Федеральное государственное автономное образовательное учреждение

высшего образования

"Российский университет транспорта" (МИИТ)

Кафедра «Цифровые технологии управления транспортными процессами»

**ОТЧЁТ**

**ПО ПРОИЗВОДСТВЕННОЙ ПРАКТИКЕ**

**Тип практики:** ПРОИЗВОДСТВЕННАЯ ПРАКТИКА

(технологическая - 1)

“Анализ и изучение тестирования ПО, его подвидов и варианты применения”

Выполнил ст. г. УВА-411 Д. Н. Романов

Принял руководитель: Н. М. Нечитайло

Москва  
2021

|  |
| --- |
| УТВЕРЖДАЮ  Руководитель ОП ВО по  направлению 09.03.01  \_\_\_\_\_\_\_\_ Э.К. Лецкий  \_\_.07.2021 г. |

**ЗАДАНИЕ НА ПРАКТИКУ**

**производственную практику**

**(технологическая - 1)**

студента группы УВА-411 Романова Даниила Николаевича

Место прохождения производственной практики: кафедра ЦТУТП ИУЦТ РУТ (МИИТ)

Наименование темы: Анализ и изучение тестирования ПО, его подвидов и варианты применения.

Необходимо изучить:

1. Тестирование ПО
2. Ручное тестирование
3. Автоматизированное тестирование
4. Selenide, Selenoid, Cucumber

Необходимо самостоятельно выполнить:

1. Разбор элемента жизненного цикла разработки ПО “Тестирование”
2. Разбор процесса автоматизированного тестирования
3. Разбор решения проблемы автоматизированного тестирования

Основные требования к проделываемой работе:

Для оформления отчета по практике должен быть использован ГОСТ 7.32-2017  «Отчет о научно-исследовательской работе. Структура и правила оформления».

Источники информации:

1. Книги.

.Материалы к защите:

1. Заполненный, подписанный и утверждённый лист индивидуального задания на производственную практику.
2. Пояснительная записка (отчёт) по производственной практике.
3. Презентация о проделанной работе.

Студент группы УВА-411 Д.Н. Романов

Профессор кафедры ЦТУТП Н.М. Нечитайло

**Оглавление**

[**1** **Вступление** 3](#_Toc94387661)

[**2** **Разбор элемента жизненного цикла разработки ПО “Тестирование”** 4](#_Toc94387662)

[**3** **Разбор процесса автоматизированного тестирования** 8](#_Toc94387663)

[**4** **Разбор проблемы автоматизированного тестирования** 10](#_Toc94387664)

[**Заключение** 12](#_Toc94387665)

[**Источники информации** 13](#_Toc94387666)

1. **Вступление**

Современный цикл разработки ПО совмещает в себе несколько простых шагов. Анализ требований, разработка приложения, его тестирование, релиз.

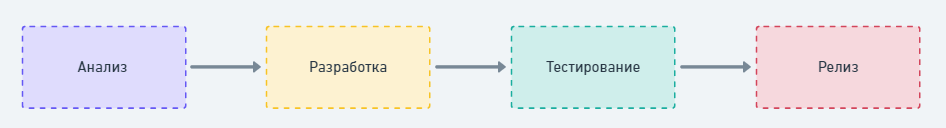


Рис. 1.1 Современный цикл разработки ПО

* Анализ.[3] Здесь происходит разговор с заказчиком, выясняются требования, пишется документация. Происходит постановка задачи на разработку, пишется техническое задание (ТЗ).
* Разработка.[3] Большой и непрерывный этап, выполняется реализация ТЗ, дополняются фичи к реализации, корректируется постановка задачи и т.д.
* Тестирование.[3] Непрерывный процесс тестирования приложения, которое находится в разработке, проверяется не только его соответствие требованиям, но и пропускная способность, отказоустойчивость, готовность к непредвиденным ситуациям и к различному поведению пользователей внутри системы.
* Релиз. [3] Непрерывная поставка оттестированного приложения заказчику для уточнения требований, возможных доработок и непосредственной эксплуатации.

1. **Разбор элемента жизненного цикла разработки ПО “Тестирование”**

Тестирование.



Рис. 2.1 Виды тестирования

* Модульное тестирование.[1] Эти тесты пишут сами разработчики для проработки маленьких кусочков кода во избежание очевидных ошибок работы кода. Покрывается около 70% кода, все методы и логические ветвления.
* Функциональное тестирование. [1] Здесь в дело вступают тестировщики, производится проверка заявленного функционала.
* Smoke тестирование. [1] Простая проверка, что наше приложение не ломается при попытке его использовать, конечный результат неважен. Успешное smoke тестирование позволяет приступить к другим тестам.
* Интеграционное тестирование (A/B тесты). [1] Позволяет протестировать конкретные тестовые сценарии из точки A в точку B. Задевает сразу несколько функций приложения. Например, авторизация -> создание объектов -> изменение -> выгрузка -> удаление -> выход из приложения.  
  Интеграционный тест считается проваленным, если не выполнился хотя-бы один пункт теста.
* Нагрузочное тестирование. [1] Его цель – сломать приложение под предельными нагрузками, чтобы знать его слабые места и устранить их по итогу.
* Системное тестирование. [1] Такое тестирование покрывает функционал интеграции с другими приложениями. Например, мы добавили в приложение Google.Maps и хотим убедиться, что интеграция с ними работает успешно, наши объекты на карте отображены корректно и функционируют.
* Регрессионное тестирование. [1] Это множество различных тестов, которые были успешными в прошлом релизе. После нового релиза их снова проверяют, чтобы убедиться, что новый функционал не сломал что-то из уже работающего. Позволяет легко выявлять недочёты, которые негативно влияют на старый оттестированный функционал. Количество регрессионных тестов увеличивается с каждым новым релизом.

Рассмотрим схему взаимодействия разработчика, тестировщика и заказчика.

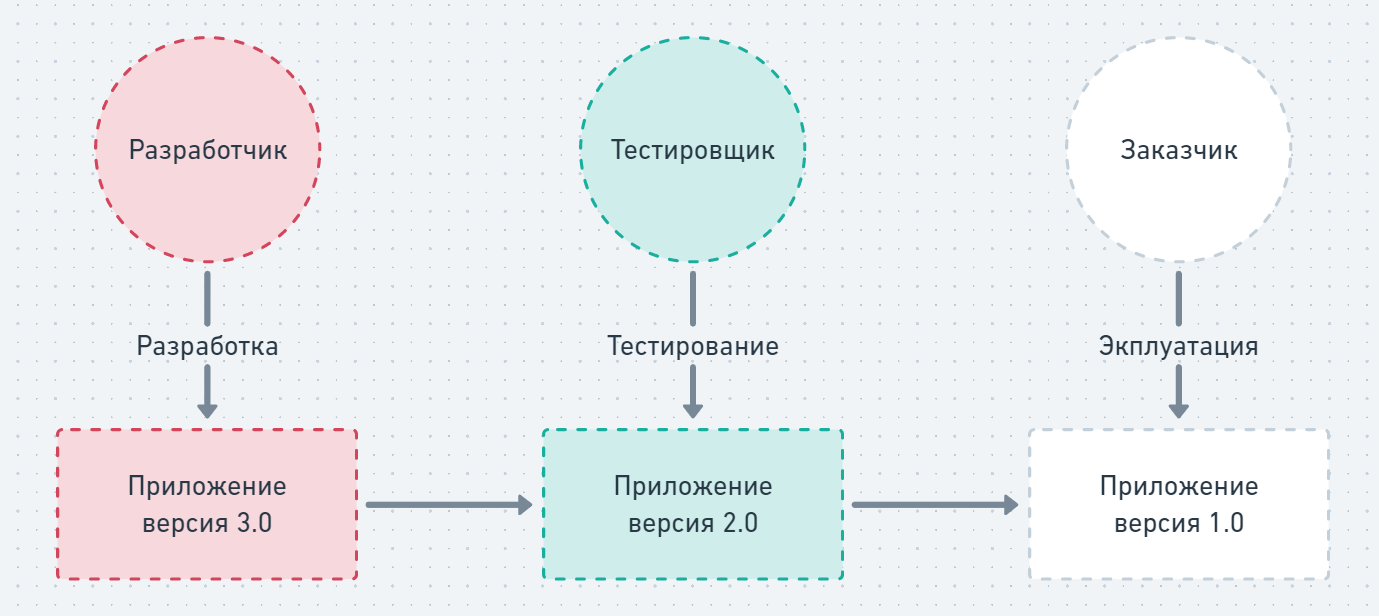


Рис. 2.2 Схема взаимодействия разработчика, тестировщика и заказчика

У заказчика всегда самая старая версия приложения. Он пишет пожелания разработчикам, недочёты, новый функционал, излагает идеи.

Тестировщик проверяет более свежую версию, проводит функциональное тестирование версии 2.0, регрессионное тестирование функционала старой версии 1.0 и т.д.

Разработчик готовит новую версию и так по кругу.

Проблемы появляются, когда у нас имеется несколько разработчиков и приложение делится на несколько маленьких, которые работают вместе (микросервисная архитектура).

Тогда наша схема преобразуется в следующую.

Например, так.

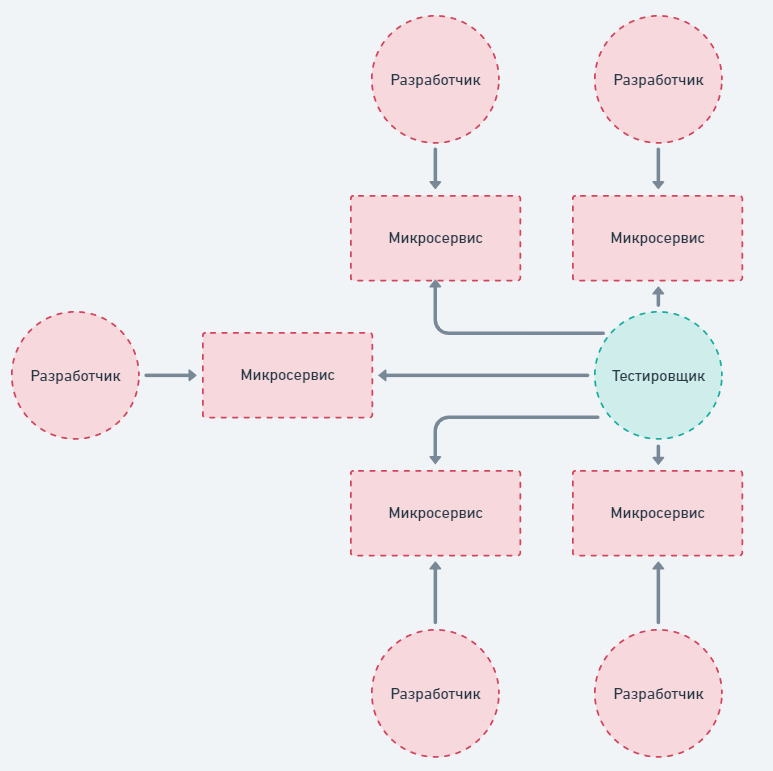


Рис. 2.3 Схема взаимодействия разработчика и тестировщика №1

Получается, что тестировщик перегружен, необходимо либо уменьшить количество разработчиков, что замедлит ход релизов, либо увеличить количество тестировщиков. Увеличим количество тестировщиков.

Получаем следующую схему взаимодействия

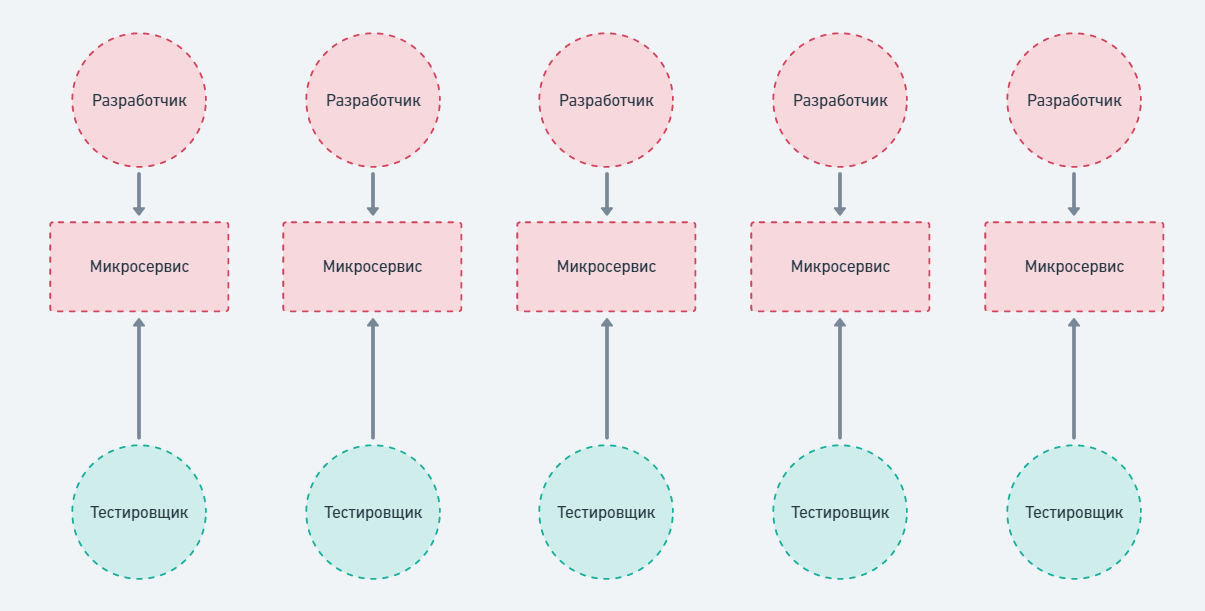


Рис. 2.4 Схема взаимодействия разработчика и тестировщика №2

Отлично. Теперь релизы будут быстрыми за счёт дополнительных четырёх тестировщиков. Но расходы компании на тестирование возросли в пять раз, что не очень хорошо, ведь если разработчик закончит микросервис, он может приступать к следующему, и тогда нам понадобится ещё один тестировщик на новый микросервис, ведь регрессионное тестирование никто не отменял.

Такая схема взаимодействия 1 микросервис = 1 тестировщик очень дорогая и нам не подходит. Потому и придумали автоматизированное тестирование.

1. **Разбор процесса автоматизированного тестирования**

Разработчики работают в прежнем режиме, пишут приложение, его отдельные части, редактируют его, внедряют новую функциональность.

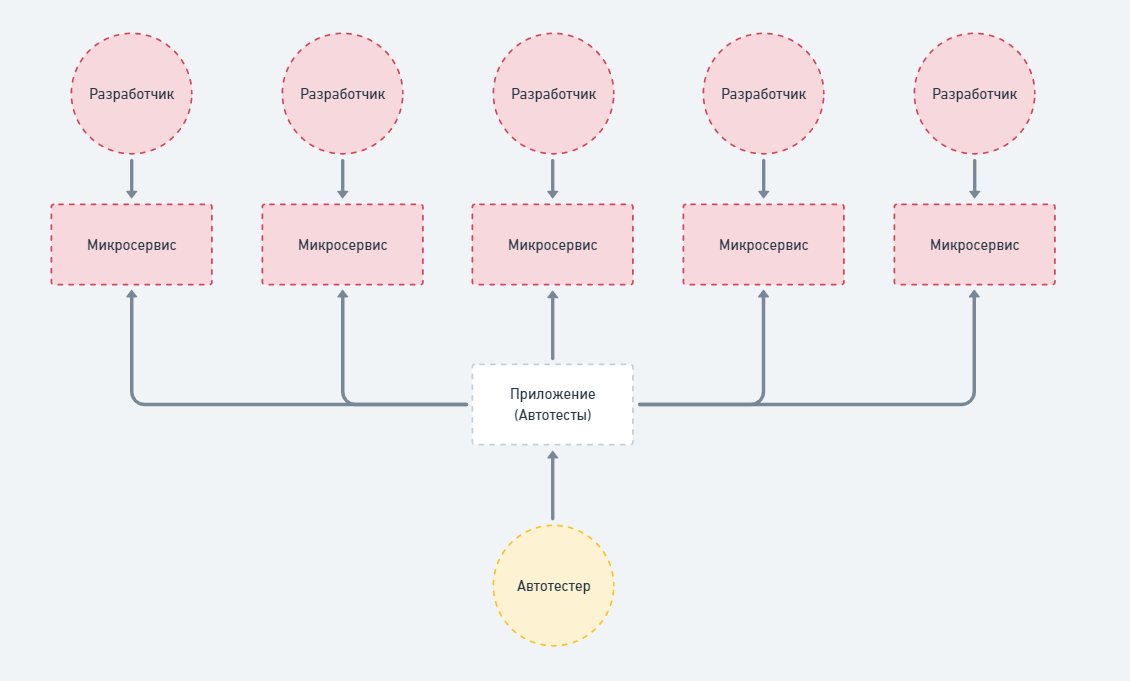
А вот ручных тестировщиков у нас уже нет. 

Рис. 3.1 Схема взаимодействия разработчика и автотестера №1

Теперь у нас есть ещё один разработчик, но он пишет приложение, которое тестирует другие приложения. Иначе говоря – автотесты.

[2]Преимущества такого подхода в том, что теперь нам не нужно постоянно нанимать новых тестировщиков для обеспечения регресса, т.к. после того, как автотестер напишет приложение, его можно запускать любое количество раз, таким образом обеспечивая непрерывное регрессионное тестирование.

[4]Автотесты невероятно гибкие и умеют практически всё, что умеет человек. Т.е. потребность ручном тестировании практически сводится к нулю.

[4]Недостатки такого подхода в том, что автотестер обходится дороже любого ручного тестировщика, а сами тесты по времени пишутся гораздо дольше ручного прогона теста.

Таким образом, когда приложение разрастается и вносятся постоянные правки, необходимо динамически редактировать поведение автотестов, постоянно их корректируя, иначе говоря – актуализировать автотесты.

Отсюда появляется большая проблема того, что с ростом приложения у автотестера не хватает времени заниматься покрытием нового функционала и одновременной актуализацией уже написанных старых тестов. Нам придётся нанимать новых автотестировщиков. Один будет занят актуализацией[2], а другие покрытием нового функционала.

Наша схема постепенно вырождается в предыдущую.

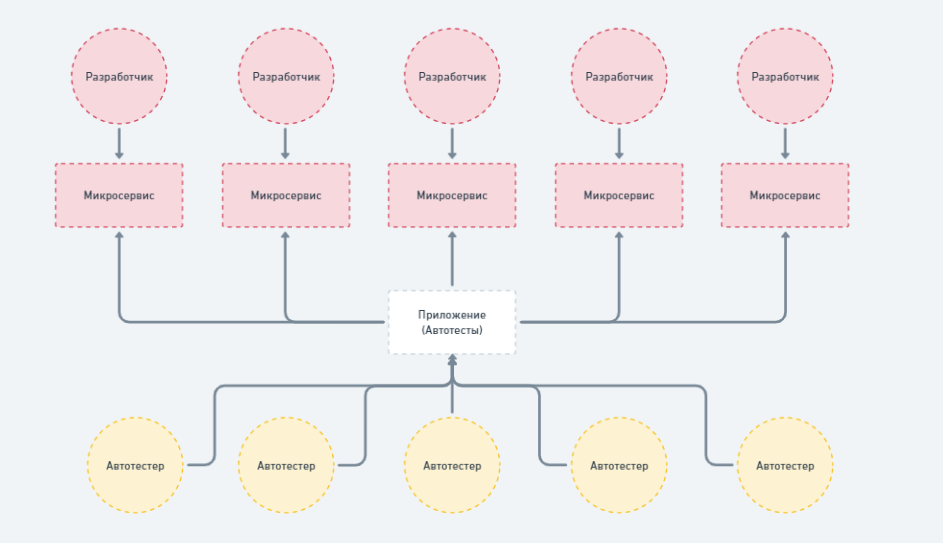


Рис. 3.2 Схема взаимодействия разработчика и автотестера №2

Было бы очень круто, если бы ручным тестировщикам, которые не знают языков программирования, можно было бы просто написать тестовый сценарий (некую последовательность действий) и он превратился бы в программный код и стал автотестом. Таким образом мы смогли бы не только сэкономить на дорогих автоматизаторах тестирования, но и облегчить жизнь ручным тестировщикам, который изо дня в день делают одно и тоже.

1. **Разбор проблемы автоматизированного тестирования**

В следующей схеме присутствует только автотестер и ручной тестировщик, подразумевается, что далее автотесты будут проверять приложение.

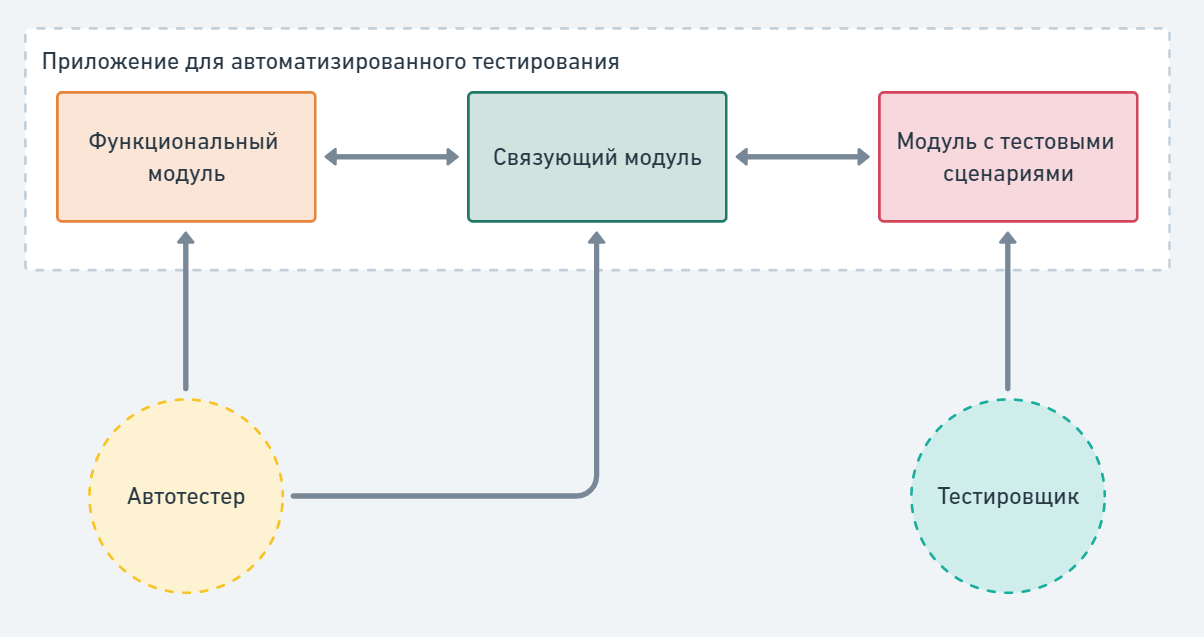


Рис. 4.1 Схема взаимодействия тестировщика и автотестера

Наше приложение состоит из трёх компонентов:

* Функциональный модуль. Содержит основные функциональные возможности автотеста, например – навести мышкой на указанный элемент, перетащить один элемент в другой, нажать на Enter, обновить страницу, кликнуть на элемент и т.д.
* Модуль с тестовыми сценариями. Содержит простой файл с инструкцией, написанной на русском языке, например, «1. Открыть браузер на странице google.com. 2. Нажать на поле поискового запроса. 3. Вести запрос “Selenide”».
* Связующий модуль. Он преобразует русский текст в вызов методов функционального модуля, таким образом текст, написанный тестировщиком, превратится в полноценный автоматизированный тест.

Наше решение – это гибрид между ручным тестированием и автоматизированным. Имеет ряд преимуществ, таких как:

* Прозрачные тестовые сценарии, написанные на русском языке
* Экономия затрат на тестирование из-за сбалансированного количества тестировщиков.
* Повышение эффективности работы ручных тестировщиков, теперь они пишут только сценарии и не прогоняют их руками. Для этого создан автотест
* Повышение эффективности автотестеров, т.к. с них уходят задачи на покрытие новой функциональности, они занимаются только обеспечением бесперебойной работы функционального модуля, а также его расширением под нужды «ручных» тестировщиков.
* Увеличивается скорость введения нового автотеста, т.к. достачно просто переиспользовать возможности функционального модуля, подставляя универсальные языковые конструкции.
* Получен конструктор для автотестов из фраз на русском языке, который буквально можно пересобирать получая новые автотесты.

**Заключение**

В ходе практики были проанализированы бизнес процессы разработки, тестирования и релиза. Выявлены узкие места. Был разобран процесс тестирования современных приложений, выявлены проблемы. Проведен анализ текущих сложностей в проведении тестирования. Предложено решение проблем с автоматизацией и ручным тестированием, была построена схема взаимодействия тестировщиков и автоматизаторов в рамках нового решения. Спроектированы и описаны ключевые модули целевого решения.

**Источники информации**

1. Основы тестирования программного обеспечения. Учебное пособие для СПО, 2-е изд., стер.
2. Software Testing Techniques, 2nd Edition, Борис Бейзер
3. Scrum: Революционный метод управления проектами, Джефф Сазерленд
4. Experiences of Test Automation: Case Studies of Software Test Automation 1st Edition, Dorothy Graham