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**

Выпускная квалификационная работа бакалавра

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

Содержание

[Введение 3](#_Toc485828482)

[Обзор существующих решений 7](#_Toc485828483)

[Конструкторы приложений 7](#_Toc485828484)

[Менеджеры пакетов 9](#_Toc485828485)

[Рассмотрение менеджеров пакетов будем проводить по следующим основным пунктам: 9](#_Toc485828486)

[1. CocoaPods 10](#_Toc485828487)

[2. Swift Package Manager 13](#_Toc485828488)

[3. NuGet 16](#_Toc485828489)

[4. Maven 18](#_Toc485828490)

[Выводы 20](#_Toc485828491)

[Архитектура и модели инструментального окружения 22](#_Toc485828492)

[Архитектура инструментального окружения 26](#_Toc485828493)

[Технологические процессы и алгоритмы 29](#_Toc485828494)

[Реализация системы 31](#_Toc485828495)

[Пример использования инструментального окружения. 33](#_Toc485828496)

[Заключение 38](#_Toc485828497)

[Список литературы 40](#_Toc485828498)

[Приложения 43](#_Toc485828499)

# Введение

Мобильные устройства стали частью повседневной жизни. На рынке множество различных компаний, производящих мобильные устройства, и этот рынок растет невероятными темпами. 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. Преимуществом такого подхода является возможность автоматизации процесса сборки и абстрагирование от данного процесса программиста. Кроме того, методология модульного программирования предполагает возможность эффективного переиспользования уже написанного исходного кода и результатов тестирования.

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 модуля, реализующего HTTP-запросы. Данный тип взаимодействия между модулями назовем "зависимостью". 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. Если модуль может каким либо образом обработать отсутствие свой зависимости, то такой тип взаимодействия между ними будем называть "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 реализация прототипа такого инструментального окружения, которое позволит использовать выше перечисленные принципы.

Для достижение поставленной цели необходимо решить следующие задачи:

* Исследовать существующие решения и проанализировать возможность их использования в рамках данной работы.
* Описать архитектуру итоговой системы
* Реализовать данной инструментальной окружение

В главе "Обзор существующих решений" рассмотрены возможные готовые решения, доступные для использования в мобильной разработке - конструкторы приложений и менеджеры пакетов. 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 менеджера пакетов CocoaPods.

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

* + Описать основные понятия инструментального окружения.
  + Спроектировать используемые модели модуля и приложения.
  + Спроектировать архитектуру системы.
  + Описать компоненты и способы их взаимодействия.
  + Описать используемые алгоритмы для разрешения зависимостей, валидации модуля и сборки.
  + 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

В данной главе обозреваются существующие технологии в современной разработка для мобильных платформ, такие как конструкторы приложений и менеджеры пакетов: конструкторы приложений позволяют «собирать» приложения для мобильных платформ из готовых «блоков»; менеджеры пакетов предоставляют возможности управления зависимостями приложения и модулей, функционал для внедрения пакетов в готовое приложение и поддержка версионности пакетов. В рамках данной работы не рассматриваются никакие реализации Case-средств для разработки, так как такие решения являются для каждой компании своеобразны «know-how», что накладывает на них определенные права доступа. Инструментальное окружение, разрабатываемое в рамках данной работы, нацелено на общедоступное использование.

## Конструкторы приложений

**Конструкторы приложений** это сервисы, которые позволяют создавать приложения 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. 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

Менеджеры пакетов - набор программного обеспечения, позволяющего управлять процессом установки, удаления, настройки и обновления различных компонентов программного обеспечения.

В мобильной разработке менеджеры пакетов являются одним из основных инструментов. Они пользуются огромным спросом и в интернете существуют огромные базы пакетов, которые могут быть использованы абсолютно бесплатно. Кроме того, создание собственных пакетов является простой и доступной процедурой, что способствует развитию данных инструментов. В данной раздел будут рассмотрены основные менеджеры пакетов, используемые в разработке для разных мобильных платформ: iOS, Andriod и Windows Phone.

## 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. Поддерживаемые платформы.
5. 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 версии 1.0. Так же было выпущено десктопное приложение. Проект CocoaPods был вдохновлен комбинацией 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 фреймворка, 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 инструкции 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’».

#### Поддерживаемые платформы

CocoaPods поддерживает работу с платформами iOS и macOS

#### 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.

#### Поддерживаемые платформы

Swift Packager Manager поддерживает работу с платформами iOS и macOS, но только с библиотекой Foundation. Это значит, что при использовании Swift Package Manager невозможно реализовать пакет с UI составляющей

#### 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 распознается автоматически пакетным менеджеров NuGet в Visual Studio.

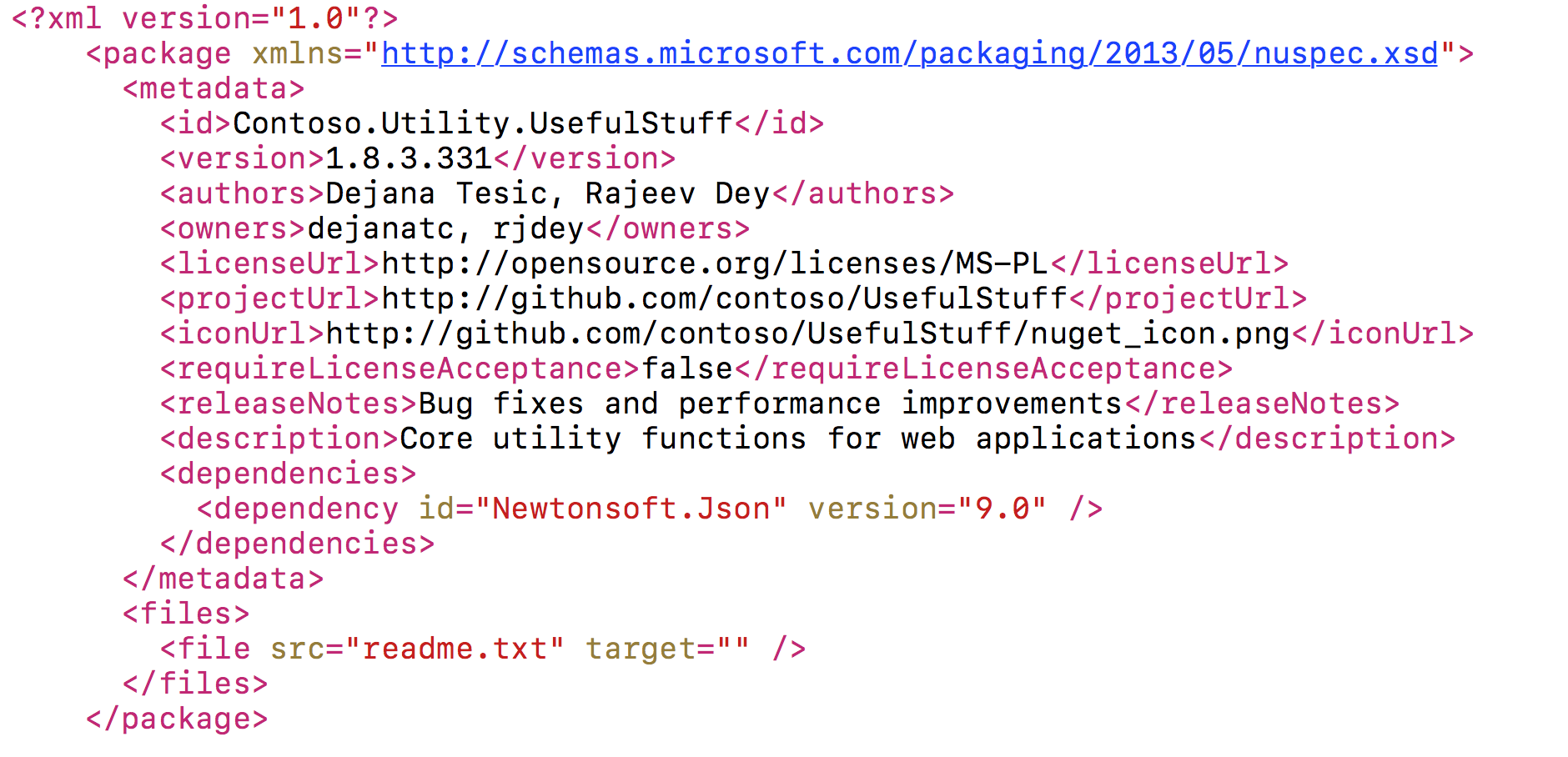


рис. 2 Пример манифеста NuGet

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 распространяется вместе с ней. Так что любой проект, созданный в Visual Studio начиная с 2011 года имеет возможность работы с NuGet «из коробки».

#### Поддерживаемые платформы

NuGet поддерживает работу с платформами работающими на базе Windows и .Net.

#### 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

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.

#### Поддерживаемые платформы

Maven поддерживает работу с любыми платформами, поддерживающими работу с Java Virtual Machine.

#### 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, полный функционал конструкторов приложений являются платной услугой, которой занимаются другие разработчики, что сложно назвать автоматизацией процесса разработки. 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** – программное обеспечение, 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

**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 функционал 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 .

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 этапе сборки готового приложения генерировать код сборки для конкретного модуля.

Манифест модуля содержит в себе следующие поля (см. Приложение 1):

* Name – уникальное имя пакета. Требуется для его идентификации в системе и именно он будет указываться в конфигурационном файле сборки.
* 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 реализованную функциональность.
  + view – UI-модуль. Это модуль экрана, в котором реализована некоторая пользовательская логика и некоторое количество экранов приложения (более одного). При указании модуля как UI-модуля должны быть описаны поля «view».
* view – название переменной, которое является носителем UI-составляющей. Для каждой мобильно операционной системы у поля «view» будут свои ограничения. Ограничения для платформы iOS будут описаны в главе «Реализация системы»
* rootClass – имя класса, в котором инициализируется объект модуля. В нем могут быть поля зависимостей и во время генерации сборки будет сгенерирован код, который протягивает все необходимые зависимости модуля
* rootPath ­ путь для фала, содержащего описание класса «rootClass».
* rootInitialize – название метода инициализации модуля. Код его вызова будет добавлен сразу после протягивания ссылок ко всем необходимым зависимостям модуля. В это методе инициализируются содержимое модуля с использованием присутствующих в модуле зависимостей.
* protocolName – имя протокола, по которому получается доступ к модулю извне. rootClass обязательно должен наследоваться от этого протокола, иначе модуль не является валидным. Код протокола должен лежать в отдельном файле.
* protocolFile – локальный путь до файла, в котором описан протокол.
* strongDependencies – словарь соответствия переменным интерфейса и модулей-зависимостей для текущего модуля. В процессе валидации для каждого модуля будет устанавливаться соответствие типа переменной и название интерфейса модуля. Кроме того каждая переменная должна быть указана, как переменная, значения которой не может быть «nil». Данное ограничение устраняет неопределенность при использовании модуля как зависимости и при сборке приложения.
* weakDependencies – словарь соответствия переменным интерфейса и модулей-зависимостей для текущего модуля. В процессе валидации для каждого модуля будет устанавливаться соответствие типа переменной и название интерфейса модуля. Кроме того каждая переменная должна быть указана, как переменная, значения которой может быть «nil». Данное ограничение устраняет неопределенность при использовании модуля как зависимости и при сборке приложения.

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.

Манифест приложения содержит в себе следующие поля:

* name – название приложения
* 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.
* resourseUrl – путь с ресурсам приложения

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

В данной главе описаны функциональные требования в инструментальному окружению и его архитектуре, сама архитектура и причины ее выбора, ее компоненты и их функции. Кроме того, описаны используемые модели и их назначение. Так же описаны технологические процессы создания нового модуля и нового приложения. Описан алгоритм для разрешения зависимостей при сборке приложения и алгоритм валидации модуля.

## Архитектура инструментального окружения

Построение архитектуры системы – важный этап разработки любой информационной системы. От этого в много зависит сложность, масштабируемость и гибкой итоговой системы. В рамках данной работы архитектура инструментального окружения должна отвечать следующим основным требованиям:

1. Разработка модуля может вестись независимо на разных машинах.
2. Каждый модуль может иметь несколько версий в процессе его разработки.
3. Сборка готового приложения может производиться на любой машине.
4. Различные модули могут храниться на разных машинах

Использование системы контроля версий (СКВ) в данном случае является отличным решением, когда модуль может разрабатываться на разных машинах. Кроме того версионность исходного кода изначально заложена в концепцию СКВ, что позволяет всегда иметь в доступе стабильную версию модуля, которая может использоваться в сборке.

В следствии некоторых ограничений мобильных платформ [21], сборка мобильных приложений для определенных мобильных платформ не может производиться на базе не предназначенной для этого операционной системы, но в то же время на хранение и обработку такого исходного кода ограничений нет. Именно на этом моменте появляется необходимость в разделении функционала системы. Если ограничений на хранение нет, то можно использовать сервер на базе любой бесплатной OC, а разработку вести на машине с необходимой операционной системой. Кроме того, репозиторий СКВ может занимать значительное дисковое пространство, что накладывает сильные ограничения на размер жесткого диска.

Все выше перечисленное и условие удаленной разработки одного модуля приводит к решению разделить функциональность системы на серверную и клиентскую. Сервер будет осуществлять хранение и дистрибуцию модулей, хранение мета-данных о модулях и координацию сборки приложений. На клиента, в свою очередь, будет осуществляться непосредственно разработка модулей и сборка готовых приложений. В процессе разработки так же будет существовать возможность использовать уже существующие модули при разработке в виде сильных или слабых зависимостей. Более подробная схема архитектуры представлена на рисунке 4.

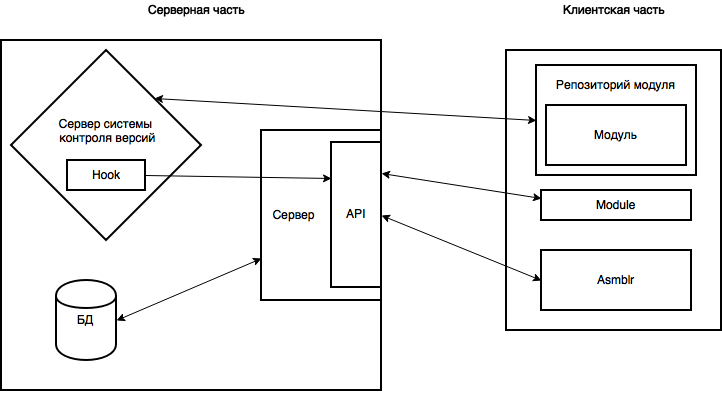


Рис. 4 Архитектура инструментального окружения

Подробнее опишем компоненты данной архитектуры:

Сервер СКВ – хранилище системы контроля версий, в котором хранится исходный код всех модулей и мета-данные о них. К каждом репозиторию «прикреплен» специальный Hook.

Hook – это скрипт, обрабатывающий события репозитория, например «push». С помощью данного скрипта существует возможность валидации модуля на основании его манифеста непосредственно после обновления его на стороне сервера 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. Только что загруженный на сервер модуля будет доступен для использования сборки, если его данные валидны. Данное решение позволит избегать ситуации, когда незаконченный модуль попадет в сборку приложения (см. Приложение 2 и 3).

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. Кроме того, в БД хранятся история о валидации модулей. Это позволяет реализовать единый функционал валидации на сервере и предоставлять данное о валидации разработчику через API.

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 файлы сборки со всеми необходимыми для сборки мета-данными (см. Приложение 4).

Module – программа, помогающая разработчику при написании модуля. В её функции входят создание манифеста, установка и обновление зависимостей, и получении сообщений валидации модуля после отправки его на сервер (см. Приложение 5).

Asmblr – программа, формирующая на основе манифеста приложения готовое к сборке приложение. Программа Asmblr работает напрямую с API сервера, так что для ее работы необходимо наличие стабильного соединения. На вход программе подается манифест приложения. Результатом ее выполнения является готовое к сборке приложение (см. Приложение 6).

## Технологические процессы и алгоритмы

В первую очередь, технологические процессы описывают поведение описывает алгоритмы взаимодействия разработчика с системой. Кроме того, из технологических процессов будут выведены алгоритмы, которые будут реализовываться самим инструментальным окружением. Это позволить в последствии оценить объем работы, которая была автоматизирована за счет использования инструментального окружения.

Технологический процесс написания модуля выглядит следующим образом:

1. Создание модуля. На данном этапе создается проект модуля, манифест для менеджера пакетов и манифест модуля.
2. Написание модуля. Описание интерфейса доступа к модулю
3. Написание манифеста модуля со всей необходимой для сборки информацией информации об интерфейсе.
4. Опционально, установка и обновление зависимостей модуля.
5. Отправка исходного кода модуля на сервер при помощи команды «push» и обработка последнего коммита специальным скриптом.

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 манифеста приложения программе-сборщику Asmblr.
3. Итог работы программы – проект приложения, в который интегрированы все необходимые зависимости, а также в котором присутствует сгенерированный файл сборки приложения.

Тогда алгоритм работы программы Asmblr будет следующим:

* Asmblr отправляет данные из манифеста на сервер.
* На стороне сервера строится граф зависимостей и генерируется файл сборки, который отправляется обратно на клиент.
* Asmblr на основании файла сборки создает проект приложения и загружает в него все необходимые зависимости.
* 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.
* 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

# Реализация системы

В данной главе будут описаны особенности реализации инструментального окружения, его принятые ограничения и правила при построении модулей и приложений.

Инструментальное окружение в рамках данной выпускной квалификационной работы разрабатывалось для сборки приложений для платформы iOS. Для того, что бы решение получилось независимым от платформы было решено реализовывать его на языке Ruby. Кроме того, менеджер пакетов CocoaPods реализован тоже на Ruby и использует библиотеку Xcodeproj[23], которая позволяет редактировать проекты Xсode[24]. Использование данной библиотеки очень сильно упрощает итоговую сборку приложения.

В качестве системы контроля версий решено было использовать Git. Git – распределенная система управления версиями. Она открытая и может работать на любой платформе. Кроме того, CocoaPods интегрирован с Git, что, в рамках данной работы, является большим плюсом. Именно в Git-репозитории будет храниться каждый отдельный модуль. Версионность модулей поддерживается средствами системы, CocoaPods и Git. Кроме того в Git поддерживается использование скриптов на события репозитория, такие как обновление репозитория, что позволяет эффективно реализовать обновление мета-данных модуля на сервере. Так как Git-Hook’а нам необходим только сигнал о том, что модуль на сервере обновился, то было решено в скрипте Hook’а отправлять POST-запрос на сервер через API с данными о репозитории, мета данные которого нужно обновить. А валидацией и самим обновлением мета-данных уже занимается сервер, что позволяет обновлять данные централизованно и логику валидации описать только один раз.

Были введены следующие ограничения при разработке модуля:

* Манифест модуля должен находиться в той же директории, что и файл спецификации пакета CocoaPods .podspec.
* Манифест модуля должен быть только один.
* Файл спецификации модуля, который будет использоваться при сборке должен носить имя пакета CocoaPods (имя дирректории + ".podspec")

Были введены следующие правила при валидации модуля на сервере:

* При анализе интерфейса используются только переменные с правами доступа {get set}
* При большем количестве зависимостей чем переменных, используются толкьо те зависимости, для которых есть переменные.
* Переменная описанная с знаком ? является слабой зависимостью по определению.
* Переменная описанная без знака или со знаком ! является сильной зависимостью по определению
* Любое несоответствие ведет к ошибке. Текст ошибки отправляется в БД для хранения.

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]. Это позволяет тестировать функциональность модуля и его зависимостей без использования законченной их реализации, что сильно упрощает процесс разработки.

## Пример использования инструментального окружения.

Для примера работы системы, реализуем приложение, которое будет сообщать нам температуру на улице.

Данное приложение будет состоять из 3х модулей:

* Модуль «weather», который будет отображать температуру на экране телефона. Модуль «weather» имеет слабую зависимость от модуль «weatherOnline». Если модуля «weatherOnline» нет, что на экране будет выводиться сообщение о том, что погода на улице хорошая.
* Модуль «weatherOnline», который будет запрашивать через http погоду с сервера. Модуль «weatherOnine» имеет сильную зависимость от модуля «http»
* Модуль «http», который реализует http запросы.

Командой вызовом программы «module.rb» с параметром «create» инициализируется создание модуля. Для него создается локальный репозиторий и средствами CocoaPods создается пустой тестовый проект, для разработки модуля.

Кроме того в корне репозитория лежит файл .modfst, который представляет из себя обычный json и содержит мета информацию о модуле (Примеры на рисунках 5, 6 и 7 ).

Рисунок 5. Содержимое манифеста модуля «http»

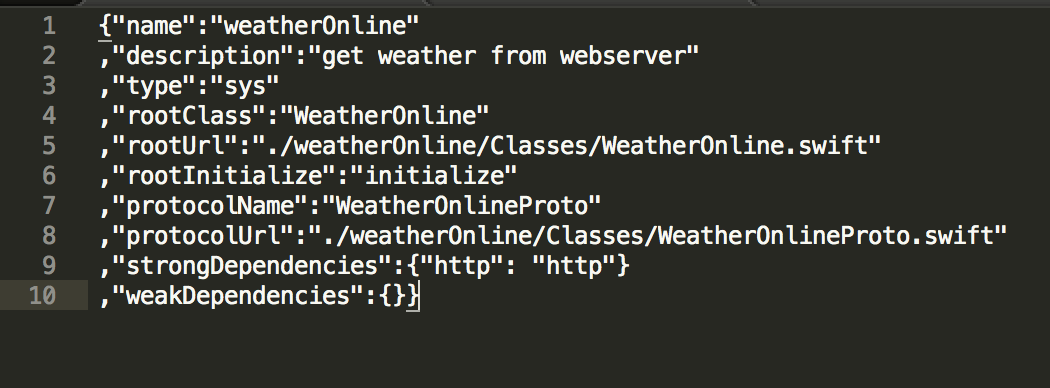


Рисунок 6. Содержимое манифеста модуля «weatherOnline»

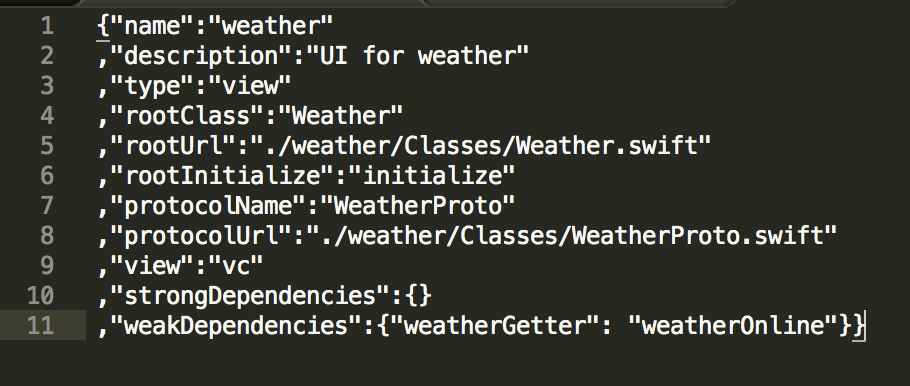


Рисунок 7. Содержимое манифеста модуля «weather»

Далее разработка каждого модуля ведется таким же образом, как и разработка обычного пакета CocoaPods.

После того, как завершилась разработка модуля «http», его исходный код отправляется на сервер, валидируется и мета-данные о нем добавляются в БД модулей.

Теперь в директории с модулем weatherOnline вызываем команду «module update», которая добавляет модуль «http» в тестовый проект модуля.

Далее ведется разработка модуля «weatherOnline». По завершении, модуль так же отправляется на сервер командой «git push», валидируется на сервере и его мета-данные попадают в БД модулей.

Теперь в директории с модулем weather вызываем команду «module update», которая добавляет модуль «weatherOnline» в тестовый проект модуля. Кроме данного модуля еще добавим в файле podspec зависимость от пакета «PureLayout», который является сторонним пакетом CocoaPods и предоставляет удобный функционал по построению UI.

Модуль «weather» тоже загружается на сервер командой «git push», валидируется и данные о нем добавляются в БД модулей.

Теперь создаем манифест приложения. Он представляет из себя тоже json-файл (Рис. 8).

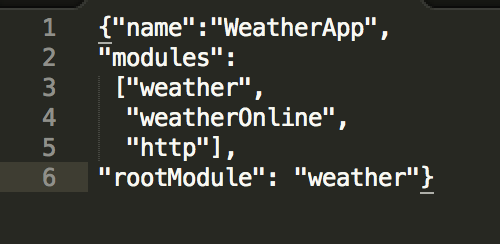


Рис. 8 Манифест приложения WeahterApp

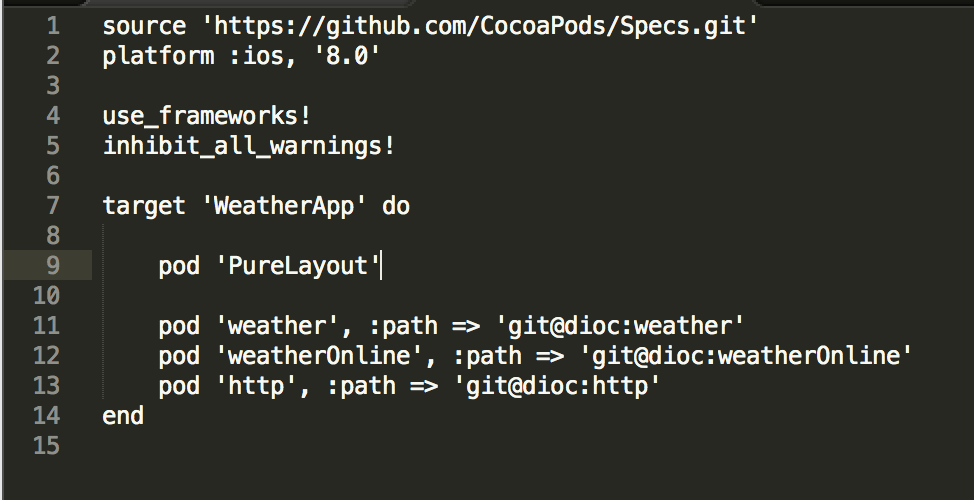
Приступаем к сборке. Запускаем программу Asmblr.rb c единственным параметром – путем к файлу с манифестом приложения. На стороне сервера формируется файл сборки приложения и отправляются обратно на клиент. На базе файл сборки формируется проект приложения и в него интегрируются все необходимые зависимости при помощи файла Podfile (Рис. 9). 

Рис. 9 Содержимое Podfile приложения WeatherApp

Далее создается файл сборки приложения, который инициализирует все необходимые зависимости и собирает готовое приложение (см. Приложение 7).

На выходе получен проект уже полностью работающего приложения, полностью собранного из уже написанных модулей без написания ни единой строчки кода. Процесс его работы показан на рисунке 9.

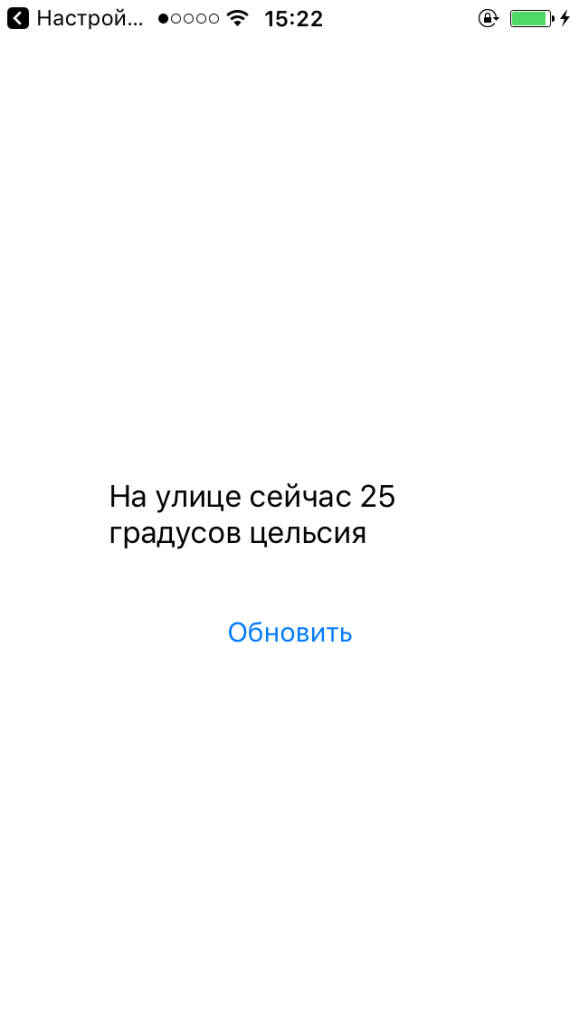


Рис. 9 Приложения WeatherApp

# Заключение

В рамках данной выпускной работы бакалавра была сформулирована проблема создания схожих по функционалу приложений и возможность ускорения сборки готовых приложений на базе уже написанного кода. Была разработана концепция слабой зависимости, которая позволяет реализовать инструментальное окружения для сборки готовых приложений для мобильных платформ. Была поставлена цель в рамках данной работы разработать такое окружения для сборки приложений для платформы iOS и цель бы достигнута.

Были рассмотрены уже существующие на рынке существующие решения. В рамках данного исследования было получено, что конструкторы приложений не подходят для данной задачи потому, что предоставляют слишком слабый функционал расширения возможностей приложения и являются в большинстве своем платными. Исследование возможностей менеджеров пакетов в свою очередь показали отсутствие в них какой либо реализации концепции слабой зависимости. Это главная причина по которой они не могу быть решением данной проблемы. Однако, менеджеры пакетов реализуют хорошие возможности управления зависимостями в приложении, что привело к решению использовать менеджеры пакетов как инструмент работы с пакетами в данной инструментальном окружении. Для текущей реализации инструментального окружения было решено использовать CocoaPods.

Были сформулированы основные понятия инструментального окружения, такие как модуль, приложение, слабые и сильные зависимости. Описана структура манифеста модуля и манифеста приложения и необходимые мета-данные для сборки приложения. Были описаны основные требования к реализации инструментального окружения. На базе этих требований было сформировано описание архитектуры будущей системы. Были писаны ее компоненты, их функционал, поведение. Так же были описаны технологические процессы создание модуля и сборки готового приложения, которые являются базовыми в рамках работы с данным инструментальным окружением.

Итогом работы является реализация инструментального окружения, всех его компонентов для сборки готовых приложений для мобильной платформы iOS. Реализована клиент-серверная архитектура системы и функционал каждой компоненты системы.

Для демонстрации жизнеспособности подхода был реализован прототип инструментального окружения для сборки готовых приложений для мобильных платформ. На базе прототипа было реализовано приложение «WeatherApp», процесс создания которого охватывал весь функционал инструментального окружения.

# 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)
21. CocoaPods/Xcodeproj [Электронный ресурс] URL: <https://github.com/CocoaPods/Xcodeproj> (дата обращения 31.05.2017)
22. Xcode – Apple Developer [Электронный ресурс] URL: <https://developer.apple.com/xcode/> (дата обращения 31.05.2017)

# Приложения

Приложение 1.

// Исходный код файла ModuleClass.rb

#!/usr/bin/env ruby

class Module

attr\_accessor :name,

:description,

:type,

:rootClass,

:rootUrl,

:rootInitialize,

:protocolName,

:protocolUrl,

:mockUrl,

:view,

:strongDependencies,

:weakDependencies

def initialize()

@name = ""

@description = "My brand new module"

@type = ""

@rootClass = ""

@rootUrl = ""

@rootInitialize = ""

@protocolName = ""

@protocolUrl = ""

@mockUrl = nil

@view = ""

@strongDependencies = Hash.new()

@weakDependencies = Hash.new()

end

def fill(json)

@name = json["name"]

@description = json["description"]

@type = json["type"]

@rootClass = json["rootClass"]

@rootUrl = json["rootUrl"]

@rootInitialize = json["rootInitialize"]

@protocolName = json["protocolName"]

@protocolUrl = json["protocolUrl"]

@mockUrl = json["mockUrl"]

@view = json["view"]

@strongDependencies = json["strongDependencies"]

@weakDependencies = json["weakDependencies"]

@name ||= ""

@description ||= ""

@type ||= ""

@rootClass ||= ""

@rootClass ||= ""

@rootInitialize ||= ""

@protocolName ||= ""

@protocolUrl ||= ""

@protocolName ||= ""

@strongDependencies ||= Hash.new()

@weakDependencies ||= Hash.new()

end

def to\_json()

mod = Hash.new()

mod["name"] = @name

mod["description"] = @description

mod["type"] = @type

mod["rootClass"] = @rootClass

mod["rootUrl"] = @rootUrl

mod["rootInitialize"] = @rootInitialize

mod["protocolName"] = @protocolName

mod["protocolUrl"] = @protocolUrl

mod["mockUrl"] = @mockUrl

mod["view"] = @view

mod["strongDependencies"] = @strongDependencies

mod["weakDependencies"] = @weakDependencies

return mod.to\_json()

end

def validate()

return @name != "" && @description != "" && @rootClass != "" && @rootUrl != "" && @rootInitialize != "" && @protocolName != "" && @protocolUrl != "" && ((@type == "view" && @view != "") || @type == "sys"))

end

end

Приложение 2.

// Исходный код программы post-update

#!/bin/sh

#

# An example hook script to prepare a packed repository for use over

# dumb transports.

#

# To enable this hook, rename this file to "post-update".

touch ~/pw

echo $PWD > ~/pw

curl 174.138.58.175:8000/update -d`echo $PWD`

$NAME = `basename $PWD`

cd ~

eval `git clone ~/$NAME`

Приложение 3.

// Исходный код программы validate.rb

#!/usr/bin/env ruby

require 'find'

require 'json'

require 'sqlite3'

require\_relative 'ModuleClass'

require\_relative 'ModuleProtoClass'

def findPod(dir)

podPaths = []

Find.find(dir) do |path|

podPaths << path if path =~ /.\*\.podspec$/

end

if podPaths.count == 1 then

return podPaths[0]

end

return nil

end

def findModule (dir)

modulepaths = []

Find.find(dir) do |path|

modulepaths << path if path =~ /.\*\.modfst$/

end

if modulepaths.count == 1 then

return modulepaths[0]

end

return nil

end

def sendReport (name, msg)

puts msg

db = SQLite3::Database.new "module.db"

db.execute("INSERT INTO validationReport (moduleName, msg)

VALUES (?, ?)", [name, msg])

exit

end

arg = ""

if ARGV.count != 2 then

exit

end

dir = ARGV[0]

commit = ARGV[1]

# puts dir

posName = dir.split('/').last.split('.').first

path = "git@dioc:#{posName}"

modPath = findModule(dir)

podPath = findPod(dir)

if modPath == nil || podPath == nil then

sendReport(posName, "module manifest or podspec didnt found")

end

if modPath.split('/').count != dir.split('/').count + 1 || modPath.split('/').count != modPath.split('/').count then

sendReport(posName, "module manifest and podspec must be in same folder")

end

mod = Module.new()

file = File.read(modPath)

data\_hash = JSON.parse(file) rescue nil

if data\_hash == nil then

sendReport(posName,"Manifest file has wrong fromat (has to be pure json)")

end

mod.fill(data\_hash)

if !mod.validate then

sendReport(mod.name == "" ? posName : mod.name,"Not all \"must have\" fields are filled in module manifest")

end

protoUrl = mod.protocolUrl.split('/')

protoUrl[0] = dir

protoUrl = protoUrl.join('/')

classUrl = mod.rootUrl.split('/')

classUrl[0] = dir

classUrl = classUrl.join('/')

proto = File.read(protoUrl) rescue nil

classRel = File.read(classUrl) rescue nil

if proto == nil then

sendReport(mod.name,"didnt find protocol in module")

end

if classRel == nil then

sendReport(mod.name,"didnt find class realisation")

end

protHead = proto.gsub(/protocol #{mod.protocolName}(| \*\n\*)(: \D[^ \.\<\>\?\/\:\;\!\\*\@\"\'\#\%\^\&\(\)\+\-\=]\*)\*{/i)

if protHead.count != 1 then

sendReport(mod.name,"wrong protocol format or multiple protocols")

end

protHead = protHead.each.next

protocol = ModuleProto.new()

protocol.name = protHead.split(' ')[1].split(':')[0].split('{')[0]

vars = proto.gsub(/var \D[^ \.\,\<\>\?\/\:\;\!\\*\@\"\'\#\%\^\&\(\)\+\-\=\[\]]\*(| \*\n\*)\*\:(| \*\n\*)\*\D[^ \.\,\<\>\?\/\:\;\!\\*\@\"\'\#\%\^\&\(\)\+\-\=]\*(|\?|\!)+ {get( set)?}/i)

vars.each do |str|

name = str.split(' ')[1].split(':')[0]

type = str.split(':')[1].split('{')[0].strip

protocol.vars[name] = type

end

puts protocol.vars

classHead = classRel.gsub(/class #{mod.rootClass}(| \*\n\*)\*(:\D[^ \.\<\>\?\/\:\;\!\\*\@\"\'\#\%\^\&\(\)\+\-\=]\*((| \*\n\*)\*,(| \*\n\*)\*\D[^ \.\<\>\?\/\:\;\!\\*\@\"\'\#\%\^\&\(\)\+\-\=]\*)\*)\*(| \*\n\*)\*{/i)

puts classHead.count

if classHead.count != 1 then

sendReport(mod.name,"wrong class format or multiple classes")

end

classHead = classHead.each.next

classInh = classHead.split(':')[1].split('{').first.split(',').map.with\_index{ |x, i| x.strip }

if !(classInh.include? protocol.name) then

sendReport(mod.name,"class does not release protocol")

end

classinit = classRel.gsub(/func #{mod.rootInitialize}(| \*\n\*)\*\(\)(| \*\n\*)\*{/i)

if classinit.count == 0 then

sendReport(mod.name,"no initialize method realized in root class")

end

db = SQLite3::Database.new("./module.db")

mod.strongDependencies.each do |key, value|

modules = db.execute("select \* from module where name = '#{value}'")

if modules.count == 1 then

protocolName = modules[0][1]

if !protocol.vars.include?(key) then

sendReport(mod.name,"no variable #{key} in protocol")

end

if !protocol.vars[key].include?(protocolName)

sendReport(mod.name,"wrong protocol for variable #{key}")

end

if protocol.vars[key].include?('?') then

sendReport(mod.name,"variable #{key} must be not optional")

end

else

sendReport(mod.name,"dependency #{value} does not exist")

end

end

mod.weakDependencies.each do |key, value|

modules = db.execute("select \* from module where name = '#{value}'")

if modules.count == 1 then

protocolName = modules[0][1]

if !protocol.vars.include?(key) then

sendReport(mod.name,"no variable #{key} in protocol")

end

if !protocol.vars[key].include?(protocolName)

sendReport(mod.name,"wrong protocol for variable #{key}")

end

if !protocol.vars[key].include?('?') then

sendReport(mod.name,"variable #{key} must be optional")

end

else

sendReport(mod.name,"dependency #{value} does not exist")

end

end

if db.execute("select \* from module where name = '#{mod.name}'").count == 1 then

db.execute("UPDATE module SET commitHash = '#{commit}'")

else

db.execute("INSERT INTO module (name, protocol, commitHash, path)

VALUES (?, ?, ?, ?)", [mod.name, protocol.name, commit, path])

end

Приложение 4.

// Исходный код программы server.rb

#!/usr/bin/env ruby

require 'webrick'

require 'sqlite3'

require 'json'

require 'set'

require\_relative 'AppClass'

require\_relative 'ModuleClass'

class Tree

attr\_accessor :name,

:commit,

:path,

:protocol,

:manifest,

:branches

def initialize()

@name = ""

@branches = []

@commit = ""

@path = ""

@protocol = ""

@manifest = Hash.new()

end

end

class Modules < WEBrick::HTTPServlet::AbstractServlet

def do\_GET(request, response)

response.status = 200

response['Content-Type'] = 'application/json'

result = Hash.new()

keys = ['name','protocol','commitHash','path']

db = SQLite3::Database.new("./module.db")

rows = db.execute("select \* from module")

modules = []

rows.each { |elem|

temp = Hash.new()

for i in 0..keys.count-1

temp[keys[i]] = elem[i]

end

modules.push(temp)

}

result["modules"] = modules

response.body = result.to\_json

end

end

class Validation < WEBrick::HTTPServlet::AbstractServlet

def do\_GET(request, response)

response.status = 200

name = request.query().keys.first

response['Content-Type'] = 'application/json'

puts request.query().keys.first

if name != nil then

db = SQLite3::Database.new("./module.db")

rows = db.execute("select \* from validationReport where moduleName = '#{name}'")

if rows.count == 1 then

response.body = '{"msg": "#{rows[0][1]}"}'

else

response.body = '{"error": "not validation msg found for that module"}'

end

else

response.body = '{"error": "invalid name"}'

end

end

end

class AppManifest < WEBrick::HTTPServlet::AbstractServlet

def do\_POST(request, response)

response.status = 200

body = request.body

data = JSON.parse(body) rescue nil

if data != nil

app = Application.new()

app.fill(data)

gottenModules = Set.new(app.modules)

db = SQLite3::Database.new("./module.db")

def getDeps(name, tree)

rows = db.execute("select \* from module where name = '#{name}'")

if rows.count == 1 then

tree = Tree.new()

tree.name = name

tree.protocol = rows[1]

tree.commit = rows[2]

tree.path = rows[3]

gottenModules.add(name)

path = "/home/git/#{name}/#{name}.modfst"

file = File.new(path, "w")

modjson = JSON.parse(file) rescue nil

if modjson != nil then

tree.manifest = modjson

mod = Module.new()

mod.fill(modjson)

mod.strongDependencies.each { |dep|

if !gottenModules.include?(dep) then

tempTree = getDeps(dep, tree)

if tempTree != nil then

tree.branches.push(tempTree)

end

end

}

end

return tree

end

return nil

end

def depth(tree, array)

locArray = array

data = Hash.new()

data["name"] = tree.name

data["protocol"] = tree.protocol

data["commit"] = tree.commit

data["path"] = tree.path

data["manifest"] = tree.manifest

if tree.branches.count == 0 then

locArray.push(data)

else

tree.branches.each { |br|

locArray = locArray + depth(br, array)

}

end

return locArray

end

rows = db.execute("select \* from module where name = '#{app.rootModule}'")

if rows.count == 1 then

tree = Tree.new()

tree.protocol = rows[1]

tree.commit = rows[2]

tree.path = rows[3]

Tree.name = app.rootModule

gottenModules.add(app.rootModule)

path = "/home/git/#{app.rootModule}/#{app.rootModule}.modfst"

file = File.new(path, "w")

modjson = JSON.parse(file) rescue nil

if modjson != nil then

tree.manifest = modjson

mod = Module.new()

mod.fill(modjson)

mod.strongDependencies.each { |dep|

if !gottenModules.include?(dep) then

tempTree = getDeps(dep, tree)

if tempTree != nil then

tree.branches.push(tempTree)

end

end

}

end

end

resultArray = depth(tree)

result["modules"] = resultArray

response['Content-Type'] = 'application/json'

response.body = result.to\_json

else

response['Content-Type'] = 'application/json'

response.body = '{"error": "invalid name"}'

end

end

end

class Update < WEBrick::HTTPServlet::AbstractServlet

def do\_POST(request, response)

response.status = 200

body = request.body

name = body.split('/').last.split('.').first

puts name

exec("git clone /home/git/#{name}.git")

dir = "~/#{name}"

commit = exec('git rev-parse HEAD')

exec('./validate.rb #{dir} #{commit}')

end

end

server = WEBrick::HTTPServer.new :Port => 8000

server.mount\_proc '/' do |req, res|

res.body = 'Hello, world!'

end

server.mount '/update', Update

server.mount '/modules', Modules

server.mount '/validation', Validation

server.mount '/app', AppManifest

trap 'INT' do server.shutdown end

server.start

Приложение 5.

// Исходный код программы Asmblr.rb

#!/usr/bin/env ruby

require\_relative 'ModuleClass'

require 'find'

require 'json'

def jsonToPodpec (hash)

result = "Pod::Spec.new do |s|

"

hash.each do |key, value|

if value.is\_a?(String) then

result += "s.#{key} = '#{value}', \n"

else

if key != "authors" then

tempres = "s.#{key} = {"

value.each do |tkey, tval|

tempres += ":#{tkey} => '#{tval}',"

end

tempres += "},"

else

tempres = "s.authors = {"

value.each do |tkey, tval|

tempres += ":#{tkey} => '#{tval}',"

end

tempres += "},"

end

tempres.sub!(",}", "}")

result+=tempres + "\n"

end

end

result += "\nend"

end

def findModule (dir)

modulepaths = []

Find.find(dir) do |path|

modulepaths << path if path =~ /.\*\.modfst$/

end

if modulepaths.count == 1 && modulepaths.split('/').count - 1 == dir.split('/').count then

puts modulepaths[0]

return modulepaths[0]

else

return nil

end

end

def findPod (dir)

podPaths = []

Find.find(dir) do |path|

podPaths << path if path =~ /.\*\.podspec$/

end

if podPaths.count == 1 then

podfile = podPaths[0]

return podfile

end

return nil

end

def install(dir)

folderPath = findPod(dir)

modulePath = findModule(dir)

if folderPath != nil && modulePath != nil then

# Server needed

end

end

def update(dir)

folderPath = findPod(dir)

modulePath = findModule(dir)

if folderPath != nil && modulePath != nil then

# Server needed

end

end

def create(dir)

folderPath = findPod(dir)

folderPath = folderPath.split("/")[0..folderPath.split("/").count-2].join("/")

folderName = folderPath.split("/")[folderPath.split("/").count-2]

if folderPath != nil then

podfile = folderPath + "/Podfile"

mod = Module.new()

puts "Write you module name (enter will set name \"#{folderName}\"): "

name = STDIN.gets.chomp

mod.name = name == "" ? folderName : name

out\_file = File.new(folderPath + "/" + name + ".modfst", "w")

puts "Module type: 0. System, 1. View"

input = STDIN.gets.chomp

typenum = Integer(input) rescue -1

while typenum != 0 && typenum != 1 do

input = STDIN.gets.chomp

b = Integer(input) rescue -1

end

type = typenum == 0 ? "sys" : "view"

mod.type = type

out\_file.puts(mod.to\_json)

out\_file.close

end

end

def validate(dir)

modulePath = findModule(dir)

if modulePath != nil then

mod = Module.new()

file = File.read(modulePath)

data\_hash = JSON.parse(file)

mod.fill(data\_hash)

if mod.name != "" then

#Запрос к серверу. Сообщение валидации

end

end

end

arg = ""

if ARGV.count >= 1 then

arg = ARGV[0]

end

dir = Dir.pwd

system('pod ipc spec ~/projects/sample1/sample2.podspec > ./pod')

puts res

file = File.read('pod')

json = JSON.parse(file) rescue nil

puts jsonToPodpec(json)

# case arg

# when "install" then

# install(dir)

# when "update" then

# update(dir)

# when "create" then

# create(dir)

# when "validate" then

# validate(dir)

# else

# puts "error"

# end

# ARGV.each do|a|

# puts "Argument: #{a}"

# end

Приложение 6.

// Исходный код программы Asmblr.rb

#!/usr/bin/env ruby

require 'net/http'

require 'uri'

require 'json'

require\_relative 'ModuleClass'

if ARGV.count >= 1 then

arg = ARGV[0]

end

dir = arg

uri = URI.parse("174.138.58.175/app")

# Full control

http = Net::HTTP.new(uri.host, uri.port)

request = Net::HTTP::Post.new(uri.request\_uri)

request.body = File.open("#{dir}",'r')

appMan = JSON.parse(request.body) rescue nil

appName = appMan["name"]

response = http.request(request)

body = response.body

if body != nil then

data = JSON.parse(body) rescue nil

if data != nil then

vcName = ""

usedVars = Hash.new()

appDelDecl = "var assembler: Assembler!"

appDelCode = "window = UIWindow(frame: UIScreen.main.bounds)\n

self.assembler = Assembler.init(window: UIWindow)"

declString = "import UIKit\nimport Foundation\n\nclass Assembler {\n"

initString = "\tfunc init(){\n"

podString = "source 'https://github.com/CocoaPods/Specs.git'\n

platform :ios, '8.0'\n

use\_frameworks!

\ninhibit\_all\_warnings!

\ntarget '#{appMan["appName"]}' do\n"

data["modules"].each { |mod|

name = mod["name"]

man = Module.new()

man.fill(mod["manifest"])

commit = mod["commit"]

path = mod["path"]

protocol = mod["protocol"]

podString += "pod '#{name}' :path => '#{path}' :commit => '#{commit}'\n"

usedVars[protocol] = name

declString += "\tvar #{name}: #{protocol}!"

initString += "\t\t#{name} = #{man.rootClass}()\n"

man.strongDependencies.each do |key, value|

if usedVars[value] != nil then

initString += "#{name}.key = #{usedVars[value]}"

end

end

man.weakDependencies.each do |key, value|

if usedVars[value] != nil then

initString += "#{name}.key = #{usedVars[value]}"

end

end

initString +="\t\t#{name}.#{man.rootInitialize}()\n\n\n"

if name == appMan["rootModule"] then

vcName = man.view

end

}

initString += "window.rootController = #{appMan}.#{vcName}\n\t\twindow.makeKeyAndVisible()\n"

initString += "\}"

resultString = declString + initString + "}"

podString += "end"

end

end

Приложение 7.

// Исходный код сборки приложения WeatherApp

class Assembler{

var http: HttpProtocol!

var weatherOnline: WeatherOnlineProtocol!

var weather: WeatherProtocol!

init(window: UIWindow) {

self.http = Http()

self.initialize()

self.weatherOnline = WeatherOnline()

self.weatherOnline.http = self.http

self.weather = Weather()

self.weather.weatherGetter = self.weatherOnline

window.rootController = self.weather.vc

}

}