**АРХИТЕКТУРА ИНСТРУМЕНТАЛЬНОГО ОКРУЖЕНИЯ СБОРКИ ГОТОВЫХ ПРИЛОЖЕНИЙ ДЛЯ МОБИЛЬНЫХ ПЛАТФОРМ**

*Тюрнин Алексей Валерьевич*

Пармский государственный национальный исследовательский университет, 614990, Россия, г. Пермь, ул. Букирева, 15, [turlex251@gmail.com](mailto:turlex251@gmail.com)

Ключевые слова: мобильная разработка, архитектура, инструментальное окружение

В мобильной разработке все важнее стало правильных пропорциях совмещать скорость и качество разработки. Кроме того, довольно часто приложения для разных заказчиков похожи друг на друга по функциональности, отличаясь лишь некоторыми особенностями или дополнительными возможностями. Каким образом можно ускорить и упростить разработку таких приложений, при этом не потеряв в качестве? Данный вопрос приводит нас к опыту использования CASE-средств в различных конструкторов, т.к. именно данную задачу они призваны решать [1].

В мобильной разработке существует множество конструкторов приложений, которые позволяют практически без знания архитектуры мобильной платформы и знания какого либо языка программирования, создавать приложения на различные мобильные платформы [2]. Кроме того, да таких задач разработчики пишут свои подобия CASE-средств, которые остаются своеобразных «know-how» самих разработчиков.

Данная проблема привела к разработке инструментального окружения, позволяющего упростить и ускорить разработку приложений для мобильных платформ. Идея состоит в том, что бы свести разработку приложения к написанию модулей, необходимых для работы приложения, указать для каждого из них, какой функционал им необходим для работы. Кроме того дать возможность некоторым модулям отсутствовать в приложении, что не привело бы к ошибкам компиляции или ошибкам времени выполнения и при это могло влиять на логику работы модулей, которые от них зависят. Данных подход позволит писать код один раз и переиспользовать его в каждом новом. И по итогу свести сборку нового приложения к простому указанию его конфигурации и конфигурации его модулей. Данный подход очень схож с концепцией CASE-средств. Но это окружение будет рассчитано в первую очередь на разработчиков, так что для него будет обязательна возможность работы с исходным кодом модулей. И в этом плане данное инструментальное окружение ближе к менеджерам пакетов, которых достаточно в открытом доступе, например CococaPods[5] и Carthage[6] для iOS, Gradle[7] и Maven[8] для Andriod и т.д. В связи с тем, что многие функции, которые будет выполнять данной окружение, совпадают с возможностями менеджеров пакетов, то оно будет использовать уже существующие решения на рынке. Это, в свою очередь, привело к решению использовать Git хранить модули в Git­-репозиториях. Git  — распределённая система управления версиями [10]. Данное решение позволяет поддерживать версионность кода и эффективный менеджмент памяти. Кроме того, использования Git является естественным при разработке.

Одна из главных задач при разработке такого инструментального окружения – описать архитектуру системы, ее компоненты и взаимодействие между ними. Кроме того, так как для сборки будут использоваться уже готовые модули, необходимо определить их спецификации и спецификацию приложения в целом.

В первую очередь введем понятие модуля, как функционально законченного фрагмента программы. При помощи описания модуля через специальные файлы-манифесты абстрагироваться от реализации конкретного модуля и обработать его отсутствие в конечной сборке приложения. Манифест модуля содержит в себе информацию о уникальном имени модуля, его «сильных» и «слабых» зависимостях, а так же дополнительная мета информация, которая позволяет на сборки генерации готового приложения генерировать код сборки для конкретного модуля. Введем понятия «сильной» и «слабой» зависимости. «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. В свою очередь «слабая» зависимость – зависимость между двумя модулями, когда один из них может использовать фунционал второго модуля, но при всем этом присутствие второго модуля не является обязательными, первый модуль имеет возможность обрабатывать его отсутствие в приложении не вызывая ошибок компиляции или времени выполнения а также может корректировать логику своей работы в зависимости от этого .

Технологический процесс написания модуля выглядит следующим образом:

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

Хук – это скрипт, обрабатывающий события Git-репозитория, например «push». С помощью данного скрипта модуль валидируется на основании его манифеста и, если модуль валиден, то данные о нем добавляются в БД модулей. Тем самым на этапе разработки существует возможность поддерживать версионность модулей на каждом этапе разработки. И, в теории, не будет происходить ситуации, когда незаконченный модуль попадет в сборку приложения. Кроме того, использование интерфейсов для доступа к функциональности модуля позволяют эффективно использовать Mock-объекты для упрощения и ускорения разработки. Mock-объект представляет собой конкретную фиктивную реализацию [интерфейса](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), предназначенную исключительно для тестирования взаимодействия и относительно которого высказывается утверждение [4].

Далее введем понятие приложения, как совокупность связанных между собой модулей. 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.

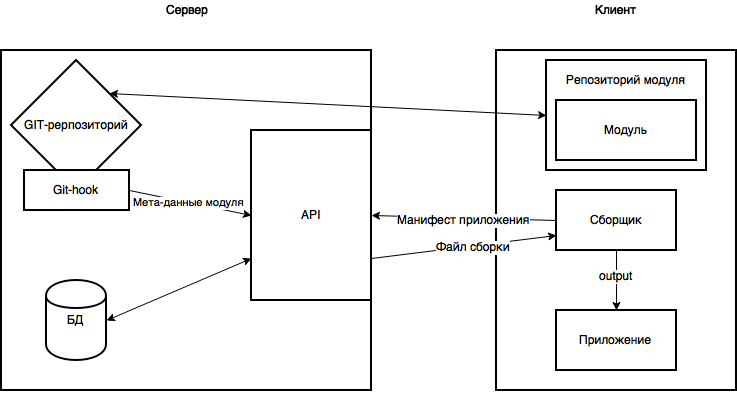
Технологический процесс сборки приложения будет выглядит следующим образом:

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

Граф зависимостей [9] — ориентированный граф, отображающий соотношение множества элементов некоторой совокупности в соответствии с выбранным транзитивным отношением над ней.

Технологический процесс, описанный выше, подразумевает использование определенный паттернов проектирования как самом приложения, так и модуля в частности. Кроме того, желательно наличие централизованного сервера, на котором будут храниться модули, мета-информация о них и, в том числе, должен быть реализован механизм разрешения зависимостей, который, при наличии большого количества модулей и связей между ними, может требовать больших вычислительных мощностей. И не мало важен доступ к этому функционалу на только на определенной машине, но и удалённо. При этом сама сборка не требует высокой вычислительной мощности и может выполняться на локальной машине. Кроме того, в следствии некоторых ограничений мобильных платформ [3], сборка iOS-приложения, написанного на Obj-C или Swift, не может производиться не на базе операционной системы macOS, но в то же время на хранение и обработку такого исходного кода ограничеий нет.

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



**Рис. 1 Схема архитектуры**

Такая архитектура системы позволяет ускорить процесс разработки за счет выделения функционала приложения в отдельные модули с унифицированным описанием и доступом, что позволяет при разработке абстрагироваться от применения кода в определенном приложении, и сконцентрироваться на самой стабильности и функциональности. В том числе процесс сборки становится практически полностью автоматизированным, что достигается сильным переиспользованием готового кода. Система имеет хороший потенциал и будет реализована для разработки приложений на платформе iOS и в последствии будет расширена и для разработки под другие платформы.

Список литературы:

1. CASE [электронный ресурс] URL: https://ru.wikipedia.org/wiki/CASE
2. Конструкторы приложений [электронный ресурс] URL: <http://apptractor.ru/develop/app-constructors>
3. Developing native iOS Apps on Windows [электронный ресурс] URL: [http://stackoverflow.com/questions/22358/how-can-i-develop-for-[электронный ресурс] URL: iphone-using-a-windows-development-machine](http://stackoverflow.com/questions/22358/how-can-i-develop-for-iphone-using-a-windows-development-machine)
4. MOCK-объект [электронный ресурс] URL: <https://ru.wikipedia.org/wiki/Mock-объект>
5. CocoaPods [электронный ресурс] URL: <https://cocoapods.org>
6. Carthage [электронный ресурс] URL: <https://github.com/Carthage/Carthage>
7. Gradle [электронный ресурс] URL: <https://gradle.org>
8. Maven [электронный ресурс] URL: <https://maven.apache.org>
9. Граф зависимостей [электронный ресурс] URL: <https://ru.wikipedia.org/wiki/Граф_зависимостей>
10. Git [электронный ресурс] URL: https://git-scm.com