рассмотрена проблема генерации однотипных приложения с разной функциональностью. Так же было введено понятие «слабой зависимости», которое является основной причиной формирования данной проблемы и необходимость ее решения в рамках управления зависимостями. Были рассмотрены несколько существующих решений связанных с управлением зависимостями. В виду отсутствия какой либо реализации концепции слабой зависимости были сформированы спецификации для модуля приложения и сборки приложения для iOS, основанных на база Cocoapods.

Планы на выпускную работу

Выпускная работа будет посвященная реализации инструментального окружения для разработки под iOS. В дальнейшем, возможно реализация такой же концепции для Android платформ, хотя существуют проблемы с разрешением зависимостей и сильным различием между менеджерами зависимостей данных платформ.

В выпускной работе будет реализован следующий инструментарий и функционал системы:

- Инструмент для формирования проекта Модуля из файла-спецификации. Необходим для генерации по манифест-файлу модуля пустого проекта, с подключенными в него внешними и внутренними зависимостями. Будет реализован на языке Ruby, библиотеки для Ruby «Xcodeproj» и CocoaPods.

- Генератор набора файлов модуля на базе концепции SOLID и наборе рекомендаций VIPER.
- Сервис для распространения пакетов с помощью CocoaPods.
- Инструмент для генерации кода сборки проекта исходя из файлов конфигурации
- Реализация концепции «слабой зависимости»
- Исследование о возможности реализации концепции «слабой зависимости» для других мобильных платформ.

Список литературы

1. Конференция Rambler iOS #6 / Станислав Цыганов / Feature toggle. / 6.04.2016
2. Шаблоны копрпоративных приложений / Матрин Фаулер, Девид Райс / Вильямс / 2016 г. 544 стр.
3. Nuget 2 Essentials/ Ozon / 2013 г. – 253 стр.
4. CocoaPods Official Site [Электронный ресурс] URL: https://cocoapods.org/ (дата обращения 12.11.2016)
5. Swift. Основы разработки приложений на iOS / Усов В.А. / Питер / 201 г. – 304 стр.
6. Swift.org [Электронный ресурс] URL: https://swift.org/ (дата обращения 13.11.2016)
7. NuGet Documentaion. Microsoft Docs [Электронный ресурс] URL: https://docs.microsoft.com/en-us/nuget/ (дата обращения 16.11.2016)
8. Maven Official Site [Электронный ресурс] URL: https://maven.apache.org/ (дата обращения 18.11.2016)
9. Руководство по Maven [Электронный ресурс] URL: http://www.apache-maven.ru/ (дата обращения 21.11.2016)