УДК 004

УРОВНИ ОРГАНИЗАЦИИ АВТОМАТИЗИРОВАННОГО ТЕСТИРОВАНИЯ МОБИЛЬНЫХ ПРИЛОЖЕНИЙ ДЛЯ ОПЕРАЦИОННОЙ СИСТЕМЫ ANDROID

© 2020 А. В. Винокуров, О. Ю. Лавлинская

Воронежский институт высоких технологий (Воронеж, Россия)

В статье рассмотрены популярные технологии тестирования мобильных приложений для Android, сделан обзор сценариев их использования и классификаций автоматических тестов, предложен комплексный подход к структуре автоматических тестов Android-приложения.

Ключевые слова: Android-приложение, тестирование, непрерывная интеграция.

Тестирование программных продуктов – один из важнейших этапов жизненного цикла разработки программного обеспечения. Рост сложности программного кода, применение практик коллективной разработки влечет рост числа различных ошибок. Выявление ошибок, различных несоответствий – прерогатива тестирования. Тесты классифицируются на ручные, автоматизированные, автоматические. Технологии тестирования развиваются в направлении разработки автоматизированных и автоматических тестов. Автотесты не только помогают отлавливать ошибки в приложении до его поставки пользователям, но и обеспечивают поддержание стройной архитектуры и качественного кода приложения. Покрытие кода тестами служит одной из важнейших метрик оценки качества разработки приложения. Написание тестов лежит в основе таких популярных техник разработки программного обеспечения, как «разработка через поведение» (BDD – behavior-driven development) и «разработка через тестирование» (TDD – test-driven development) [1] .

Классификация тестов, применяемых при разработке для Android, задача нетривиальная, поскольку на сегодняшний день не существует однозначной терминологии, позволяющей описать всё существующее многообразие библиотек, фреймворков и технологий, применяемых для тестирования.

Традиционно тесты для Android принято разделять по способу выполнения на *локальные* и *инструментальные [2]*. Локальные тесты выполняются при сборке проекта, тогда как инструментальные требуют наличия эмулятора или подключённого устройства, на котором будет произведён запуск тестовой сборки приложения и выполнение тестов. Традиционно локальные тесты представляют собой обычные unit-тесты языков Java и Kotlin с использованием фреймворка JUnit и размещаются в папке src/tests. Тогда как инструментальные тесты в структуре проекта выделены в отдельную папку src/androidTests и используют программные интерфейсы тестирования из Android SDK и библиотеку Espresso, позволяющие работать с объектами пользовательского интерфейса приложения при запуске теста на эмуляторе или устройстве.

Несмотря на то, что такая классификация заложена в структуру проекта и описана в документации Android SDK, а её компоненты «из коробки» предоставляются в AndroidX Test Framework, её сложно назвать однозначной и исчерпывающей. Прежде всего, выполнение инструментальных тестов на эмуляторах и физических устройствах – это не характерная особенность, а характерный недостаток данной группы тестов, поскольку выполняется медленно и задействует избыточное количество ресурсов системы. В целях оптимизации тестирования и выполнения инструментальных тестов как локальных компанией PivotalLabs в 2010 году был разработан фреймворк Robolectric, позволяющий запускать инструментальные тесты на локальной виртуальной машине Java без установки приложения, а также без использования эмуляторов или внешних подключённых устройств. Начиная с версии 4.0, данный инструмент интегрирован с AndroidX Test Framework и используется

Лавлинская Оксана Юрьевна – Воронежский институт высоких технологий, канд. техн. наук, доцент ВИВТ, lavlin2010@yandex.ru.

Винокуров Артем Викторович – Воронежский институт высоких технологий студент магистратуры, vinartyom@yandex.ru.

повсеместно. Таким образом сглаживается, «фундаментальное» различие двух категорий тестов для Android-приложения.

Сейчас целесообразно говорить скорее о том, что инструментальные тесты – это тесты, которые могут быть выполнены на эмуляторе, устройстве или на группе устройств или эмуляторов, управляемых системой непрерывной интеграции (CI-системой).

Вторым недостатком такой классификации является чрезвычайно низкий уровень абстракции, который привязывается к файловой структуре проекта и выбору подключаемых зависимостей. В результате теряется фокус внимания на цели тестирования и на первый план выходят особенности реализации.

С учётом сказанного более значимой является группировка тестов по объекту тестирования и по целям тестирования:

- unit-тесты;
- ui-тесты;
- интеграционные тесты;
- функциональные тесты;
- тесты обработки внешних событий;
- тесты безопасности;
- тесты производительности;
- тесты локализации;

и др.

Самыми распространёнными и востребованными являются первые четыре группы тестов, их можно встретить практически в любом мобильном приложении.

С unit-тестами для Android ситуация более-менее понятна: они унаследованы от «обычной» java-разработки и представляют собой модульные тесты отдельных методов и классов с использованием библиотеки JUnit и сопутствующих инструментов, таких как Mockito. Как правило, unit-тесты не используют специфические для Android объекты, такие как Context, не взаимодействуют с UI или внешними ресурсами (например, базой данных, удалённым сервисом).

UI-тесты – категория более размытая. С одной стороны, сюда, на первый взгляд, попадают все тесты, взаимодействующие с объектами пользовательского интерфейса посредством библиотеки Espresso или аналогичных инструментов. С другой стороны, не все подобные тесты проверяют непосредственно UI приложения. Так некоторые из них могут проверять соблюдение бизнес-логики, взаимодействие компонентов приложения, соответствие элементов UI требованиям юзабилити или, например, соответствие времени отклика приложения на действие пользователя установленным требованиям технического задания. Всё это требует различных подходов и средств реализации.

В связи с этим хорошая практика – отделять в проекте «чистые» UI-тесты от интеграционных, функциональных и иных кейсов, не связанных с простой проверкой видимости, расположения и работоспособности элементов UI. Можно даже предложить критерий такого разграничения: если тест нельзя или нецелесообразно выполнять локально, как обычный unit-тест, с помощью Robolectric, то, вероятно, перед нами не UI-тест, а нечто совершенно иное. Например, интеграционный тест.

Под интеграционным тестированием понимают тестирование нескольких классов (модулей) приложения в их взаимосвязи друг с другом. Согласно приводимому М. Фаулером и ставшему классическим определению: «Интеграционные тесты проверяют, функционируют ли корректно независимо разработанные части приложения, когда они подключены друг к другу». Вместе с тем тот же автор отмечает, что на практике понятие интеграционного тестирования весьма туманно из-за существующих неоднозначных стандартов в индустрии.

Тем не менее, можно выделить определённые признаки, наличие любого из которых показывает, что мы имеем дело с интеграционным тестом:

- тестирование производится методом «белого ящика» и использует абстракции интерфейсов межкомпонентного или пользовательского взаимодействия;
- в тестировании участвует не менее двух компонентов, взаимодействие которых проверяется;
- тесты запускаются по расписанию под управлением среды непрерывной интеграции;
- требуют выполнения затратных операций без использования «заглушек» (mock) или запуска сборки приложения на устройстве или эмуляторе.

Большинство интеграционных тестов являются инструментальными, но не все. Единственная разновидность интеграционных тестов, выполняемая средствами локального тестирования, в Android унаследована от «классической» Java-разработки и представляет собой тесты, проверяющие доступность и корректность

работы сервисов и прочих «внешних» ресурсов, а также тесты соблюдения контрактов между компонентами приложения. Такие тесты Х. Воке, консультант и ведущий разработчик компании «ThoughtWorks», описывает как примеры типичных интеграционных тестов. Их реализация для Android принципиально не отличается от аналогичных тестов в любом веб- или ином приложении ничем, кроме более ограниченного набора объектов тестирования.

Вместе с тем, ошибочным является мнение некоторых авторов (например, Джарад Н.) [3], что для написания «интеграционного теста» Android-приложения достаточно просто в unit-тесте заменить mock-объекты View и Presenter реальными экземплярами классов с примерами реальных данных. Такие тесты, во-первых, избыточны для корректно спроектированного приложения (использование или неиспользование mock-объекта не должно оказывать влияние на поведение теста), во-вторых, они продолжают тестировать отдельные методы, а не проверяют корректность работы компонентов при. их взаимодействии друг с другом. По сути подобные тесты останутся модульными тестами, просто с искажённой абстракцией и нарушенной методологией тестирования.

Функциональное тестирование, как следует из его названия, проверяет функциональность приложения, то есть способность приложения решать задачи нужные пользователям в пределах, определённых техническим заданием. Функциональные тесты различны по своей реализации и используемым технологиям, однако, большинство из них выполняется средствами инструментального тестирования и требует создания условий, подобных сценариям обычного использования приложения пользователем.

Стоит подробнее остановиться на выполнении тестов в системе непрерывной интеграции. Наиболее популярные CI-системы, такие как Jenkins и TeamCity позволяют настраивать запуск интеграционных и функциональных тестов мобильного android-приложения. Обычно для этого требуется создание соответствующей конфигурации, а также установка и небольшая дополнительная настройка Robolectric, как связующего звена между CI и тестовым инструментарием Android.

Используя подобные тесты в своём проекте, стоит позаботиться об исследовании покрытия кода тестами. Наиболее известным инструментом для этого является библиотека JaCoCo от Eclipse Foundation, позволяющая не только рассчитать граф покрытия кода тестами, но и вывести результаты в форме удобных для чтения отчётов.

Стоит отметить, что примерно в одно время с релизом JaCoCo версии 0.1.0 в отечественной литературе предпринимались попытки предложить альтернативные пути оценки покрытия кода тестами путём применения алгоритма траверса графов (Е. Е. Хатько) [4], однако, по имеющейся информации, практической реализации данное исследование не получило, что отчасти может объясняться масштабом поставленной автором цели — не просто исследовать покрытие кода тестами, но разработать алгоритм для автоматической генерации тестовых сценариев. Представляется, что подобная планка и тогда, и сегодня не более достижима, чем эффективность конструкторов мобильных приложений для создания качественного мобильного приложения, созданного без участия профессиональных разработчиков.

Нельзя не упомянуть в связи с рассматриваемой проблематикой и о такой специфической особенности процесса разработки мобильных приложений, как наличие вендорских платформ-агрегаторов (Google Play Market, AppStore, Windows Store) для размещения приложений. Корпорации-вендоры мобильных устройств, фактически, являются монопольными операторами указанных площадок, через которые распространяются приложения для их устройств, и принимают на себя часть ответственности за качество распространяемого контента. Это означает, что размещаемые приложения проходят проверку, которая часто включает автоматическое тестирование определённого набора кейсов.

Так в случае Google Play Market целями такого тестирования являются: проверка соответствия заявленных разработчиком характеристик приложения загруженному файлу apk или App Bundle, выявление некоторых дефектов производительности, которые разработчику следует оптимизировать, проверка соответствия интерфейса приложения критическим, с точки зрения вендора, требованиям

гайдлайнов Material Design, изучение пригодности приложения для запуска на разных моделях и типах устройств, соответствие размера загруженного приложения его доступной функциональности (наличие неиспользуемых ресурсов и недостижимых классов), а также проверка пригодности приложения для использования лицами с ограниченными возможностями. Успешное прохождение такого тестирования и устранение разработчиком выявленных дефектов является необходимым условием публикации приложения в открытом доступе.

Несколько более упрощённые механизмы вендорского интеграционного тестирования публикуемых приложений используют и альтернативные площадки для публикации приложений Android, например, Huawei AppGallery, Samsung Galaxy Apps, Amazon Appstore.

Такая политика вендоров, с одной стороны, усложняет процесс распространения приложений, с другой стороны, снижает потребность в использовании собственных систем непрерывной интеграции и собственном интеграционном и функциональном тестировании.

Несмотря на очевидную значимость и влияние на контроль качества приложений, вендорское тестирование часто не включают ни в одну из существующих классификаций автоматических тестов android-приложения. Не учитывается или учитывается не в полной мере при построении классификаций и роль фреймворков автоматического тестирования.

Так для небольшого приложения, как правило, достаточно покрытия кода unit- и ui-тестами, а также успешного прохождения приложением проверок в консоли разработчика на площадке-агрегаторе. Возможно, ещё понадобятся проверки доступности внешних ресурсов и корректной обработки внешних событий, если эта функциональность имеет значение для приложения.

Если же мы создаём крупный и сложный проект для большой аудитории из разных стран, то нам понадобится CI-система и запуск интеграционных и функциональных тестов с её помощью. Также нам могут дополнительно понадобиться автоматические тесты функциональности, отвечающие за безопасность, производительность нашего приложения на разных устройствах, тесты локализации и иные тесты, соответствующие специфике приложения. Код и все необходимые кейсы будут покрыты тестами, а интеграционное и функциональное тестирование будет вынесено в среду непрерывной интеграции.

Однако можно ли сказать, что на этом потребность приложения в автоматическом тестировании будет исчерпана, и мы нигде не просчитались? Нет. Потому что запуск наших тестов в системе непрерывной интеграции будет выполняться посредством Robolectric, то есть на уровне локальной JVM под управлением CI-системы. Наши интеграционные тесты, которые должны моделировать пользовательскую активность, на самом деле будут выполняться даже не в Android Runtime, и все вызовы будут выполняться на уровне кода, а не пользовательского интерфейса. Такие тесты не показательны.

Но есть и иной существенный недостаток в прямом выполнении интеграционных и функциональных тестов для Android в CI-системе посредством Robolectric [5]. А именно, то, что для репрезентативности результатов таких тестов необходимо, чтобы они выполнялись на разных устройствах и конфигурациях. Приложение, которое должно запускаться на планшете, должно быть протестировано на планшете. Здесь же этого, очевидно, не происходит. Инфраструктура же Android настолько фрагментирована и разобщена, что часто без тестирования приложения на разных типах экранов, разных типах и моделях устройств гарантировать надёжность приложения не представляется возможным.

Подводя итоги, для автоматизации интеграционных и функциональных тестов мобильного приложения нужна возможность полноценно запускать тестовые сборки приложения на некотором количестве виртуальных машин, симулирующих работу эмулятора Android, либо на таком же количестве физических устройств. По сути, это должен быть кластер тестирования Selenium Grid Hub с эмуляторами или физическими устройствами в качестве ячеек.

На сегодняшний день существует целый ряд программных решений, позволяющих это сделать: Robotium, Appium и Calabash. Каждый из этих фреймворков автоматизации мобильного тестирования обладает своей спецификой, которую следует учитывать при их использовании в своём проекте.

Так Robotium ориентирован на написание тестов «серого ящика» для

Android-приложений.

Calabash – кроссплатформенный фреймворк, написанный на языке программирования Ruby, и ориентированный на идеологию Behavior-Driven Development. Поддерживаются различные конфигурации и модели как Android, так и iOs устройств.

Appium – кроссплатформенный фреймворк, предоставляющий широкие возможности тестирования как Android, так и iOs, и даже гибридных веб-приложений. Appium является наиболее востребованным инструментом в своей отрасли, поскольку, во-первых, поддерживает язык Java, а, во-вторых, относительно легко интегрируется с Selenium Grid Hub, развёрнутым в CI-системе, или с облачным кластером вроде BrowserStack или SauceLabs. Фактически, Appium – это распространение идеологии и интерфейсов Selenium WebDriver на область мобильной разработки.

Для более удобного написания и запуска Appium-тестов можно добавить в проект библиотеку Selenide, которая значительно сокращает и делает более читаемым код UI тестов на Java.

Подводя итог, отметим, что для инфраструктуры автоматического тестирования Android-приложений более значимой является не линейная классификация тестов по какому-либо одному признаку, а выделение следующих *уровней тестирования Android-приложения*, исходя из комплексного понимания объектов тестирования [6], целей тестирования и используемых технологий:

1) Обычные JUnit тесты, которые будут выполняться при каждой сборке приложения, и которые проверяют корректность кода отдельных классов и методов.

2) Инструментальные UI-тесты, а также интеграционные тесты внешних ресурсов и контрактов, которые не требуют имитации действий пользователя с приложением, которые не должны выполняться при каждой сборке (доступность базы данных не зависит от сборки приложения), но которые целесообразно запускать по расписанию в системе непрерывной интеграции.

3) Интеграционные и функциональные тесты приложения с помощью Appium или его аналогов, выполняемые в кластере Selenium Grid Hub, запущенном в системе непрерывной интеграции, либо в облачном кластере, и имитирующие активность пользователей в приложении.

4) Тестирование приложения оператором площадки-агрегатора при его опубликовании (набор выполняемых тестов индивидуален для каждой площадки).

5) Автоматические нагрузочные тесты и иные специфические методы тестирования, которые должны выполняться отдельно и зависят от конкретных задач проекта.

ЛИТЕРАТУРА

1. Build effective unit tests. URL: https://developer.android.com/training/testing/unit-testing?hl=ru (дата обращения: 03.11.2020)
2. Fowler M. IntegrationTest. 2018. URL: https://martinfowler.com/bliki/IntegrationTest.html (дата обращения: 24.10.2020)
3. Jarad N. How to do TDD in Android? Part 3 – Mocking & Integration testing. 2016. URL: https://medium.com/mobility/how-to-do-tdd-in-android-part-3-mocking-integration-testing-60b057840db6 (дата обращения 23.10.2020).
4. Хатько Е. Е. Об одном методе тестирования «мобильных» приложений / Е. Е. Хатько // Труды МФТИ. – 2012. – Т. 4. – № 3. – С. 132-140.
5. Vocke H. The Practical Test Pyramid. 2018. URL: https://martinfowler.com/articles/practical-test-pyramid.html (дата обращения 23.10.2020).
6. Курченкова Т. В. Пример реализации мобильного приложения в рамках концепции "CITIZEN HEALTHCARE" / Т. В. Курченкова, О. Ю Лавлинская // Информационные системы и технологии. – 2018 – № 1 (105). – С. 44-50.

LEVELS OF AUTOMATED TESTING STRUCTURE FOR ANDROID APPLICATION

© 2020 A. V. Vinokurov, O. Yu. Lavlinskaya

Voronezh Institute of High Technologies (Voronezh, Russia)

The article observes popular technologies of automated testing android apps, their use cases and automated tests categorizations, suggests the comprehensive way to categorize Android automated tests.

Keywords: Android app, testing, continuous integration.