ВЫЧИСЛИТЕЛЬНЫЕ СИСТЕМЫ И СЕТИ

УДК 004.75

К ВОПРОСУ ОБ ЭФФЕКТИВНОСТИ АВТОМАТИЗАЦИИ ТЕСТИРОВАНИЯ WEB-, DESKTOP- И МОБИЛЬНЫХ ПРИЛОЖЕНИЙ

Е.Д. ЗАЙЦЕВ[1], Д.М. ЗАЙЦЕВ[2]

[1]*Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники», ул. П. Бровки, 6, Минск, 220113, Беларусь*

[2]*Учреждение образования «Белорусская государственная академия связи», ул. Ф. Скорины, 8/2, Минск, 220114, Беларусь*

Поступила в редакцию 6 октября 2018

В статье раскрываются вопросы эффективности автоматизации тестирования web-, desktop- и мобильных приложений на проектах различной продолжительности. Приведено описание разных методологий разработки и необходимости автоматизации при их использовании. Выполнен обзор основных плюсов и минусов функционального и автоматического тестирования приложений, даны советы по выбору тестовой стратегии в различных ситуациях.

Ключевые слова: тестирование, автоматизация тестирования, программирование, разработка ПО.

Введение

В современном мире все больше внимания уделяется автоматизации различных процессов с целью перекладывания рутинных задач на компьютеры, уменьшения денежных затрат на персонал, исключения фактора человеческой ошибки. Но не все специалисты даже высокого уровня ясно представляют себе плюсы и минусы автоматизации как в общем, так и для конкретного проекта в частности. Все это обусловливает актуальность выбранной темы.

Ярким примером рутинной задачи является тестирование приложений, как web-, так и desktop- и мобильных. Если разработчики пишут 10 модулей для приложения в неделю, то за первые 5 рабочих дней тестировщику необходимо проверить 10 модулей. За следующие 5 дней – уже 20 модулей, потому что следует убедиться, что, во-первых, новые 10 модулей работают правильно, а во-вторых, что они не «сломали» ничего из уже существующего функционала. В самом начале у тестировщика не так много работы, но с каждой неделей ее объем растет, и в какой-то момент он и руководство приходят к выводу, что справиться с данным объемом работы в отведенные сроки невозможно. В таком случае вывод напрашивается сам собой: либо нанять дополнительных тестировщиков, либо автоматизировать тестирование на проекте. Именно об автоматизации тестирования и пойдет речь в данной статье. Попробуем ответить на вопрос, действительно ли она избавляет от всех проблем на проекте и должна ли быть применена повсеместно.

Большое количество руководителей проектов полагает, что автоматическое тестирование – это безусловное благо. Если разработчики и/или автоматизаторы тестирования пишут тесты – это хорошо, а лучшее качество достигается большим количеством тестов. В реальном же мире тестирование чаще проводится вручную. Такое положение дел не всегда связано с некомпетентностью, глупостью или банальной ленью разработчиков. Естественно, по сравнению с функциональным тестированием, автоматизированное имеет как достоинства, так и явные недостатки, речь о которых пойдет ниже.

Теоретический анализ

В начале 2018 года в компании EPAM Systems была собрана статистика по нескольким тысячам активных и завершенных проектов разной продолжительности с целью определить, как часто применяется автоматизация тестирования и насколько она оправдывает себя в зависимости от продолжительности проекта. Результаты данного исследования представлены в табл. 1 (проанализированы UI, API, DB, End-to-End, сценарные и подобные тесты, но не учитываются Unit- и интеграционные тесты – на подавляющем большинстве проектов они есть и пишутся разработчиками вместе с функционалом для внутреннего контроля качества).

Таблица 1. Статистика наличия и эффективности автоматизации тестирования на проектах в зависимости от их продолжительности

Тип проекта	Продолжительность	Количество проектов	Процент использования автоматизации	Эффективность автоматизации
Короткий (в основном мобильная разработка)	до 0,5 лет	1281	9,2	низкая
Средний (в основном разработка web- или desktop-приложения)	0,5–2 года	3497	54,4	средняя, сильно зависит от проекта и способа разработки
Длительный (в основном поддержка существующего проекта)	от 2 лет	2863	86,9	высокая

Как видно из содержания таблицы, для краткосрочных проектов (в основном это мобильные приложения, так как они не обладают большим функционалом), как правило, автоматизация не используется. В подобных случаях имеет смысл покрывать автотестами только самый критичный функционал (например, финансовые операции, осуществляемые с помощью мобильных приложений), потому что от стадии, когда структура UI станет стабильной, до момента сдачи проекта проходит очень мало времени, чтобы разработка системы для автоматизации тестирования как минимум окупилась, как максимум – принесла экономическую выгоду.

В проектах средней продолжительности, в основном это разработка web- или desktop-приложения, все не так однозначно. Если используется каскадная модель разработки ПО (старт = определение требований → проектирование → реализация → тестирование → сдача проекта → поддержка (при необходимости) = финиш (рис. 1), то в автоматизации нет необходимости и практического смысла, так как тестирование проводится один раз (в случае, когда поддержка проекта не требуется) после разработки, и гораздо выгоднее направить больше сил на ручное тестирование. Такие модели используются чаще всего в военной сфере и медицине, когда все требования известны заранее и не меняются в процессе разработки. Тестирование в таком случае необходимо гораздо более качественное, а если использовать автоматизацию, то увеличение кода приведет к увеличению количества багов, что является критичным в подобных сферах. Как и в случае с короткими проектами, есть смысл «покрыть» тестами самый критичный функционал, в котором человек может допустить очень «дорогую» ошибку [1].

С другой стороны, если используются гибкие методологии разработки (например, Scrum) или проект модульный, то автоматизация позволяет сократить затраты и разгрузить ручных тестировщиков. Суть Scrum (общий пул задач → выбор задач на итерацию → разработка, тестирование, покрытие автотестами выбранного функционала → анализ итерации → выбор задач на итерацию и так далее по кругу (рис. 2) заключается в том, что вся разработка разделена на промежутки времени равной длины – итерации (обычно 1–4 недели). Из общего пула задач выбирается столько, сколько команда способна сделать за итерацию, основываясь на своих ощущениях, предыдущих итерациях, предполагаемых отпусках сотрудников и проектных рисках, и фиксируется, запрещая вносить изменения. Это позволяет избежать лишних рисков, связанных с недостаточным количеством времени на разработку или

тестирование, и, как следствие, переноса невыполненной работы в следующую итерацию. Это также способствует тому, чтобы готовый функционал отдавался не большими частями и редко, а маленькими кусочками в конце каждой итерации, что позволяет заказчику в режиме реального времени оценивать соответствие продукта его требованиям и при необходимости вносить корректировки на будущее. Как следует из вышеизложенного, сам пул задач можно изменять, адаптируясь к новым требованиям, поэтому методология и называется гибкой. В таком случае в рамках одной итерации происходят минорные изменения функционала, что позволяет автотестам отслеживать стабильность системы в связи с доработками без необходимости писать их с нуля или полностью переделывать. С модульными системами примерно такая же ситуация. Каждый модуль – отдельная подсистема, слабо зависящая от остальных, что позволяет «покрыть» их автотестами независимо друг от друга и достичь лучших результатов путем распараллеливания задач, а после того как между двумя модулями появляются связи, дополнительно написать тесты на их взаимодействие. Поэтому каждый отдельно взятый модуль является практически неизменным и стабильным, что позволяет сократить время, затрачиваемое на поддержку автоматизации на данном проекте [2, 3].

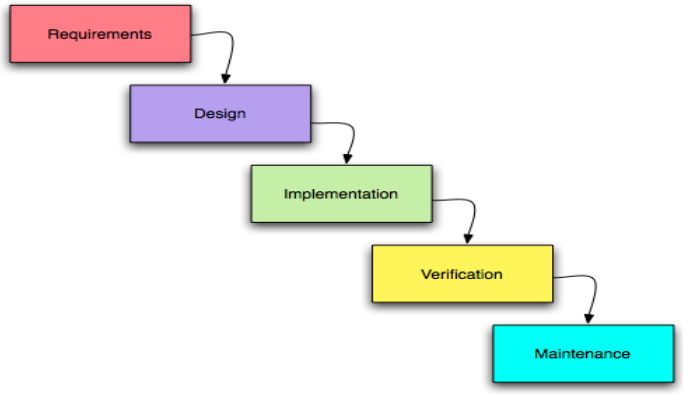

Рис. 1. Каскадная методология разработки (Waterfall)

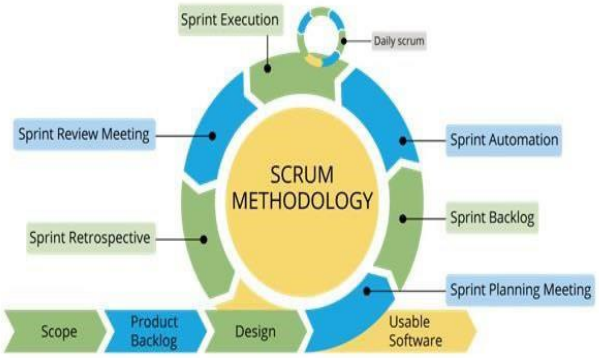

Рис. 2. Гибкая методология разработки (Scrum)

Проекты с наибольшей продолжительностью чаще всего представляют собой поддержку или изменение и доработку существующего функционала. Их высокая степень автоматизации обуславливается глобальной неизменностью функционала, статичностью как кода, так и структуры приложения. Это позволяет «покрыть» существующий функционал автотестами и переложить на них, например, регрессионное тестирование, чтобы максимально быстро обнаруживать баги в измененном коде и отправлять его на доработку. Однако несложно заметить, что высокая степень автоматизации не достигает ста процентов. Как показывает

практика, не имеет смысла автоматизировать сценарии, которыми пользуются крайне редко – это позволяет найти разумный компромисс между стоимостью (которая в большинстве случаев высока) и процентом «покрытия» тестами.

Результаты и их обсуждение

Следующие далее утверждения были получены и подтверждены на основе анализа различных наблюдений и исследований в IT-сфере, в частности в компаниях EPAM Systems, A1QA и Wrike [4, 5, 6].

Преимущества автоматических тестов

Автотесты имеют ряд достоинств в сравнении с функциональным тестированием.

1. Стоимость проведения теста – автоматические тесты в подавляющем большинстве случаев быстрее и дешевле проверки человеком (только выполнение сценария, не разработка) на несколько порядков, и чем больше тестов и количество их запусков, тем существеннее этот разрыв. Именно разработка автотестов, конечно, дороже разового ручного тестирования.

2. Постоянная готовность к работе – автотесты можно запускать бесчисленное количество раз и в любой последовательности, у них неограниченный рабочий день, они не устают и не болеют (исключая проблемы аппаратной части, но это не вина автотестов).

3. Наличие автотестов значительно ускоряет добавление нового функционала и рефакторинг кода – вместо того чтобы тратить время на попытку продумать и оценить последствия изменения кода, можно его просто изменить и запустить тесты, и если они завершаются успешно, то с высокой долей вероятности эти изменения ничего не нарушили в приложении.

4. Быстродействие – автотесты выполняются гораздо быстрее, чем те же проверки, выполненные вручную. В реальности бывают и относительно медленные тесты, но их количество стараются минимизировать, так что можно назвать исключением, подтверждающим правило.

5. Лаконичность – если тест завершился неудачно и система логирования настроена грамотно, то разработчик быстрее может понять, где произошла ошибка, при каких условиях и значениях, какие шаги были выполнены до этого. Машина не забывает описать какую-то часть своего пути, не путает введенные значения.

6. Точность – компьютер выполняет именно то, что ему укажет пользователь или разработчик, а значит, автотесты проверяют именно то, что описано в сценарии, во всех деталях и мелочах. Они не отвлекаются, не путают и не забывают.

7. Тестирование, неподвластное человеку – некоторые виды тестирования очень сложно либо невозможно выполнить одному или группе функциональных тестировщиков. Например, нагрузочное тестирование, которое симулирует миллионы запросов, или unit-тестирование, в котором каждый тест отвечает за атомарную часть приложения, который конечный пользователь не может вызвать в отрыве от всего остального.

8. «Покрытие» приложения – автотесты позволяют «покрыть» огромное количество сценариев и находить проблемы в наиболее нетривиальных функциях приложения или сценариях, до которых тестировщик никогда бы не «добрался», кроме как случайно.

9. Упрощение тестирования при смене окружения – если структура приложения не меняется и основные элементы приложения, за которые и «цепляются» автотесты, остаются такими же, то кросс-платформенное тестирование – тестирование в разных операционных системах, с другими версиями библиотек – будет гораздо быстрее. А если меняется только back-end часть приложения, то чаще всего изменения невозможно корректно протестировать «руками».

Преимущества функционального тестирования

Ручное тестирование тем не менее превосходит автоматизированное по многим аспектам.

1. Разовая скорость – протестировать какой-то модуль вручную в первый раз легко и быстро. Автоматический тест нужно прежде разработать. Это всегда медленнее разовой ручной проверки.

2. Тестирование внешнего вида – автотест не знает, что вот эта «кнопка» должна быть чуть зеленее, а чекбокс выходит на 2 пикселя влево и очень режет глаз. В то же время человек сразу это заметит и создаст баг, чтобы разработчики исправили эту проблему.

3. Гибкость – ручное тестирование можно проводить разнообразнее, и изменение способа тестирования в рамках неизменного или практически неизменного UI практически ничего не стоит. Протестировать в Safari – пожалуйста, на Chromebook – нет проблем, IE6 – придется запустить виртуальную машину, но также вполне возможно.

4. Адаптируемость – умение быстро подстраиваться под изменения. Когда внешний вид, логика и/или структура продукта резко изменяются и все начинает работать совсем не так, как раньше, ручные тестировщики легко могут забыть, как они тестировали прежде. Они просто будут тестировать то, что есть сейчас. Автоматические тесты в той же ситуации потеряются в новом приложении, что влечет за собой множество ошибок, которые необходимо будет исправить, изменение логики тестов, проверок и «якорей», прежде чем двигаться дальше.

5. Отступление от изначального маршрута – ручное тестирование позволяет найти проблемы, которые находятся за пределами тестируемого сценария, при необычной последовательности действий или нажатии другой кнопки.

6. Осмысленность – хотя каждая вещь в отдельности может быть абсолютно корректна, люди гораздо легче понимают, что вместе эти вещи не имеют никакого смысла (в большинстве случаев относится к unit-тестированию, при котором взаимодействие модулей не проверяется).

7. Предоставление вспомогательных данных – недостаточно сказать: «Сценарий не пройден!» – важно правильно объяснить, в чем заключается проблема, и уметь ответить на дополнительные вопросы разработчика. При отсутствии очень подробной системы логирования, которая в таком случае будет иметь в отчете много «мусора», функциональные тестировщики смогут сделать это лучше компьютера.

8. «Человечность» – для большинства продуктов конечным пользователем является человек, по этой причине доверять его тестирование лучше человеку, ведь он сможет дать качественный отзыв по поводу простоты и удобства его использования.

Если сравнивать преимущества ручного и автоматизированного тестирования, то можно найти пересекающиеся пункты. Это не ошибка, так как в зависимости от того, как хорошо организовано автоматическое и/или функциональное тестирование, эти аспекты могут являться как плюсами, так и минусами для каждой сферы. Слабая автоматизация при сильных функциональных тестировщиках – минус для первых и плюс для вторых. Если отдел тестирования набран из людей без соответствующего технического образования, которые с компьютером на «вы» и без особого взаимопонимания с разработчиками, – минус для отдела функционального тестирования.

Тестирование парадоксально по своей природе, особенно когда речь идет о полном тестировании. На самом деле в этом случае говорят о *достаточном* тестировании, потому что затраты на тестирование растут экспоненциально. За день можно протестировать приложение, условно, на 90 %, за неделю – на 95 %, за месяц – на 99 %, за год – на 99,9 %. В большинстве случаев такие затраты нецелесообразны, и на проекте устанавливается некий допустимый уровень, при котором тестирование можно считать успешным.

Гораздо важнее **устранять проблемы быстро**, чтобы добиться более высокого качества приложения, чем вкладывать все ресурсы в тестирование. В таком случае и клиент будет доволен хорошей обратной связью, и денег будет потрачено меньше. Важно найти баланс.

Если разрабатывать нужно быстро, при этом приложение меняется предсказуемо и локально, то **функциональное тестирование лучше автоматизации**, так как проверить несколько сценариев человеку проще и быстрее, чем автоматизировать их.

Исходя из вышесказанного, **автоматизация крайне неэффективна на коротких дистанциях**, потому что к тому моменту, как все будет настроено и сценарии будут переведены в код, разработка может уже закончиться, и автоматизация тестирования будет являться убыточной на проекте (рис. 3). Стоит учесть, что на графике изображена стоимость в текущий момент времени, так что момент, когда общие затраты (площадь под графиком) уравняются, находится правее точки пересечения.

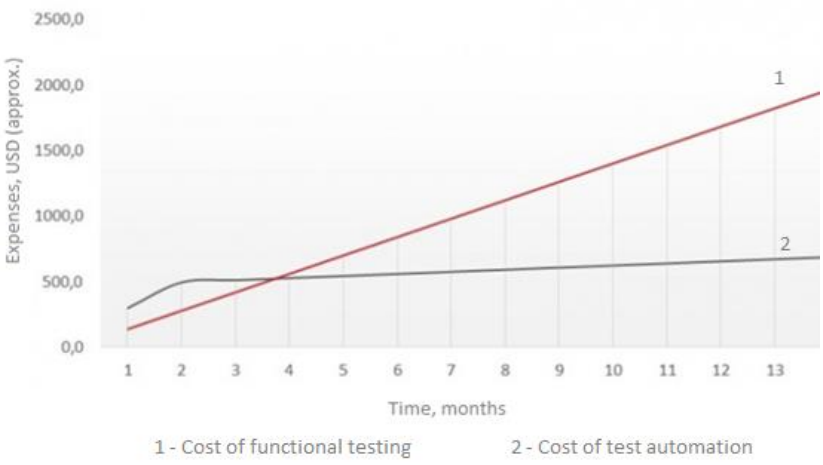

Рис. 3. Соотношение временных и денежных затрат функционального и автоматизированного тестирования

Обратная ситуация, когда проект является долгосрочным. Для поддержания качества кода необходим рефакторинг, в ходе которого, теоретически, разработчик может «сломать» какой-то компонент. И чем глобальнее рефакторинг, тем выше шанс. А это значит, что тестировщикам необходимо провести регрессионное тестирование как минимум модуля, код которого менялся, как максимум – всего приложения. Регрессия занимает довольно продолжительное время, а параллельно добавляется новый функционал, который также необходимо протестировать. Поэтому **для обеспечения стабильного базового уровня качества приложения в долгосрочных проектах выгодно производить автоматизацию тестирования** [4, 5].

Недостатки автоматизации тестирования

Проблема: в приложении происходит глобальный рефакторинг, разделенный на множество пересекающихся сегментов. Необходимо постоянное тестирование, чтобы убедиться, что ничего не «сломалось» и можно продолжать рефакторинг.

Решения:

1. Нанять больше тестировщиков, чтобы они занимались регрессионным тестированием.

2. «Покрыть» приложение достаточным количеством автотестов, которые будут следить за работоспособностью приложения.

Первый вариант требует постоянных денежных затрат, связанных с расширением штата сотрудников. Второй вариант требует больших изначальных затрат, но в последствии они уменьшаются и остается только поддержка.

Однако автотесты не всегда являются решением, а иногда сами становятся проблемой.

Автоматизация тестирования – это разработка, требующая соответствующих ресурсов и квалификации. А если этим занимаются сами разработчики приложения, то в итоге это выливается в замедление разработки самого продукта. Причем замедление идет не на «незаметные» 15–20 %, а во вполне ощутимые 2–3–5 раз.

Тесты тоже можно написать плохо. Тест – такой же код, которому часто необходим рефакторинг, написанный такими же разработчиками. А, как известно, людям свойственно ошибаться. При плохо написанной системе логирования, по не прошедшему тесту сложно понять, где проблема, и необходимо тратить дополнительное время на поиск неисправности, которое можно было бы потратить на ее исправление. Иногда тест ничего не проверяет, в таком случае он бесполезен. Иногда не проходит в случае отсутствия ошибки, что влечет за собой выделение ресурсов на его доработку. Хуже, когда тест проходит при наличии ошибки. В таком случае ресурсы на разработку теста были потрачены впустую, и конечный пользователь получает некорректно работающее приложение.

«Быстрые» тесты на самом деле медленные. Каждый конкретный тест, конечно, выполняется очень быстро, особенно если речь идет о unit-тестировании, при котором локально тестируется атомарный кусочек кода. Допустим, для выполнения подобного теста необходимо 5 мс. Интеграционные тесты и API-тесты, хоть и медленнее unit-тестов, потому что требуют взаимодействия модулей, доступ к базе данных и подобного, также сравнительно быстрые, допустим, 1 секунда. Наиболее медленными являются UI-тесты, в среднем 30–60 секунд в зависимости от сценария, что все равно является очень хорошим результатом относительно функционального тестирования. Но, когда количество тестов хоть сколько-нибудь существенно, быстрые тесты оказываются очень даже медленными. Для больших систем необходимо огромное количество тестов – счет может идти на десятки или сотни тысяч. Даже если взять практически идеальный вариант распределения тестов (90 % unit-тестов, 7,5 % интеграционных/API-тестов и 2,5 % UI-тестов), то при 50 000 тестов на проекте мы получим $45\,000 \cdot 0{,}005 + 3750 \cdot 1 + 1250 \cdot 60 = 313\,500$ секунд ≈ 87 часов, и это только для одного полного запуска тестов. Поэтому возникает вопрос: а стоит ли постоянно запускать все тесты, тормозя таким образом разработку?

Миллион тестов не является миллионом тестов: 3000 тестов в сумме – на веб-интерфейс, API, бизнес-логику и unit-тесты – могут проверять примерно 1000 сценариев. Если сценарий UI-теста затрагивает 20 модулей, то он дублирует проверки минимум 20 unit-тестов, несколько интеграционных тестов, которые проверяют взаимодействие этих модулей, а также может в себе содержать более мелкие сценарии, на которые также уже написаны тесты.

И самым ироничным недостатком является то, что **автоматическая регрессия не отменяет «ручную»**, вопреки главной задумке. А все потому, что автоматические тесты проверяют далеко не все. Когда тестов много, сложно уследить, что конкретно проверяет каждый тест. Если есть непрошедший тест, то это означает, что в приложении баг. Если все тесты прошли – это не значит ничего. Также тесты не проверяют сценарий на наличие в нем здравого смысла, а делают то, что написал разработчик. А убедиться в том, что приложение не только рабочее, но и им удобно пользоваться, должен функциональный тестировщик. И такой тест не один, а большинство, но попадает ли конкретный тест в это «большинство» – неизвестно. Как результат – проверить вручную нужно все [4, 6].

Заключение

Как итог, можно выделить несколько основных выводов из статьи:

1. Автоматизация тестирования не является решением всех проблем на любых проектах, то есть так называемой «серебряной пули» не существует.
2. Как у автоматизированного, так и у функционального тестирования есть свои плюсы и минусы.
3. Для автоматизации тестирования необходимы квалифицированные кадры, обладающие как теоретическими, так и практическими навыками в данной сфере.
4. На целесообразность использования автоматизации тестирования влияет большое количество различных факторов: продолжительность проекта, бюджет, ROI (Return of Investments) и другие.
5. Менеджменту необходимо правильно расставлять приоритеты при выборе стратегии тестирования, а не слепо следовать за «трендами».

TO THE QUESTION ABOUT THE EFFICIENCY OF TEST AUTOMATION FOR WEB, DESKTOP AND MOBILE APPLICATIONS

E.D. ZAITSEV, D.M. ZAITSEV

Abstract

The article reveals questions about the efficiency of test automation for web, desktop and mobile applications. The description is given for different development methodologies and the need of automation in pair with them. Made a review of main pros and cons of functional and automation testing, given tips on choosing a test strategy in different situations.

Список литературы

1. Каскадная модель // Википедия, свободная энциклопедия [Электронный ресурс]. – Режим доступа : https://ru.wikipedia.org/wiki/Каскадная_модель/. – Дата доступа : 02.09.2018.

2. A brief Introduction to Scrum Methodology // MTC EduHub [Электронный ресурс]. – Режим доступа : https://mtceduhub.com/a-brief-introduction-to-scrum-methodology/. – Дата доступа : 02.09.2018.

3. Scrum // Википедия, свободная энциклопедия [Электронный ресурс]. – Режим доступа : https://ru.wikipedia.org/wiki/Scrum/. – Дата доступа : 02.09.2018.

4. Мамонов, Д. Блеск и нищета автоматизации тестирования [Электронный ресурс] / Д. Мамонов. – СПб. : Wrike, 2017. – Режим доступа : https://habr.com/company/wrike/blog/321290/. – Дата доступа : 02.09.2018.

5. Черняк, М. Оценка эффективности автоматизации тестирования [Электронный ресурс] / М. Черняк. – 2015. – Режим доступа : http://www.a1qa.ru/blog/otsenka-effektivnosti-avtomatizatsii-testirovaniya/. – Дата доступа : 02.09.2018.

6. Шульга, В. Автоматизированное тестирование – «убийца» ручного тестирования, модный тренд или серебряная пуля? [Электронный ресурс] / В. Шульга. – 2018. – Режим доступа : https://habr.com/company/epam_systems/blog/349270/. – Дата доступа : 02.09.2018.