МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Тольяттинский государственный университет»

Институт математики, физики и информационных технологий
(наименование института полностью)

Кафедра Прикладная математика и информатика
(наименование)

09.04.03 Прикладная информатика
(код и наименование направления подготовки)

Информационные системы и технологии корпоративного управления
(направленность (профиль))

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ)

на тему «Автоматизация тестирования при проектировании мобильных приложений по технологии Agile»

Студент О.А. Нестерова
(И.О. Фамилия) (личная подпись)

Научный руководитель д.т.н, доцент, С.В. Мкртычев
(ученая степень, звание, И.О. Фамилия)

Тольятти 2020

Оглавление

Введение

В настоящее время технология Agile набирает все большую популярность в сфере разработки программного обеспечения, в том числе – для мобильных устройств.

Для автоматизации приложений, проектируемых с применением гибких методологий, чаще всего используется методика Scripting. Она подходит для автоматизации тестирования API, но является неоптимальным решением для подготовки автотестов GUI. В результате тестирование графического интерфейса проводится вручную.

В течение коротких итераций на Agile-проекте возможно провести только базовые проверки GUI. Такой подход допустим для веб- и десктоп-приложений, которые рассчитаны на работу с ограниченным количеством браузеров и платформ.

В отличие от них, мобильные приложения разрабатываются под разные мобильные платформы, версии операционных систем и конфигурации устройств. Из-за того, что тестирование ограничивается базовыми проверками, многие дефекты GUI попадают в продакшен-версию приложения и обнаруживаются конечными пользователями. Для мобильных приложений такая ситуация может привести к получению негативных отзывов от пользователей и, как следствие, к коммерческому провалу.

Таким образом, **актуальность** исследования обусловлена необходимостью разработки методики, которая позволит автоматизировать тестирование API и графического интерфейса мобильного приложения.

Объектом исследования является процесс тестирования мобильных приложений, проектируемых по технологии Agile.

Предмет исследования – методика автоматизации тестирования мобильных приложений, проектируемых по технологии Agile.

Целью работы является исследование и разработка методики автоматизации тестирования мобильных приложений, проектируемых по технологии Agile.

Гипотеза исследования: применение предлагаемой методики позволит повысить эффективность процесса тестирования мобильных приложений, проектируемых по технологии Agile.

Для достижения цели и проверки поставленной гипотезы нужно решить следующие **задачи**:

1. Проанализировать существующие методы и виды тестирования программного обеспечения и оценить возможность их применения для мобильных приложений.

2. Проанализировать существующие методики и инструменты автоматизации тестирования и оценить возможность их применения для мобильных приложений.

3. Разработать методику автоматизации тестирования МП по технологии Agile.

4. Выполнить апробацию и обосновать применение разработанной методики для повышения эффективности тестирования мобильных приложений.

Новизна исследования заключается в разработке методики автоматизации, которая основана на комбинации двух существующих методик для подготовки разных типов тестов.

Методы исследования: системный анализ, методы тестирования программного обеспечения, эксперимент.

Публикации по теме исследования:

1. Нестерова О.А. Методика автоматизации тестирования мобильных приложений на Scrum-проектах // Студенческий форум: электрон. научн. журн. 2020. № 16(109);

2. Нестерова, О.А. Автоматизация тестов как часть Agile-подхода в тестировании мобильных приложений // Студенческий форум: электрон. научн. журн. – 2019. – № 39 (90).

Практическая значимость диссертационного исследования заключается в возможности применения разработанной методики автоматизации для тестирования МП.

Основные этапы исследования: исследование проводилось с 2018 по 2020 гг. в несколько этапов:

На первом этапе (констатирующем этапе) – формулировалась тема исследования, выполнялся сбор информации по теме исследования из различных источников, проводилась формулировка гипотезы, определялись постановка цели, задач, предмета исследования, объекта исследования и выполнялось определение проблематики данного исследования.

Второй этап (поисковый этап) – в ходе проведения данного этапа осуществлялся анализ методик тестирования, была разработана модель методики тестирования, проводилось написание и публикация научных статей по теме исследования в сборниках научных статей.

Третий этап (оценка эффективности) – на данном этапе осуществлялась оценка эффективности и проверка адекватности предлагаемой методики тестирования, были сформулированы выводы о полученных результатах по проведенному исследованию.

На защиту выносятся:

1. Методика автоматизации тестирования мобильных приложений.

2. Результаты апробации разработанной методики.

Диссертационная работа состоит из введения, трех глав, заключения, списка используемой литературы и приложения.

В первой главе описана специфика мобильных приложений, рассмотрены методы и виды тестирования. Также описана специфика тестирования мобильных приложений на Agile-проектах.

Во второй главе описаны область применения автоматизированного тестирования, методики и инструменты автоматизации тестирования мобильных приложений. Рассмотрена методика Scripting, применяемая для подготовки автотестов на Agile-проектах.

Третья глава посвящена разработке методики автоматизации тестирования мобильных приложений, проектируемых по технологии Agile. В этой же главе представлены результаты апробации разработанной методики.

В заключении содержатся итоги выполненной работы. В приложении представлен тестовый сценарий, созданный при подготовке автотеста API.

Работа изложена на 74 с. с приложением, содержит 22 рисунка и 4 таблицы.

Глава 1 Методологические основы тестирования мобильных приложений

1.1 Специфика мобильных приложений

Мобильное приложение (МП) – программное обеспечение, предназначенное для работы на смартфонах, планшетах и других мобильных устройствах, разработанное для конкретной платформы [18].

Существует несколько типов МП [18]:

– нативные;

– браузерные (мобильные веб-приложения);

– гибридные.

Нативные МП создаются для работы с конкретной платформой. Их код пишется на языках программирования, которые являются «родными» для мобильных платформ. Для Android это Java, для iOS – Swift или Objective-C. Такие МП физически устанавливаются на мобильное устройство и распространяются через магазины мобильных приложений.

Браузерные МП – это оптимизированные версии веб-приложений, первоначально разработанных для ПК. МП этого типа являются мультиплатформенными – запускаются через браузер на мобильном устройстве с любой операционной системой (ОС) и не используют его программное обеспечение (ПО).

Гибридные МП сочетают в себе черты первых двух типов – они используют веб-технологии, но требуют установки на устройство и имеют доступ к его функциям. Гибридные МП используют встроенную оболочку приложения, которая содержит веб-представление (web-view) для запуска веб-приложения внутри приложения для конкретной платформы.

Независимо от типа, МП обладает рядом особенностей, которые отличают его от веб- и настольных приложений. Далее будут рассмотрены ключевые моменты [20, 25, 37]:

– взаимодействие с основными функциями устройства-коммуникатора. Смартфон – это, в первую очередь, телефон и никакие приложения не должны блокировать возможность принять/ совершить звонок, получить/ отправить sms;

– работа с приложением в постоянно меняющихся условиях (например: изменение уровня внешнего освещения, нестабильную связь с Интернетом, переключение между Wi-Fi и 3/4G, расход заряда батареи устройства, перевод приложения в фоновый режим);

– короткий цикл разработки. Это вызвано необходимостью часто выпускать обновления для обеспечения совместимости с более новыми моделями мобильных девайсов и операционных систем для них;

– большое разнообразие мобильных устройств, размеров и диагоналей экранов, версий ОС, на которых может быть использовано приложение, и графических оболочек (для Android);

– сильная конкуренция среди производителей мобильного ПО, из-за которой у пользователей формируются высокие ожидания.

Исходя из этого, пользователи оценивают качество мобильного приложение по таким критериям как

– корректное выполнение заявленных в названии и описании задач,

– интуитивная понятность и хорошая скорость отклика всех элементов управления,

– бесперебойная работа в любых условиях.

Неудобство использования, несовместимость с популярными моделями девайсов, функциональные ошибки, проблемы графического интерфейса и прочие недочеты приводят к негативным отзывам и резкому снижению рейтинга приложения [19].

Поэтому качество МП – необходимое условие его востребованности и конкурентоспособности. Важно выделить именно те тесты, которые наиболее критичны для конкретного приложения с целью сокращения затрат компании и снижения риска появления ошибок.

Сложность МП не позволит провести все возможные тесты, поэтому нужно использовать приоритеты зон тестирования, среди которых можно выделить наиболее критические [26, 27, 28]:

– пользовательский интерфейс – необходимо убедиться, что все элементы имеют удобный размер, в приложении нет пустых экранов, МП поддерживает стандартные жесты;

– аппаратные ресурсы – нужно тщательно проверять обработку проблемных ситуаций (например: установка приложения при нехватке памяти, недостаточный объем памяти для работы приложения в активном или фоновом режиме);

– проверки различных версий ОС и разрешений экрана (например: корректность отображения элементов МП на AMOLED- и retina дисплее, в ландшафтной и портретной ориентации, невозможность установки приложения на девайс с неподдерживаемой версией ОС);

– реакция на внешние прерывания – в первую очередь это входящие звонки и смс, переход девайса в спящий режим, push-уведомления других приложений, подключение дополнительных устройств отключение и включение Wi-Fi и мобильного интернета;

– обратная связь с пользователем – отклик элементов на действия пользователя должен быть понятным и своевременным, реакция кнопок на нажатие должна соответствовать их состоянию (активная, нажатая, заблокированная), при попытке удалить данные должны появляться предупреждающие алерты с возможностью отменить действия;

– платный контент – стоимость должна соответствовать предоставляемому функционалу, покупки не должны теряться при обновлении приложения и так далее;

– локализация – сюда относятся максимальное количество символов, которые можно ввести в заполняемые поля, корректность перевода, формат отображения дат и специфичных для конкретного языка символов;

– обновления – основные проверки – сохранение пользовательских данных, функционирование урезанных версий приложения, созданных для работы с более старыми версиями ОС;

– соответствие МП правилам и соглашениям конкретной ОС – необходимо проверять корректность названия и описания, формат установочного файла, поддержку требований различных магазинов приложений (для Android) [38];

– обработка случайных и непредсказуемых событий – мобильные девайсы часто оказываются в условиях, в которых получают хаотичную информацию (например, когда происходит разблокировка устройства, лежащего в кармане), потому МП должно адекватно обрабатывать случайный ввод [28];

– имитация реальных условий использования – необходимо проверять работу мобильного приложения при нестабильной связи с интернетом;

Перечень критических зон может сокращаться или расширяться в зависимости от специфики конкретного МП. Например, для приложения, которое разрабатывается для использования в конкретной стране, не требуется проводить тестирование локализации.

1.2 Методы и виды тестирования мобильных приложений

Базовые принципы тестирования сформулированы в классических книгах по тестированию [11, 13, 16]. Авторы выделяют два подхода к тестированию программных продуктов – метод черного ящика и стеклянного (белого) ящика.

Тестирование методом стеклянного ящика направлено на проверку внутренних аспектов работы приложения. При этом целью является

обнаружение не синтаксических ошибок (для их поиска обычно используется компилятор), а более сложных для локализации логических дефектов.

Подобные проверки позволяют абстрагироваться от внешнего проявления ошибок и обнаружить их первопричину, упрощает поиск и диагностику скрытых проблем. Это необходимый этап тестирования, но его недостаточно для оценки качества МП.

При тестировании методом белого ящика приложение исследуется в синтетических условиях, без учета влияния реальной среды исполнения и в отрыве от пользовательских сценариев. Для пользователей МП большое значение имеют простота и удобство графического интерфейса, дизайн экранов и прочие внешние аспекты, которые невозможно проверить на уровне кода.

Кроме того, при тестировании методом белого ящика фокус делается на реализованной функциональности, в результате повышается вероятность пропустить нереализованные требования.

При тестировании черным ящиком программа рассматривается как объект с неизвестной внутренней структурой. Таким тестированием занимаются QA-специалисты. При этом они фокусируются не на коде, а на том, как приложение обрабатывает различные входные данные.

Черноящичный подход применим только при наличии открытых интерфейсов МП – пользовательского и (или) программного (API). Поведение приложения сравнивается с тем, что описано в требованиях к нему.

Метод черного ящика направлен на поиск ошибок, у которых есть внешние проявления. Это проблемы, связанные с функционалом ПО, производимыми им вычислениями и допустимым диапазоном данных, которые могут быть обработаны приложением. Работа внутренних компонентов системы проверяется неявно, посредством анализа внешних проявлений дефектов

Основной плюс и в то же время минус такого подхода – взаимодействие с программой с позиции конечного пользователя. С одной стороны, это позволяет сосредоточиться на проверке функционала приложения и выявить наиболее заметные и критичные проблемы в его работе. С другой – применение черноящичного тестирования в чистом виде ведет к одностороннему видению программного продукта.

Методы черного и белого ящика не являются взаимоисключающими – они гармонично дополняют друг друга и таким образом компенсируют имеющиеся недостатки.

Поэтому в более современных книгах по тестированию [15, 23, 30] предлагается использовать комбинированный метод – тестирование серым ящиком. Это подход, который сочетает элементы первых двух методов. С одной стороны, тестировщик использует паттерны поведения конечного пользователя, с другой – частично знает, как устроен бэк-энд тестируемой программы и активно применяет это знание.

Метод серого ящика широко применяется для тестирования МП. Он подразумевает одинаковое внимание к внешней (графический пользовательский интерфейс), так и к внутренней (взаимодействие с сервером) частям приложения.

Тестирование методом серого ящика позволяет локализовать дефекты одновременно и по принципу связанности с определенными строками кода, и по последовательности пользовательских действий в результате которых воспроизводится проблема. Эти сведения помогают более быстро и точно оценивать дефекты по серьезности и приоритету, определять масштаб их проявления и прогнозировать последствия их исправления.

В рамках этого метода для проверки работы МП обычно применяются различные виды тестирования. В зависимости от цели их можно разделить на три группы [7]:

– функциональные,

– нефункциональные,

– связанные с изменениями.

Функциональное тестирование подразумевает проверку того, что те или иные функции реализованы в данной версии приложения и работают в соответствии с требованиями.

Нефункциональное тестирование направлено на проверку нефункциональных особенностей МП. Ниже рассмотрены основные виды нефункционального тестирования.

Инсталляционное (установочное) тестирование направлено на выявление проблем, которые влияют на установку МП. Включает проверку таких ситуаций как установка в новой среде (новая модель девайса или версия ОС), обновление текущей версии, изменение установленной версии на более старую, повторная установка после неудачной попытки, удаление приложения.

Тестирование удобства использования (usability) призвано определить, насколько функционал программного продукта понятен для пользователя и нравится ему. МП можно считать удобным для использования, если пользователь делает то, что ему кажется наиболее правильным и при этом у него не возникает никаких вопросов и сомнений в своих действиях [5, 15]. При работе с МП не должно возникать «тупиковых ситуаций», когда пользователь не полностью понимает, что происходит в приложении и (или) не может контролировать происходящее.

Тестирование интерфейса подразумевает проверку того, насколько корректно работает пользовательский интерфейс и его компоненты [32].

Тестирование безопасности проверяет, способно ли МП противостоять действиям злоумышленникам – попыткам доступа к конфиденциальной информации или внедрению вредоносного кода.

Тестирование совместимости позволяет определить способность программы работать в конкретном окружении (модель и конфигурация девайса, версия ОС, подключаемое оборудование – внешняя клавиатура, гарнитура). Сюда же можно отнести проверку корректного

функционирования приложения в различных ориентациях конкретного устройства, проверку модулей приложения, их дизайна на предмет соответствия конкретной ОС [22].

Тестирование производительности сводится к исследованию того, с какой скоростью приложение реагирует на нагрузку различного характера и интенсивности – обычную, растущую и стабильно высокую (стрессовую).

Тестирование локализации проводится для проверки качества и корректности адаптации МП к работе на определенном языке, а также с учетом культурных особенностей (например, формат даты, специфичные для языка разделители и символы, единицы изменения веса, расстояния).

По степени автоматизации тестирование делится на ручное и автоматизированное. В первом случае все проверки проводятся вручную, во втором – тест-кейсы частично или полностью выполняет специализированное инструментальное средство. При этом тестировщик не исключается из процесса полностью – он занимается разработкой тестовых сценариев, подготовкой данных, анализом и оценкой результатов выполнения, подготовкой отчетов о найденных ошибках.

Также тестирование мобильных приложений может классифицироваться по субъекту исполнения. Когда продукт уже собран воедино, но еще слишком «сырой» для демонстрации внешним пользователям., внутри компании-разработчика проводится альфа-тестирование.

К выполнению привлекаются как профессиональные тестировщики, так и сотрудники других отделов организации. Альфа-тестирование применяется для проверки жизнеспособности идеи проекта и отслеживания наиболее критических ошибок в коде МП [7].

Бета-тестирование подразумевает активное привлечение заказчика и (или) конечных пользователей. Продукт передается бета-тестировщикам, когда он уже достаточно стабилен, но может содержать дефекты, выявить

которые возможно только при использовании МП в реальных условиях. Бета-тестирование может быть организовано в закрытом или открытом формате.

При закрытом бета-тесте доступ к приложению получает ограниченное число участников, в открытом может принять участие любой желающий. В первом случае проще контролировать круг лиц, которым доступно приложение. Во втором – появляется возможность охватить более широкую аудиторию, получить больший объем обратной связи и обеспечить приближенную к реальной нагрузку на серверную часть МП.

Тестирование мобильного приложения проводится после каждого внесения изменений – исправления дефектов, добавления или исключения функциональности.

Дымовое тестирование представляет собой небольшой набор проверок, который позволяет убедиться, что после сборки нового или исправленного кода приложение возможно установить, запустить и использовать по назначению.

Регрессионное тестирование проводится после того, как в приложении или среде исполнения были сделаны изменения, для подтверждения того, что реализованная ранее функциональность работает корректно.

Тестирование сборки позволяет убедиться, что выпущенная версия соответствует критериям качества, необходимым для начала тестирования.

Санитарное тестирование является подвидом регрессионного тестирования и направлено на проверку того, что конкретная функция работает согласно требованиям, заявленным в спецификации.

В зависимости от глубины проводимых проверок и функциональное, и нефункциональное тестирование можно разделить на два подвида – тестирование критического пути и расширенное.

Тестирование критического пути нацелено на исследование той части функциональности, которая «используется типичными пользователями в повседневной деятельности» [15].

В случае с тестированием МП сюда можно отнести установку и запуск приложения, авторизацию, переходы между основными экранами и события, связанные с выполнением базовой функции конкретного МП – сохранение и обработка определенного вида данных, отправка определенных запросов и тому подобное.

На рисунке 1 показана суть тестирования критического пути [15].

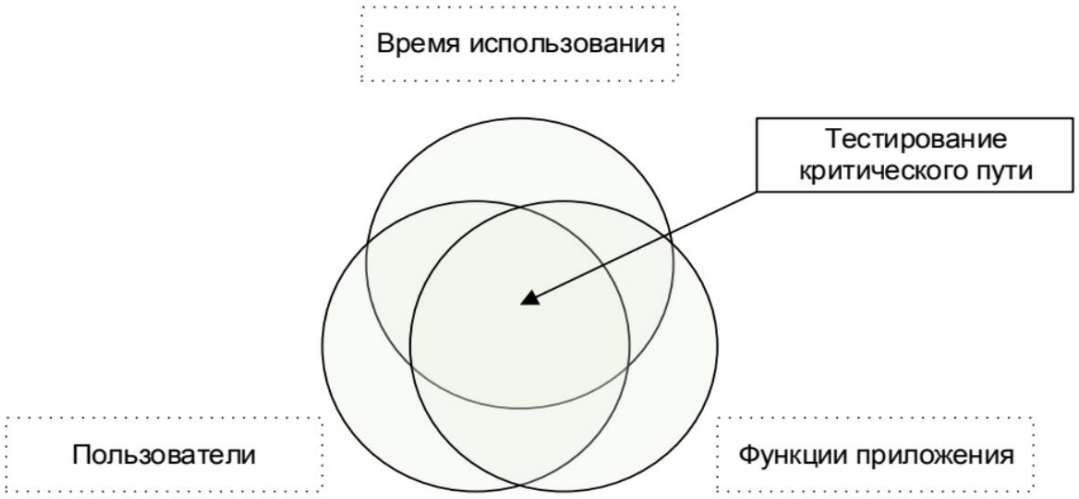

Рисунок 1 – Суть тестирования критического пути

Расширенное тестирование, напротив, направлено на проверку всей описанной в требованиях функциональности. При выполнении тестов учитывается приоритет функциональности – сначала проверяется более важная, затем менее важная. При наличии достаточного количества временных и человеческих ресурсов тестируются даже самые низкоприоритетные случаи.

Также в рамках расширенного тестирования могут проверяться нетипичные и маловероятные тест-кейсы и сценарии использования свойств и функций приложения, более поверхностно затронутых при тестировании критического пути.

В зависимости от характера используемых кейсов проверки всех описанных выше видов могут относиться как к позитивному, так и к негативному тестированию.

Первое подразумевает исследование работы МП в соответствии с инструкцией, то есть, с выполнением корректных действий и вводом валидных данных. Второе, наоборот, направлено на проверку того, как приложение обрабатывает ошибочные действия пользователя. Например, отправку невалидных данных или пропуск обязательных шагов.

Исходя из сказанного выше, метод серого ящика соответствует задачам мобильного тестирования – его применение позволяет уделить одинаковое внимание фронт-эндной и бэк-эндной частям программного продукта, его интерфейсу и производительности.

Но в то же время этот метод подразумевает проведение большого объема проверок и, как следствие, требует существенных временных затрат.

1.3 Специфика тестирования МП на Agile-проектах

Большинство мобильных приложений создаются в постоянно меняющихся условиях, когда нужно регулярно дорабатывать функционал, обновлять дизайн, переписывать работающий код для совместимости с новыми версиями мобильных ОС, а также соответствия растущим требованиям пользователей.

Поэтому в разработке мобильных приложений все большую популярность набирает технология Agile, которая подразумевает быструю реализацию работающего функционала, его частое обновление и тесное сотрудничество с заказчиком.

Agile или гибкая методология разработки (agile software development) – группа методологий разработки программного обеспечения, основанных на итеративной поэтапной разработке, где требования и решения развиваются

посредством сотрудничества между самоорганизующимися межфункциональными командами [17].

Ключевые принципы методологии описаны в Agile Manifesto [29] – программном документе сообщества «Agile Alliance», разработанном в феврале 2001 года.

В основе Agile лежат 12 принципов:

1. «Наивысший приоритет – удовлетворение потребностей заказчика, которое достигается благодаря регулярной и ранней поставке ценного программного обеспечения.

2. Изменение требований приветствуется, даже на поздних стадиях разработки.

3. Работающий продукт следует выпускать как можно чаще, оптимальная периодичность – от пары недель до пары месяцев.

4. На протяжении всего проекта разработчики и представители бизнеса должны ежедневно работать вместе.

5. Над проектом должны работать мотивированные профессионалы. Чтобы работа была сделана, создайте условия, обеспечьте поддержку и полностью доверьтесь им.

6. Непосредственное общение является наиболее практичным и эффективным способом обмена информацией как с самой командой, так и внутри команды.

7. Работающий продукт – основной показатель прогресса.

8. Инвесторы, разработчики и пользователи должны иметь возможность поддерживать постоянный ритм бесконечно.

9. Постоянное внимание к техническому совершенству и качеству проектирования повышает гибкость проекта.

10. Простота как искусство минимизации лишней работы.

11. Самые лучшие требования, архитектурные и технические решения рождаются у самоорганизующихся команд.

12. Команда должна систематически анализировать возможные способы улучшения эффективности и соответственно корректировать стиль своей работы» [29].

На рисунке 2 [14] показана разница в организации процесса тестирования ПО с применением каскадной методологии и Agile.

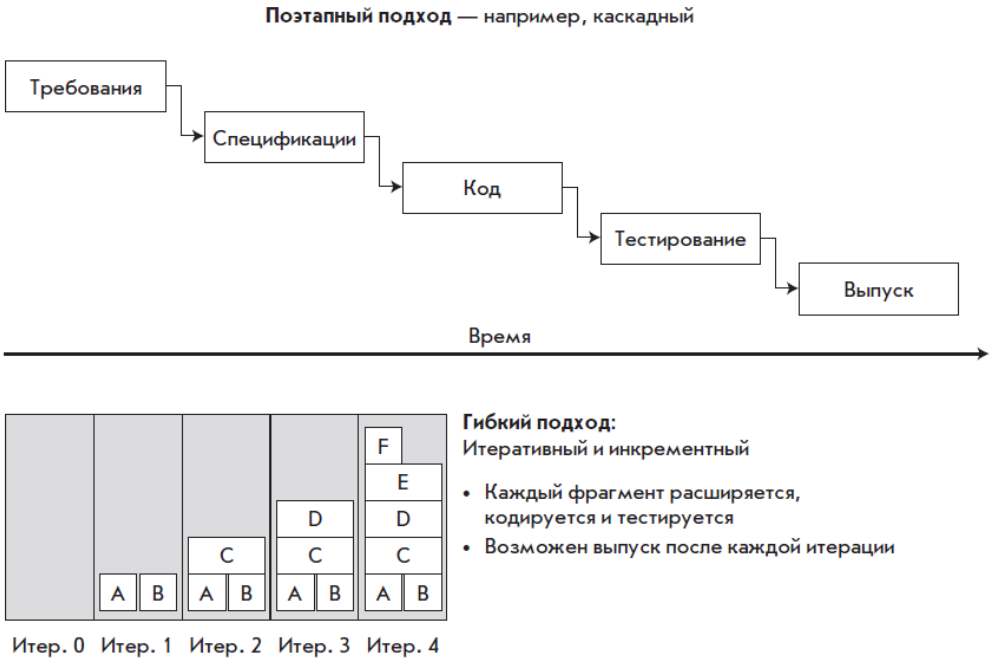

Рисунок 2 – Разница в организации процесса тестирования при применении традиционной и гибкой методологий

При применении каскадной методологии приложение выпускается сразу с полным набором функциональностей. Если разработка ведется с использованием гибкой методологии – функционал приложения разрабатывается и тестируется поэтапно, в каждой итерации к существующим функциональностям добавляется новый фрагмент.

Agile-подход к тестированию подразумевает следующие изменения в работе QA-команды [14]:

– тестирование перестает быть изолированной фазой в создании ПО и активно применяется на всех стадиях жизненного цикла (ЖЦ) продукта – начиная с планирования. Таким образом, QA-специалисты могут

своевременно выявлять наиболее простые в исправлении ошибки (неточности, неоднозначные детали и прочие проблемы документации), составить представление о программном продукте задолго до написания кода и выявить его потенциально слабые места;

– объем тестовой документации сокращается до минимума: на смену подробным тест-кейсам приходят более высокоуровневые и универсальные тест-планы и чек-листы. Формат чек-листа позволяет не расписывать проверки досконально, за счет этого документацию проще поддерживать в актуальном состоянии, а в работе тестировщика увеличивается доля исследовательского тестирования;

– на всех стадиях разработки поддерживается обратная связь между специалистами по тестированию и остальными членами команды (разработчики, бизнес-аналитики, дизайнеры, проектный менеджер). Благодаря этому у каждого из участников проекта формируется более полное и всестороннее видение продукта;

– быстрая отдача от тестирования – найденные баги подлежат оперативному исправлению, что позволяет поддерживать «чистоту кода» и избежать накопления устаревшего и сложно поддерживаемого кода;

– тестирование – неотъемлемая часть критерия готовности: степень готовности ПО определяется с учетом количества, приоритета и серьезности обнаруженных проблем. Например, критерием готовности МП к выпуску может служить отсутствие в продукте дефектов с приоритетом выше «незначительного» (Minor) или «среднего» (Normal).

Применение тестирования на всех стадиях ЖЦ приложения создает условия для реализации методологии BDD (сокр. от англ. Behavior-driven development, дословно «разработка через поведение»), которая основана на совмещении в процессе разработки чисто технических интересов и интересов бизнеса [31].

Согласно этой методологии, для общения членов проектной команды между собой используется предметно-ориентированный язык. Основу

последнего представляют конструкции из естественного языка, понятные неспециалисту и описывающие поведение программного продукта и ожидаемые результаты.

Под ожидаемым результатом понимается поведение ПО, представляющее ценность для бизнеса. Для его описания используется спецификация поведения (англ. behavioral specification), которая имеет следующую структуру:

1) заголовок – описание бизнес-цели в сослагательном наклонении;

2) краткое описание пользовательской истории с указанием роли пользователя;

3) один или несколько сценариев, каждый из которых раскрывает ситуацию пользовательского поведения.

Такой подход позволяет формировать единое понимание продукта всеми участниками процесса разработки ПО. За счет этого BDD позволяет сформулировать требования к продукту, которые сделают его технически реализуемым, полезным с точки зрения бизнеса и удобным для конечного пользователя.

Взаимодействие по принципам BDD показано на рисунке 3 [40].

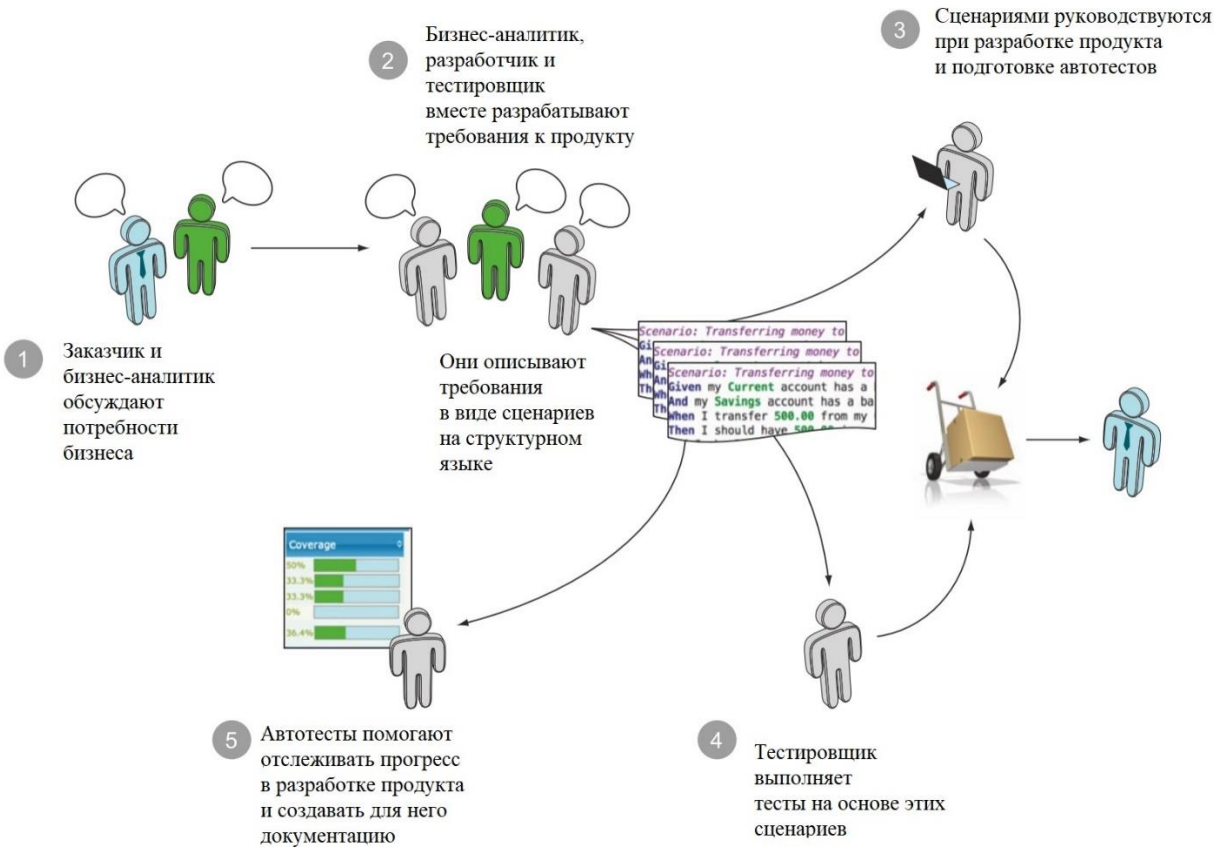

Рисунок 3 – Применение методологии BDD при разработке ПО

Методология BDD противопоставляется традиционному подходу к организации взаимодействия внутри проектной команды, при котором источником ошибок в работе продукта часто становятся различные трактовки требований бизнес-аналитиками, разработчиками и тестировщиками.

Традиционный подход показан на рисунке 4 [40].

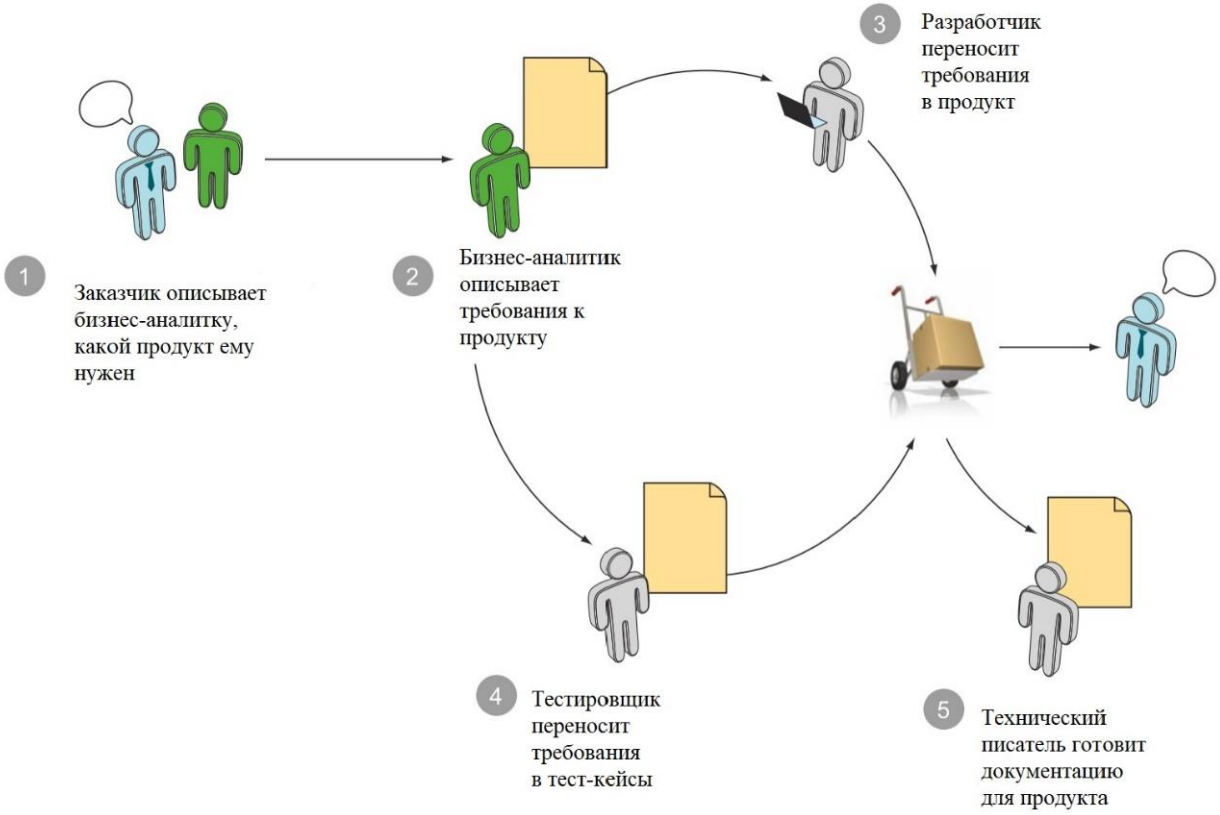

Рисунок 4 – Традиционный подход к организации взаимодействия внутри проектной команды

Перечисленные принципы и изменения затрагивают все стадии разработки приложения:

– проектирование. Отказ от исчерпывающего описания всех аспектов ПО позволяет быстро вносить изменения в документацию без ущерба для цельности общей идеи приложения. Основные задачи QA-специалиста на этапе проектирования – тесное общение с бизнес-аналитиками, изучение проектной документации и подготовка тестовых сценариев;

– разработка. За счет частого выпуска новых версий МП постоянно эволюционирует и не теряет актуальности, когда выходят новые модели смартфонов и планшетов, мобильных ОС и «оболочек». Тестировщики выполняют ручные проверки, которые направлены на поиск существенных (блокирующих, критических и важных) ошибок. Параллельно создаются автоматизированные функциональные тесты;

– завершение итерации. Активное общение с заказчиком позволяет своевременно получать обратную связь и, как следствие, оперативно устранять ошибки, обеспечивать согласованность работы всех составляющих приложения. На этой стадии деятельность QA-специалистов сосредоточена на воспроизведении и анализе проблем, которые найдены и зафиксированы на этапах проектирования и разработки.

Таким образом, применение гибкой методологии позволяет выстроить процесс разработки МП с учетом специфики приложений этого типа.

Выводы по главе 1

1. Мобильные приложения обладают рядом особенностей, которые определяют специфику их тестирования.

2. При тестировании МП используется метод серого ящика, потому что он подразумевает одинаковое внимание к внешней (графический пользовательский интерфейс), так и к внутренней (взаимодействие с сервером) частям приложения.

3. Применение метода серого ящика в тестировании МП подразумевает проведение большого объема проверок в течение короткого цикла разработки.

4. Разработка МП с применением гибкой методологии позволяет учитывать специфику этого приложений на всех этапах разработки.

5. Применение тестирования на всех стадиях ЖЦ приложения на Agile-проекте создает условия для реализации методологии BDD, которая позволяет формировать единое понимание продукта всеми участниками процесса разработки ПО.

Глава 2 Методики автоматизации тестирования мобильных приложений на Agile-проектах

2.1 Определение автоматизированного тестирования и область его применения

На портале «ПроТестинг» дано следующее определение автоматизированного тестирования ПО: «это процесс верификации программного обеспечения, при котором основные функции и шаги теста, такие как запуск, инициализация, выполнение, анализ и выдача результата, выполняются автоматически при помощи инструментов для автоматизированного тестирования» [4].

Автоматизация приносит проектной команде ряд преимуществ:

– экономический эффект от уменьшения затрат по сравнению с выполнением ручного тестирования;

– улучшение качества за счет исключения человеческого фактора, повторного использования автотестов на разных этапах разработки, регулярной проверки функциональности в процессе разработке и того, что QA-специалисты смогут уделить больше внимания проверкам, не покрытым автотестами;

– оптимизация объема тестирования – возможность проверить больший объем функциональности за то же время, а также применить более креативные методы тестирования;

– сокращение времени тестирования – регулярный запуск регрессионных тестов позволяет быстро обнаруживать старые дефекты, а уменьшение времени за счет автоматизации дает возможность быстрее выводить программный продукт на рынок.

При этом происходит оптимизация ключевых задач тестирования:

– тестирование критически важной функциональности (например: функциональности, наличие ошибок в которой связано со многими рисками с точки зрения бизнес-логики или безопасности пользовательских данных);

– проведение регрессии – объем дымового тестирования увеличивается с выпуском каждой новой версии, но вся его суть сводится к проверке того факта, что ранее работавшая функциональность продолжает работать корректно;

– установка и настройка тестового окружения – автотесты пишутся для часто повторяющихся рутинных операций (например, проверка содержимого конфигурационных файлов или реестра) и подготовки приложения для запуска в определенной среде и с определенными настройками для проведения основного тестирования;

– тестирование безопасности – проверка прав доступа, паролей, уязвимостей текущих версий МП и так далее, то есть, быстрое выполнение очень большого количества проверок, в процессе которого нужно учесть большое количество параметров;

– тестирование скорости и надежности работы приложения под разной нагрузкой – создание нагрузки с точностью и интенсивностью, которые недоступны человеку, а также быстрый анализ большого объема данных или сбор большого набора параметров работы приложения;

– модульное тестирование – проверка правильности работы большого количества атомарных участков кода и взаимодействий таких участков кода;

– интеграционное тестирование – проверка взаимодействия компонентов в ситуации, когда почти нечего наблюдать, так как все тестируемые процессы проходят на более глубоких уровнях, чем пользовательский интерфейс;

– выполнение проверок, которые требуют сложных и точных математических расчетов;

– использование комбинаторных техник тестирования – генерация комбинаций значений и многократное выполнение тест-кейсов с использованием полученных комбинаций в качестве входных данных;

– различные «технические задачи» (например: проверка работы с базами данных, корректности протоколирования, файловых операций, поиска, корректности форматов и содержимого генерируемых приложением документов).

Несмотря на все плюсы автоматизации стоит помнить и о ее потенциальных рисках. Частые изменения функционала, кода, структуры баз данных и других компонентов приложения подразумевают регулярное обновление написанных ранее автотестов. В результате автоматизация тестирования МП по времени нередко сопоставима с проведением ручных проверок.

Вопреки названию автоматизированное тестирования не является полностью автономным процессом и подразумевает активное участие человека. Помимо запуска автотестов жизненный цикл автоматизированного тестирования включает оценку выгоды применения автотестов, поиск инструментальных средств, написание скриптов и подготовку тестовых данных, анализ результатов выполнения автотестов.

Исходя из сказанного выше, эффективность работы команды тестирования во многом зависит от того, какие именно задачи было решено автоматизировать и как эта автоматизация была проведена [9, 9, 10, 12, 24].

Любое приложение (в том числе – мобильное) можно условно разделить на три уровня:

– уровень компонентов (Unit layer) включает код МП (например, переменные, функции, методы, библиотеки);

– уровень функциональности (Functional layer) – это бизнес-логика приложения, то есть, практический результат работы, для получения которого оно создано;

– уровень графического интерфейса пользователя (GUI layer) включает компоненты МП, видимые конечному пользователю (например, экраны, кнопки, выпадающие списки).

Автоматизация приносит максимальный эффект процессу тестирования, если она затрагивает все уровни тестируемого МП. На рисунке 5 показано, какие типы автотестов соответствуют всем уровням приложения.

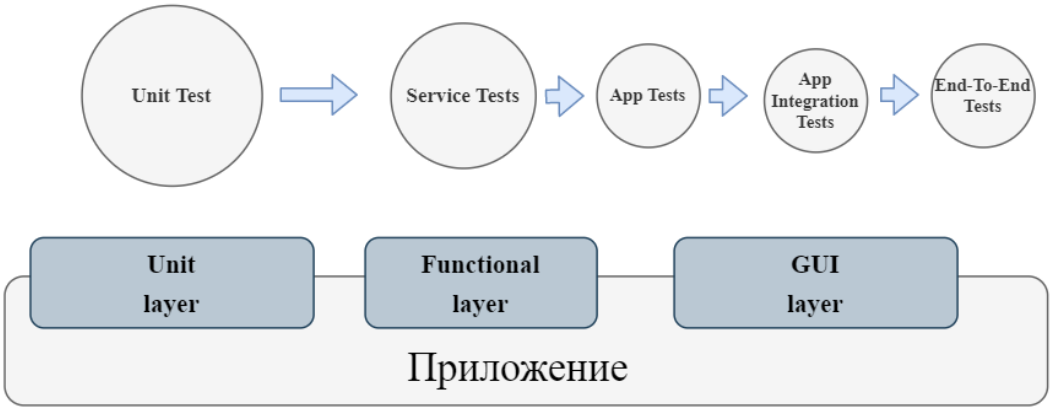

Рисунок 5 – Типы автотестов для разных уровней приложения

Автоматизация начинается на уровне компонентов. Автоматизированные юнит-тесты (Unit Tests) создаются для каждой новой возможности, добавленной в приложение. Они позволяют быстро обнаруживать ошибки в коде.

Следующая ступень – уровень функциональности. Ему соответствуют сервисные автотесты (Service Tests), которые направлены на тестирование классов, образующих компонент в составе нового функционала. Такие тесты запускаются только после успешного завершения юнит-тестирования.

Это тесты, предназначенные для проверки функциональности «в чистом виде». Обычно они запускаются на уровне API без использования графического интерфейса. Если нужно протестировать взаимодействие с внешними сервисами, которые не могут гарантировать предоставление

данных или по каким-то причинам недоступны, используются эмуляторы внешних сервисов.

На уровне интерфейса выполняются автотесты для проверки приложения в целом (App Tests), интеграции приложений (App Integration Tests) и полные сценарные тесты (End-to-End Test).

Автотесты для проверки приложения в целом отличаются глубиной проработки и большим объемом. Их цель – убедиться в корректности работы всего приложения. Если приложение имеет объемный функционал, для тестирования оно может быть разбито на несколько отдельных приложений, предоставляющие пользователю разные возможности.

В описанном выше случае также применяются автотесты интеграции приложений, предназначенные для проверки взаимодействия «приложений внутри приложения» и корректности переключения между ними.

Полные сценарные тесты представляют собой автоматизированные GUI-тесты, которые запускаются для всей системы, воспроизводят типичные пользовательские пути или полные сценарии взаимодействия.

2.2 Алгоритм автоматизации тестирования

В общем виде алгоритм автоматизации тестирования выглядит следующим образом:

Шаг 1: определение цели автоматизации;

Шаг 2: выбор существующей методики автоматизации или разработка новой;

Шаг 3: выбор инструментов автоматизации;

Шаг 4: подготовка тестовой инфраструктуры (например: библиотек кода, систем отчетности, баз данных);

Шаг 5: написание и отладка набора автотестов для основной архитектуры МП (как правило, это проверки для проведения приемочного тестирования);

Шаг 6: подготовка более детализированных автотестов для тестирования критической функциональности, регрессионного тестирования;

Шаг 7: однократный или многократный (в зависимости от целей разработки конкретного набора автотестов) прогон автотестов;

Шаг 8: анализ автоматически сформированных отчетов, оценка качества тестируемого МП;

Шаг 9: поддержка автотестов – проверка адекватности логики новых тестов, обновление параметров в ранее созданных тестах, адаптация тестовой инфраструктуры для проверки измененных версий МП или другого приложения;

Шаг 10: при необходимости – передача автотестов заказчику.

Применение описанного алгоритма позволяет провести автоматизацию тестирования с учетом особенностей конкретного МП.

2.3 Методики автоматизации тестирования МП

Прежде чем выбрать инструментальное средство, необходимо определиться с методикой автоматизации.

В настоящее время применяются четыре методики:

– запись и воспроизведение скриптов,

– написание сценария,

– тестирование под управлением данными,

– тестирование на основе ключевых слов.

Запись и воспроизведение скриптов (Record and Play) – подразумевает использование утилит для записи действий пользователя в приложении. Программа преобразует запись в код и генерирует автотесты, которые в последствии выполняются без участия человека.

Основные преимущества – простота применения, не требуются знания в области программирования. А главный недостаток – необходимость создания новых автотестов после внесения любых изменений в интерфейс

МП. Обычно эта методика активно применяется для проведения дымового тестирования, а также однократного прогона однотипных тестов в разных окружениях.

Написание сценария (Scripting) – методика заключается в использовании тестовых сценариев, написанных на языках, специально разработанных для автоматизации тестирования ПО [20]. Основное отличие от методики «запись и воспроизведение скриптов» – код тестов создается людьми, а не программой.

Это требует серьезных временных и финансовых затрат, поскольку разработкой занимаются программисты или тестировщики с высокой квалификацией. С другой стороны – такие тесты проще поддерживать и масштабировать, поскольку при написании кода вручную автор учитывает возможные изменения в названиях структурных элементов МП, а также может согласовать эти изменения с остальными участниками проектной команды.

Тестирование под управлением данными (Data-driven testing). Это методология создания скриптов и их верификации на основе данных, которые содержатся в базе данных или хранилище. Используется в случаях, когда нужно реализовывать однотипные проверки для различных комбинаций входных данных.

Тестирование на основе ключевых слов (Keyword-based testing) – методика написания автоматизированных тестовых сценариев, использующая подающиеся на вход файлы не только для хранения тестовых данных и ожидаемых результатов, но и ключевых слов, относящихся к тестируемому приложению. Ключевые слова интерпретируются специальными процедурами, вызываемыми из управляющего сценария для данного теста [17].

Тесты, подготовленные в рамках этого подхода, представляют собой не программный код, а последовательность действий с их параметрами. Как и

первый подход, позволяет создавать автотесты тестировщикам, которые не имеют навыков программирования.

При этом автотесты под управлением ключевыми словами стабильнее и легче в поддержке, чем тесты типа Record and Play. Для описания Keyword-based тестов используются ключевые слов, для реализации применяются фреймворки.

При выборе методики автоматизации тестирования для конкретного продукта нужно учитывать такие факторы: особенности предметной области и тип МП, а также методологию разработки приложения.

2.4 Инструменты автоматизации тестирования МП

Автоматизация тестирования приводит к усложнению схемы этого процесса. Схема ручного тестирования включает три компонента – тесты, приложение и тестировщик, выступающий в качестве «посредника» между ними. Он преобразует шаги тест-кейсов в действия с тем или иным интерфейсом приложения (API, GUI, Net и прочие).

Поэтому, чтобы автоматизировать тесты, недостаточно использовать единственный инструмент, функции которого ограничиваются их запуском. Также необходимы инструментальные средства, которые будут формировать тест-кейсы, взаимодействовать с интерфейсами тестируемого приложения и другими инструментами автоматизации [8, 21].

Поэтому схема автоматизированного тестирования включает комплекс инструментов. В зависимости от функциональных возможностей и механизма работы их можно разделить на несколько групп [21]:

– драйверы,

– надстройки,

– фреймворки запуска,

– комбайны.

Выбор конкретных инструментов зависит от типа МП, тестирование которого необходимо автоматизировать. Существуют кросс-платформенные средства автоматизации и специализированные программы, предназначенные для работы с конкретной платформой. Первые применяются для автоматизации МП любых типов, вторые – для автоматизации нативных и гибридных МП.

На рисунке 6 [21] показано изменение схемы тестирования при его автоматизации, а также обозначено место каждого из перечисленных выше типов инструментов в процессе автоматизированного тестирования.

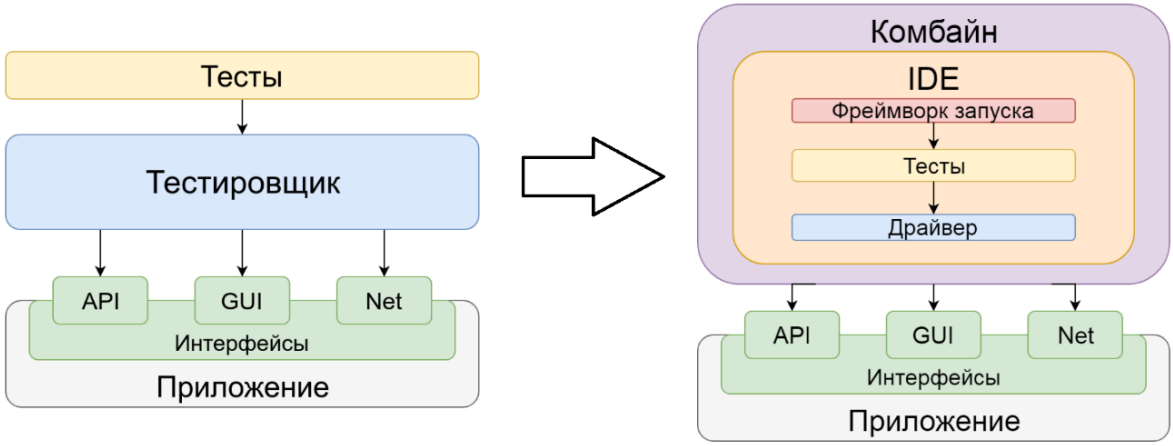

Рисунок 6 – Изменение схемы тестирования при автоматизации

В данном контексте драйвер – это «программа, выдающая себя за аппаратное обеспечение и обеспечивающая связь между программами по стандартному интерфейсу. Другими словами, этот инструмент предоставляет API для одного из интерфейсов приложения» [21].

Например, драйвер для графического интерфейса воспринимает команды через API и отправляет их в тестируемое МП, где команды превращаются в соответствующие действия с графическими элементами.

При тестировании нативных и гибридных МП используются GUI-драйверы XCTest для iOS, UIAutomator и Espresso для Android. Для создания

автотестов графического интерфейса для браузерных МП применяется Selenium WebDriver.

XCTest поддерживает версии iOS от 9.0 и выше. Для создания автотестов нужен доступ к исходному коду тестируемого МП. Тесты для этого драйвера пишутся на тех же языках, что и iOS-приложения – Swift и Objective-C.

В базовой комплектации XCTest тесты можно запустить только на симуляторе, но с помощью сторонних утилит это можно сделать и на реальных устройствах. С помощью рекордера, встроенного в интерфейс Xcode, XCTest позволяет записывать GUI-тесты, находить графические элементы и их свойства.

UIAutomator позволяет работать с версиями Android начиная с Android 4.3 (API level 18) и не требует внедрения своего кода в проект. Этот инструмент поддерживает такие возможности Android как поворот экрана, снятие скриншотов и нажатие на кнопку Home. За счет этого UIAutomator широко применяется для функционального End-to End тестирования.

Утилита UIAutomator Viewer взаимодействует с приложениями, запущенными на эмуляторе или на реальном девайсе, получает данные о GUI-элементах и показывает их локаторы.

Espresso поддерживает версии Android начиная с Android 2.3.3 (API level 10) и предназначен для тестирования методом белого ящика. Притом утилита не может самостоятельно работать с системой Android и другими приложениями. Для запуска драйверу нужен доступ к исходному коду МП. Espresso можно использовать совместно с UIAutomator, сочетая в одном тесте команды обоих инструментов.

Надстройкой называется программа, которая взаимодействует с приложением через один или несколько драйверов, повышает удобство их использования или расширяет их возможности. Надстройка может обладать такими функциями как

– модификация поведения драйвера без изменения API (например: валидация данных, ожидания выполнения действия в течение заданного времени);

– повышение уровня абстракции API путем упрощения сложных команд, реализации альтернативных стилей программирования и так далее.;

– унификация драйверов через предоставление единого интерфейса для них, например, для использования одного и того же кода тестов для использования с приложением на iOS и Android.

Наиболее известными надстройками являются Appium и Calabash.

Appium подходит для автоматизированного тестирования МП вне зависимости от платформы, типа приложения и версии системы. Программа поддерживает описанные ранее драйвера XCTest, UIAutomator и Espresso, предоставляет возможности:

– писать автотесты на любом популярном языке программирования, в том числе – на «неродных» для МП языках;

– создавать и запускать тесты для любых типов МП (нативные, гибридные, веб);

– работать с любым тестовым фреймворком;

– тестировать приложения без доступа к коду.

Надстройка Calabash представлена в виде двух инструментов – Calabash iOS и Calabash Android. Оба поддерживают языки Ruby и JRuby. Calabash Android работает без подступа к коду МП, а утилита для iOS требует подключения Calabash framework.

Фреймворк запуска (далее – фреймворк), в отличие от драйверов и настроек, не является прослойкой между тестами и МП. Это программа, которая служит для формирования и запуска набора тестов, а также сбора результатов их выполнения.

Помимо этого, в задачи фреймворка входит группировка, упорядочение и распараллеливание исполняемых тестов, формирование отчетов об их выполнении. Наиболее популярные фреймворки – xUnit и Cucumber.

Комбайн – это утилита, объединяющая в себе драйверы, фреймворки и возможности разработки. В автоматизированном тестировании чаще всего используются комбайны Xamarin, UITest, Squish и Ranorex.

Xamarin является сервисом для разработки (в основном на языке C#) и тестирования МП. У этого комбайна есть инструменты автоматизации тестирования, а также собственные фермы мобильных устройств.

Ranorex позволяет тестировать МП на эмуляторах и реальных устройствах. Тесты для него пишутся на языках C# и VB.NET. Доступен только для Windows, имеет рекордер для тестов.

У Squish есть собственный рекордер и IDE. Языки для написания тестов: Ruby, Perl, Python, JavaScript.

2.5 Методика автоматизации тестирования на Agile-проектах и оценка ее применимости для МП

Технология Agile подразумевает частый выпуск новых версий приложения, это определяют специфику тестирования на Agile-проектах. На подготовку тестовой документации, исполнение тестов и анализ результатов выделяется гораздо меньше времени, чем на проектах с традиционным подходом к разработке.

По этой причине большая часть проводимых тестировщиками проверок связана с реализацией новых функциональных возможностей, разработка и тестирование выполняются параллельно. Это позволяет с самого начала итерации заложить обеспечение качества и снизить риск того, что новые возможности нарушат работу существующего функционала.

Одним из способов оптимизации процесса тестирования становится применение автотестов. При этом основной целью автоматизации становится возможность быстро оценить состояние приложения и оперативно передать эту информацию разработчикам.

На Agile-проекте автоматизированное тестирование, как и тестирование в целом, является не изолированной задачей или этапом в работе над приложением, а непрерывным процессом, который вписан во все стадии жизненного цикла ПО.

Для обеспечения качественной обратной связи автотесты должны выполняться часто и быстро, а их результаты – быть достоверными и достаточно детализированными [26].

Минимальным критерием готовности новой версии приложения к выпуску считается отсутствие регрессионных дефектов. Поэтому одним из ключевых моментов автоматизации тестирования на Agile-проектах является частый запуск регрессионных тестов [3].

Как правило, автотесты для регрессии включают несколько наборов тест-кейсов, которые отличаются по количеству и степени детализации тест-кейсов:

– пакет «дымовых» автотестов для проверки того, что приложение успешно загружается и запускается, а также проверки нескольких ключевых сценариев работы с приложением. «Дымовой» набор запускается при каждом развертывании приложения;

– пакет функциональных автотестов применяется для более детальной проверки работы приложения, обычно включает несколько тестовых наборов для различных целей. Такие наборы по мере необходимости запускаются в различных окружениях и проверяют стабильность работы приложения. Подобные автотесты запускаются несколько раз в день;

– пакет регрессионных автотестов предназначен для тестирования приложения как единого целого. Такая проверка позволяет убедиться, что различные части приложения, которые обращаются к другим приложениям, различным базам данных, сторонним библиотекам и внешним ресурсам, работают корректно.

Этот набор предназначен не для проверки всех возможностей приложения (их работа ранее проверена функциональными автотестами), а

для тестирования переходов из одного состояния в другое, а также наиболее популярных пользовательских сценариев.

Для подготовки перечисленных автотестов используется методика Scripting. Технически она позволяет создавать автотесты и для графического интерфейса. Но последние обладают рядом особенностей, из-за которых написание таких тестов с использованием Scripting становится неоптимальным решением.

Принято говорить о следующих недостатках автотестов GUI:

– хрупкость – для определения графических элементов и взаимодействия с ними в тестах прописываются локаторы, поэтому после изменения существующих элементов или замены их новыми тесты перестают работать;

– ограниченность тестирования – графический интерфейс не всегда позволяет тестировщику полностью проверить функциональность, поскольку он предоставляет недостаточно деталей, необходимых для верификации;

– относительно низкая скорость выполнения (по сравнению с юнит и API тестами) – так как тесты проводятся через GUI, время загрузки графических элементов заметно увеличивает общее время выполнения тестов, в итоге возрастает и время получения обратной связи;

– наименьшая окупаемость – из-за перечисленных выше проблем, автотесты графического интерфейса становятся невыгодными с финансовой точки зрения.

В условиях Agile автотесты GUI могут утратить актуальность еще до их первого запуска. Ручное создание автотестов для новых графических элементов занимает много времени, а выполнение откладывается до следующей итерации. К тому моменту графический интерфейс приложения снова может измениться, и написанные ранее автотесты придется переделывать.

Поэтому на Agile-проектах часто происходит отказ от GUI-тестов в пользу тестирования на уровне API [3, 14], где автотесты можно запустить сразу после юнит-тестов.

Такой подход к автоматизации допустим для веб- и десктоп-приложений, которые рассчитаны на работу с ограниченным количеством браузеров и платформ.

Но МП проектируются под разные мобильные платформы, версии их операционных систем и конфигурации девайсов. Ручное тестирование МП позволяет проверить только базовые сценарии и только в наиболее популярных окружениях.

Поэтому автоматизации тестирования, которая не охватывает проверки графического интерфейса, неприменима для большинства мобильных приложений. Для МП, которые разрабатываются по технологии Agile, отсутствие автотестов GUI особенно критично.

Типичная длительность итерации на Agile-проекте составляет две недели. За это время невозможно протестировать графический интерфейс МП вручную. В результате многие дефекты GUI попадают в продакшен-версию МП и обнаруживаются конечными пользователями. Такая ситуация может привести к получению негативных отзывов от пользователей и, как следствие, к коммерческому провалу приложения.

Для решения этой проблемы необходимо разработать новую методику автоматизации тестирования, которая позволит автоматизировать тестирование и API, и графического интерфейса МП.

Общий принцип этой методики можно сформулировать так: использование Scripting для создания тестов API, Record and Play – для автотестов GUI.

Методику «запись и воспроизведение» имеет смысл применять для тестирования новых или часто изменяемых экранов МП. Использование программ-драйверов позволяет автоматизировать тестирование новых элементов GUI сразу после их разработки, в рамках одной итерации. Также

данная методика позволяет быстро подготовить новые автотесты для покрытия внеплановых доработок в дизайне или его кардинальном изменении.

Ручное создание автотестов для новых экранов занимает больше времени, чем генерация кода с помощью драйвера, и подразумевает выполнение автотестов только в следующей итерации. До этого придется тестировать новые экраны вручную и, как следствие, ограничиться проверкой основных сценариев.

В итоге возникает двойной риск. Во-первых, после ввода новой версии МП в эксплуатацию пользователи могут обнаружить значительные дефекты в сценариях, не покрытых ручными тестами. Во-вторых, после запуска автотестов в следующей итерации может выяснится, что их необходимо доработать. В этом случае фактическое применение автотестов откладывается еще на две недели.

Поэтому методика «написание сценария» больше подходит для автоматизированной проверки API и функциональностей, которые не подвергаются изменениям или меняются незначительно.

При подготовке автотестов GUI нужно убедиться, что они минимально пересекаются с функциональными автотестами. Последние должны покрывать большую часть позитивных и негативных сценариев, направленных на проверку взаимодействия с сервером, базой данных и различными внешними сервисами.

Автотесты графического интерфейса не должны пересекаться с функциональными проверками. Исключение – тесты, связанные с формированием корректных запросов при взаимодействии пользователя с различными контролами.

Параллельная разработка и запуск автотестов для API и GUI помогает максимально быстро оценить его качество и выявить наиболее крупные дефекты всех типов – структурные ошибки в коде, проблемы функционала и графического интерфейса.

Второе преимущество использования комбинированной методики – возможность комплексно анализировать и чинить проблемы, связанные одновременно с функционалом и компонентами GUI. Это позволяет избежать ситуации, когда дефекты на соответствующих уровнях обнаруживаются и исправляются асинхронно, из-за чего починка на стороне бэк-энда может привести к появлению новых проблем графического интерфейса и наоборот.

Выводы по главе 2

1. Автоматизированное тестирования подразумевает активное участие человека, которое по времени сопоставимо с организацией и проведением ручного тестирования. Поэтому эффективность автоматизации зависит от того, какие именно задачи было решено автоматизировать и как они были выполнены.

2. При выборе методики автоматизации тестирования для конкретного МП нужно учитывать такие факторы: особенности предметной области и тип приложения, а также методологию, по которой оно разрабатывается.

3. При автоматизации тестирования на Agile-проектах чаще всего используется методика Scripting, которая не охватывает проверки графического интерфейса. Такой подход неприемлем для тестирования МП, особенно тех, которые разрабатываются по гибкой методологии. В течение короткой Agile-итерации невозможно вручную провести полноценное тестирование GUI.

4. Необходимо разработать новую методику автоматизации тестирования, которая позволит автоматизировать тестирование и API, и графического интерфейса МП. Ключевой принцип этой методики: использование Scripting для создания тестов API, Record and Play – для автотестов GUI.

Глава 3 Разработка методики автоматизация тестирования МП на Agile -проекте. Апробация разработанной методики

3.1 Разработка методики автоматизация тестирования МП

3.1.1 Постановка задачи на разработку методики автоматизации тестирования

Анализ методики Scripting показал, что ее применение не является оптимальным решением для автоматизации тестирования МП на Agile-проектах.

Для тестирования МП необходимо использовать автотесты GUI. Это позволит провести полное тестирование графического интерфейса МП в течение короткой Agile-итерации. При этом автотесты GUI являются не заменой, а дополнением для автотестов API. Поэтому автоматизация проверок графического интерфейса не должна сокращать время на создание автотестов API.

Также при постановке задачи на разработку методики автоматизации МП нужно учитывать специфику последних:

– мобильная разработка подразумевает необходимость часто изменять или дорабатывать работающий дизайн, переписывать код фронт-энда, чтобы обеспечить корректную работу приложения с меняющимися параметрами мобильных устройств (размер и разрешение экрана, обновление сенсоров и так далее);

– большая часть нативных и гибридных МП проектируется для работы на iOS и Android, поэтому код для клиентской части МП пишется на двух разных языках. Соответственно, для автоматизации проверок фронт-энда потребуются отдельные наборы автотестов для каждой платформы.

Исходя из вышесказанного можно сформулировать следующие требования к новой методике:

1. возможность разрабатывать большой объем автоматизированных тестов GUI в сжатые сроки;

2. возможность параллельно создавать и запускать автотесты для API и графического интерфейса мобильного приложения.

Применение методики, разработанной с учетом данных требований, позволит повысить эффективность процесса тестирования МП.

3.1.2 Методика автоматизации тестирования МП на Agile -проекте

На основе анализа методики Scripting и с учетом специфики тестирования мобильных приложений, разработана методика автоматизации тестирования МП, проектируемых по технологии Agile.

На рисунке 7 представлена диаграмма вариантов использования предлагаемой методики.

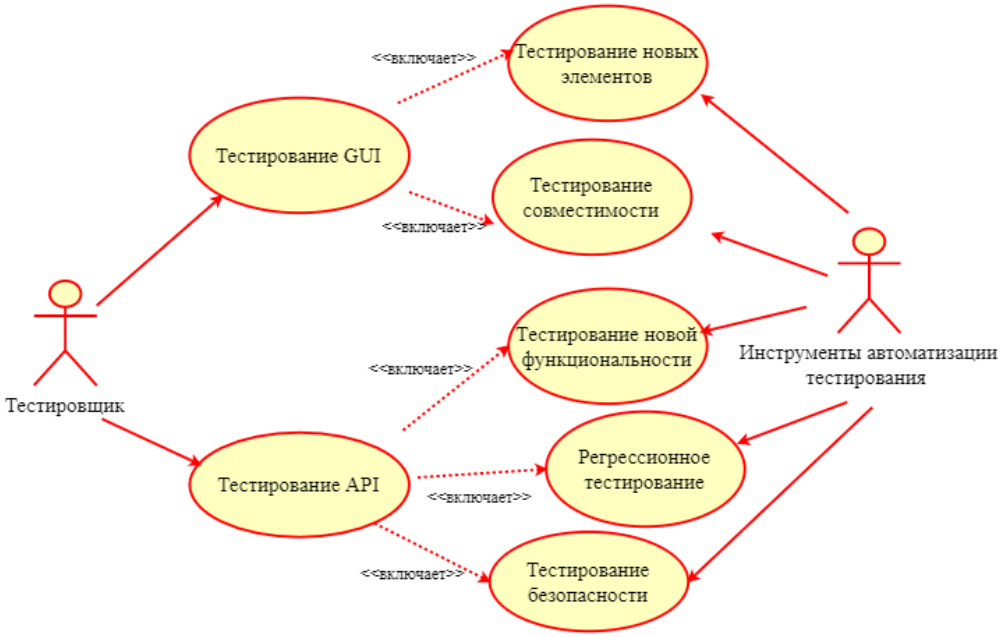

Рисунок 7 – Диаграмма вариантов использования методики

В предлагаемой методике для автоматизации тестирования на разных уровнях МП применяются две разные методики:

– для создания автотестов GUI используется методика Record and Play;

– для создания автотестов API используется методика Scripting.

Автоматизированное тестирование параллельно выполняется на GUI и API уровнях приложения.

Тестирование GUI применительно к МП включает следующие проверки:

– тестирование работоспособности новых элементов графического интерфейса;

– проверка функционирования элементов GUI на различных моделях мобильных девайсов, версиях ОС и графических оболочек, перечисленных в матрице тестовых девайсов.

Такая матрица составляется на этапе проектирования МП и включает устройства с набором параметров, которые наиболее популярны у потенциальных пользователей. Пример матрицы тестовых девайсов представлен на рисунке 8.

Платформа	Тип устройства	Наименование	Модель	ОС	Ширина	Высота	Архитектура	Память
Android	Smartphone	Google Pixel 3 XL	Pixel 3 XL	10	1440	2960	arm64-v8a	64
Android	Smartphone	Samsung Galaxy A50	SM-A505U	9	1080	2340	arm64-v8a	64
Android	Tablet	Samsung Galaxy Tab S4	SM-T830	8.1.0	1600	2560	arm64-v8a	64
Android	Tablet	Samsung Galaxy Tab S6 (WiFi)	SM-T860	9	1600	2560	arm64-v8a	128
IOS	Tablet	Apple iPad 2	A1395	9.3.5	768	1024	armv7f	32
IOS	Tablet	Apple iPad Mini 2	A1489	9.3.1	1536	2048	arm64	16
IOS	Smartphone	Apple iPhone 11	MWL72LL/A	13.1.3	828	1792	arm64e	64
Android	Tablet	ASUS Nexus 7 - 2nd Gen (WiFi)	ME571K	6	1200	1920	armeabi-v7a	32
Android	Smartphone	Galaxy S8 Unlocked	SM-G950U	7	1440	2960	arm64-v8a	64
Android	Smartphone	HTC One M9 (AT&T)	6735A	5.0.2	1080	1920	arm64-v8a	32
Android	Smartphone	LG Nexus 5	D820	5.0.1	1080	1920	armeabi-v7a	16
Android	Smartphone	LG Nexus 5	D820	6	1080	1920	armeabi-v7a	16
Android	Smartphone	Motorola Moto G - 3rd Gen	MotoG3	6	720	1280	armeabi-v7a	8
Android	Smartphone	Samsung Galaxy Note5 (AT&T)	SM-N9120/	7	1440	2560	arm64-v8a	32
Android	Tablet	Samsung Galaxy Tab A 10.1"	SM-T580	7	1200	1920	armeabi-v7a	16
Android	Tablet	Samsung Galaxy Tab S2 8.0" (W	{SM-T710}	5.1.1	1536	2048	armeabi-v7a	32
Android	Tablet	Samsung Galaxy Tab S2 8.0" (W	SM-T713	6.0.1	1536	2048	arm64-v8a	32
IOS	Tablet	Apple iPad Air	A1474	10.3.3	1536	2048	arm64	16
IOS	Tablet	Apple iPad Air 2	A1566	11,1	1536	2048	arm64	16
IOS	Smartphone	Apple iPhone 6	A1549	10.3.1	750	1334	arm64	16
IOS	Smartphone	Apple iPhone 6 Plus	A1522	10.2.1	1080	1920	arm64	16
IOS	Smartphone	Apple iPhone 6s	A1633	11	750	1334	arm64	16
IOS	Smartphone	Apple iPhone 8	A1863	12	750	1334	arm64	64
IOS	Smartphone	Apple iPhone 8 Plus	A1864	11	1080	1920	arm64	64
IOS	Smartphone	Apple iPhone XR	MRYR2LL/A	12.4.1	828	1792	arm64e	64

Рисунок 8 – Матрица тестовых девайсов

Тестирование API мобильного приложения включает два типа проверок:

– функциональное тестирования, которое включает тестирование новой функциональности и регрессионное тестирование;

– тестирование безопасности.

Для апробации разработанной методики были выбраны следующие инструментальные средства:

1. Интегрированные среды разработки (ИСР) IntelleJ IDEA, Android Studio и Xcode. Первая использовалась для создания автотестов для бэк-эндной части МП. Остальные две ИСР применялись для подготовки автотестов фронт-энда МП, представленного версиями под мобильные платформы iOS и Android;

2. Фреймворк Cucumber [34] для создания кода автотестов на основе сценариев, написанных на предметно-ориентированном языке;

3. Фреймворк Allure для генерации отчетов о выполнении автотестов.

Диаграмма компонентов методики, включающая перечисленные инструменты, представлена на рисунке 9.

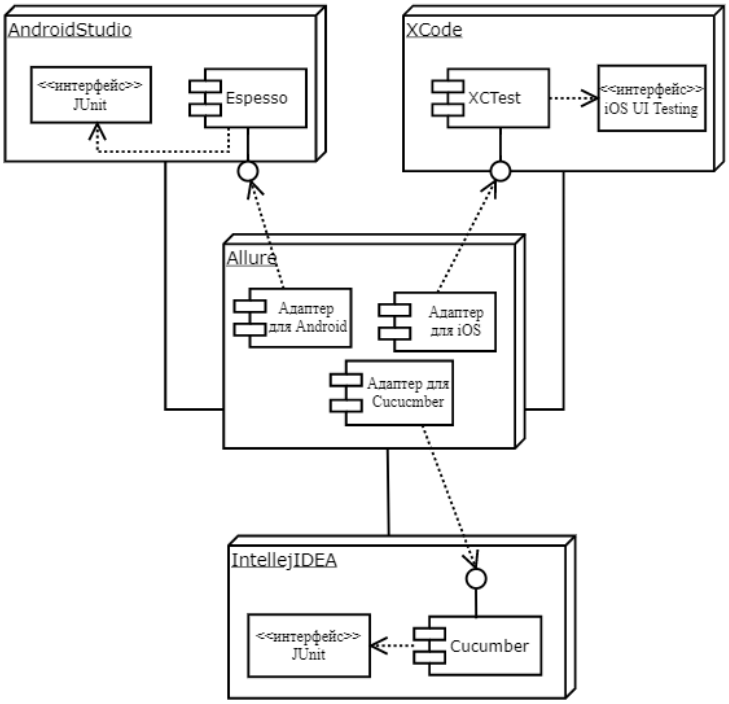

Рисунок 9 – Диаграмма компонентов методики автоматизации МП

После выполнения всех автотестов их результаты собираются в общий отчет, на основе которого проводится анализ эффективности процесса тестирования.

Для оценки эффективности применения методики будут использоваться тестовые метрики:

– тестовое покрытие (Test Coverage) [33];

– количество дефектов по приоритету (Bugs by Priority);

– количество тест-кейсов, не выполненных в ходе текущей итерации (Not Run Test Cases).

Тестовое покрытие оценивается как для набора тестовых устройств в целом, так и для каждого конкретного девайса.

3.2 Апробация методики автоматизации тестирования МП

3.2.1 Анализ предметной области тестируемого МП и условия проведения эксперимента

Апробация разработанной методики автоматизации проведена на нативном МП для iOS и Android, которое разрабатывается по методологии Scrum с применением BBD-подхода.

В работе использовано следующее определение Scrum: «одна из гибких технологий, позволяющая в жестко фиксированные и небольшие по времени итерации, называемые спринтами (sprints), предоставлять конечному пользователю работающий продукт с новыми бизнес-возможностями, для которых определен наибольший приоритет» [39].

МП предназначено для администрирования информационной системы (ИС) сети книжных магазинов. Это многопользовательское приложение для получения контролируемого доступа к ИС и выполнению операций, набор которых зависит от роли пользователя.

В приложении определены следующие роли:

– владелец сети магазинов (Owner),

– главный администратор сети магазинов (Main administrator),

– администратор филиала (Department administrator),

– сотрудник магазина (Employee).

Ниже перечислены основные сущности, представленные в структуре данных МП:

– пользователь (User),

– роль (Role),

– покупатель (Customer),

– книга (Book),

– магазин (Department),

– издательство (Publishing office),

– заказ от покупателя (Order),

– поставка от издательства (Supply).

На рисунке 10 показаны варианты использования МП для пользователя с ролью владелец магазина.

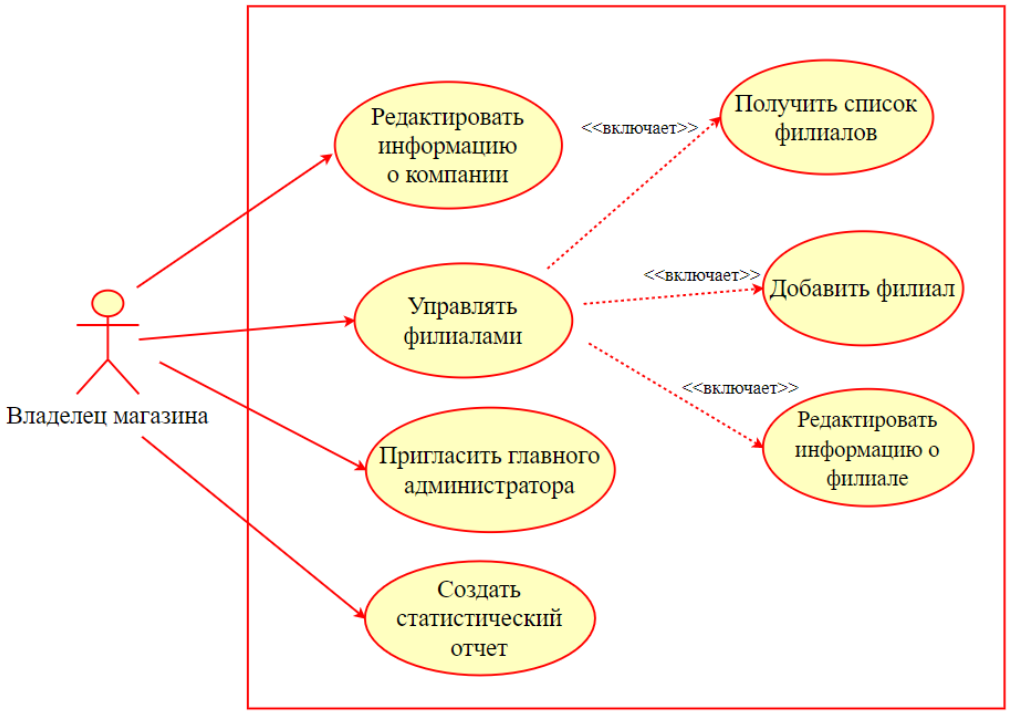

Рисунок 10 – Варианты использования МП для владельца магазина

На рисунке 11 показаны представлены варианты использования для пользователя с ролью главный администратор.

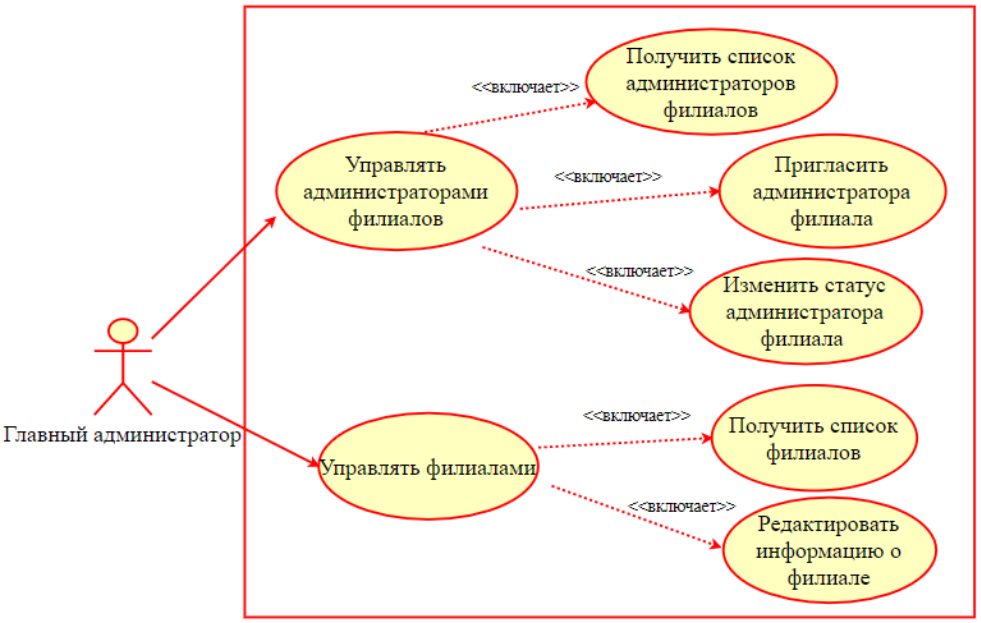

Рисунок 11– Варианты использования МП для главного администратора

На рисунке 12 представлена диаграмма вариантов использования для пользователя «администратор филиала».

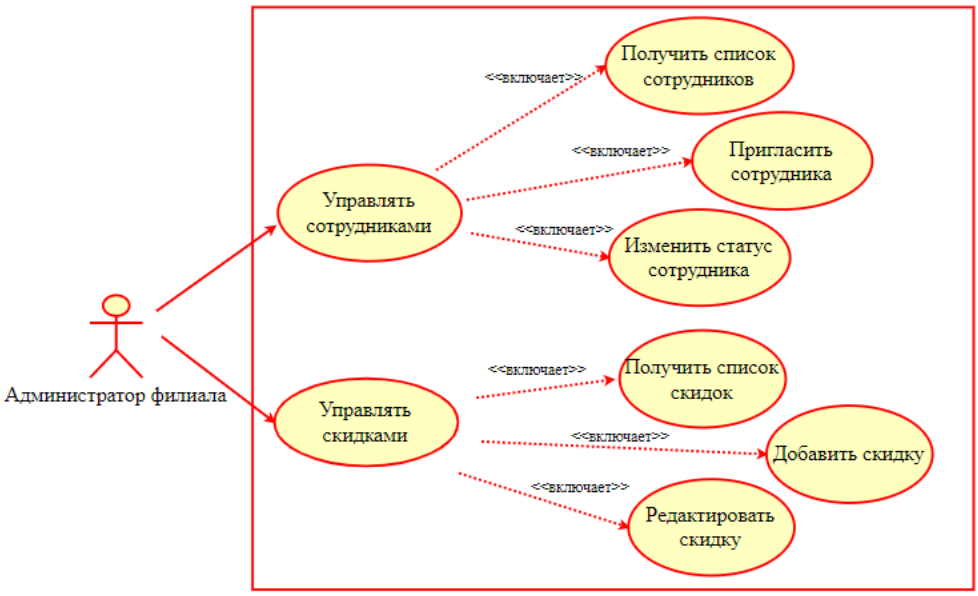

Рисунок 12 – Варианты использования МП для администратора филиала

На рисунке 13 показана диаграмма вариантов использования для пользователя «сотрудник магазина».

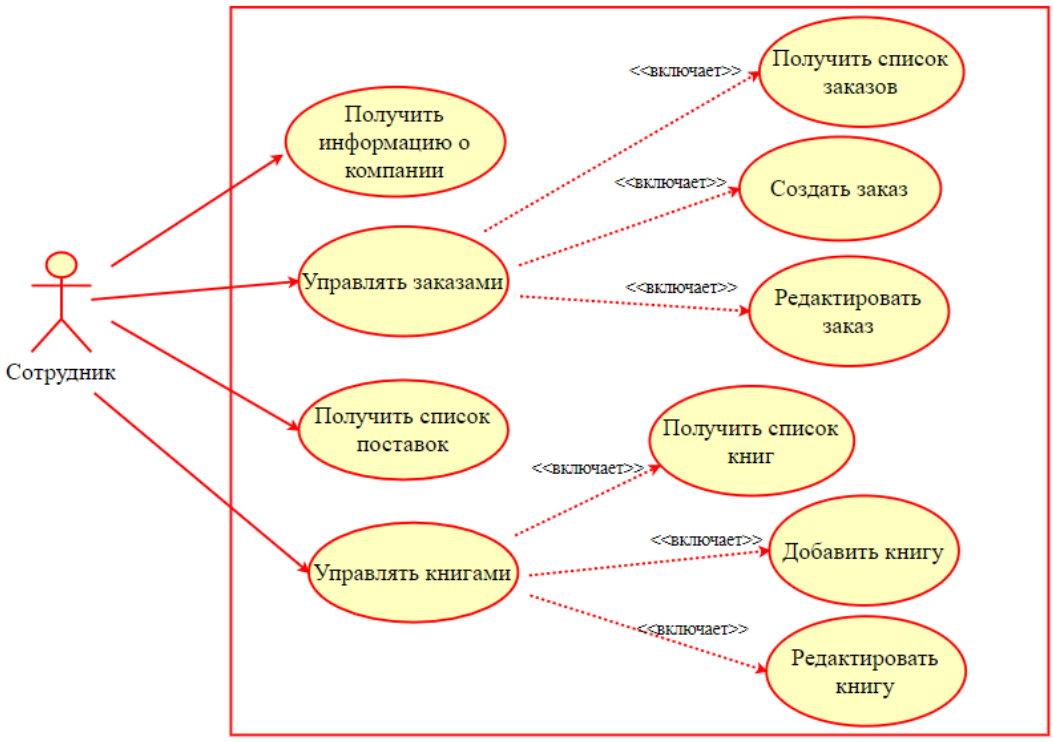

Рисунок 13 – Варианты использования МП для сотрудника магазина

Эксперимент проводился после реализации в МП новой функциональности, доступной всем пользователям МП – возможности переноса книг из списка поставки в список книг магазина, который отображается на соответствующем экране (Book list). Для этой функциональности был добавлен новый экран «Доставленные книги» (Delivered books) и несколько новых запросов:

– получение списка доставленных книг,

– получение конкретной книги из списка доставленных книг,

– добавление книги из списка доставленных книг в список книг магазина.

Прочие условия проведения эксперимента описаны в таблице 1.

Таблица 1 – Условия проведения эксперимента

Методика автоматизации	Record and Play	Scripting
Платформа	iOS, Android	iOS, Android
Метод тестирования	Метод серого ящика	Метод серого ящика
Уровень МП	GUI	API
Тестируемая часть системы	Фронт-энд	Бэк-энд
Вид тестирования	Графического интерфейса, совместимости	Новой функциональности, регрессионное, безопасности

Блок-схема алгоритма проведения автоматизированного тестирования с применением разработанной методики представлена на рисунке 14.

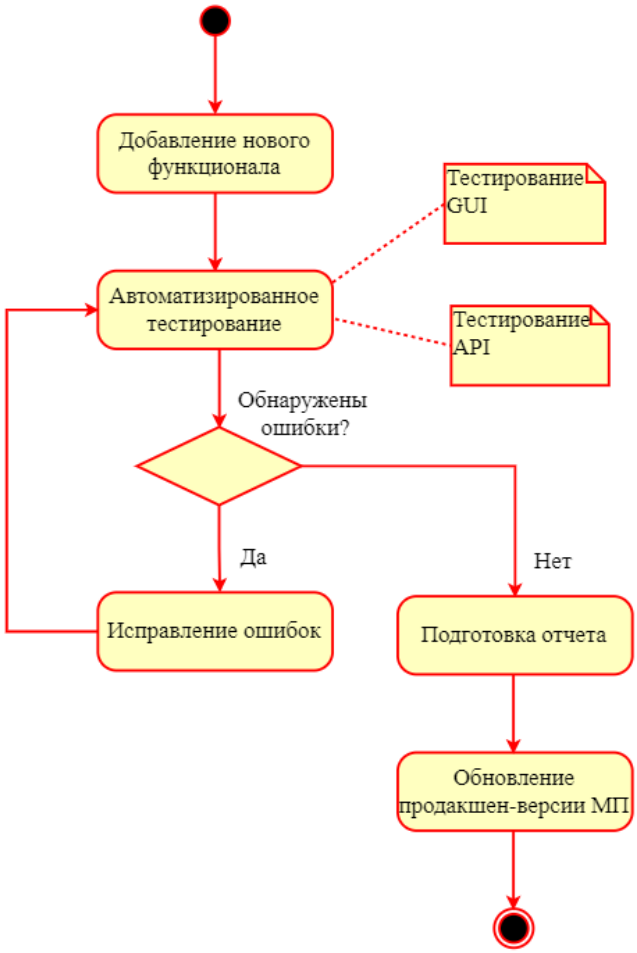

Рисунок 14 – Алгоритм проведения тестирования с применением разработанной методики

Далее описана апробация разработанной методики для тестирования GUI и API.

3.2.2 Апробация методики автоматизации для тестирования GUI

Для записи автотестов с применением методики Record and Play для МП используются специальные драйверы, встроенные в платформенные средства разработки. Для Android это драйвер Espresso в составе Android Studio, для iOS – драйвер XCTest, встроенный в Xcode.

Алгоритм создания таких автотестов одинаковый для обеих платформ:

Шаг 1: в платформенной ИСР выбрать реальное устройство или эмулятор для запуска тестируемого приложения;

Шаг 2: включить запись действий пользователя, вручную выполнить нужный тест;

Шаг 3: убедиться, что в записи, созданной драйвером, нет лишних шагов;

Шаг 4: после завершения записи выбрать опцию, которая запускает генерацию кода автотеста;

Шаг 5: запустить автоматическое выполнение подготовленного теста и убедиться, что он работает корректной;

Шаг 6: с помощью фреймворка Allure сгенерировать отчет о выполнении автотеста.

В таблице 2 более детально описано окружение, которое было создано для подготовки и запуска автотестов с применением методики Record and Play для iOS и Android.

Таблица 2 – Описание окружения для работы с автотестами Record and Play

Платформа	iOS	Android
Операционная система	macOS	Windows
Приложение	.swift файлы с исходным кодом приложения, исполняемый ipa.файл	.java файлы с исходным кодом приложения, исполняемый apk.файл
Язык разработки приложения	Swift	Java
Среда разработки	Xcode 11.4.1	Android Studio 3.6.6
Инструмент автоматизации	Надстройка Appium версия 1.15.1	Надстройка Appium 1.15.1
	Драйвер XCTest	Драйверы Espresso 3.2.0 и UIAutomator2

На рисунке 15 показаны результаты запуска автотестов в Android Studio.

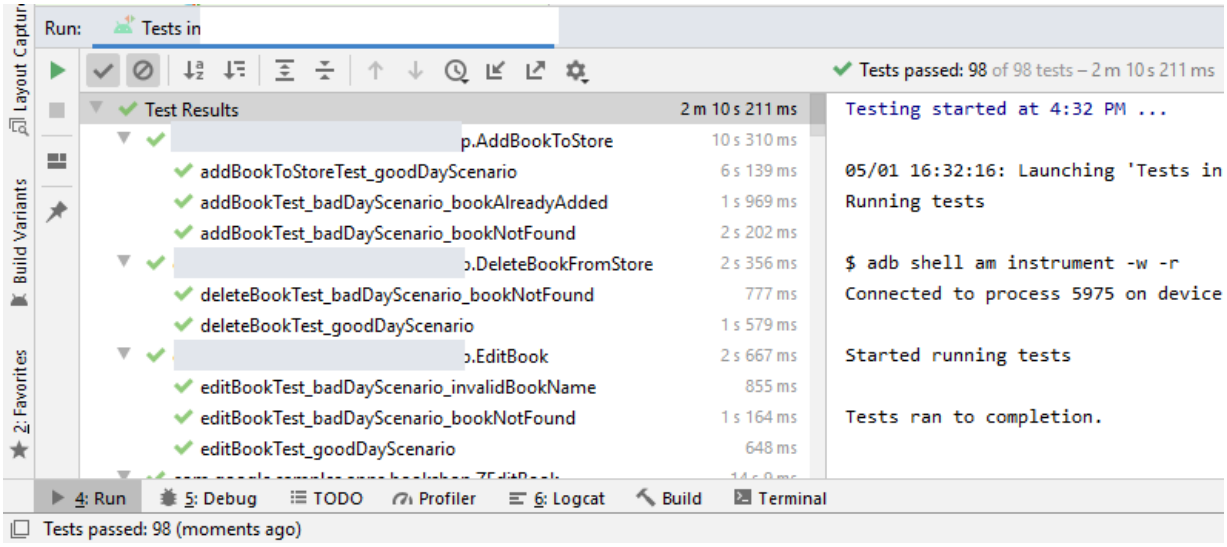

Рисунок 15 – результаты запуска автотестов Record and Play в Android Studio

На рисунке 16 показан фрагмент автотеста, записанного в xCode.

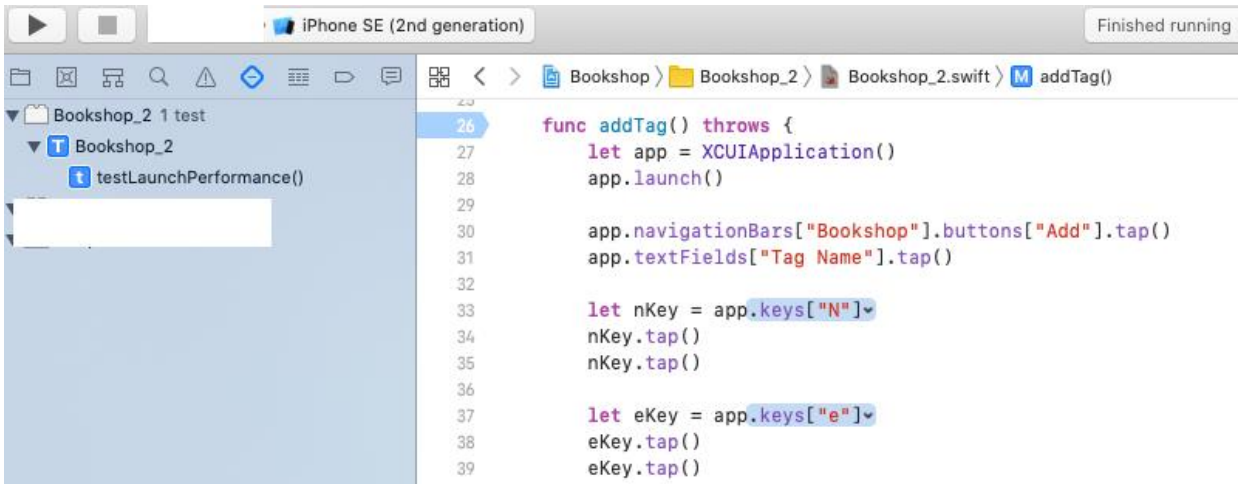

Рисунок 16 – работа с автотестами Record and Play в xCode

Далее приведен пример автотеста, записанного с помощью Espresso в Android Studio. Этот тест предназначен для проверки новых элементов GUI – экрана «Доставленные книги» и кнопки добавления книги в список магазина.

```
@LargeTest
@RunWith(AndroidJUnit4.class)
public class AddBookToStore {

    public int POSITION = 3;

    @Rule
```

```
    public ActivityTestRule<BookshopActivity> mActivityTestRule = new
ActivityTestRule<>(BookshopActivity.class);

    @Test
    public void BookshopActivityTest3() {
        POSITION = new Random(new Date().getTime()).nextInt(12);
        ViewInteraction tabView = onView(
                allOf(withContentDescription("Book list"),
                        childAtPosition(
                                childAtPosition(
                                        withId(R.id.tabs),
                                        0),
                                1),
                        isDisplayed()));
        tabView.perform(click());

        ViewInteraction recyclerView = onView(
                allOf(withId(R.id.plant_list),
                        childAtPosition(
                                withClassName(is("android.widget.FrameLayout")),
                                0)));
        recyclerView.perform(actionOnItemAtPosition(POSITION, click()));

        ViewInteraction floatingActionButton = onView(
                allOf(withId(R.id.fab),
                        childAtPosition(
                                childAtPosition(
                                        withId(R.id.nav_host),
                                        0),
                                2),
                        isDisplayed()));
        floatingActionButton.perform(click());

        ViewInteraction appCompatImageButton = onView(
                allOf(childAtPosition(
                        allOf(withId(R.id.toolbar),
                                childAtPosition(
                                        withId(R.id.toolbar_layout),
                                        1)),
                        0),
                        isDisplayed()));
        appCompatImageButton.perform(click());

        ViewInteraction tabView2 = onView(
                allOf(withContentDescription("Store"),
                        childAtPosition(
                                childAtPosition(
                                        withId(R.id.tabs),
                                        0),
                                0),
                        isDisplayed()));
        tabView2.perform(click());

        ViewInteraction recyclerView2 = onView(
                allOf(withId(R.id.Book_list),
                        childAtPosition(
                                withClassName(is("android.widget.FrameLayout")),
                                0)));
        recyclerView2.perform(actionOnItemAtPosition(0, click()));
    }
```

```
    private static Matcher<View> childAtPosition(
            final Matcher<View> parentMatcher, final int position) {

        return new TypeSafeMatcher<View>() {
            @Override
            public void describeTo(Description description) {
                description.appendText("Child at position " + position + " in parent
");
                parentMatcher.describeTo(description);
            }

            @Override
            public boolean matchesSafely(View view) {
                ViewParent parent = view.getParent();
                return parent instanceof ViewGroup && parentMatcher.matches(parent)
                        && view.equals(((ViewGroup) parent).getChildAt(position));
            }
        };
    }
}
```

Ниже приведен пример теста, созданного в xCode. Тест предназначен для проверки реализованных ранее элементов GUI, связанных с функциональностью «добавление нового тега для книг».

```
func testExample() throws {
      let app = XCUIApplication()
      app.launch()

      app.navigationBars["Bookshop"].buttons["Add"].tap()
      app.textFields["Tag Name"].tap()

      let nKey =
app/*@START_MENU_TOKEN@*/.keys["N"]/*[[".keyboards.keys[\"N\"]",".keys[\"N\"]"],[[[-
1,1],[-1,0]]],[0]]@END_MENU_TOKEN@*/
      nKey.tap()
      nKey.tap()

      let eKey =
app/*@START_MENU_TOKEN@*/.keys["e"]/*[[".keyboards.keys[\"e\"]",".keys[\"e\"]"],[[[-
1,1],[-1,0]]],[0]]@END_MENU_TOKEN@*/
      eKey.tap()
      eKey.tap()

      let wKey =
app/*@START_MENU_TOKEN@*/.keys["w"]/*[[".keyboards.keys[\"w\"]",".keys[\"w\"]"],[[[-
1,1],[-1,0]]],[0]]@END_MENU_TOKEN@*/
      wKey.tap()
      wKey.tap()
```

```
        let spaceKey =
app/*@START_MENU_TOKEN@*/.keys["space"]/*[[".keyboards.keys[\"space\"]",".keys[\"sp
ace\"]"],[[[-1,1],[-1,0]]],[0]]@END_MENU_TOKEN@*/
        spaceKey.tap()
        spaceKey.tap()

        let tKey =
app/*@START_MENU_TOKEN@*/.keys["t"]/*[[".keyboards.keys[\"t\"]",".keys[\"t\"]"],[[[-
1,1],[-1,0]]],[0]]@END_MENU_TOKEN@*/
        tKey.tap()
        tKey.tap()

        let app2 = app
        app2/*@START_MENU_TOKEN@*/.keys["a"]/*[[".keyboards.keys[\"a\"]",".keys[\"a\"]"
],[[[-1,1],[-1,0]]],[0]]@END_MENU_TOKEN@*/.tap()

        let gKey =
app2/*@START_MENU_TOKEN@*/.keys["g"]/*[[".keyboards.keys[\"g\"]",".keys[\"g\"]"],[[[-
1,1],[-1,0]]],[0]]@END_MENU_TOKEN@*/
        gKey.tap()
        gKey.tap()
        app2/*@START_MENU_TOKEN@*/.buttons["Done"]/*[[".keyboards",".buttons[\"done\"
]",".buttons[\"Done\"]"],[[[-1,2],[-1,1],[-1,0,1]],[[-1,2],[-
1,1]]],[0]]@END_MENU_TOKEN@*/.tap()
        app/*@START_MENU_TOKEN@*/.otherElements["PopoverDismissRegion"]/*[[".other
Elements[\"dismiss popup\"]",".otherElements[\"PopoverDismissRegion\"]"],[[[-1,1],[-
1,0]]],[0]]@END_MENU_TOKEN@*/.tap()

        let bookshopButton = app.navigationBars["New tag"].buttons["Bookshop"]
        bookshopButton.tap()
        app.tables.children(matching: .cell).element(boundBy: 0).children(matching:
.staticText).matching(identifier: "-").element(boundBy: 0).tap()
        bookshopButton.tap()

    }
```

При необходимости автотесты, созданные с применением методики Record and Play, можно доработать. Например, добавить в код автотестов значения идентификаторов и имена локаторов элементов GUI, не распознанные драйвером.

Для этого нужно использовать надстройку Appium и совместимые с ней драйверы. При работе с тестируемым МП использовались UIAutomator2 для Android и XCTest для iOS.

Алгоритм работы с Appium выглядит следующим образом:

Шаг 1: запустить сервер Appium;

Шаг 2: в настройках Appium указать возможности (Capabilities), которые будут использованы при запуске сессии – платформу, версию ОС, имя реального девайса или эмулятора, название нужного драйвера и путь до исполняемого файла для тестируемого МП;

Шаг 3: начать сессию. После подключения программы к девайсу и запуска МП перейти на нужный экран и выбрать графический элемент, для которого нужно узнать локатор и (или) идентификатор.

Шаг 4: скопировать данные элемента в ранее записанные автотесты, в которых задействован этот элемент;

Шаг 5: запустить обновленный тест и убедиться, что он работает корректно.

На рисунке 17 показана конфигурация Appium для запуска драйвера UIAutomator.

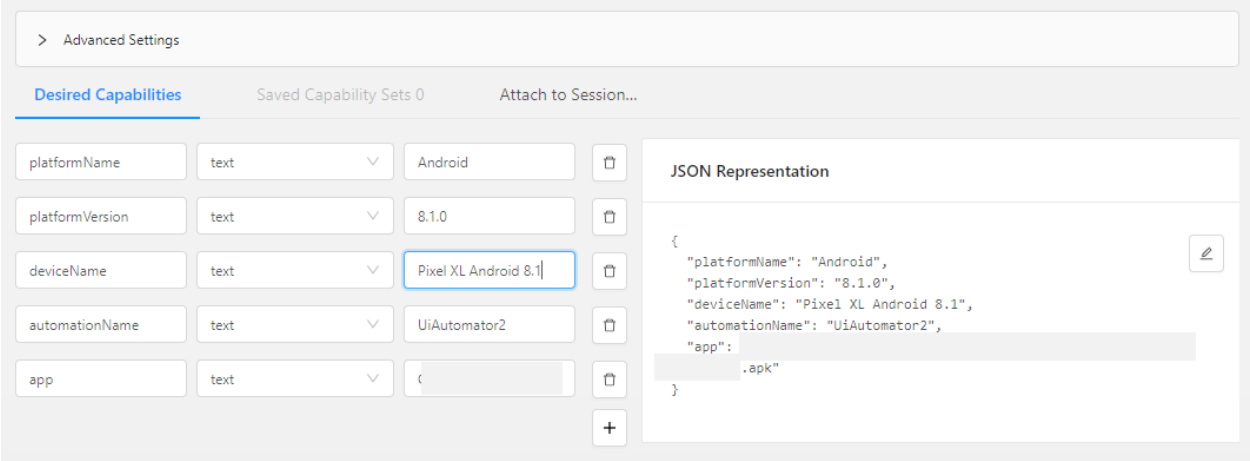

Рисунок 17 – Конфигурация Appium для запуска драйвера UIAutomator

На рисунке 18 показано отображение локаторов графических элементов в интерфейсе программы UIAutomator.

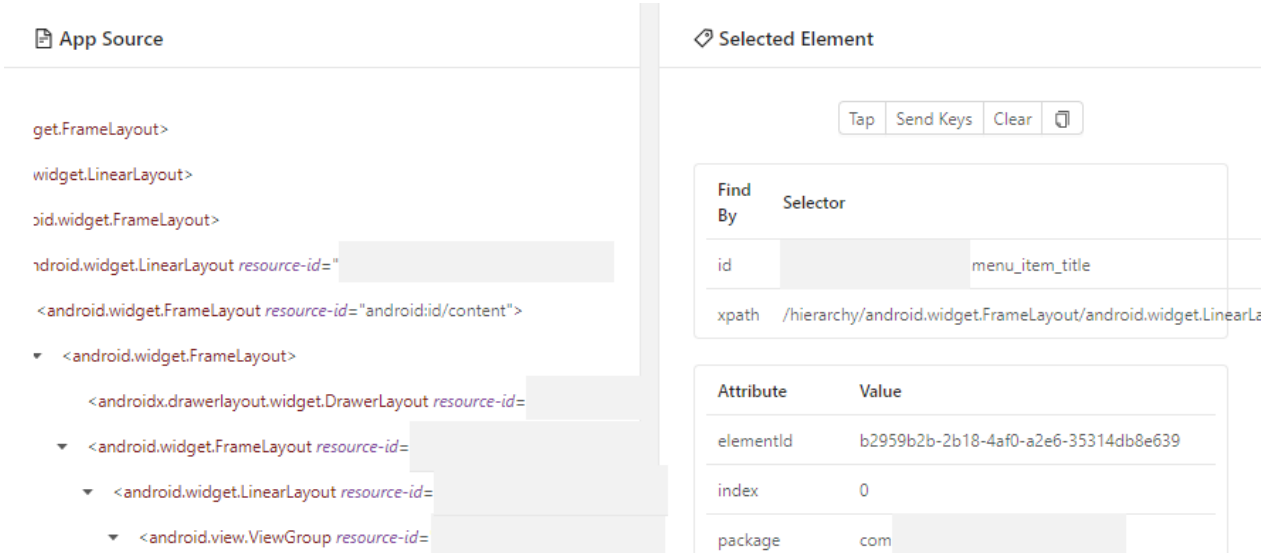

Рисунок 18 – Отображение локаторов элементов GUI в UIAutomator

Автотесты, созданные с применением методики Record and Play, были выполнены на всех тестовых устройствах из матрицы девайсов, подготовленной при проектировании МП.

Это позволило убедиться, что новые элементы графического интерфейса корректно функционируют на всех популярных окружениях.

3.2.3 Апробация методики для автоматизации тестирования API

Код бэк-эндной части МП не привязан к конкретной мобильной платформе, поэтому автотесты API являются универсальными для iOS и Android. Для работы с такими автотестами используется среда разработки, поддерживающая язык, на котором написан код приложения.

В соответствии с принципами BDD, подготовка автотестов с применением методики Scripting начинается с написания тестового сценария. Сущности и функциональности тестируемого МП описываются с помощью ключевых слов предметно-ориентированного языка, который одинаково понятен разработчикам, тестировщикам и бизнес-аналитикам [32].

Так как код бэк-энда МП, описанного в условиях проведения эксперимента, реализован на Java, при апробации методики использовалась

среда Intellej IDEA. На рисунке 19 показана подготовка тестового сценария в Intellej IDEA.

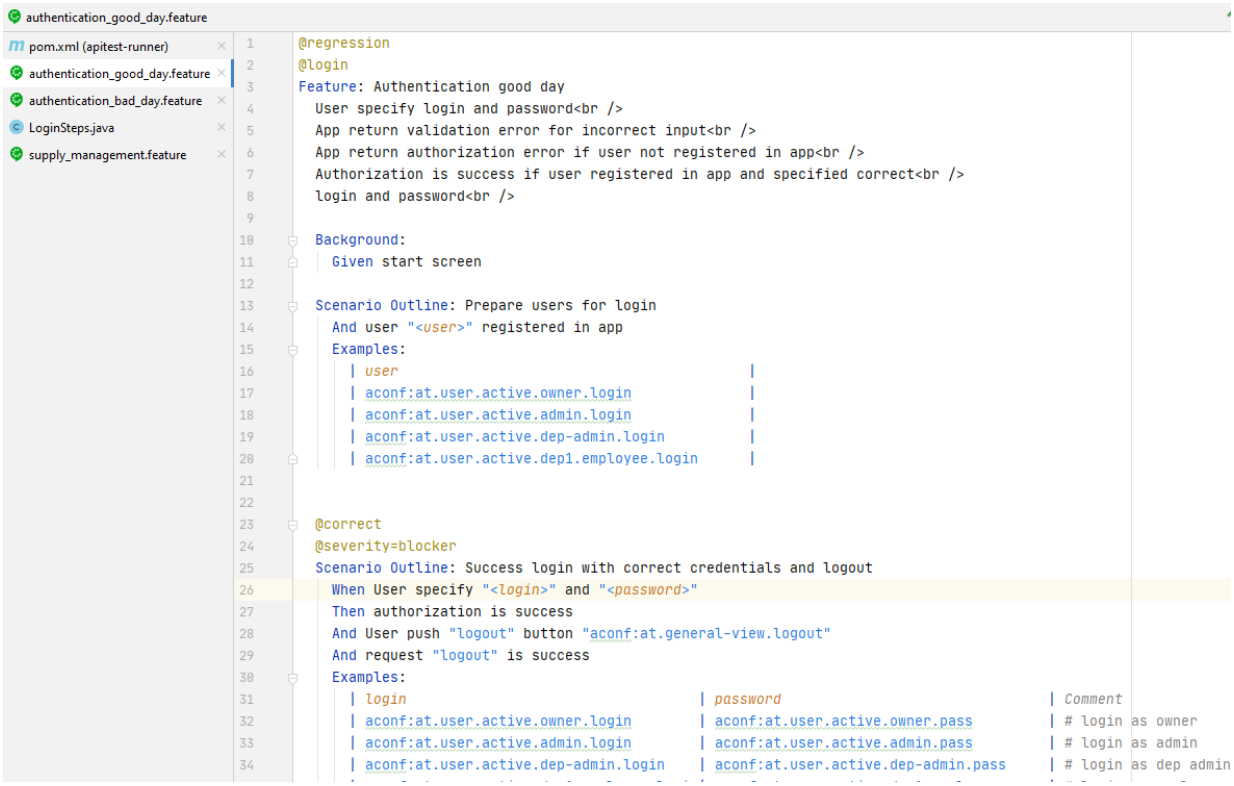

Рисунок 19 – Подготовка тестового сценария в IntelliJ IDEA

Алгоритм автоматизированного тестирования с применением методики Scripting состоит из следующих шагов:

Шаг 1: в ИСР создать сценарий тестирования на предметно-ориентированном языке Gherkin , совместимом с фреймворком Cucumber [32];

Шаг 2: написать код автотеста на основе подготовленного сценария;

Шаг 3: с помощью фреймворка сборки Maven настроить конфигурацию запуска сценариев;

Шаг 4: запустить полученный автотест в среде разработки;

Шаг 5: с помощью фреймворка Allure сгенерировать отчет о выполнении автотеста.

Описание окружения для подготовки и запуска автотестов Scripting представлено в таблице 3.

Таблица 3 – Описание окружения для работы с автотестами Scripting

Операционная система	Windows
Методология	BDD
Среда разработки	IntelleJ IDEA
Язык написания сценариев	Gherkin based
Язык написания кода автотестов	Java 8
Инструмент автоматизации	Фреймворк Cucumber 5.5.0, библиотека cucumber-java 1.2.5,
	Фреймворк автоматической сборки Maven 3.6.1

Далее приведен пример тестового сценария для проверки авторизации (Authentication) в МП из регрессионного набора тестов. В сценарий включены как позитивные (correct), так и негативные (fail) кейсы.

```
@regression
@login
Feature: Authentication good day
  User specify login and password<br />
  App return validation error for incorrect input<br />
  App return authorization error if user not registered in app<br />
  Authorization is success if user registered in app and specified correct<br />
  login and password<br />

  Background:
    Given start screen

  Scenario Outline: Prepare users for login
    And user "<user>" registered in app
    Examples:
      | user                                |
      | aconf:at.user.active.owner.login    |
      | aconf:at.user.active.admin.login    |
      | aconf:at.user.active.dep-admin.login |
      | aconf:at.user.active.dep1.employee.login |


  @correct
  @severity=blocker
  Scenario Outline: Success login with correct credentials and logout
    When User specify "<login>" and "<password>"
```

```
  Then authorization is success
  And User push "logout" button "aconf:at.general-view.logout"
  And request "logout" is success
  Examples:
    | login                          | password                          | Comment
    | aconf:at.user.active.owner.login       | aconf:at.user.active.owner.pass       | # login as
owner
    | aconf:at.user.active.admin.login       | aconf:at.user.active.admin.pass       | # login as
admin
    | aconf:at.user.active.dep-admin.login    | aconf:at.user.active.dep-admin.pass    | # login as
dep admin
    | aconf:at.user.active.dep1.employee.login| aconf:at.user.active.dep1.employee.pass | # login
as employee

 @fail
 @severity=blocker
 Scenario Outline: Failed login in case of incorrect input
  When User specify "<login>" and "<password>"
  Then app return validation error
  Examples:
    | login                  | password       | Comment
    | val:                   | val:qwasd46gun  | # empty login, correct password
    | val:test_user@gmail        | val:123456     | # correct login, incorrect password
    | val:test_user@gmail.com    | val:           | # correct login, empty password
    | val:@gmail.com             | val:qwasd46gun  | # login with no email name, correct password
    | val:test_usergmailgmail.com | val:qwasd46gun  | # login with no @, correct password
    | val:test_user@.com         | val:qwasd46gun  | # login with incomplete domain name, correct
password

 @fail
 @severity=blocker
 Scenario Outline: Log In by not registered user
  When User specify <login> and <password>
  Then app return authorization error
  Examples:
    | login                      | password        |
    | "val:unregistered_user@gmail"   | "val:123456"     |
```

Пример более объемного тестового сценария для регрессионного тестирования приведен в приложении А. Ниже представлен код теста, написанный на основе сценария Authentication на языке Java8:

```
@Slf4j
public class LoginSteps extends ASteps {

 @Given("start screen")
```

```
public void startScreen() {
    getUrl(StorageUtils.getStr("aconf:ui.url"));
}

@Then("app return validation error")
public void appReturnValidationError() {
  try {
    createResponseStorageAttachment(
        auth2(user.get("login"), user.get("password"))
    );
  } catch (IOException e) {
    createExceptionStackTraceAttachment(e);
  }
}

@Then("authorization is success")
public void authorizationIsSuccess() {
  try {
    createResponseStorageAttachment(
        authCheck(user.get("login"), user.get("password"))
    );
  } catch (IOException e) {
    createExceptionStackTraceAttachment(e);
  }
}

@Then("app return authorization error")
public void appReturnAuthorizationError() {
  try {
    createResponseStorageAttachment(
        auth(user.get("login"), user.get("password"))
    );
  } catch (IOException e) {
    createExceptionStackTraceAttachment(e);
  }
```

```
  }

  @When("User specify {string} and {string}")
  public void userSpecifyCorrectLoginAndPassword(String login, String password) {
    user = ImmutableMap.of(
        "login", StorageUtils.getStr(login),
        "password", StorageUtils.getStr(password)
    );
    createMapAsJsonAttachmentWithName(user, "Current User");
  }

  @And("user {string} registered in app")
  public void userRegisteredInApp(String user) {
    Map userModel = ImmutableMap.of(
        "login", StorageUtils.getStr(user)
    );
    createMapAsJsonAttachmentWithName(userModel, "Verified User");
  }
}
```

3.2.4 Результаты применения и оценка эффективности разработанной методики

Результаты выполнения автоматизированного тестирования МП, полученные при апробации разработанной методики, представлены в таблице 4.

Таблица 4 – Отчет о выполнении автоматизированного тестирования

Вид проверки	Описание проверки	Результат тестирования
Тестирование GUI	Проверка наличия новых элементов графического интерфейса и их функционирования	Соответствует спецификации

Продолжение таблицы 4

Вид проверки	Описание проверки	Результат тестирования

Тестирование GUI	Проверка работоспособности новых элементов GUI на разных устройствах	Обнаружены незначительные дефекты на некоторых комбинациях устройство + платформа + версия ОС
Тестирование API	Проверка API-части новой функциональности на соответствие спецификации	Соответствует спецификации
	Регрессионная проверка ранее созданных функциональностей, затронутых при добавлении новой функциональности	Обнаружены незначительные дефекты
	Тестирование безопасности передачи данных при отправке новых запросов	Обнаружены незначительные дефекты, связанные с нарушением ролевой модели

Применение методики позволило

– запустить автотесты GUI на всех тестовых устройствах и благодаря этому обнаружить дефекты, специфичные для конкретных девайсов и версий ОС;

– выполнить тестирование API с использованием комбинаторных данных, что помогло оперативно выявить проблемы при прохождении менее частотных сценариев, связанные с добавлением функциональности.

Для оценки эффективности примененной методики было сделано сравнение показателей тестовых метрик, полученных до и после апробации разработанной методики. В результате проведенного анализа были выявлены следующие изменения:

– тестовое покрытие увеличилось на 25 % для всех тестовых устройств и на 60 % для каждого конкретного устройства;

– количество не пройденных тест-кейсов уменьшилось с 827 до 307 для всех девайсов и с 1938 до 704 для отдельных девайсов;

– количество обнаруженных ошибок увеличилось в 6 раз – с 12 до 72. Но рост произошел в основном за счет дефектов с незначительным и тривиальным приоритетом. Их число составило 35 и 26 соответственно.

Такая динамика указывает на то, что сценарии критического пути были покрыты до внедрения автоматизации. Применение разработанной методики позволило увеличить тестовое покрытие за счет запуска тестов на всех тестовых устройствах и выполнения менее частотных сценариев, которые при ручном тестировании не проверялись или проверялись не полностью.

На рисунке 20 представлена диаграмма, иллюстрирующая стартовые и итоговые показатели метрики «тестовое покрытие» для всего набора тестовых устройств и для каждого отдельного устройства.

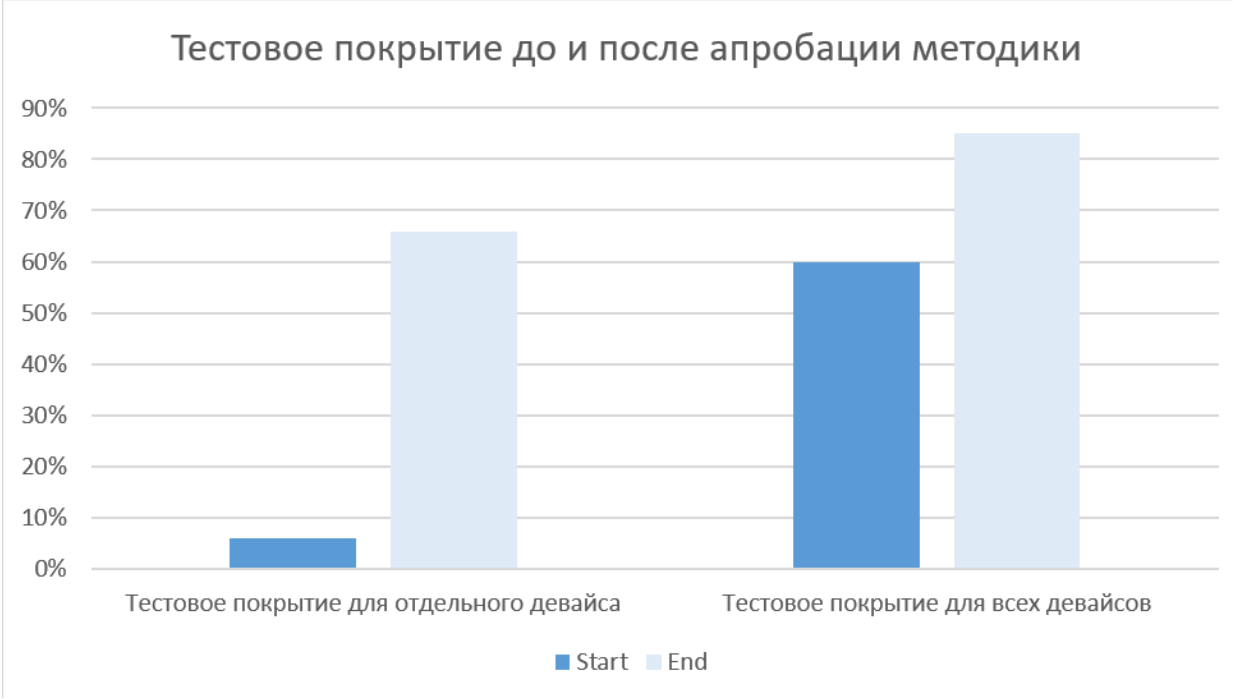

Рисунок 20 – Тестовое покрытие до и после применения методики

На рисунке 21 представлен фрагмент отчета о выполнении всех подготовленных наборов автотестов API после починки обнаруженных дефектов, сгенерированный с помощью фреймворка Allure.

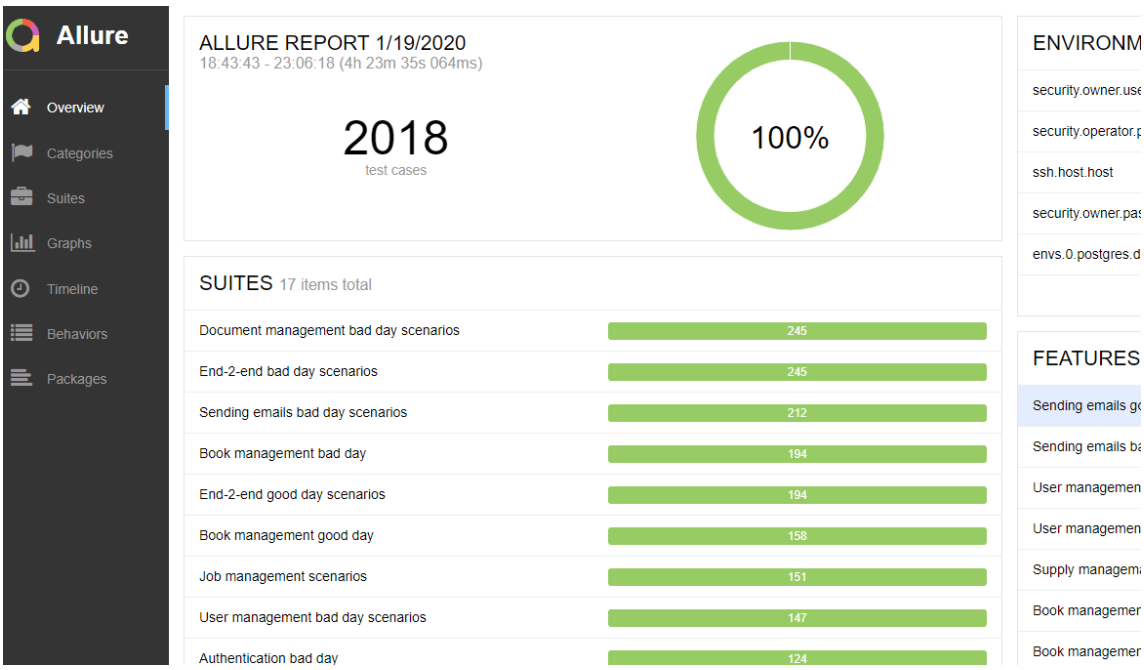

Рисунок 21 – Фрагмент отчета о выполнении автотестов API

На рисунке 22 показан фрагмент отчета о выполнении конкретного автотеста.

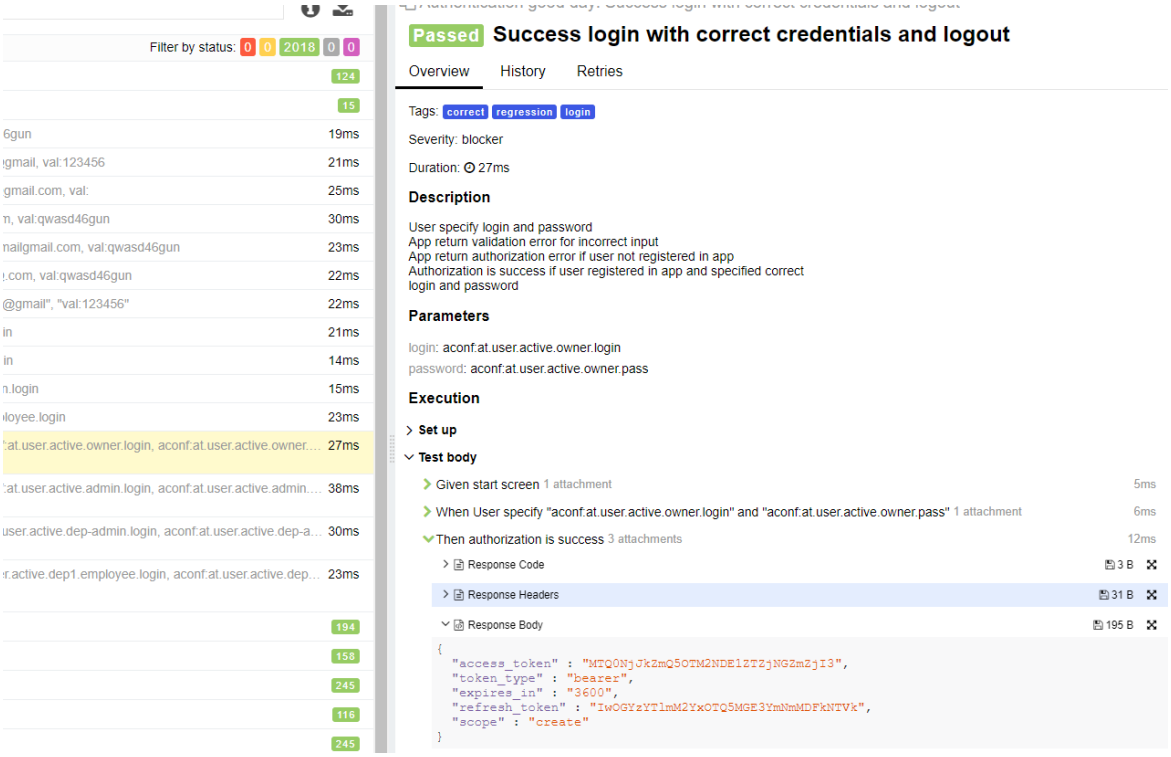

Рисунок 22 – Отчет о выполнении автотеста «авторизация в МП»

Характер и качество изменений доказывают эффективность примененной методики автоматизации тестирования. Внедрение автотестов позволило проверять большинство сценарных ветвлений в течение двухнедельной Scrum-итерации.

Таким образом, апробацию методики можно считать успешной.

Выводы по главе 3

1. Для повышения эффективности тестирования МП на Agile-проектах должна использоваться методика автоматизации, разработанная с учетом специфики мобильных приложений.

2. Предлагаемая методика автоматизации тестирования включает в себя тестирование GUI с применением методики Record and Play и тестирование API с применением методики Scripting.

3. Для автоматизации процесса тестирования МП используются специализированные средства автоматизации: встроенные в платформенную среду разработки драйверы для записи автотестов Record and Play, а также фреймворк, реализующий подход BDD.

4. Для проведения автоматизированного тестирования API подготовлены тестовые сценарии, написанные на предметно-ориентированном языке Gherkin.

5. На основе предлагаемой методики проведено тестирование GUI и API мобильного приложения с применением инструментов автоматизации, что повышает эффективность процесса благодаря увеличению тестового покрытия.

Заключение

Целью диссертационной работы является исследование и разработка методики автоматизации тестирования МП, которые проектируются по технологии Agile.

При проведении исследования получены следующие основные выводы и результаты:

1. Произведен анализ источников по предмету исследования, который подтвердил недостаточность работ, посвященных автоматизации тестирования МП на Agile-проектах, что подтвердило актуальность темы исследования

2. Произведен анализ методики Scripting, применяемой для автоматизации тестирования на Agile-проектах, который показал, что ее недостаточно для полноценного тестирования МП;

3. Разработана и реализована методика автоматизации тестирования МП на Agile-проектах, основанная на применении двух разных методик для разных видов тестирования: методики Record and Play для тестирования графического интерфейса и Scripting для тестирования API;

Для внедрения методики были использованы специальные драйверы, встроенные в платформенные средства разработки Android Studio и Xcode, а также фреймворк Cucumber, интегрированный в среду разработки IntelleJ IDEA.

4. Для оценки эффективности разработанной методики сделано сравнение показателей тестовых метрик, полученных при тестировании МП до и после применения методики автоматизации.

Как показал анализ полученных результатов, внедрение методики позволило увеличить тестовое покрытие МП на 25 % для всех тестовых устройств и на 60 % для каждого конкретного устройства, а также

увеличение числа найденных дефектов с 12 до 72 за счет большего количества дефектов с незначительным приоритетом.

Полученные изменения указывают на повышение эффективности процесса тестирования МП при применении разработанной методики.

Таким образом, в работе решена актуальная научно-практическая проблема разработки методики автоматизации тестирования МП, проектируемых по технологии Agile.

Гипотеза исследования подтверждена, поставленная цель достигнута.

Список используемой литературы и используемых источников

1. Абдулназаров, Ф.М. Тестирование мобильных приложений, функционирующих на Android / Ф.М. Абдулназаров, О.А. Анарбеков // Современные технологии поддержки принятия решений в экономике: Сборник трудов III Всероссийской научно-практической конференции студентов, аспирантов и молодых ученых. Национальный исследовательский Томский политехнический университет, Юргинский технологический институт; под ред. А.А. Захаровой. – 2016. – С. 196–198.

2. Автоматизация тестирования [Электронный ресурс]. URL: https://ru.wikipedia.org/wiki/Автоматизированное_тестирование (дата обращения: 10.05.2020).

3. Автоматизация тестирования и Agile [Электронный ресурс]. URL: https://habr.com/ru/company/otus/blog/351104 (дата обращения: 10.05.2020).

4. Автоматизированное тестирование программного обеспечения – основные понятия [Электронный ресурс] URL: http://www.protesting.ru/automation/ (дата обращения: 10.05.2020).

5. Атисков, А.Ю. Тестирование эргономики пользовательского интерфейса мобильных приложений / А.Ю. Атисков, И.И. Давидович // Научный вестник НГТУ, том 57. – 2014. – № 4. – С. 119–130.

6. Винниченко, И.В. Автоматизация процессов тестирования. – Спб.: Питер, 2005. – 203 с.

7. Виды тестирования [Электронный ресурс]. URL: http://www.protesting.ru/testing/testtypes.html (дата обращения: 10.05.2020).

8. Градусов, А.Б. Сравнительный анализ современных программных средств автоматизированного тестирования мобильных приложений / А.Б. Градусов, И.И. Хмеляр // Постулат. – 2018. – № 6 (32). – С. 38–43.

9. Дастин, Э. Автоматизированное тестирование программного обеспечения. Внедрение, управление и эксплуатация: Пер. с англ. / Э. Дастин, Дж. Рэшка, Дж. Пол. – М.: из-во «Лори». – 2003, 289 с.

10. Зайцев, Е.Д. К вопросу об эффективности автоматизации тестирования web-, desktop- и мобильных приложений / Е.Д. Зайцев, Д.М. Зайцев // Проблемы инфокоммуникаций. – 2018. – № 2 (8). – С. 56–63.

11. Калберстон, Р. Быстрое тестирование: Пер. с англ./ Роберт Калберстон, Крис Браун, Гэри Кобб. – Издательский дом «Вильямс», 2002. – 384 с.

12. Калын, М.М. Автоматизация тестирования: постановка целей, плюсы и минусы // Синергия наук. – 2017. – Том 1. – С. 45–51.

13. Канер, С. Тестирование программного обеспечения. Фундаментальные концепции менеджмента бизнес-приложений: Пер. с англ./ Сэм Канер, Джек Фолк, Енг Кек Нгуен. – К.: Издательство «ДиаСофт», 2001. – 544 с.

14. Криспин, Л. Гибкое тестирование. Практическое руководство для тестировщиков ПО и гибких команд / Криспин Лайза, Грегори Джанет. – М: Изд-во Вильямс, 2010. – 464 с.

15. Куликов, С.С. Тестирование программного обеспечения. – Минск: Четыре четверти, 2015. – 294 с.

16. Майерс, Г. Искусство тестирования программ: Пер. с англ. под ред. Б.А. Позина. – М.: Финансы и статистика, 1982. – 176 с.

17. Материалы ISTQB [Электронный ресурс]. URL: https://www.rstqb.org/ru/istqb-downloads.html (дата обращения: 10.05.2020).

18. Мобильное приложение [Электронный ресурс]. URL: https://ru.wikipedia.org/wiki/Мобильное_приложение (дата обращения: 10.05.2020).

19. Мун, Д.Е. Проблемы тестирования мобильных игр на примере «Pokémon GO» / Д.Е. Мун, И.О. Семенов // E-SCIO. – 2019. – № 5 (32). – С. 629-633.

20. Наталуха, А. Чек-лист начинающих Android- и iOS тестировщиков // Tester’s life. – 2013. – № 2. – С. 14–15.

21. Путеводитель по инструментам автотестирования мобильных приложений [Электронный ресурс]. URL: https://habr.com/ru/company/badoo/blog/347986/ (дата обращения: 10.05.2020).

22. Савин, Д.А. Обзор критериев тестового покрытия для мобильных web-приложений / Д.А. Савин // Альманах научных работ молодых ученых университета ИТМО в 5 томах. – СПб: Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики, 2016. – С. 256–258.

23. Савин, Р. Тестирование Дот Ком, или Пособие по жестокому обращению с багами в интернет-стартапах / – М.: Дело, 2007. – 312 с.

24. Салапаев, А.И Автоматизация тестирования: для чего? // Прогрессивные технологии и экономика в машиностроении. – 2018. – С. 68–70.

25. Сергеева, А.И. Особенности организации тестирования мобильных приложений. Общий план тестирования / А.И. Сергеева // Системный администратор: Издательский дом «Положевец и партнеры» (Москва). – 2016. – № 10 (167). – С. 65–69.

26. Соколов, С.А. Особенности при тестировании мобильных приложений / С.А. Соколов, А.В. Шутов // Постулат. – 2018. – № 6 (32). – С. 122–125.

27. Хозя, А. Тестирование мобильных приложений: tips & tricks [Электронный ресурс]. URL: https://habr.com/company/badoo/blog/269189/ (дата обращения: 10.05.2020).

28. Шлыков, К. Особенности тестирования мобильных приложений в целом [Электронный ресурс]. URL: http://www.enterra.ru/blog/mobile_qa/ (дата обращения: 10.05.2020).

29. Agile Manifesto (русскоязычная версия) [Электронный ресурс]. URL: http://agilemanifesto.org/iso/ru/principles.html (дата обращения: 10.05.2020).

30. Bach, J. Lessons Learned in Software Testing / Cem Kaner, James Bach, Bret Pettichord. – Wiley, 2001.

31. BDD [Электронный ресурс]. URL: https://ru.wikipedia.org/wiki/BDD_(программирование) (дата обращения: 10.05.2020).

32. Bourque, P. SWEBOOK v 3.0 Guide to the Software Engineering Body of knowledge / Pierre Bourque, Richard E. (Dick) Fairley– IEEE Computer Society Products and Service, 2014.

33. Copeland, L. A Practitioner's Guide to Software Test Design: Artech House Publishers, 2003.

34. Cucumber [Электронный ресурс]. URL: https://cucumber.netlify.app/docs/cucumber/ (дата обращения: 10.05.2020).

35. Davis, C. Agile Metrics in Action: Manning Publication, 2015.

36. Gherkin Reference [Электронный ресурс]. URL: https://cucumber.netlify.app/docs/gherkin/reference/ (дата обращения: 10.05.2020).

37. Pashuk, Alesia Android app testing specifics [Электронный ресурс]. URL: https://www.scnsoft.com/blog/android-app-testing-specifics (дата обращения: 10.05.2020).

38. Pashuk, Alesia Mobile testing process: How to make it efficient. [Электронный ресурс]. URL: https://www.scnsoft.com/blog/mobile-testing-process-how-to-make-it-efficient (дата обращения: 10.05.2020).

39. Scrum [Электронный ресурс]. URL: https://ru.wikipedia.org/wiki/SCRUM (дата обращения: 10.05.2020).

40. Smart, J.F. BDD in action Behavior-Driven Development for the whole software lifecycle: Manning Publication, 2015.

Приложение А

Тестовый сценарий для проверки существующей функциональности

Сценарий для проверки функциональности «Добавление поставки» (Add supply):

```
@regression
Feature: Supply management good day
  As a department administrator I want to create supply from publishing office

  Scenario: Prepare user for supply
    And user "aconf:at.user.active.dep-admin.login" registered in app

  Scenario Outline:
    Given book with "<name>" created in catalog
    Examples:
      | name                                  |
      | aconf:at.supply-view.create-form.values.book1 |
      | aconf:at.supply-view.create-form.values.book2 |
      | aconf:at.supply-view.create-form.values.book3 |

  @severity=blocker
  Scenario: Successfully login with correct credentials
    Given success previous scenario
    When User specify "aconf:at.user.active.dep-admin.login" and "aconf:at.user.active.dep-admin.pass"
    Then authorization is success

  @severity=blocker
  Scenario: Successfully push button "Create supply"
    Given success previous scenario
    And User watch "supplying" view "aconf:at.supply-view.view"
    When User push "Create supply" button "aconf:at.supply-view.create"
    Then User can see "Create supply" form "aconf:at.supply-view.create-form"
    Then request "create-form" is success

  @severity=blocker
  Scenario: Successfully creation supply in draft status
    Given success previous scenario
    When User fills "publishing office" field as "aconf:at.supply-view.create-form.values.p-office"
    And User add book "aconf:at.supply-view.create-form.values.book1"
    And User add book "aconf:at.supply-view.create-form.values.book2"
    And User add book "aconf:at.supply-view.create-form.values.book3"
    And User fills "Description" field as "val:Some test description"
    And User push "Save draft" button "aconf:at.supply-view.create-form.save"
    Then request "save" is success

  @severity=blocker
```

```
Scenario: Successfully view supply in draft status
  Given success previous scenario
  When User open "supply" object "store:createdSupply"
  Then field "publishing office" is equal to "aconf:at.supply-view.create-form.values.p-office"
  And supply contains book "aconf:at.supply-view.create-form.values.book1"
  And supply contains book "aconf:at.supply-view.create-form.values.book2"
  And supply contains book "aconf:at.supply-view.create-form.values.book3"
  And field "Description" is equal to "val:Some test description"
  And supply has status "DRAFT"

@severity=blocker
Scenario: Successfully editing supply in draft status
  Given success previous scenario
  And User open "supply" object "store:createdSupply"
  When User remove book "aconf:at.supply-view.edit-form.values.book2" from supply
  And User fills "Description" field as "val:Some another test description"
  And User push "Save draft" button "aconf:at.supply-view.create-form.save"
  Then request "save" is success

@severity=blocker
Scenario: Successfully view supply in draft status
  Given success previous scenario
  When User open "supply" object "store:createdSupply"
  Then field "publishing office" is equal to "aconf:at.supply-view.edit-form.values.p-office"
  And supply doesn't contain book "aconf:at.supply-view.create-form.values.book2"
  And field "Description" is equal to "val:Some another test description"
  And supply has status "DRAFT"

@severity=blocker
Scenario: Successfully send supply email to publishing office
  Given success "Successfully creation supply in draft status" scenario
  When User open "supply" object "store:createdSupply"
  And User push "Sent to publishing office" button "aconf:at.supply-view.edit-form.publish"
  Then supply has status "SENT"
  And publishing office receive email from "aconf:at.notifier.email"
  And email contains "book" "aconf:at.supply-view.create-form.values.book1"
  And email contains "book" "aconf:at.supply-view.create-form.values.book3"
  And email contains "description" "val:Some another test description"

@severity=blocker
Scenario: Successfully logout
  Given success "Successfully login with correct credentials" scenario
  And User push "logout" button "aconf:at.general-view.logout"
  Then request "logout" is success
```