Введение.

Формирование

В текущей предметной области дадим определения некоторым основным понятиям, которые будут использоваться в Дипломной работе

**Приложение** –

**Модуль** - логическая часть приложения с четко определенной функциональностью, является самодостаточной частью приложения, выполняющий четко определенную задачу и умеющий, при необходимости, инициировать переход на к другим модулям-экранам.

**Модуль-экран** - модель, который имеет UI составляющую.

**Зависимость** -

## Описание

## Что из себя представляют пакеты?

## Как пакеты интегрируются в текущее приложение?

## Работа со слабыми зависимостями

Существующие решения

# CocoaPods

## Описание

CocoaPods - это менеджер зависимостей уровня приложений для Objective-С, 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

В каком виде представлены "пакеты":

Спецификация:

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 спкцификации много возможных полей, но не все из них одинаково полезны. Рассмотрим толкьо некоторые из них:

spec.version - поле, определяющее номер версии Pod библиотеки. Пример: '3.1.0'

spec.source - содержит в себе информацию о пути к репозиторию, а так же номеру верисс библиотеки в репозитории. Имеет вид: { :git => 'https://github.com/tonymillion/Reachability.git', :tag => 'v3.1.0' }

spec.source\_files - регулярное выражение, описывающее все необходимые файлы для библиотеки. Пример: 'Reachability/common/\*.swift'

spec.{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'

spec.framework - Список системных фреймворков необходимых для этой библиотеки. Пример: 'SystemConfiguration'

spec.{OS}.framework - Список системных фреймворков необходимых для этой библиотеки. {OS} может быть ios, osx. Пример: 'UIKit', 'Foundation'

spec.dependency - В каждом таком поле указываются зависимости текущей библиотеки от других. Например: 'RestKit/CoreData', '~> 0.20.0'

spec.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.

spec.compiler\_flags = '-Wno-incomplete-implementation -Wno-missing-prototypes'

spec.subspec - Тоже самое описание спецификации, но только длля "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

В каком виде представлено приложение и как в него интегрируются "пакеты":

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)

## Что из себя представляют пакеты?

## Как пакеты интегрируются в текущее приложение?

## Работа со слабыми зависимостями

# Package Manager

## Описание

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

Пакеты состоит из файлов исходного кода и 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.

## ~~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?

## Работа 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, 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.

Any type other than Dependency and DotnetCliTool, however, are not recognized by the NuGet Package Manager in Visual Studio.

Nupkg представляет из себя архив, который содержит в себе с !!!

## Как пакеты интегрируются в текущее приложение?

## Работа 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, что-нибудь еще)

**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.

## Как пакеты интегрируются в текущее приложение?

## Работа со слабыми зависимостями