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 «Исследование методов и технологий автоматизированного тестирования облачного IT-продукта, построенного на микросервисной архитектуре»

Студент	Е.С.Алексеева (И.О. Фамилия)	(личная подпись)
Научный руководитель	канд.пед.наук, доцент, Т.А.Агошкова (ученая степень, звание, И.О. Фамилия)	

Тольятти 2021

Оглавление

Введение 4
Глава 1 Теоретические основы методов и технологий автоматизированного тестирования облачного IT-продукта, построенного на микросервисной архитектур 11
1.1 Характеристики и модели облачных вычислений 11
1.2 Особенности микросервисной архитектуры 16
1.3 Анализ преимуществ и недостатков, а также особенностей применения автоматизированного тестирования 23
Глава 2 Анализ методов проведения автоматизированного тестирования облачного IT-продукта, построенного на микросервисной архитектуре........ 29
2.1 Классификация критериев эффективности тестирования IT-продукта 29
2.2 Методы автоматизированного тестирования облачного IT-продукта, построенного на микросервисной архитектуре 32
2.3 Сценарии автоматизированного тестирования облачного IT-продукта, построенного на микросервисной архитектуре 38
2.4 Методологии разработки программного обеспечения с интегрированным процессом тестирования.. 43
Глава 3 Апробация методов проведения автоматизированного тестирования облачного IT-продукта, построенного на микросервисной архитектуре........ 52
3.1 Информационные модели автоматизированного тестирования облачного IT-продукта, построенного на микросервисной архитектуре . 52
3.2 Алгоритм тестирования облачного IT-продукта, построенного на микросервисной архитектуре и разработанного на основе методологии BDD 55
3.3 Инструментальные средства для тестирования облачного IT-продукта, построенного на микросервисной архитектуре и разработанного на основе методологии BDD.. 58

3.4. Сценарий автоматизированного тестирования облачного IT-продукта, построенного на микросервисной архитектуре, с использованием тестовых дублёров ... 62
Глава 4 Анализ эффективности использования тестовых дублёров как части IT-продукта ... 69
4.1. Преимущества поставки тестовых дублёров как части IT-продукта . 69
4.2 Оценка трудозатрат на установку и конфигурацию тестовых дублеров на оборудовании заказчика ... 71
Заключение ... 75
Список используемой литературы ... 77
Приложение А Набор тест-кейсов, созданный в библиотеке Cucumber-JVM 81

Введение

В последнее время особую актуальность в сфере информационных технологий приобретают вопросы качества программного продукта, связано это с большой конкуренцией среди компаний-разработчиков IT-продуктов. Качество разрабатываемого программного продукта напрямую зависит от того, насколько результат разработки соответствует ожиданиям заказчика. Основным этапом, предшествующим разработке программного продукта, является выявление требований, параметров и критериев, которые будут обеспечивать качество IT-продукта, необходимое заказчику.

Тестирование программного продукта является одним из основных способов управления качеством в сфере информационных технологий. Недостаточно качественно организованный этап тестирования может нанести большой урон всему проекту. Выявление и устранение ошибок во время сопровождения IT-продукта влечёт за собой дополнительные расходы в среднем в 200 раз больше, чем на стадии тестирования, а в случае позднего обнаружения дефектов итоговый бюджет проекта может увеличиться на 30-40%. Наиболее простым, распространённым и дешёвым является процесс ручного тестирования, но он не гарантирует необходимое качество IT-продукта, поэтому всё чаще используют автоматизированное тестирование, однако его широкое использование не всегда целесообразно и возможно.

С целью определения критерия соответствия готового программного продукта заявленным характеристикам, необходимо протестировать разработанный IT-продукт разными методами. Это позволит выявить ошибки, устранение которых повысит качество разрабатываемого продукта. Традиционные методы ручного тестирования уже не могут обеспечить необходимое качество современных программных систем. К основным недостаткам ручного тестирования можно отнести человеческий фактор, трудоемкость повторного тестирования программы после внесения изменений, невозможность нагрузочного тестирования.

В связи с рассмотренными причинами с каждым днём появляется все больше методов, инструментов и технологий автоматизированного тестирования, направленных на улучшение качества и уменьшение затрат на этап тестирования IT-продуктов. Главным преимуществом автоматизации тестирования является уменьшение времени на тестирование больших программных модулей, так как проработать огромный объем данных вручную гораздо труднее, чем один раз составить сценарий выполнения тестирования. Кроме того, созданный тестовый сценарий целесообразно применять далее для организации такого же вида тестирования, что в будущем значительно сэкономит время. Немаловажным фактором можно отметить то, что при автоматизированном тестировании возможно быстро смоделировать пользовательскую нагрузку на всю систему, что обеспечит разностороннюю проверку корректности функционирования системы и, следовательно, гарантирует высокое качество программного продукта.

По мере того, как новые тенденции в различных областях появляются день ото дня, требуется новая архитектура для различных приложений. Пользователям требуется интерактивный, богатый и динамичный опыт работы приложений на различных платформах. Этим требованиям удовлетворяют приложения, имеющие высокую доступность, масштабируемость и простоту выполнения на облачной платформе. Большинство организаций хотят обновлять свои приложения часто, несколько раз в день. Монолитные приложения имеют ограничение для поддержки таких требований. Микросервисы представляют собой новую архитектуру, в которой большие и сложные программные приложения состоят из одного или нескольких небольших сервисов. Они могут работать независимо друг от друга. Такие микросервисы слабо связаны друг с другом, каждый из них отвечает за эффективное выполнение только одной задачи. Микросервисы очень полезны для приложений в облачных вычислениях. Использование микросервисов в облачных вычислениях позволяет повысить популярность облака. Использование микросервисов предлагает больше вариантов и

возможностей для независимого развития сервиса. В 2021 году оно призвано ускорить процесс разработки мобильных и веб-приложений.

Поскольку организации создают все больше микросервисов, они все чаще сталкиваются с проблемами интеграции. Последствия могут быть разными – от увеличения времени на разработку нового функционала, до полностью нерабочей программы. В крайних случаях ошибки согласованности API – интерфейсов обнаруживаются только во время эксплуатации приложения, и ведут к долгой и дорогой его доработке и новому циклу тестирования.

На протяжении десятилетий развития разработки программного обеспечения к вопросам тестирования в системе обеспечения качества продукта подходили с разных точек зрения. Большой вклад в развитие тестирования как отдельного этапа в жизненном цикле разработки программного обеспечения внесли как отечественные, так и зарубежные учёные.

Отечественный учёный Куликов С.С. считает, что современный период развития тестирования характеризуется такими особенностями: гибкие методологии разработки и тестирования, глубокая интеграция процесса тестирования со стадией разработки, повсеместное использование автоматизации, огромный выбор технологий и инструментальных средств, кросс функциональность команды (команда, в которой тестировщик и разработчик во многом могут исполнять работу друг друга).

Согласно исследованию компании IBM, стоимость исправления программных ошибок со временем возрастает. Например, один час работы разработчика стоит 25 $, а процесс устранения дефектов на этапе архитектурного планирования занимает 2 часа. Поэтому общая стоимость составляет 50 $. В случае обнаружения дефекта на втором этапе разработки стоимость его устранения возрастет в пять раз и составит 250 $. Стоимость устранения ошибок на этапе тестирования увеличится в десять раз (500 $), на этапе бета-тестирования - в 15 раз (750 $). После релиза общая стоимость

увеличится в 30 раз и будет равна 1500 $. Очевидно, что стоимость устранения одного и того же дефекта может варьироваться от 50 до 1500 $, что зависит от времени его обнаружения [23].

Определяя место и роль тестирования в обеспечении качества программного продукта, автор книги «Как тестируют в Google» Уиттакер Джеймс считает, что тестирование и разработка должны идти рука об руку. Тестирование не рассматривается как отдельный этап, это часть самого процесса разработки. Качество появляется только тогда, когда разработка и тестирование существуют вместе. «Наконец, выбирая между автоматизированным и ручным тестированием, мы отдаем предпочтение первому. Если это можно автоматизировать, и проблема не требует человеческого внимания и интуиции, то это нужно автоматизировать. Только проблемы, которые явно требуют оценки человеком (например, красив ли пользовательский интерфейс или не нарушает ли раскрытие данных конфиденциальность), должны доставаться ручному тестированию» [19].

Такие авторы, как Э.Дастин, Дж. Рэшка, Дж.Пол в своей популярной книге «Автоматизированное тестирование программного обеспечения. Внедрение, управление и эксплуатация" большое внимание уделяют рассмотрению целесообразности внедрения автоматизации тестирования в принципе. Автоматизация не преподносится как лекарство от всех бед и волшебное средство, делающее тестирование однозначно успешным. Достаточно подробно рассматриваются в книге ложные ожидания, часто связанные с автоматизацией, анализируются факторы, при которых внедрение автоматизации может оказаться неуспешным. При этом описываются те преимущества, которые на самом деле могут дать средства автоматизации. В данной книге автоматизация рассматривается как весьма серьёзный процесс, требующий и планирования, и специально выделенных ресурсов [7].

За рубежом активное развитие автоматизированного тестирования стало возможным благодаря большим постоянным инвестициям. Иностранный ученый Хоффман в четко определил компромисс между стоимостью и

преимуществами автоматизации тестирования, рассматривая различные факторы ROI (окупаемость инвестиций). На основании его расчетов можно сделать вывод, что доход от автоматизации обычно виден в следующем выпуске, после текущего, т. е. запуск и повторный запуск автоматизированных тестовых случаев дает значительную экономию. Эффективные решения по автоматизации тестирования могут привести к более чем 100% -ому ROI в десяти циклах повторного запуска тестового набора [21].

Кроме того, исследование, проведенное Коллинзом, показало, что автоматизация тестирования работает очень хорошо, если команды находят правильный способ внедрения автоматизации тестирования в свои проекты, и представил некоторые стратегии, чтобы минимизировать риск при внедрении автоматизации тестирования [20].

Исследование научных трудов и публикаций показал, что автоматизированное тестирование – это серьёзный процесс, который должен идти параллельно с разработкой программного обеспечения, требующий детального планирования и специально выделенных ресурсов. Создание алгоритма внедрения автоматизированного тестирования позволит значительно повысить качество IT-продукта, скорость разработки и выпуска обновлений.

Исходя из вышеизложенного, теоретическую основу исследования составляют труды следующих отечественных учёных: Гвоздева В.А., Котляров В.П., Коликова Т.П., Куликов С.С., Синицын С. В., Налютин Н. Ю. , Маглинец Ю.А. а также зарубежных специалистов: Бек К., Криспин Л., Калбертсон Р., Грэхем Л., Ньюмэн С., Э.Дастин, Уиттакер Дж., Dani A., Hadas Schwartz C., Yaron T., Bures M., Shlomo M., Sulabh T., Ritu S., Bharti S., Flemstrom D., Potena P., Ulewicz S., Vogel-Heuser B.

Целью диссертационной работы является создание эффективного алгоритма проведения автоматизированного тестирования облачного IT-продукта, построенного на микросервисной архитектуре, для повышения качества решения и скорости доставки обновлений заказчику.

Для достижения обозначенной цели необходимо решить следующие задачи:

1. Рассмотреть теоретические аспекты существующих средств и методов автоматизированного тестирования IT-продукта, построенного на микросервисной архитектуре;

2. На основе рассмотренной теории проанализировать методику проведения автоматизированного тестирования облачного IT-продукта;

3. Разработать алгоритм проведения автоматизированного тестирования облачного IT-продукта, построенного на микросервисной архитектуре, позволяющий повысить качество и скорость доставки обновлений заказчику.

Обозначенная цель определила объект и предмет исследования:

Объект исследования – методы проведения автоматизированного тестирования облачного IT-продукта, построенного на микросервисной архитектуре.

Предмет исследования – разработка сценария автоматизированного тестирования облачного IT-продукта, построенного на микросервисной архитектуре для повышения качества продукта и скорости доставки обновлений заказчику.

Методологическую основу исследования составляют следующие методы:

– метод моделирования, который используется для построения модели IT-продукта и анализе его свойств на основе построенной модели.

– метод абстрагирования, который позволяет исключить из рассмотрения при тестировании незначительные свойства объекта и уделить внимание наиболее значимым характеристикам объекта.

– метод визуализации данных, используемый для построения графиков и схем, позволяющий более наглядно представлять полученные результаты исследования.

Гипотезой исследования является возможность создания тестового дублёра, способствующего повышению качества тестирования на всех этапах разработки и тестирования IT-продукта.

Научная новизна исследования состоит в разработке стратегии автоматизированного тестирования IT-продукта, построенного на микросервисной архитектуре, с применением интеллектуального тестового дублёра; исследование преимуществ, предоставляемых поставкой данного тестового дублёра как части бизнес-решения IT-продукта.

Теоретическая значимость заключается в выявлении нового способа проведения автоматизированного тестирования облачного IT-продукта, построенного на микросервисной архитектуре, с использованием тестового дублёра. Полученные выводы могут быть применены для дальнейшего исследования факторов, влияющих на качество IT-продукта.

Практическая значимость исследования заключается в способах применения тестового дублёра, а также способах его интеграции в стратегию автоматизированного тестирования IT-продукта.

Глава 1 Теоретические основы методов и технологий автоматизированного тестирования облачного IT-продукта, построенного на микросервисной архитектур

1.1 Характеристики и модели облачных вычислений

В соответствии с данными Американского национального института стандартов и технологий (NIST) облачные вычисления следует рассматривать как средство для использования удобного и широкодоступного сетевого доступа к общему хранилищу настраиваемых вычислительных ресурсов. Такими ресурсами могут быть сети, сервера, приложения и услуги, которые предоставляются по требованию заказчика и могут быть использованы с небольшими усилиями по управлению или взаимодействию с поставщиком услуг облачного хранилища.

Можно выделить пять важных характеристик и три модели обслуживания облачных моделей.

К основным важным характеристикам облачных моделей относятся:

- самообслуживание по требованию. Клиент самостоятельно конфигурирует вычислительные возможности (например, серверное время и сетевое хранилище), без взаимодействия с поставщиком услуг;
- широкий доступ к сети. Сеть может использоваться через стандартные механизмы на гетерогенных платформах тонких или толстых клиентов (например, мобильными телефонами, планшетами, ноутбуками и рабочими станциями);
- объединение ресурсов. Используя многоклиентскую модель, можно объединить вычислительные ресурсы провайдера. Различные физические и виртуальные ресурсы могут динамически назначаться и перераспределяются в зависимости от требований потребителей. Несмотря на то, что клиент как правило не может контролировать и

знать точное расположение используемых ресурсов, он может указать местоположение на более высоком уровне абстракции (например, страна, область или конкретный центр обработки данных). В качестве примеров ресурсов можно рассматривать хранение, обработку, память и пропускную способность сети;

- эластичность. Данная характеристика подразумевает гибкую настройку облачных мощностей, возможность их сократить в случае необходимости. В соответствии со спросом такое сокращение может произойти автоматически для более быстрого масштабирования системы. Для потребителя облачные вычислительные ресурсы не ограничены и могут быть предоставлены в любой момент;
- учет потребления. Автоматический контроль и оптимизация используемых ресурсов в облачных системах происходит на определённом уровне абстракции, который соответствует типу услуги (таких как, хранение, обработка, пропускная способность и активные учетные записи пользователей). Использование таких услуг можно легко отследить и проконтролировать. С целью обеспечения прозрачности для поставщика и потребителя составляются отчёты по данным услугам.

NIST выделяет три сервисные модели облачных систем:

- инфраструктура каксервис (IaaS),
- платформа каксервис (PaaS),
- программное обеспечение каксервис (SaaS).

Каждая сервисная модель сконфигурирована из следующих ресурсов:

- аппаратные средства – сервера, сетевые устройства, хранилища данных;
- виртуализация - комплекс различных вычислительных ресурсов или их логического объединения, которое абстрагировано от аппаратной реализации, и обеспечивающее при этом логическую изоляцию друг

от друга вычислительных процессов, выполняемых на одном физическом ресурсе [25];

– платформа – программное обеспечение, необходимое для запуска приложения (Apach, PHP, .NET, JRE);

– приложение – программа, созданная для решения бизнес-задач.

Конфигурация IT-продукта до внедрения облачных сервисов представлена на рисунке 1.

Рисунок 1 — Конфигурация решения до внедрения облачных сервисов

В соответствии с рисунком 1 все составляющие IT-продукта являются собственными вычислительными ресурсами [27].

Рассмотрим конфигурацию IT-продукта в случае внедрения облачных сервисов:

– инфраструктура как сервис (IaaS): покупатель получает доступ к вычислительной мощности, хранилищу данных, сетевым ресурсам, он не управляет облачной инфраструктурой, но может решать, какие операционные системы запускать на выделенной мощности, объем дискового пространства, и приложения, которые будут установлены

на операционную систему. Заказчик также может влиять на конфигурацию сетевых компонентов, как брандмауэр. Виртуализация также становится частью облака. Пример: Amazon предлагает оплату по факту использования более чем для 160 облачных сервисов. Есть возможность оплачивать отдельные нужные сервисы, причем оплачивать только за время их фактического использования;

– платформа как сервис (PaaS): возможно использование программных языков и инструментов, которые поддерживаются поставщиком PaaS. Заказчик уже не имеет контроля над инфраструктурой, в которой работает приложение, но может ее конфигурировать под бизнес-задачи. Приложение остается под полным контролем заказчика. Пример: веб-хостинг является известным PaaS сервисом, который позволяет разворачивать сайты на основе различных технологий, и использовать базы данных, которые не требуется устанавливать самостоятельно;
– программное обеспечение как сервис (SaaS): заказчик может использовать технологии от поставщика услуг, работающие в облачной инфраструктуре. Данные приложения доступны из браузера для всех, в любом месте, в любое время, на любом устройстве. Все, что может изменить заказчик – локализацию и визуальное отображение продукта. Пример: почтовый сервис является типичным представителем SaaS. Пользователи не знают, как и где хранятся их письма, но сервис со всем содержимым им всегда доступен.

Конфигурация решения с сервисным уровнем IaaS представлена на рисунке 2.

В соответствии с рисунком 2 сервисный уровень IaaS предполагает, что виртуализация и аппаратные средства располагаются в облачном пространстве [28]. Компании могут по требованию как увеличить, так и уменьшить арендуемую мощность, делая возможным обработку максимальных нагрузок без капитальных затрат.

Рисунок 2 — Конфигурация решения с сервисным уровнем IaaS

Конфигурация решения на уровне PaaS представлена на рисунке 3.

Рисунок 3 — Конфигурация решения с сервисным уровнем PaaS

В соответствии с рисунком 3 в собственности заказчика остаётся только приложение, а платформа, виртуализация и аппаратные средства становятся частью облачного пространства.

Конфигурация решения с сервисным уровнем SaaS представлена на рисунке 4.

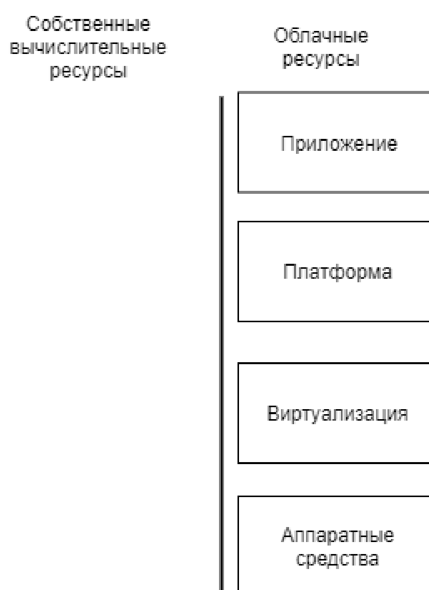

Рисунок 4— Конфигурация решения с сервисным уровнем SaaS

В соответствии с рисунком 4 при использовании сервисного уровня SaaS все ресурсы находятся в облачном пространстве, они открыты и доступны в любое время и на любом устройстве.

Таким образом, рассмотрев все три модели использования облачного пространства, можно сформулировать вывод о том, что при любых требованиях заказчика и возможностях IT-компании можно выбрать удобную конфигурацию использования облачного пространства. Удобный и быстрый доступ к ресурсам позволит с минимальными усилиями по контролю или согласованию с поставщиками услуг разработать и протестировать IT-продукт.

1.2 Особенности микросервисной архитектуры

По мере того, как новые тенденции в различных областях появляются день ото дня, требуется новая архитектура для различных приложений. IT-

компаниям требуется подход к разработке, удовлетворяющий следующим требованиям: кросс-платформенные приложения, способные работать быстро и качественно на разных устройствах; частые обновления; быстрый вывод решения на рынок – минимальное время от возникновения идеи до первой рабочей версии; оптимальный выбор технологий, наиболее подходящий для решения конкретных проблем; широкое переиспользование функциональности [30].

Микросервисная архитектура — модель архитектуры программного обеспечения, которая ориентирована на взаимодействие небольших, слабо связанных между собой компонент— микросервисов. Такие модули легко изменяемы, могут работать независимо друг от друга, каждый из них отвечает за эффективное выполнение только одной бизнес-функции.

Выделяют следующие особенности микросервисов:

- небольшой размер, ориентированность на то, чтобы качественно выполнять только одну работу;
- границы микросервисов(модулей) определяются бизнес-границами, которые определяют область кода для конкретных функций и выполняемых операций;
- изолированность: микросервис можно развернуть в облачном пространстве, например — Platform asa Service(PAAS). При другом рассмотрении он может быть рассмотрен как процесс своей собственной операционной системы;
- гибкость развёртывания: позволяет не просто выбрать самый простой инструмент для всей системы, а индивидуально для каждого модуля определить оптимальную технологию развёртывания, не ограничиваясь универсальным алгоритмом для всей программы. В ситуации, когда какому-то модулю проектируемой системы необходима более высокая производительность, целесообразно выделить другой набор технологий, который более удобен для достижения требуемого уровня производительности. Иногда

оправданным является решение об изменении способа хранения данных для разных подсистем;

- устойчивость: в ситуациях, когда непредвиденно отказывает один из блоков подсистемы, но без последующих отказов, неработающий модуль следует изолировать, но сохранить работоспособность всей остальной системы. Микросервисы могут быть написаны на разных языках и могут использовать разные технологии хранения данных;
- масштабирование: в случае применения традиционных больших монолитных архитектур разрабатываемую программу приходится расширять всю сразу. Так происходит потому, что один небольшой блок ограничен в производительности, но, если из-за него задерживается отклик всего огромного монолитного приложения, то расширять следует все как единое целое. В ситуации с небольшими модулями расширяют только некоторые, те, которые действительно в этом нуждаются. Такой подход позволяет запускать остальные блоки системы на менее мощном оборудовании. При работе с системами предоставления услуг по требованию, например, тех, которые предоставляет Amazon Web Services, часто используют такое масштабирование по требованию для тех модулей приложения, которые в этом нуждаются. Это позволяет существенно сократить расходы;
- простота развертывания: так как микросервисы построены вокруг бизнес-функций, то их можно развернуть независимо друг от друга через полностью автоматизированный механизм развертывания. Возможно изменить только один микросервис и провести развёртывание изолированно от всей системы. Такой подход даёт возможность обновлять микросервис быстрее. Можно быстро изолировать неработающий модель и вернуться на предыдущую версию разработки, если возникли какие-то неполадки в обновлённой системе. Кроме того, новый функционал можно намного быстрее

донести до заказчика. То качество, что микросервисная архитектура позволяет устранить максимальное количество неполадок и задержек при запуске приложения в эксплуатацию, и стала одной из основных причин, по которой такие организации, как Amazon и Netflix, воспользовались ею.

Одним из важных преимуществ, которое появляется при использовании микросервисной архитектуры, становится возможность повторного использования функциональности. Разработка на микросервисной архитектуре позволяет по-разному использовать любую функциональную возможность и в различных целях [26].

При разработке и тестировании современных IT-систем, вопрос изолированности от внешних сервисов всегда актуален. Внешний сервис может быть недоступен на этапе разработки, либо его функционал разрабатывается параллельно и на него нельзя полагаться. Особенно остро этот вопрос встает на этапе планирования автоматизированного тестирования, так как проверять необходимо не только штатное поведение своей системы, но и исключительные случаи: например, недоступность внешнего сервиса, или случай, когда внешний сервис отвечает ошибкой.

Даже при минимальной зависимости от внешних сервисов, микросервисная архитектура предполагает поэтапную разработку и тестирование микросервисов. В таком случае внешней зависимостью уже является взаимосвязь с соседним микросервисом, который может быть еще в стадии планирования. В данной ситуации целесообразно использовать тестовый дублёр.

Джерард Месзарос предложил термин «тестовый дублер» (TestDouble) - общее имя для объектов, используемых для замены настоящих микросервисов IT-продукта в целях проведения более эффективного тестирования. Тестовый дублер – общий термин, для более точных реализаций используются специальные имена. Выделяют 3 категории тестовых дублеров:

– фиктивный модуль (fake) - простейший, самый примитивный тип тестового дублера, не выполняет никаких функций, а просто реализует некоторый интерфейс;

– заглушки (stubs). Сложнее фиктивного модуля, представляют из себя функцию отображения входных параметров в выходных. Заглушки являются минимальными реализациями интерфейсов и базовых классов. Методы, возвращающие пустоту, обычно не содержат реализации вообще, тогда как методы, возвращающие значения, обычно возвращают жестко определенные значения. Используется для того, чтобы тестировался сам метод тестируемого класса;

– имитация (mock) - используется для имитации работы некоторого класса или модуля. Имитация содержит более сложные реализации, обычно осуществляя взаимодействия между различными членами унаследованного ею типа.

Логическая схема программы, где требуется авторизация пользователя, с использованием фиктивного модуля представлена на рисунке 5.

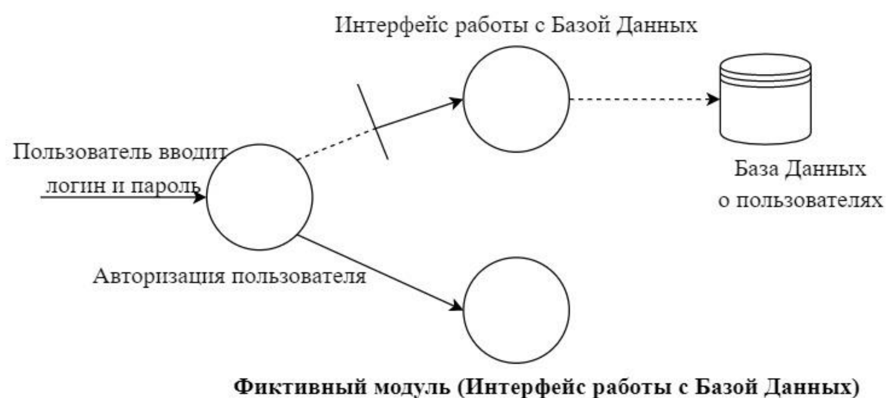

Рисунок 5 - Логическая схема работы программы с использованием фиктивного модуля

В соответствии с рисунком 5 фиктивные модули не содержат реализации и в основном используются, когда необходимы только значения параметров.

Пустые значения могут считаться фиктивными модулями, но фиктивные модули как таковые являются производными от интерфейсов и базовых классов, не отягощенными реализацией.

Логическая схема работы программы по получению средних оценок студентов с использованием заглушки представлена на рисунке 6.

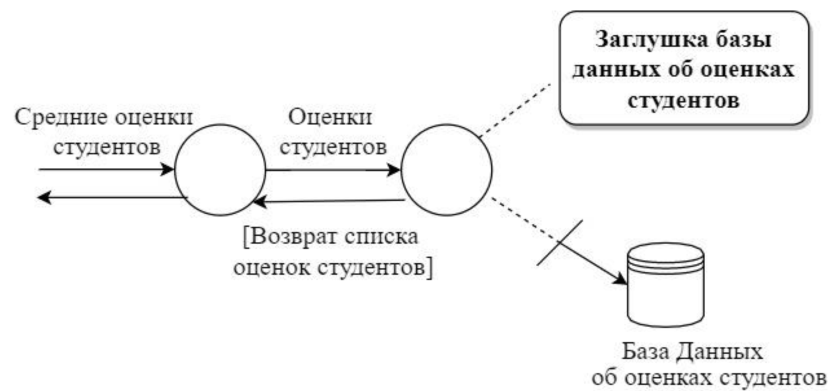

Рисунок 6 - Логическая схема работы программы с использованием заглушки

В соответствии с рисунком 6 основная задача заглушки - возврат в тестируемый модуль некоторого детерминированного значения.

Логическая схема работы программы, посылающей уведомления, с использованием имитации представлена на рисунке 7.

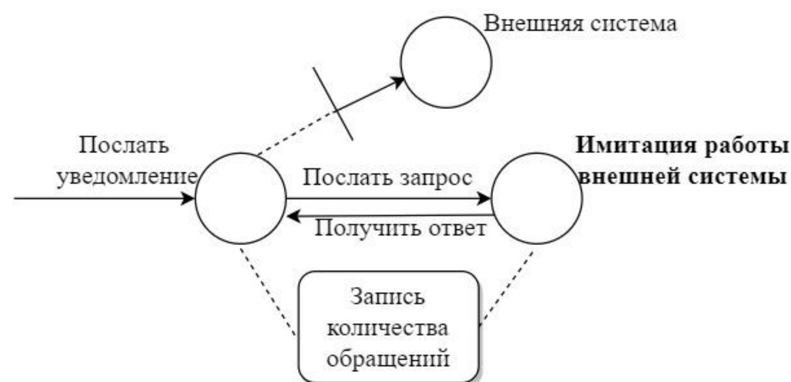

Рисунок 7 - Логическая схема работы программы с использованием имитации

В соответствии с рисунком 7 имитация проверяет корректность передаваемых параметров, порядок вызовов и возвращает результаты некоторым интеллектуальным образом. Имитация не является рабочим решением, но может напоминать такое, хотя и с некоторыми упрощениями [4].

Выделяют также:

– макет: динамически создается библиотекой макетов (прочие типы подразумевают работу с кодом от разработчика теста). Разработчик теста не видит реального кода, реализующего интерфейс, или базовый класс, но может настроить макет на предоставление возвращаемых данных, ожидание вызова определенных элементов и так далее. В зависимости от своей настройки макет может вести себя как фиктивный модуль, заглушка или шпион;
– тестовый шпион: подобен заглушке, но помимо выдачи клиенту экземпляра, для вызова элементов, шпион также записывает, какие элементы вызывались, так что модульные тесты могут удостоверяться, что элементы вызывались в соответствии с ожиданиями. Одной из форм этого может быть служба электронной почты, которая записывает, сколько сообщений было отправлено [18].

Итак, при автоматизированном тестировании IT-продукта, построенного на микросервисной архитектуре, целесообразно использовать тестовые дублёры, которые выглядят и ведут себя как микросервисы, но на самом деле они упрощены. Главное преимущество использования тестовых дублёров – возможность изолированно и поэтапно тестировать каждый микросервис, а в дальнейшем и взаимосвязь микросервисов между собой. Такой подход уменьшает сложность, но повышает качество тестирования IT-продукта.

1.3 Анализ преимуществ и недостатков, а также особенностей применения автоматизированного тестирования

Главными факторами, инициирующими внедрение автоматизированного тестирования, можно назвать повышение качества разрабатываемого IT-продукта и экономия времени, затрачиваемого на непосредственно процесс тестирования. Повышение качества подразумевает то, что при автоматизированном тестировании можно охватить больший набор тест-кейсов, и/или тесты, которые невозможно выполнить вручную, и тот фактор, что при автоматизированном покрытии возможно минимизировать ущерб от ошибок человека в результатах тестирования. В связи с тем, что на выполнение автоматизированных тестов уходит намного меньше времени, чем на аналогичное тестирование вручную, происходит экономия времени. Интернет-энциклопедия «Википедия» даёт следующее определение термину автоматизированное тестирование программного обеспечения: автоматизированное тестирование программного обеспечения — этап процесса тестирования при контроле качества во время разработки программного обеспечения. При этом используются программные средства для прохождения тестового набора и проверки результатов выполнения. Всё это в комплексе позволяет сократить время тестирования и упростить его процесс.

К преимуществам автоматизированного тестирования можно отнести:

- сокращение времени на прохождение тестового набора по сравнению с ручным тестированием;
- возможность проведения некоторых тестовых нагрузок, которые невозможно провести вручную;
- повышение независимости тестирования от человека (исключается человеческий фактор ошибки);
- возможность проводить тестирование в любое время, в том числе и во вне рабочее время специалиста по тестированию [3].

Кроме преимуществ, необходимо рассмотреть и недостатки автоматизированного тестирования, основные из них следующие:

– автоматизированное тестирование требует значительных трудозатрат и предварительной подготовки технологий и персонала;
– необходим более опытный и квалифицированный персонал;
– после проведения тестирования следует провести тщательный анализ результатов;
– любое изменение в структуре кода программы влечёт за собой модификации в кодировке автоматических тестов;
– в результате одного ошибочного автотеста могут получиться некорректные результаты для прохождения последующих тестов;
– само написание автоматического теста необходимо сделать качественно, без ошибок, соблюдая правильный логическую схему реализации;
– не все функциональные требования в системе можно проверить автоматизированными тестами с помощью выбранного инструментария.

Итак, целесообразность внедрения автоматизированного тестирования не всегда оправдана и понятна [1]. Например, необходимость автоматизации очевидна, если стоит вопрос о прохождении часто повторяемых идентичных тестов, а также при нагрузочном тестировании, тестировании быстродействия компонентов системы или других ситуациях, когда достаточно сложно или невозможно обеспечить необходимые условия для прохождения теста вручную. Однако необходимость автоматизации часто подвергается сомнениям, если встаёт вопрос об автоматизации тестирования функциональных критериев.

Также важную роль играет виртуализация. Она является ключевым фактором в вопросах, связанных с управлением тестовой средой. Настройка виртуальной машины предоставляет дополнительное пространство как специалистам по разработке, так и тестировщикам для организации

тестирования исследуемого приложения. Виртуализация применяется для уменьшения накладных расходов, связанных с различными конфигурациями операционной системы и программными компонентами. Участники часто используют документ для сбора различных требований к среде тестирования, с целью спланировать управление своей существующей средой или создать новою [22].

Лиза Криспен и Джанет Григори в своей книге описали матрицу, помогающую убедиться, что все тесты, которые требуются для обеспечения качества продукта, выполнимы. Секторы тестирования представлены на рисунке 8.

Рисунок 8 - Секторы тестирования по Лизе Криспен и Джанет Григори

В соответствии с рисунком 8 квадрат Q1 - модульные тесты. Цель использования – проверить работу некоторой функции или метода. К этой категории тестирования относится и тестирование, основанное на методологии разработки под контролем тестирования (Test-Driven Design (TDD)) и все виды тестового покрытия на основе свойств (Property-Based Testing). При таких тестах не происходит запуск сервиса, тестирование

проходит без использования внешних систем или подключений к сети. Следовательно, таких тестов запускается очень много, но их прохождение происходит достаточно быстро, если логически тесты правильно сконструированы. В современных условиях прогнозируется прохождение многих тысяч таких тестов за время меньшее, чем 1 минута. Главная задача при модульном тестировании – получение мгновенных ответов за запросы, которые говорят о корректном функционировании написанного кода конкретной функции или метода.

Идея разработки интеграционных тестов состоит в том, чтобы без участия пользователя провести непосредственное тестирование сервиса. В случае написания программы по принципу монолитной системы, целесообразно тестировать коллекции классов, которые обеспечивают сервис интерфейсу пользователя. Если же разрабатываемый продукт состоит из нескольких микросервисных модулей, тестирование сервиса используется для оценки возможностей конкретного изолированного сервиса. Главный фактор, влияющий на такое разделение – желание изолировать тест от остального функционала с целью более качественного тестирования.

Квадрат Q2. Функциональные тесты помогают разработчикам в тестировании системы, но на более абстрактном уровне. Они формируются на основе функциональных требований заказчика и тестируют работу определённого функционала. Такое тестирование необходимо провести для всей разрабатываемой системы. Запускать и контролировать такое тестирование удобнее всего через графический интерфейс пользователя в браузере. Но немаловажным является и проверка других функций взаимодействия с пользователем, например, выкладывание файла. Результатом функционального тестирования является уверенность в том, что протестированный продукт работает в производственном режиме.

Квадрат Q3. Исследовательские тесты проверяют, способна ли система корректно работать с точки зрения пользователя. Такое тестирование можно выполнять только вручную, так как необходимо оценить удобство и

правильность созданного функционала. Чаще всего такое тестирование проводится совместно с представителями заказчика во время так называемой UAT фазы (User Acceptance Testing). Во время такого тестирования совместно с заказчиком возможно не только выявить недостатки текущего решения, но и запланировать разработку нового функционала в создаваемой программе.

Квадрат Q4. Нагрузочное тестирование требует применения специальных программных инструментов и предполагает проверку соответствия созданного IT-продукта заданным нефункциональным требованиям. К таким требованиям относят свойства, определяющие как что программа должна демонстрировать при работе, или какие-то ограничения, которые она должна соблюдать, не влияющие на поведение системы. Например, максимальная производительность, устойчивость, удобство использования, расширяемость, надежность, условия эксплуатации, поддерживаемость.

Данные требования очень важны для бизнеса, наравне с функциональными – к примеру, медленное открытие стартовой страницы интернет-магазина грозит потерей клиентов. Другой вариант - разглашение или утеря личных данных пользователей влечёт за собой судебные издержки и репутационные потери. [8]

Итак, тестирование проводится по множеству причин: найти ошибки в коде программы, убедиться в том, что созданный программный продукт надежен, и что он решает поставленные перед ним задачи. Знание всех функциональных и нефункциональных требований, планируемых интеграций, безопасности, облегчает разработку системы и позволяет лучше выбрать как архитектуру, так и используемый инструментарий для тестирования.

В результате изучения теоретических основ и методов проведения автоматизированного тестирования облачного IT-продукта, построенного на микросервисной архитектуре можно сделать вывод о том, что для создания качественного, конкурентоспособного программного продукта необходим комплексный подход к выбору необходимых информационных ресурсов и

технологий. Выбрав оптимальную модель использования облачного пространства, можно получить удобный и быстрый доступ к ресурсам без дополнительных денежных и временных затрат.

Микросервисная архитектура IT-продукта позволяет быстрее разрабатывать и тестировать каждый микросервис, который нацелен на выполнение только одной задачи. При такой организации разработки программных продуктов целесообразно использовать автоматизированное тестирование, где ответы на запросы исследуемого микросервиса будут посылать тестовые дублёры, имитирующие связь с сотрудничающими подсистемами. Рассмотрев категории тестовых дублёров и их логическую схему работы, становится возможным определиться с оптимальным видом тестового дублёра и настроить его для сотрудничества с разрабатываемым микросервисом.

Однако, следует отметить, что внедрение автоматизированного тестирования требует детального анализа, рациональной обоснованности, квалифицированных сотрудников и значительных временных и экономических затрат.

Глава 2 Анализ методов проведения автоматизированного тестирования облачного IT-продукта, построенного на микросервисной архитектуре

2.1 Классификация критериев эффективности тестирования IT-продукта

Тестирование программного обеспечения подразумевает последовательное прохождение программы через созданные тестовые наборы (тест-кейсы).

Эффективным является тест, который обнаруживает хотя бы одну ошибку. Таким образом, целесообразно создавать такие тест-кейсы, каждый из которых с максимальной вероятностью сможет обнаружить ошибку. Формирование набора тест-кейсов имеет важное значение, поскольку тестирование является одним из наиболее трудозатратных этапов (занимает от 30 до 60 % общей трудоемкости) создания программного продукта [14].

Классификация основных критериев формирования тестовых наборов при тестировании программного обеспечения представлена на рисунке 9.

Рисунок 9 - Классификация критериев тестирования

В соответствии с рисунком 9 выделяют следующие критерии:

– стохастические критерии: применяются для тестирования комплексных программных модулей – в ситуациях, если набор детерминированных тестов имеет огромную мощность. Изначально составляются программы - имитаторы случайных потоков входной информации{x}. Далее независимым способом генерируются значения {y} для соответствующих входных значений {x} и в итоге формируется тестовый набор {x,y};
– критерии тестирования потоков управления: предполагают проверку системы по принципу "белого ящика". Данный подход требует понимания написанного кода программы в соответствии со спецификацией в виде потокового графа управления. Данный вид критериев оправдан во время планирования модульного и интеграционного тестирования (Unit testing, Integration testing);
– критерии тестирования потоков данных предполагают формирование наборов тест-кейсов, которые в совокупности обеспечивают однократную проверку каждого экземпляра класса входных данных. Происходит сопоставление классов данных с режимами функционирования тестируемого блока или микросервиса приложения. Такой подход существенно уменьшает вариации переборов, которые следует учитывать при разработке тестовых наборов. Кроме ограничений на величины входных данных, необходимо учитывать ограничения, связанные с данными во всевозможных комбинациях. Так же важным фактором является проверка функционирования системы при появлении сбоев в значениях или структурах входных данных. Контроль аналогичных ошибок - процесс очень трудоёмкий и затратный по времени, что влечёт ограничении в использовании данного критерия;
– функциональные критерии являются самыми важными в процессе тестирования. Они обеспечивают контроль степени соответствия

требованиям заказчика в программном продукте, отражают взаимодействие тестируемой системы с окружением. Работают по модели "черного ящика".

К наиболее применяемым видам функционального тестирования следует отнести тестирование функций и тестирование спецификаций.

Тестирование функций является самым популярным и многочисленным критерием, по которому проверяется каждая функция (способ), реализуемая программным модулем (классом). Тестирование функций объединяет в себе особенности структурных и функциональных критериев и базируется на принципе "полупрозрачного ящика", при котором явно прослеживаются не только входы и выходы тестируемого модуля, но также состав и структура используемых методов (функций, процедур) и классов.

В случае тестирования спецификации важно спланировать набор тестов, обеспечивающий покрытие каждого пункта требований заказчика минимум один раз. Так как спецификация чаще всего включает тысячи требований, то тестирование спецификации должно проверить их все, затрагивая все требования к каждому методу. Кроме того, проверка должна произойти и в отношении всех требований к классам и системе в целом.

Мутационные критерии реализуют подход, который позволяет на основе незначительных ошибок сделать вывод о совокупном количестве ошибок в разрабатываемой системе. Мутациями считаются небольшие неисправности, а мутанты – это программы, отличающиеся друг от друга мутациями. Принцип мутационного тестирования основан на том, что в разрабатываемую программу P вносят мутации, т.е. искусственно создают программы-мутанты P1, P2... Затем программа P и ее мутанты тестируются на одном и том же наборе тестов (X,Y).

В случае успешного прохождения набора тестов формируется суждение о корректной работе программы P. Кроме того, фиксируются все внесенные в программы-мутанты ошибки, делается вывод о том, что набор тестов (X,Y)

соответствует мутационному критерию, а тестируемая система считается корректной [1].

Итак, рассмотрев основные критерии формирования тестового набора можно сделать вывод о том, что основная сложность в тестировании – определить и сформировать достаточное количество тестовых наборов, которые позволят всесторонне проверить разрабатываемую систему, но не будут избыточны.

2.2 Методы автоматизированного тестирования облачного IT-продукта, построенного на микросервисной архитектуре

Основной проблемой тестирования является определение того, какое количество тестов будет достаточным для вывода о корректной работе системы, а также определение тех видов тестов, которые позволят всесторонне проверить каждую отдельную функцию и весь функционал в целом.

С целью решения этой задачи выделяют следующие методики:

- эквивалентное разделение (Equivalence Partitioning),
- метод анализа граничных значений (Boundary Value Analysis),
- метод составления диаграмм причинно-следственных связей (Cause/Effect) [1].

Эквивалентное разделение подразумевает разделение входных данных на классы эквивалентности. Идея в том, чтобы исключить набор входных данных, которые заставляют систему вести себя одинаково и давать аналогичный результат при тестировании программы. Идея - уменьшение числа избыточных тестовых сценариев, избавляясь от тех, которые генерируют один и тот же результат и не всегда обнаруживают дефекты в функциональности программы [28].

На рисунке 10 представлен пример выделения классов эквивалентности (цветами изображены области корректных и некорректных значений, а кружками — сами классы эквивалентности):

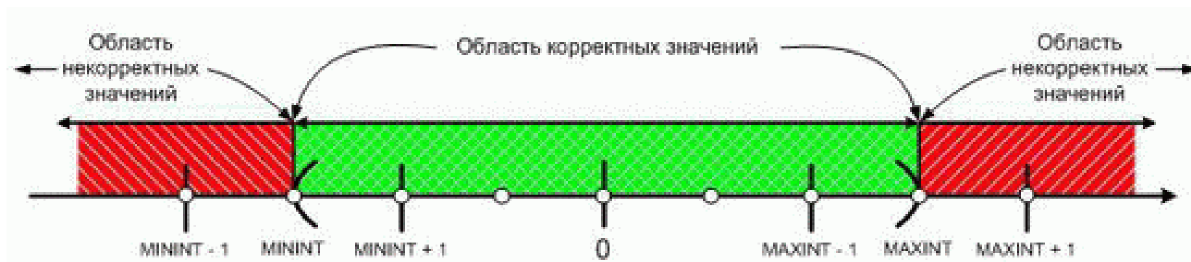

Рисунок 10 - Выделение классов эквивалентности

Метод анализа граничных значений предполагает подход, при котором анализировать следует не только граничные значения, но и данные вблизи (выше и ниже) выделенных границ. Как правило, методики тестирования классов эквивалентности и анализа граничных значений следует совмещать. В качестве недостатка такого метода следует отметить то, что при таком тестировании не покрываются все возможные комбинации множества входных данных и возникает большая вероятность пропустить критическое значение, вызывающее отказ системы.

Составление функциональных диаграмм помогает составлять результативные наборы тест-кейсов, выявляющие большое количество ошибок. Генерация тестовых наборов по данному методу происходит в следующей последовательности:

– изучение спецификации (выделяются причины (классы эквивалентности) и следствия (выходное ограничение либо изменение программы), выстраивается их взаимосвязь);
– в соответствии с 1-м этапом составляется булевый граф (демонстрируются события, их следствия и совокупность связей между ними);
– на основе графа составляется таблица решений (происходит преобразование булевого графа в таблицу решений путем методического отслеживания состояний условий на схеме);

– формирование тестового комплекта по столбцам таблицы решений [16].

Пример булевого графа тестирования представлен на рисунке 11.

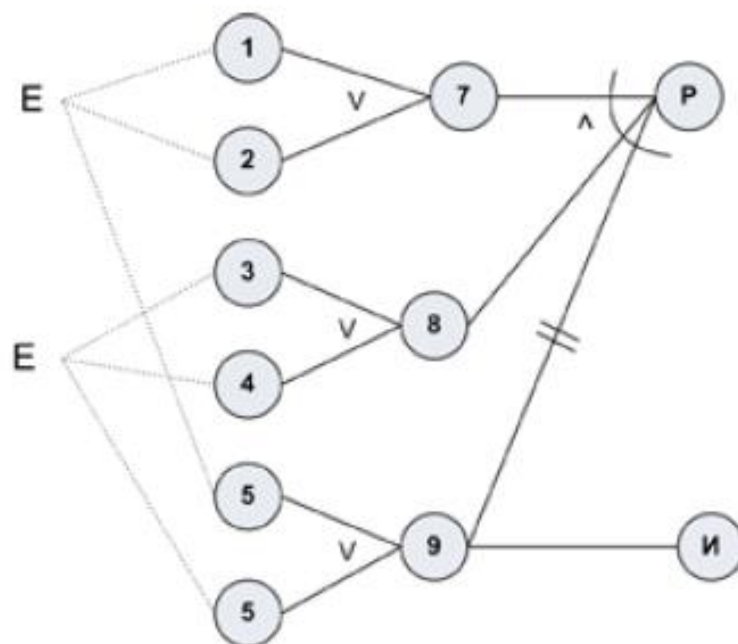

Рисунок 11 - Пример булевого графа тестирования

Рассмотренный метод обеспечивает высокую результативность формируемых тестовых наборов, но рассмотренный процесс занимает неоправданно много времени и при ручном исполнении является чрезвычайно трудоёмким. Этап преобразования булевого графа в таблицу решений и составление тестового набора по составленной таблице можно частично автоматизировать.

Для определения необходимой методики тестирования необходимо составить спецификацию процедуры тестирования. Весь процесс проектирования и реализации автоматизированного тестирования IT-продукта представлен на рисунке 12.

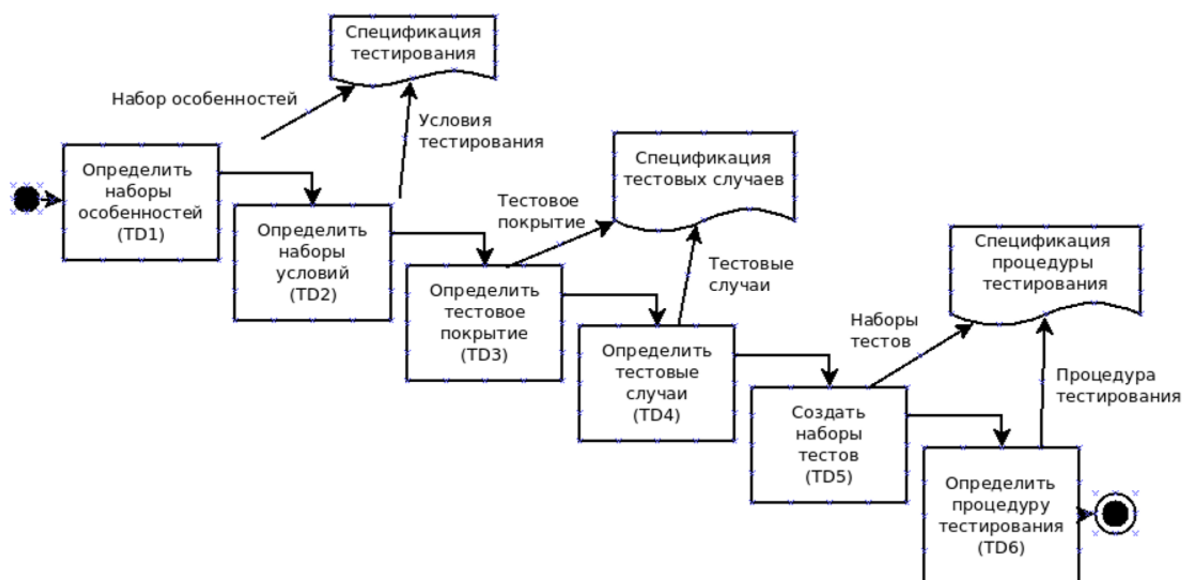

Рисунок 12 - Проектирование и реализация процесса тестирования

В соответствии с рисунком 12 первоначальным этапом является определение набора особенностей и набора условий. Затем исследуется область тестового покрытия и составляется ранжирование тестовых случаев [29]. Далее составляется спецификация процедуры тестирования, включающая описание процедуры тестирования и окончательный набор тестов. Целесообразно разбивать тестовые наборы на более мелкие бизнес-артефакты, которые могут быть повторно использованы в различных тестовых случаях.

Очень важным аспектом в тестировании является возможность многократного использования тестовых наборов. Автоматизированное тестирование используется для повторного применения созданных тестов. Жизненный цикл теста представлен на рисунке 13.

Итак, в соответствии с рисунком 13 жизненный цикл любого теста состоит из последовательных этапов: новый тест, который в последствии становится либо устаревшим, либо пригодным для повторного использования. Далее возможны варианты: тест может требовать повторного запуска и оставаться пригодным для повторного использования, а может стать устаревшим, что говорит о завершении жизненного цикла.

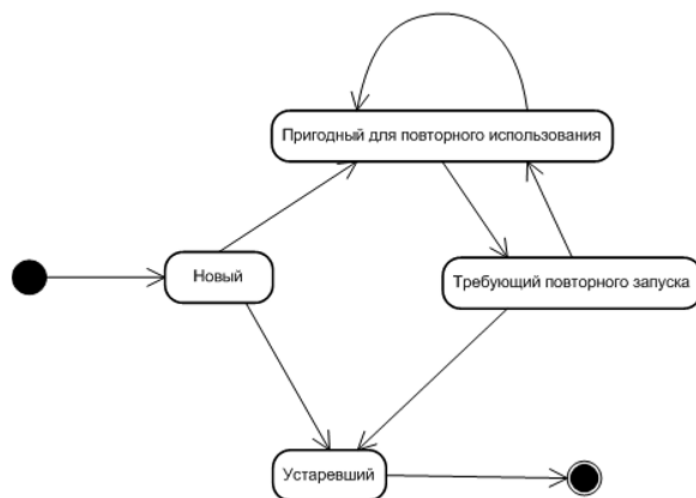

Рисунок 13 - Жизненный цикл теста

К необходимости модификации созданного тестового набора приводят три ситуации: составление новых актуальных тест-кейсов, прохождение тест-кейсов, модификация кода. Старый тест перестаёт быть актуальным и происходит создание нового теста, если изменяется входная информация или сопутствующие условия. Изменение входных данных влечёт за собой изменение траектории и, как следствие, результирующих данных. Таким образом, следует рассмотреть различные варианты повторного использования теста.

На первом уровне созданный тест не предполагается использовать повторно. Формируется новый тест-кейс (например, путем удаления или изменения старого теста).

На втором уровне допускается повторное использование только входных данных теста. Рассматриваются случаи, когда часть тестируемой программы включается в работу раньше вновь созданных команд, тогда входные данные могут быть использованы повторно для покрытия этих элементов. Но после изменений в системе и/или спецификации на разработку алгоритм и результирующие данные могут существенно отличаться от результатов предыдущих запусков.

На третьем уровне возможно повторное использование как входных, так и выходных данных теста. Функциональное тестирование относится к этому уровню. Если в системе происходит изменение кода с сохранением основного функционала, то становится возможным многократное применение существующих функциональных тестов для проверки корректности системы [27]. В результате повторного прохождения тестового набора итоговые результаты должны быть идентичны результатам предыдущих запусков тестов.

Четвёртый уровень является наивысшим уровнем повторного использования теста и допускает повторное использование входных данных, выходных данных и траектории теста. Бывают случаи, когда на траектории теста не изменился ни один оператор, тогда нецелесообразно запускать такие тесты несколько раз, так как и результирующие данные, и траектория останутся прежними [11]. Идея целесообразности внедрения тестирования состоит в том, что количество тестируемых случаев будет увеличиваться с каждым завершенным модулем проекта.

Итак, проанализировав методы автоматизированного тестирования облачного IT-продукта, построенного на микросервисной архитектуре, можно сформулировать вывод, что выбор конкретного метода зависит от спецификации программного продукта, уровня автоматизации тестирования и критериев эффективности работы готовой системы. Проблему планирования тестирования можно рассматривать как проблему практического решения минимизации затрат при написании IT-продуктов. Мониторинг записей ранее выполненных тестовых случаев может повлиять на результаты новых тестовых случаев, особенно на уровне интеграционного тестирования, где тестовые примеры являются более взаимозависимыми.

2.3 Сценарии автоматизированного тестирования облачного IT-продукта, построенного на микросервисной архитектуре

Микросервисная архитектура стала лучшей альтернативой для монолитных, сложных и негибких приложений. Монолитное приложение имеет слабую масштабируемость, а обновление на уровне процедур требует полного обновления системы.

При тестировании IT-продукта, построенного на микросервисной архитектуре, вся система разделяется на более мелкие, модульные, совместно работающие компоненты. Их гораздо проще создавать, обновлять и тестировать, чем приложение в целом. Кроме того, эта архитектура обеспечивает большую масштабируемость, что очень эффективно для большого количества развертываний на различных платформах.

Особенностями автоматизированного тестирования IT-продуктов, построенных на микросервисной архитектуре являются:

- необходимость автоматизированной инфраструктуры, обеспечивающей управление жизненным циклом, именование, адресацию и масштабирование микросервисов в зависимости от текущей загрузки;
- необходимость непрерывной интеграции разработанных и/или модифицированных микросервисов в существующую систему требует всестороннего тестирования как отдельных микросервисов, так и их совместного функционирования в комплексе с другими микросервисами [29].

Весь процесс разработки и тестирования микросервисов представлен на рисунке 14.

В соответствии с рисунком 14 при тестировании микросервисов большое значение следует уделить компонентному тестированию, тестированию контейнера с микросервисом и интеграционному тестированию.

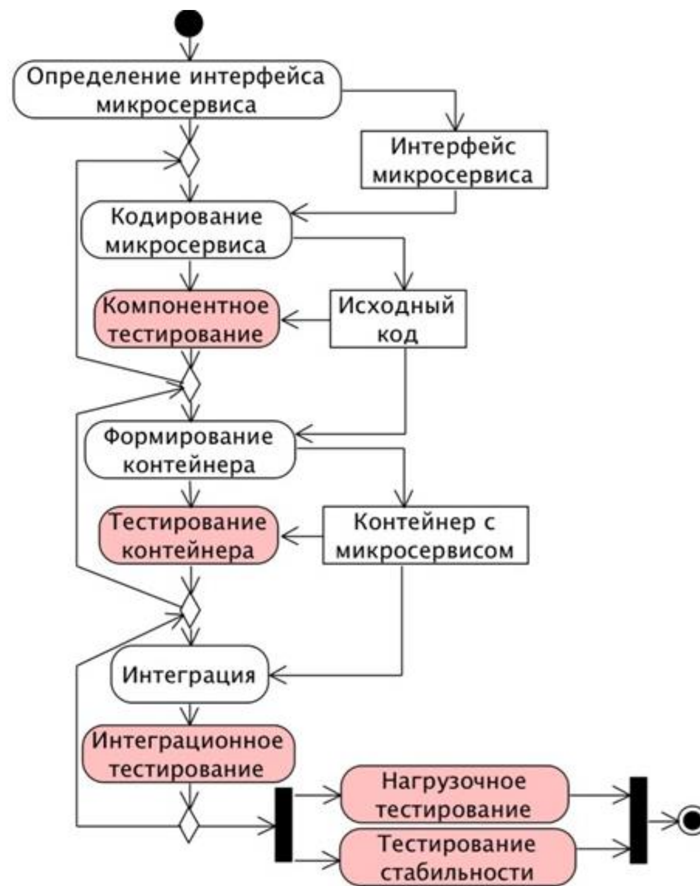

Рисунок 14 - Процесс разработки и тестирования микросервисов

Компонентное тестирование подразумевает тестирование каждого микросервиса по отдельности, изолированно. Логическая схема компонентного тестирования микросервисных систем представлена на рисунке 15.

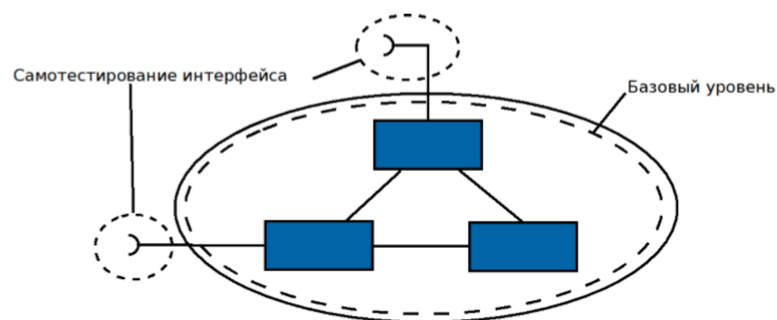

Рисунок 15 - Компонентное тестирование микросервисных систем

На рисунке 15 наглядно изображено, что компонентное тестирование предполагает самотестирование интерфейса каждого микросервиса внутри базового (отдельного) уровня системы.

В компонентном тестировании микросервисов следует рассмотреть:

– функциональное компонентное тестирование нацелено на диагностику работы каждого микросервиса в зависимости от функциональных требований заказчика;
– нагрузочное компонентное тестирование состоит в оценке работоспособности и максимальной устойчивости каждого микросервиса при определённой нагрузке;
– компонентное тестирование безопасности включает в себя тестирование безопасности и изолированности отдельного микросервиса.

Тестирование контейнера с микросервисом предполагает следующий сценарий. Каждый микросервис вызывается независимо друг от друга и при этом проверяются их ответы. Любые зависимости должны помочь микросервису функционировать, но не должно быть взаимодействия с другими сервисами. Это позволит избежать любого непредсказуемого поведения, которое может быть вызвано влиянием другого сервиса. При тестировании проверяется ответ или другой результат, который зависит от входных данных. Каждый потребитель должен получать один и тот же результат от микросервиса, даже если его внутренняя реализация изменяется. Каждый микросервис должен быть способен к гибкому изменению функционала, но ранее реализованный функционал не должен меняться и тянуть за собой изменение вызывающего микросервиса.

Каждый микросервис имеет несколько экземпляров времени выполнения. При тестировании контейнера с микросервисом каждый экземпляр должен быть настроен, развернут, масштабирован и отслежен [28].

Далее необходимо провести интеграционное тестирование микросервисных систем. Интеграционное тестирование — это этап

тестирования системы, состоящей из двух и более модулей и направленный на обнаружение ошибок в реализации и интерпретации интерфейсного взаимодействия между модулями. Интеграционное тестирование сочетает в себе следующие виды:

- функциональное интеграционное тестирование обеспечивает тестирование взаимодействия отдельных микросервисов между собой:

  а) протоколы и форматы обмена сообщениями,

  б) взаимные блокировки и совместное использование общих ресурсов;
- нагрузочное интеграционное микросервисное тестирование оценивает состояние и работоспособность системы при автоматическом развертывании или масштабировании микросервисов;
- интеграционное тестирование безопасности микросервисов предполагает тестирование безопасности взаимодействия между микросервисами, а также проверку на возможность перехвата сообщений злоумышленниками извне или внутри системы.

В процессе интеграционного тестирования происходит сборка микросервисов (модулей). Различают следующие сценарии [18]:

- монолитный, предполагающий одновременную интеграцию всех микросервисов в единую систему. Для имитации работы отсутствующих, но необходимых для тестирования модулей необходимо использовать тестовые дублёры.
- инкрементальный, когда предполагается поэтапное тестирование путем постепенной интеграции двух или более логически связанных модулей. Постепенно присоединяются другие связанные модули до момента полной интеграции. Далее происходит комплексное тестирование всей системы.

Выделяют две стратегии добавления модулей: «снизу вверх» и «сверху вниз».

«Снизу вверх» (восходящая интеграция): каждый модуль на более низких уровнях тестируется с более высокими модулями, пока не проверится работа всех модулей. Часто прибегают к использованию тестовых дублёров (заглушек) с получением в итоге целого приложения в соответствии с рисунком 16.

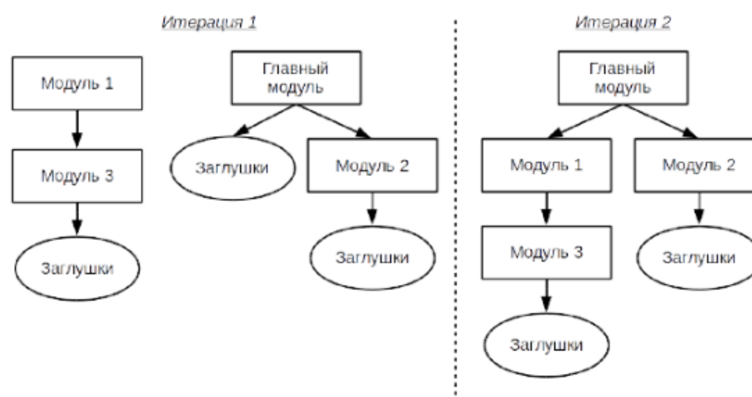

Рисунок 16 - Схема восходящей интеграции

"Сверху вниз" (нисходящая интеграция) предполагает последовательное присоединение модулей к главному модулю, с использованием тестовых дублёров (заглушек) в соответствии с рисунком 17.

Часто самым удобным способом становится использование смешанной интеграции (Mixed Integration), когда сначала сборка идёт по восходящей интеграции (модули группируются в блоки), а далее по нисходящей интеграции (блоки присоединяются к управляющему модулю). Смешанная интеграция очень удобна и позволяет избежать недостатков традиционных подходов [8].

Выделяют теорию «Большого взрыва» («Big Bang» Integration), которая предполагает одномоментную интеграцию всех модулей в одну систему и реализацию интеграционного тестирования. Данный подход существенно

сокращает временной фактор тестирования, однако может привести к мгновенному возникновению множества критических ошибок.

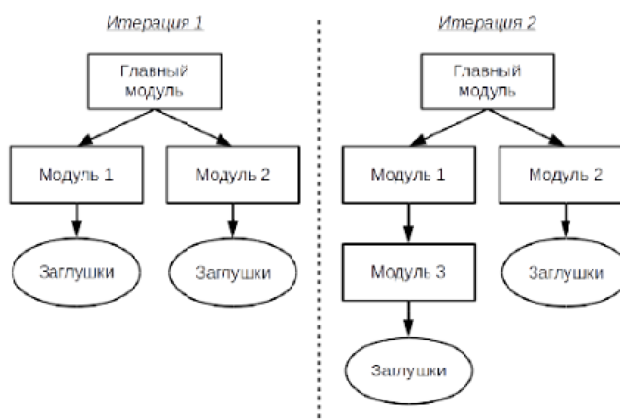

Рисунок 17 - Схема нисходящей интеграции

Таким образом, при выборе любого сценария тестирования микросервисного IT-продукта необходимо всестороннее тестирование как отдельных микросервисов, так и их совместного функционирования в комплексе с другими микросервисами. Эффективное тестирование микросервисного IT-продукта невозможно без использования тестовых дублёров, которые специально разрабатываются для удовлетворения необходимых требований недоступных модулей.

2.4 Методологии разработки программного обеспечения с интегрированным процессом тестирования

В данный момент в IT индустрии существует большое множество методологий разработки программного обеспечения, но самыми популярными считаются - каскадная модель (Waterfall Model) и различные вариации гибкой разработки (Agile).

Традиционной и самой консервативной моделью разработки и тестирования ПО является каскадная модель (Waterfall Model). Суть данной

модели состоит в соблюдении заранее спланированных этапов разработки и тестирования. Строгая иерархическая последовательность означает, что невозможно приступить к следующей стадии без успешного завершения предыдущей. Сейчас такую модель применяют редко и используют только в том случае, если все функциональные и нефункциональные требования и критерии заранее определены и изменению не подлежат.

Этапы разработки программного обеспечения по каскадной модели представлены на рисунке 18.

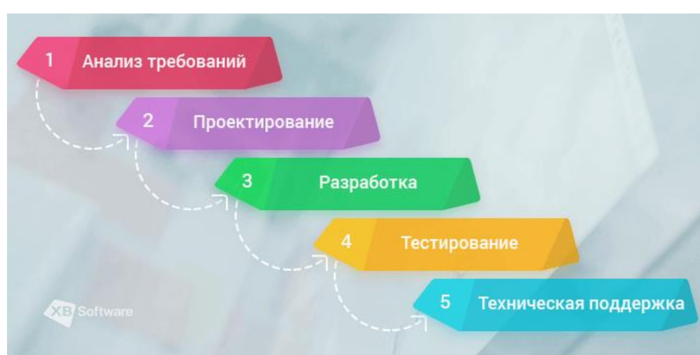

Рисунок 18 - Этапы разработки программного обеспечения по каскадной модели (Waterfall Model)

При работе с помощью каскадной модели разработки ПО юнит–тесты и интеграционные тесты не обязательны, так как процесс тестирования начинается уже после завершенного процесса разработки. Но такие виды тестирования позволяют выявить недостатки программного продукта, которые необходимо исправить до даты завершения проекта. Таким образом, получается неэффективный процесс возврата к процессу разработки, затем снова стадия тестирования и так несколько раз.

При использовании методологии гибкой разработки (Agile), процесс разработки программного обеспечения становится максимально гибким, разработка идёт малыми итерациями [10].

Цель применения методологий гибкой разработки - сделать новую функциональность как можно быстрее доступной потребителю, позволяя быстрее заказчику начать получать прибыль от разрабатываемого продукта.

Процесс методологии гибкой разработки представлен на рисунке 19.

Рисунок 19 - Процесс разработки программного обеспечения по методологии Agile

С целью повышения качества готового IT-продукта и ускорения процесса выполнения проектов были созданы следующие модели разработки программного обеспечения:

- разработка через тестирование (Test Driven Development, TDD);
- разработка на основе поведения (Behavior Driven Development, BDD);
- разработка через приемочное тестирование (Acceptance Test-Driven Development, ATDD).

Разработка через тестирование (Test Driven Development, TDD) предполагает создание разработчиком автоматизированных проверочных тестов для каждого модуля (функции) перед написанием самой программы. Тесты создаются на основе требований, указанных в спецификации и включают в себя проверку условий, которые могут или выполняться, или нет. Когда они выполняются, считается, что тест пройден. Прохождение теста подтверждает поведение, ожидаемое программистом.

Последовательность действий при разработке через тестирование следующая:

– написать тест для разрабатываемой функциональности;
– написать код, который предполагается тестировать;
– провести рефакторинг нового и старого кода.

Цикл разработки в соответствии с TDD представлен на рисунке 20.

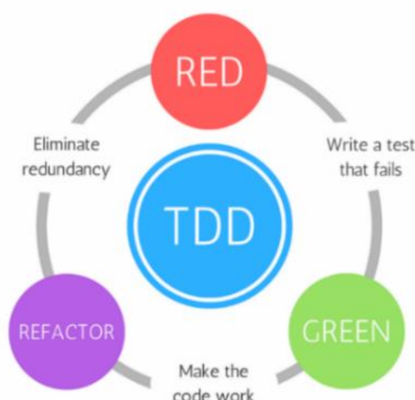

Рисунок 20 - Процесс разработки программного обеспечения по методологии TDD

В соответствии с рисунком 20 первая фаза (RED) заключается в том, что программист пишет тест и метод-заглушку на тестируемом классе, необходимые только для того, чтобы тест запустился. Функция-заглушка (тестовый дублёр) — функция, не выполняющая никакого определённого действия, возвращающая либо пустой результат, либо входные данные в неизменном виде. Заглушку следует разработать такую, чтобы тест не прошёл — тогда делается вывод о том, что тест корректно реагирует на ошибочный запрос.

На следующем этапе (GREEN) тестовый дублёр редактируется таким образом, чтобы тест начал работать правильно (при этом пишется минимальное количество кода).

Третья фаза (REFACTORING) подразумевает процесс рефакторинга - модификация внутренней структуры IT-продукта, которая не влечёт за собой изменения в её внешнем поведении. Цель такой модификации – усовершенствовать и упростить логический порядок кода разрабатываемого продукта. Данный этап обязателен при разработке программного обеспечения. Специалисты по разработке сначала пишут новый функционал и тестовые наборы для его проверки, а затем делают рефакторинг, усовершенствуя логичность выполнения операций. Автоматическое модульное тестирование позволяет убедиться, что рефакторинг не нарушил существующую функциональность.

Разработка на основе поведения (Behavior Driven Development, BDD) - техника разработки, при которой анализируется не результат выполнения какого-либо модуля, а та работа, которую он выполняет. BDD предполагает описание тестировщиком или бизнес-аналитиком пользовательских сценариев на естественном языке —на языке бизнеса. Написание тестов в виде целых предложений на языке бизнес-функций облегчает как заказчикам и аналитикам, так и разработчикам понимание целей создания IT-продукта и составление технической документации [7].

Цикл разработки ПО в соответствии с BDD представлен на рисунке 21.

Рисунок 21- Цикл разработки программного обеспечения по методологии BDD

Исходя из рисунка 21 каждая спецификация действует как входная точка в цикл разработки и описывает, как должна вести себя система с точки зрения пользователя (в пошаговом виде). Основное внимание уделяется не тестам модулей или объектов, а целям пользователей и пошаговым операциям, предпринимаемым ими для достижения этих целей. В данном виде тестирования тестовые дублеры уже не эмулируют работу программных модулей микросервиса. Вместо этого эмулируются более крупные объекты – целые системы итогового решения, а именно:

- подсистемы, с которыми программа будет интегрироваться;
- микросервисы, которые еще не готовы;
- любые элементы системы для проверки специальных сценариев (негативные, нагрузочное тестирование).

Разработка через приемочные тесты (Acceptance Test-Driven Development, ATDD). Цикл разработки в соответствии с ATDD представлен на рисунке 22.

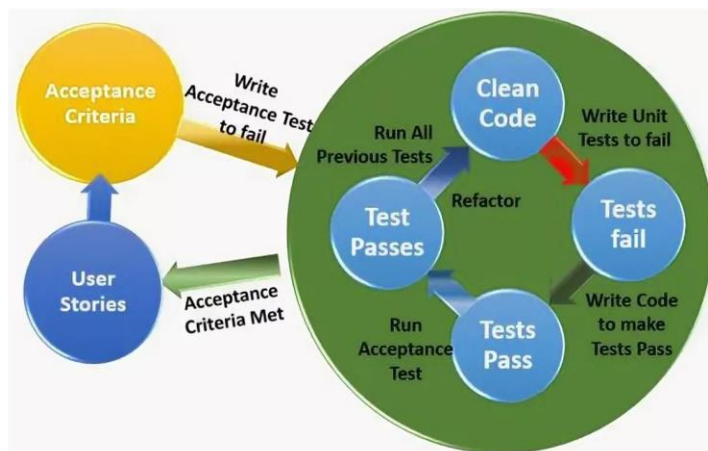

Рисунок 22 - Цикл разработки программного обеспечения по методологии ATDD

Идея методологии ATDD, изображённой на рисунке 22, заключается в том, что прежде, чем что-то разрабатывать, необходимо определить критерий

выполненного запроса и критерий того, что получен корректный ответ на запрос. Данные критерии позволяют на самых ранних стадиях понять, что именно при тестировании готового программного обеспечения следует считать корректным результатом.

Создаются сценарии на основе выявленных критериев, эти сценарии должны отражать поведение проектируемой системы. Когда все шаги сценария исполнены и конечный результат совпадает с ожидаемым, значит задача считается решённой. Комплект созданных сценариев и является приемочным тестированием.

Приёмочные тесты позволяют проверить разрабатываемый продукт путём использования пользовательского интерфейса, но они не отражают оптимальное внутреннее устройство и технические характеристики разработки. Используя данную методологию, все участники разработки (заказчики, специалисты по разработке и тестированию) вместе составляют техническую документацию на основе критических тестовых требований, что даёт возможность в кратчайшие сроки разрабатывать качественный IT-продукт.

Разработка по ATDD подразумевает, что все участники, вовлеченные в процесс создания IT-продукта, точно знают, какие требования к проекту и что необходимо сделать. Приемочные испытания составлены в соответствии с требованиями клиента и могут также использоваться при документировании требований. Команда разработчиков может использовать приемочные тесты, чтобы убедиться, что программа работает корректно и что система, над которой они работают, действительно решает задачи клиентов. Применение тестовых дублеров в данной методологии полностью идентично BDD методологии – также эмулируются крупные подсистемы итогового решения для проверки специальных и негативных тестов.

Каждая из рассмотренных методологий предоставляет уникальный набор возможностей командам разработки и ставит перед ними свои задачи, но все они нацелены на «тестирование как часть проектирования». Все

вышеперечисленные методологии разработки программного обеспечения так или иначе используют тестовые дублеры (testing doubles), но они особенно важны для BDD и ATDD методик. Одним из основных преимуществ тестовых дублёров является возможность убедиться, что основные сценарии использования системы работают и соответствуют всем функциональным и нефункциональным требованиям. Данный тестовый набор полезен не только при тестировании разрабатываемого продукта, но также и после установки на оборудование заказчика, с целью убедиться, что на его конфигурации оборудования все работает так, как и было запланировано. В процессе приемочного тестирования пользователями (User Acceptance Testing) тестовый дублер, входящий в состав выдаваемого программного продукта, является эффективной и зачастую необходимой опцией, позволяющей без задержек и дополнительных согласований проверить весь функционал программы на работоспособность с помощью уже написанных и отлаженных BDD и ATDD тестов.

Проанализировав методы проведения автоматизированного тестирования облачного IT-продукта, построенного на микросервисной архитектуре, можно сделать вывод, что выбор критериев и подмножества сгенерированных тестовых случаев или их ранжирование для выполнения может привести к более эффективному использованию выделенных ресурсов тестирования. Сгенерированный тестовый набор для тестирования программного продукта может иметь разные показатели качества и поэтому не имеет одинаковых сценариев для выполнения.

Определение свойств тестовых случаев и измерение их значения для выполнения можно считать основным ключом к решению задачи оптимизации процесса тестирования. Определение критических критериев и желаемых целей зависит от нескольких факторов, таких как размер тестов, их сложность, разнообразие, а также процедура тестирования. Количество прогнозируемых повторных запусков тест-кейсов является одним из ключевых факторов при обосновании использования автоматизированного тестирования.

В результате изучения новых методологий разработки программных продуктов можно сделать вывод о том, что методология BDD позволяет сделать процесс тестирования более удобным, дешевым и полезным, повышая при этом качество готового IT-решения.

Использование тестовых дублёров как части готового решения, построенного на микросервисной архитектуре, позволяет изолированно разрабатывать и тестировать сервисы, построенные вокруг бизнес-функций, они могут быть развернуты независимо через полностью автоматизированный механизм развертывания. Эти микросервисы могут быть написаны на разных языках и могут использовать разные технологии хранения данных. Необходимость непрерывного интеграционного тестирования разработанных или изменённых микросервисов требует всестороннего комплексного подхода к автоматизации тестирования.

Процесс тестирования должен быть адаптирован к изменению адресации, а также к масштабированию микросервисов в зависимости от текущей загрузки.

Глава 3 Апробация методов проведения автоматизированного тестирования облачного IT-продукта, построенного на микросервисной архитектуре

3.1 Информационные модели автоматизированного тестирования облачного IT-продукта, построенного на микросервисной архитектуре

На этапе проектирования IT-продукта целесообразно создавать информационные модели, отражающие общие принципы и алгоритмы работы создаваемой системы. Одной из таких информационных моделей является диаграмма вариантов использования, на которой представляется общая концептуальная схема разрабатываемой системы. Выделяют следующие цели составления диаграммы вариантов использования:

- на самых ранних стадиях проектирования определить границы создаваемой системы с учётом её назначения;
- установить функциональные требования;
- спроектировать стартовую концептуальную модель системы с целью её усовершенствования на основе физических и логических связей;
- создать техническую документацию для более предметного общения разработчиков с заказчиками.

Составленная диаграмма показывает, какие варианты использования разрабатываемого IT-продукта должны быть доступны. Кроме того, определяется взаимодействие с системой извне. Лицами, осуществляющими такое взаимодействие, могут считаться любые объекты, субъекты или системы. Они представляются как внешние сущности или актёры, инициирующими определённое поведение создаваемой системы.

Диаграмма вариантов использования тестирования микросервиса с использованием тестового дублёра представлена на рисунке 23. Варианты использования микросервиса: формирование запроса на чтение данных,

формирование запроса на модификацию данных, обработка ответов. Варианты использования тестового дублёра: обработка запроса, формирование ответов на запросы.

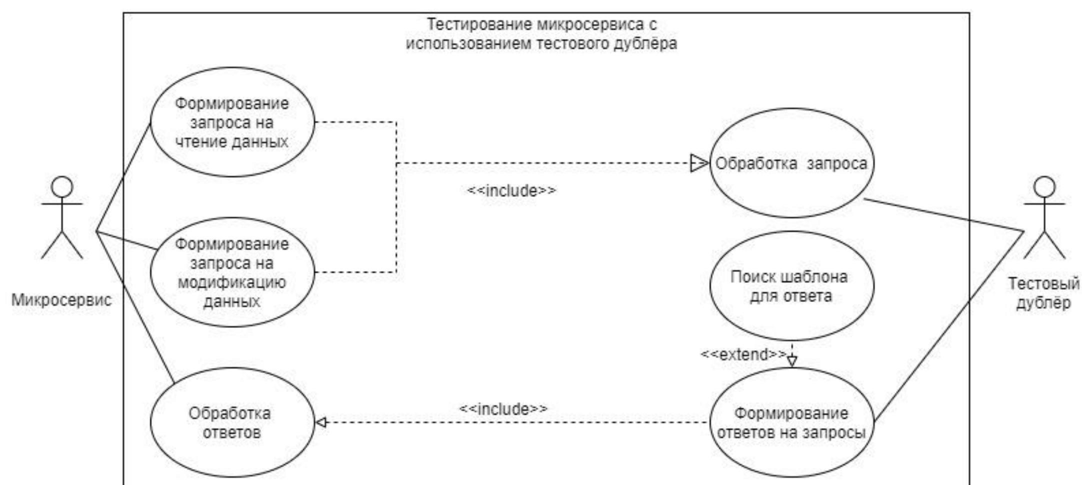

Рисунок 23 - Диаграмма вариантов использования процесса тестирования микросервиса

На рисунке 23 представлен микросервис, который взаимодействует с тестовым дублёром. При формировании запроса на чтение данных или формирование запроса на модификацию данных ВСЕГДА следует Обработка запроса. А формирование ответов на запросы ВСЕГДА влечёт за собой обработку ответов. Поэтому такое отношение носит характер «включение» или <<include>>.

При формировании ответов на запросы появляется вариант поиска шаблона для ответа. Но, так как шаблон может быть найден НЕ ВСЕГДА, отношение носит характер «расширение» или <<extend>>.

Диаграмма классов – это диаграмма, на которой представлено графическое изображение набора элементов, представленное в виде связного графа вершин (сущностей) и путей (связей). Диаграммы представляются как одна из форм статического описания состояния системы с точки зрения ее проектирования, отражая ее структуру. При этом диаграмма классов не

показывает действия объектов в динамике, здесь отображаются только статическое состояние классов, интерфейсов и их отношений.

Пример диаграммы классов взаимодействия запросов и ответов с использованием тестового дублёра изображена на рисунке 24:

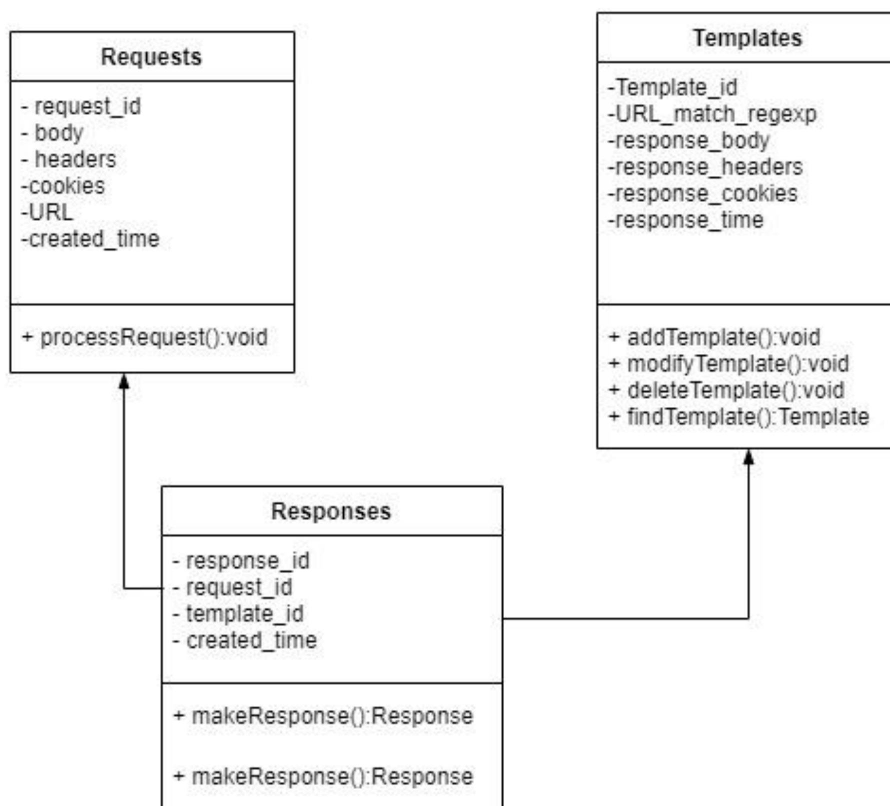

Рисунок 24 - Диаграмма классов тестового дублёра

На рисунке 24 в диаграмме классов выделены 3 класса: Запросы (Requests), Ответы (Responses), Шаблоны (Templates). В каждом классе выделены атрибуты-свойства класса и методы- операции класса. Взаимодействие между классами носит характер ассоциации, так как ни один класс не включает в себя другой, и класс Responses ссылается на класс Templates и на класс Requests (атрибуты класса Responses включают response_id и request_id). Таким образом, тестовый дублёр на каждый входящий запрос ищет шаблон ответа, и если шаблон не найден, то должна вернуться ошибка ответа.

Итак, на основании диаграммы вариантов использования и диаграммы классов процесса тестирования микросервиса с использованием тестового дублёра, становится очевидно, что проектируемая программная система взаимодействует с внешней системой, работу которой имитирует тестовый дублёр. На каждый запрос тестовый дублёр ищет шаблон для ответа, но если шаблон не найден, то должна вернуться ошибка ответа.

3.2 Алгоритм тестирования облачного IT-продукта, построенного на микросервисной архитектуре и разработанного на основе методологии BDD

Методологию BDD можно описать как более удобный для пользователя сценарий тестирования с использованием естественного языка, чтобы все заинтересованные стороны могли лучше понять процесс тестирования. В соответствии с методологией BDD тест-кейсы в приемочном тесте должны охватывать все функции программы по различным сценариям пользователей.

BDD-методология в IT-продукте, построенном на микросервисной архитектуре, — это сотрудничество клиента, разработчиков и тестировщиков. BDD — это разработка, которая учитывает как технические интересы, так и бизнес-требования. Такой подход обычно применяется для описания работы интерфейсов приложений, а так как микросервисы — детали реализации системы, то BDD прекрасно походит и для разработки микросервисов.

BDD методология подразумевает принцип, состоящий в том, что перед написанием тест-кейса следует изначально определить и сформулировать на предметно-ориентированном языке результат от проектируемой функциональности. Затем все составленные тесты перестраиваются специалистами в BDD-сценарий тестирования.

В соответствии с методологией BDD необходимо определить следующие пункты:

– с чего начинается процесс тестирования;

– какую функциональность нужно тестировать, а какую нет;
– какое количество проверок совершается одномоментно;
– какой тест является проверкой;
– в каком случае тест считается незавершённым или результат некорректным.

В соответствии с вышеизложенным, данная методология предполагает, что имена тестов должны представлять из себя целые предложения, которые начинаются с глагола в сослагательном наклонении и отражают суть бизнес-цели. Бизнес-цель – это результат, который должен быть получен пользователем в результате выполнения определённых шагов.

BDD-сценарий состоит из предложений, построенных из некоторых элементов:

– конструкция Given определяет начальные условия совершения операции (определяет то, что изначально «дано»). Например, окно ввода команды, поисковая строка и т.д;
– слово When определяет действия, совершаемые пользователем или подсистемой и инициирующими процесс тестирования функции (отвечает на вопрос «когда?»);
– фраза Then описывает ожидаемый результат тестирования (например, переход на другую страницу или выборка по заданным критериям).

На рисунке 25 представлен пример сценария входа в систему тестируемой системы:

```
Feature: To check that home page has loaded in Sistem Informasi SPI
 Skenario: To check that home page has loaded
 Given I am on Sistem Informasi SPI
 When I click on the Login Link
 And input username and password
 Then I should be on the Home SPI page
```

Рисунок 25 - Вход в систему тестируемой системы в соответствии с методологией BDD

В соответствии с рисунком 25 сценарий входа в тестируемую систему описывает, что, пользователь находится на сайте «System Informatsi SPI» (GIVEN). При нажатии на клавишу «Авторизоваться» (WHEN) и вводе имени пользователя и пароля (AND) попадает на страницу Home SPI page (THEN).

Подход к автоматизированному тестированию по методологии BDD значительно отличается от стандартного. Сравнение алгоритмов подхода к автоматизированному тестированию представлено на рисунке 26.

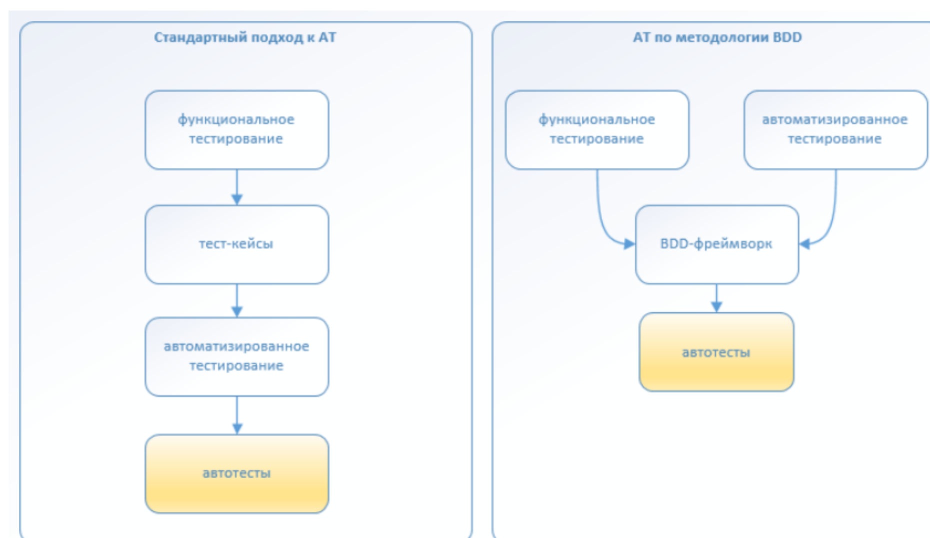

Рисунок 26 - Подходы к автоматизированному тестированию

В соответствии с рисунком 26 очевидно, что при автоматизации тестирования в соответствии с методологией BDD автотесты создаются одновременно при участии и функциональных тестировщиков, и специалистов по тестированию, что позволяет сэкономить рабочее время и бюджет IT-проекта. Кроме того, к данному процессу планирования тестирования целесообразно привлекать менеджеров IT-продукта с бизнес-аналитиками для того, чтобы они указывали, какому функционалу необходимо уделить особое внимание и автоматизировать в первую очередь с точки зрения бизнес-значимости функционального критерия.

Особое значение следует уделить приёмочному тестированию. Схема организации приёмочного тестирования по методологии BDD представлена на рисунке 27.

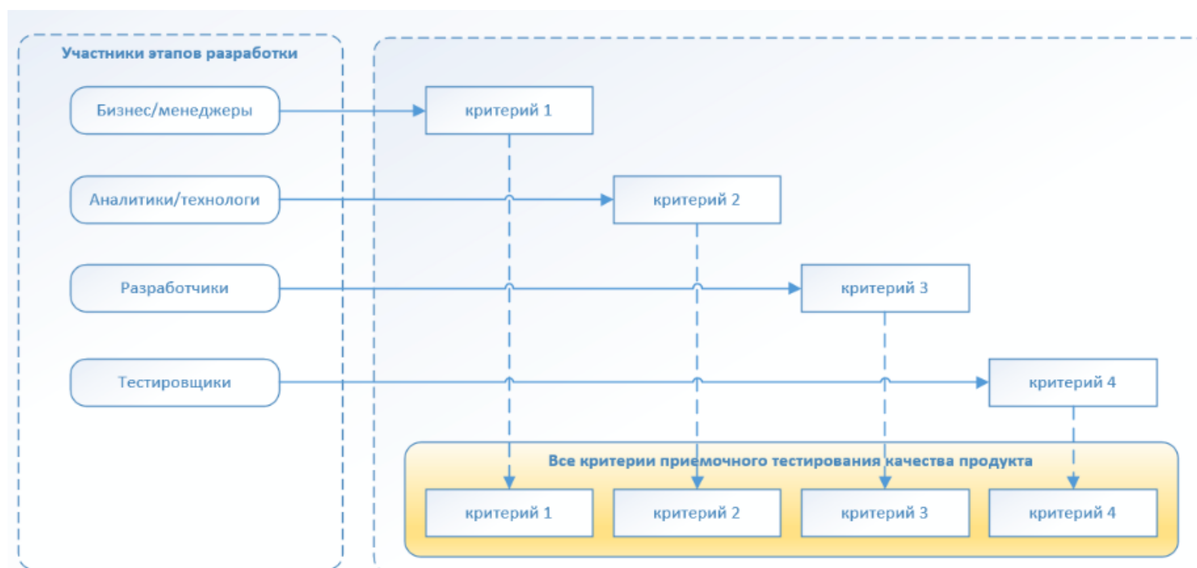

Рисунок 27 - Формирование приёмочного тестирования по методологии BDD

Анализ рисунка 27 показывает, что полный набор критериев качества IT-продукта формируется не только тестировщиками, как это происходит зачастую, а всеми участниками процесса разработки. Данный подход наиболее полезен и эффективен с точки зрения автоматизации, он дает быстрое и очень эффективное составление BDD-сценария тестирования.

Итак, сценарии работы системы, написанные в соответствии с методологией BDD, понятны и разработчикам, и клиентам (пользователям системы).

3.3 Инструментальные средства для тестирования облачного IT-продукта, построенного на микросервисной архитектуре и разработанного на основе методологии BDD

Спецификация, составленная на основе пользовательского поведения, требует использования ограниченного набора предложений, синтаксис которых ограничен. В связи с данным фактором инструментальные средства, поддерживающие BDD, строятся в соответствии со следующими особенностями:

- парсер разбивает спецификацию на некоторые части, например, по ключевым словам языка Gherkin. В результате формируется набор предложений, построенных в соответствии с BDD-синтаксисом;
- каждое предложение определяет один шаг в тестировании;
- процессор регулярных выражений осуществляет захват той части предложения, в которой содержатся входные параметры, инициирующими операцию. Остальная часть предложения не используется, она необходима только для понимания происходящего события. Захваченные параметры конвертируются и отправляются на вход к определённой операции.

В соответствии с данными принципами работают такие программные средства, как Cucumber, JBehave и JGiven.

Cucumber – кроссплатформенное приложение для автоматизации тестирования на основе подхода BDD. Это специально разработанная библиотека для тестирования, которая позволяет разрабатывать тестовые наборы на естественном языке с последующей конвертацией в текстовый файл с расширением .feature. В Cucumber для написания тестов используется язык Gherkin, который определяет структуру теста и набор ключевых слов. Шаблоном составления сценария тестирования являются ключевые команды, такие как Given, When, Then.

JBehave – это приложение, написанное на языке Java, помогает автоматизировать приемочные тесты и поддерживает процесс разработки в BDD стиле. Бизнес-требования и их приемочные критерии описываются в виде пользовательских сценариев. Далее эти сценарии читаются построчно как обычный текстовый файл. Для того, чтобы сопоставлять команды с

предложением Gherkin, используются Java-аннотации, которые представлены во фреймворке JBehave. JGiven — это удобный инструмент BDD для Java. Разработчики пишут сценарии на простом языке Java, используя свободный API для конкретного домена. JGiven генерирует отчеты, которые могут быть прочитаны экспертами в предметной области.

Достоинства и недостатки рассмотренных инструментальных средств представлены в таблице 1.

Таблица 1 - Сравнительный анализ инструментальных средств, работающих в соответствии с методологией BDD

Приложение	Достоинства	Недостатки
Cucumber-JVM	1. Подробная документация 2. Поддержка множества языков при описании сценариев 3. По результатам прохождения тестов генерируется отчет в достаточно детальной и удобочитаемой форме 4. Поддерживает функции	Не поддерживает параллельные тесты JUnit. Однако будет работать с параллельными сборками Maven 3
JBehave	1. Открытый исходный код 2. Имеет множество дополнительных конфигураций для точной настройки инструмента BDD в соответствии с вашими предпочтениями	1. Поддерживает только истории, а не функции 2. отсутствуют ключевые особенности языка Gherkin, такие как backgrounds, doc strings и tags
JGiven	1. Работает со всеми существующими IDE и инструментами сборки для Java 2. Генерирует HTML-отчеты, которые могут быть легко прочитаны и понятны 3. Поддержка параметризованных шагов 4. Поддержка тегов для организации сценариев	Больше подходит для модульных и интеграционных тестов, чем для автоматических системных и регрессионных тестов.

Таким образом, все рассмотренные инструменты для тестирования работают в соответствии с методологией BDD, являются Java-приложениями, в которых для написания сценариев тестирования используются шаги Given, When, Then. Но самым распространённым и удобным с точки зрения

доступности технической документации, а также для получения сгенерированных и понятных отчетов о тестировании целесообразно использовать Cucumber-JVM. В данной библиотеке каждому шагу соответствует аннотация, которая с помощью регулярного выражения связывает метод, над которым объявлена, со строкой в текстовом описании сценария. Этапы тестирования формируются в сценарии (Scenario), которые описывают определённую функциональность (Feature).

Структура проекта с использованием библиотеки Cucumber-JVM представлена на рисунке 28.

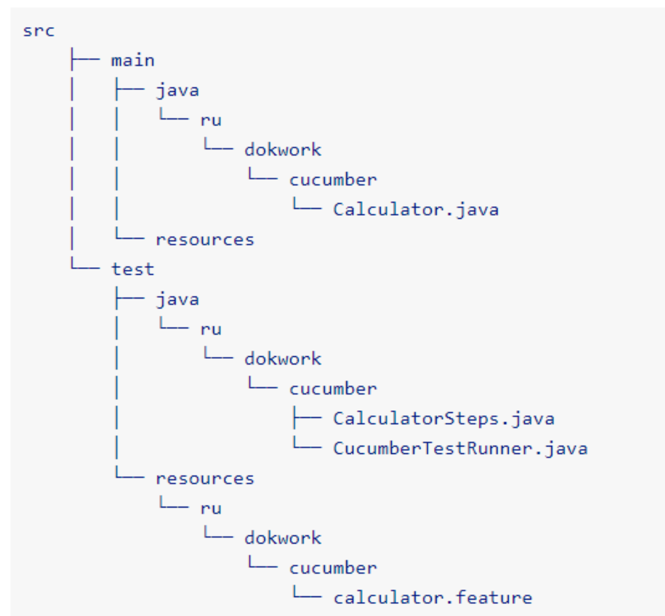

Рисунок 28 – Структура проекта с использованием библиотеки Cucumber-JVM

Итак, на основании рисунка 28 можно сделать вывод, что основной файл - это текстовый.feature файл с описанием сценариев и .java файлы с описанием реализации шагов выполнения сценариев. Никаких обязательств, кроме расширения, на имена этих файлов не накладывается. Но отсюда следует еще один момент - шаг, описанный в одном текстовом файле, будет использоваться во всех java-реализациях.

Таким образом, можно сделать вывод о том, что существует множество инструментальных средств для тестирования, но окончательный выбор зависит от методологии разработки, языка программирования и удобства использования.

3.4. Сценарий автоматизированного тестирования облачного IT-продукта, построенного на микросервисной архитектуре, с использованием тестовых дублёров

Для составления сценария тестирования рассмотрим этап тестирования любого Интернет-магазина. Интернет-магазин взаимодействует с тремя микросервисами: сервис доставки товаров, сервис оплаты и сервис уведомлений о текущем состоянии заказа. Моделируемая система взаимодействия микросервисов представлена на рисунке 29.

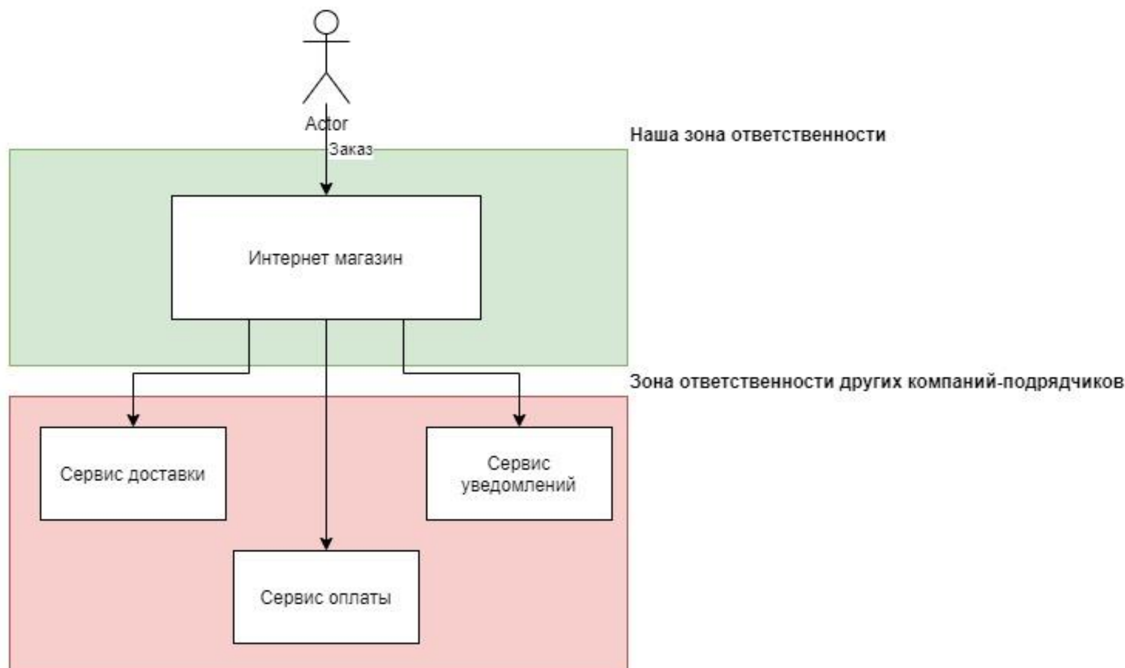

Рисунок 29 – Моделируемая система взаимодействия Интернет-магазина с сотрудничающими микросервисами

В соответствии с рисунком 29 нашей зоной ответственности является только работа Интернет-магазина. Но для тестирования его работы необходимо провести интеграционное тестирование с внешними сотрудничающими микросервисами, за которые отвечает сторонняя компания.

Для нахождения наибольшего количества ошибок при тестировании смоделированной системы необходимо поэтапно проверять взаимодействие Интернет-магазина с каждой внешней подсистемой. Для этого целесообразно заглушить все связанные микросервисы, кроме тестируемых, чтобы в область тестирования попало минимально допустимое количество операций.

Диаграмма последовательности взаимодействия Интернет-магазина с внешними микросервисами представлена на рисунке 30.

На рисунке 30 представлен жизненный цикл смоделированного Интернет-магазина и его взаимодействие с внешними сотрудничающими микросервисами (система оплаты, система доставки, система уведомлений). Для того, чтобы сфокусироваться только на тестировании определённых микросервисов, необходимо заглушить все остальные подсистемы. Таким образом, чтобы проверить взаимодействие Интернет-магазина с микросервисом «система оплаты», нужно заглушить «систему доставки» и «систему уведомлений». Тогда может возникнуть необходимость в использовании тестового дублёра (сервиса-заглушки) для имитации комплексного взаимодействия всей системы.

Этот способ позволит изолировать тестируемый микросервис, но гарантирует то, что другие микросервисы доступны и возможно настроить подключение тестируемой подсистемы к сотрудничающим сервисам-заглушкам. Далее заглушки конфигурируются на отправку обратных ответов с целью сымитировать взаимодействие.

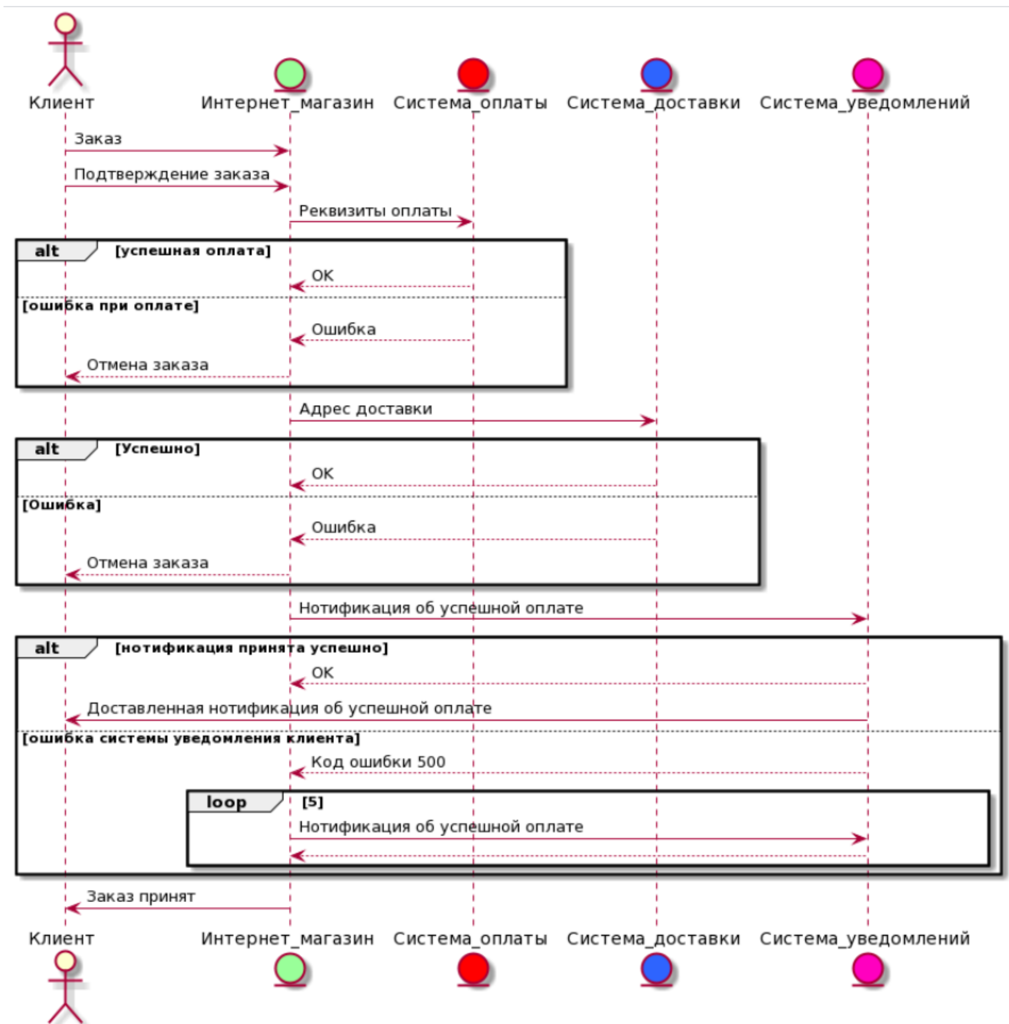

Рисунок 30 – Диаграмма последовательности взаимодействия Интернет-магазина с сотрудничающими микросервисами

В случае успешного оформления и подтверждения заказа Интернет-магазин посылает запрос микросервису «сервис оплаты». Ответным сообщением может быть как подтверждение оплаты («ОК»), так и ошибка при оплате («ошибка»). В результате ошибки происходит отмена заказа, и взаимодействия с системой доставки и системой уведомлений не происходит. Для интеграционного тестирования Интернет-магазина с микросервисом «система доставки» необходимо получить ответ от «сервиса оплаты». Имитацию его деятельности выполняет тестовый дублёр, посылая ответные сообщения об успешной оплате заказа. Аналогично происходит настройка

тестового дублёра для «системы доставки» для тестирования взаимодействия Интернет-магазина с внешним микросервисом «система уведомлений».

Для тестирования исследуемой системы необходим следующий набор тест-кейсов:

- тестирование успешного заказа. Пользователь выполняет шаги: создаёт заказ, вводит данные карты для оплаты, подтверждает заказ. Ожидаемый результат поведения тестируемой системы:
  а) тестовый дублер платежной подсистемы получил запрос;
  б) проверяем запрос, полученный тестовым дублером платежной подсистемы на соответствие спецификации;
  в) проверяем, что тестовый дублер подсистемы доставки получил запрос;
  г) проверяем запрос, полученный тестовым дублером подсистемы доставки на соответствие спецификации "Успешная доставка";
  д) проверяем, что тестовый дублер подсистемы уведомлений получил запрос;
  е) проверяем запрос, полученный тестовым дублером подсистемы уведомлений клиента на соответствие спецификации "Успешная оплата";
  ж) проверяем статус заказа = "Подтвержден".
- проверка ввода неверных данных при оплате заказа. Пользователь выполняет шаги: создаёт заказ, вводит неверные данные карты для оплаты, подтверждает заказ. Ожидаемый результат поведения тестируемой системы:
  а) проверяем, что тестовый дублер подсистемы уведомлений получил запрос;
  б) проверяем запрос, полученный тестовым дублером подсистемы уведомлений клиента на соответствие спецификации "Ошибка оплаты";

в) проверяем, что тестовый дублер подсистемы доставки не получил запрос;
г) проверяем статус заказа = «В процессе»;
д) проверяем ошибку = "Введены неверные данные для оплаты".

– проверка недоступности доставки. Пользователь выполняет шаги: создаёт заказ, вводит верные данные карты для оплаты, подтверждает заказ. Ожидаемый результат поведения тестируемой системы:
  а) проверяем, что тестовый дублер подсистемы уведомлений получил запрос;
  б) проверяем запрос, полученный тестовым дублером подсистемы уведомлений клиента на соответствие спецификации "Успешная оплата";
  в) проверяем, что тестовый дублер подсистемы доставки получил запрос;
  г) проверяем, что тестовый дублер подсистемы доставки ответил ошибкой;
  д) проверяем статус заказа = "Отменен";
  е) проверяем ошибку = "Невозможно назначить доставку";
– проверка работы сервиса уведомлений (недоступность уведомлений). Пользователь выполняет шаги: создаёт заказ, вводит верные данные карты для оплаты, подтверждает заказ. Ожидаемый результат поведения тестируемой системы:
  а) проверяем, что тестовый дублер подсистемы уведомлений получил запрос;
  б) проверяем запрос, полученный тестовым дублером подсистемы уведомлений клиента на соответствие спецификации "Успешная оплата";
  в) проверяем, что тестовый дублер подсистемы уведомлений ответил ошибкой;
  г) проверяем статус заказа = "Подтвержден";

д) проверяем, что в поле «Ошибка» у заказа выходит ошибка «Сбой системы уведомлений»;
е) ждем 5 минут, проверяем, что тестовый дублер подсистемы уведомлений клиента получил всего 5 запросов.

Идея рассмотренных тест-кейсов состоит в поиске максимального количества ошибок при интеграционном тестировании Интернет-магазина с внешними сотрудничающими микросервисами. Следует отметить, что при тестировании сервиса уведомлений важным моментом является настройка времени ожидания (тест-кейс 1.4, шаг № 6, для примера мы взяли время ожидания 5 минут). Для правильной интеграции системы с нижестоящими микросервисами данный этап является ключевым. Если время ожидания сделать слишком маленьким, то потенциально работоспособный сервис можно посчитать неработающим. В обратном случае, если настроить слишком большое время ожидания, то вся система может «зависать» на очень долгое время, неприемлемое для комфортной работы. Идеальным вариантом является настройка времени ожидания для всех вызовов, адресуемых за пределы своей системы. Для проведения автоматизированного тестирования смоделированной системы был использован Cucumber-JVM. В связи с тем, что шаблоном написания сценария тестирования являются шаги, такие как Given, When, Then, все тест-кейсы, рассмотренные в таблице 1, необходимо представить в соответствующем виде. В приложении А представлены созданные тест-кейсы в библиотеке Cucumber-JVM.

Итак, после запуска всех четырёх тест-кейсов, представленных в Приложении А, приложение Cucumber-JVM сообщает о корректной работе всей системы. Успешное выполнение всех тестов представлено на рисунке 31.

На основании рисунка 31 можно сделать вывод о том, что все четыре сценария пройдены успешно, выполнено 40 шагов последовательного тестирования смоделированной системы.

```
✔ Done: Scenarios 4 of 4  (12 s 529 ms)

4 Scenarios (4 passed)
40 Steps (40 passed)
```

Рисунок 31 – Результат успешного завершения тестирования смоделированной системы в библиотеке Cucumber-JVM

Таким образом, проведение тестирования в соответствии с методологией BDD на основе пользовательских сценариев является удобным и понятным как для заказчика, так и для IT-специалиста. Изолированность каждого внешнего микросервиса позволяет поэтапно тестировать взаимодействие смоделированной системы с каждым сотрудничающим компонентом. Тестовые дублёры, имитирующие ответные вызовы, позволяют провести тестирование своей системы наиболее комплексно и оперативно.

Итак, в результате проведения апробации методов проведения автоматизированного тестирования облачного IT-продукта, построенного на микросервисной архитектуре, можно сделать вывод о том, что практическое составление BDD-сценария проведения тестирования программного продукта происходит при совместном участии заказчика, разработчиков и тестировщиков. В результате определяется полный набор критериев качества готового IT-продукта, составляется набор тест-кейсов, написанных на языке BDD-синтаксиса. Следующим важным этапом является определение инструментального средства, работающего в соответствии с методологией BDD. После проведения тестирования должен быть сгенерирован понятный отчёт о результатах тестирования в соответствии с составленным сценарием, а также общем времени проведения тестирования.

Глава 4 Анализ эффективности использования тестовых дублёров как части IT-продукта

4.1. Преимущества поставки тестовых дублёров как части IT-продукта

В результате проведения автоматизированного тестирования облачного IT-продукта, построенного на микросервисной архитектуре, с использованием тестовых дублёров (сервисов-заглушек) становится очевидно, что такой способ позволяет провести тестирование наиболее комплексно.

Но комплексное тестирование полностью готового IT-продукта может быть необходимым не только во время разработки, но и во время внедрения, установки, и сопровождения готового решения у заказчика.

Таким образом, поставка тестовых дублёров как части готового IT-продукта имеет следующие преимущества:

– позволяет проверить правильность установки и настройки IT-продукта на оборудовании заказчика. Дело в том, что тестовый стенд, на котором проводились разработка и тестирование функциональных возможностей, может существенно отличаться от оборудования и конфигурации у заказчика. Различия могут быть как в модели конкретного оборудования (серверов, сетевого оборудования), так и в конфигурации сети (наличие дополнительных систем безопасности и т.д.). Используя BDD тесты, созданные на этапе тестирования IT-продукта, мы сможем убедиться, что в рамках нашей зоны ответственности все подсистемы работают согласно требованиям заказчика, и интеграция со всеми внешними системами происходит строго по согласованной спецификации.
– позволяет проверить работу всей IT-системы в сложных условиях (нагрузочное тестирование, пограничные ситуации). Во время приемочного тестирования заказчик проверяет систему на

соответствие всем требованиям согласно спецификации, в том числе обязательным является проведение нагрузочного тестирования. В связи с тем, что это чрезвычайно трудозатратно, взаимодействие с другими подсистемами может быть затруднено (в случае, если они не готовы), или вовсе невозможно (работа с платежными подсистемами или подсистемами уведомления клиента по почте или телефону всегда оплачивается отдельно). В таких случаях целесообразно использовать тестовых дублеров.

– даёт возможность получить дополнительную выгоду. Заказчик заинтересован самостоятельно проверить систему на этапе приемочного тестирования, но перед ним встают те же самые проблемы, что и у команды тестирования во время разработки IT-продукта. Имея готовое решение для комплексного тестирования, становится возможным продать его заказчику, сократив таким образом его время на разработку собственной автоматизированной системы проверки, без которой не обойтись при использовании Agile-методологии и частого обновления рабочих серверов. Даже если заказчик и не заинтересован в использовании нашего сценария тестирования готового IT-продукта, тестовые дублеры могут быть полезны уже для его собственного алгоритма проверки IT-системы.
– позволяет существенно упростить работу команды поддержки и внедрения IT-продукта. При внедрении IT-продукта могут возникать неожиданные непредвиденные обстоятельства: например, нарушения требований спецификации у соседних внешних систем, с которыми IT-продукт интегрируется. В таком случае наличие готовых сконфигурированных тестовых дублеров облегчает исследование данных инцидентов, сокращая расследование и исправление выявленных дефектов, повышая удовлетворенность клиента купленным продуктом.

Итак, рассмотрев преимущества использования и поставки тестовых дублёров как части готового IT-продукта, следует отметить один недостаток: разработка, установка и конфигурация тестового дублёра занимает определённое время. Следовательно, необходимо учитывать трудозатраты, связанные с использованием тестовых дублёров.

4.2 Оценка трудозатрат на установку и конфигурацию тестовых дублеров на оборудовании заказчика

Важным этапом в разработке и тестировании любого IT-продукта является оценка стоимости работ. Целесообразно провести примерную оценку стоимости разработки и внедрения тестовых дублеров на оборудовании заказчика, если тестовые дублёры не были включены в IT-продукт на этапе составления спецификации и сметы. Для оценки стоимости внедрения тестовых дублеров на оборудовании заказчика целесообразно использовать трёхточечный метод или взвешенная трёхточечная оценка.

Трёхточечный метод (PERT-оценка) — способ оценки времени и усилий в деятельности по управлению проектами. В соответствии с трёхточечным методом вычисление происходит по следующей формуле:

$$A = \frac{\mathrm{B}est\ case + 4 * \mathrm{Likely\ case} + \mathrm{W}orst\ case}{6} \tag{1}$$

где A – взвешенная оценка (дни);

Best case - оптимистичная оценка (дни);

Likely case - нормальная оценка или наиболее вероятное время исполнения (дни);

Worst case - пессимистичная оценка (дни).

Данный способ расширяет двукратный метод, который учитывает только наилучший и наихудший временной прогноз. Появляется дополнительный критерий «наиболее вероятное время исполнения».

Объём работы по внедрению тестовых дублеров на оборудовании заказчика включает следующие виды работ:

– планирование работы с заказчиком – включает в себя обсуждение плана работ, их согласование со всеми структурами заказчика;
– установка тестового дублера – сама по себе установка необходимых микросервисов, проверка их корректной функциональности;
– конфигурирование тестового дублера – настройка на эмуляцию требуемого поведения. Зависит от самой задачи, простая настройка для функциональной задачи, и нетривиальная для нагрузочного сценария;
– непредвиденные обстоятельства – существует неопределённость относительно точного содержания всех элементов в оценке, как будет выполнена работа, какие будут условия для работы, поэтому для учета данных рисков делается поправка, на основе прошлого опыта оценщика – средняя поправка определяется в 30%.

Исходная информация для расчёта взвешенной оценки представлены в таблице 2.

Таблица 2– Исходные данные для расчёта взвешенной оценки трёхточечным методом

Вид работы	Оптимистичная оценка (дни);	Нормальная оценка (дни);	Пессимистичная оценка (дни);	Взвешенная оценка (дни);
Планирование работы с заказчиком	1	2	3	2
Установка тестового дублера	1	2	3	2
Конфигурирование тестового дублера	1	3	5	3
Непредвиденные обстоятельства, 30%			0,3	2,1
Итого	3	7	11,3	9,1

Итак, на основании данных таблицы 2, в среднем 9 дней может потребоваться для внедрения тестового дублера на оборудовании заказчика. Кроме того, каждый раз, когда возникнет необходимость в проверке нового сценария работы IT-системы, операцию конфигурирования тестового дублёра придется повторять, а это дополнительные временные и денежные затраты.

Аналитики IT-рекрутингового агентства ATLAS составили таблицу средней заработной платы IT-специалистов в России и зарубежных странах, данные их исследований представлены на рисунке 32.

Наименование	Минимум (руб.)	Средний уровень (руб.)	Максимум (руб.)
Программист JavaScript	92250	146000	212500
HTML-верстальщик	44000	58500	83750
Программист PHP	90500	140250	204750
Программист Java	97500	162500	245000
Программист IOS (Swift / ObC)	112500	175000	275000
Программист 1С	95000	150000	197500
Системный администратор	52500	72500	107500

Рисунок 32 – Средняя заработная плата IT-специалистов в России за 1 календарный месяц

Итак, если взять среднюю заработную плату за 1 календарный месяц (22 рабочих дня) Java-программиста в России, равную 162500 рублей, а установка тестового дублера занимает в среднем 9 дней, получается, что итоговая стоимость работы по установке и конфигурированию тестового дублёра составляет 162500*9/22=66477 рублей, не учитывая командировочные и прочие расходы.

Без использования тестового дублёра для тестирования разрабатываемой системы необходимо ждать, когда все сотрудничающие микросервисы будут разработаны и сконфигурированы на интеграцию, этот процесс может занимать несколько месяцев. Кроме того, после каждого обновления любого из микросервисов потребуются дополнительные трудозатраты на поддержку работоспособности всей системы.

Таким образом, поставка тестового дублера как части комплексного IT-продукта позволяет не только упростить процесс внедрения, поддержки и тестирования IT-продукта у заказчика, но и облегчает самому заказчику процесс тестирования, а также открывает возможности дополнительного заработка на продаже собственных инструментов тестирования заказчику. В результате предварительного согласования с заказчиком о разработке IT-продукта с созданием тестового дублёра итоговая стоимость IT-решения для компании-разработчика увеличивается несущественно. Но продать заказчику такой IT-продукт можно существенно дороже, аргументируя тем, что это комплексная IT-система, настроенная и протестированная со связанными подсистемами.

Кроме того, с помощью тестовых дублёров как части итогового IT-решения можно проверять работоспособность системы на всех этапах разработки и тестирования, даже если сторонние системы еще не готовы.

Тестовые дублёры, имитирующие ответные вызовы, позволяют провести тестирование своей системы наиболее комплексно и оперативно. Без них тестирование совместимости с сотрудничающими сервисами становится непредсказуемым и более дорогим. Изолированность каждого внешнего микросервиса позволяет поэтапно протестировать взаимодействие смоделированной системы с каждым сотрудничающим компонентом.

Комплекс всех рассмотренных мер ускоряет процесс внедрения и сроки приемочного тестирования IT-продукта, повышая качество готового IT-решения, а также лояльность и удовлетворённость заказчика.

Подтвердилась гипотеза о возможности создания тестового дублёра, способствующего повышению качества тестирования на всех этапах разработки и тестирования IT-продукта.

Заключение

В ходе проведенного исследования были рассмотрены основные методы и особенности автоматизированного тестирования IT-продукта, построенного на микросервисной архитектуре. Выявлено, что использование облачного пространства предоставляет удобный и быстрый доступ к ресурсам, что позволяет с минимальными управленческими усилиями или действиями по взаимодействию с поставщиком услуг разработать и протестировать IT-продукт.

При проведении исследований в диссертации получены следующие теоретические и прикладные результаты.

- в результате рассмотрения теоретических основ существующих средств и методов автоматизированного тестирования IT-продукта были установлены основные преимущества и недостатки, а также выделены особенности проведения автоматизированного тестирования облачного IT-продукт, построенного на микросервисной архитектуре. Выделены критерии эффективности автоматизированного тестирования, а также критерии формирования тестовых наборов;
- анализ методов проведения автоматизированного тестирования позволил выделить методы и инструментальные средства, позволяющие наиболее полно и быстро протестировать облачный программный продукт;
- на основе анализа методов проведения автоматизированного тестирования облачного IT-продукта, построенного на микросервисной архитектуре, было установлено, что использование тестовых дублёров, имитирующих ответные вызовы сотрудничающих подсистем, позволяет провести тестирование своей системы наиболее комплексно и оперативно;

- результатом рассмотрения современных методологий разработки программных продуктов с интегрированным процессом тестирования стало выделение методологии, которая позволяет наиболее быстро и качественно составить план тестирования, понятный как заказчикам, так и команде разработчиков и тестировщиков;
- рассмотрен ряд инструментальных средств, предоставляющих возможности для проведения автоматизированного тестирования облачного IT-продукта. Выделены достоинства и недостатки с точки зрения удобства использования;
- составлен сценарий проведения автоматизированного тестирования облачного IT-продукта, построенного на микросервисной архитектуре с использованием тестовых дублёров;
- в рамках апробации выделенных методов, сценария и инструментального средства для проведения автоматизированного тестирования был составлен набор тест-кейсов для тестирования смоделированного облачного IT-продукта, построенного на микросервисной архитектуре. Описан ход и результат тестирования;
- в результате анализа использования тестовых дублёров были выделены преимущества данного подхода, проведена оценка трудозатрат на установку и конфигурацию тестового дублёра на оборудовании заказчика. Дан ряд рекомендаций для повышения качества, надёжности и скорости доставки обновлений заказчику.

Таким образом, предложенный сценарий (алгоритм) проведения автоматизированного тестирования облачного IT-продукта, построенного на микросервисной архитектуре, позволяет значительно улучшить и оптимизировать итоговую стоимость, качество и время выхода на рынок программного обеспечения. Это улучшение очень значительно в нынешней ситуации, когда отрасли программного обеспечения сталкиваются с жесткой международной конкуренцией и пытаются сократить свои бюджеты и графики.

Список используемой литературы

1. Бейзер Б. Тестирование чёрного ящика. Технологии функционального тестирования программного обеспечения и систем. — СПб.: Питер, 2015. -320 с.
2. Бек К. Экстремальное программирование. Разработка через тестирование TDD. —Питер СПб, 2019.— 224с.
3. Блэк Р. Ключевые процессы тестирования. Планирование, подготовка, проведение, совершенствование. —Лори, 2019. —544с.
4. Гвоздева В.А., Лаврентьева И.Ю. Основы построения автоматизированных информационных систем. – М.: Форум, Инфра-М, 2017. – 320 с.
5. Грегори Д., Криспин Л. Agile-тестирование. Обучающий курс для всей команды. —Манн, Иванов и Фербер, 2019. —528с.
6. Грэхем Л. Разработка через тестирование для iOS. – М.: ДМК Пресс, 2018. – 272 с.
7. Дастин Э., Рэшка Дж., Пол Дж. Автоматизированное тестирование программного обеспечения. Внедрение, управление и эксплуатация. — Лори, 2019. —567с.
8. Криспин Л. Грегори Дж. Гибкое тестирование. Практическое руководство для тестировщиков ПО и гибких команд. —М.: Вильямс, 2016. — 464 с.
9. Котляров В.П., Коликова Т.П. Основы информационных технологий. Основы тестирования программного обеспечения. — Бином. Лаборатория знаний, Интернет-университет информационных технологий,2016. —288с.
10. Канер К., Фолк Д., Нгуен Е. Тестирование программного обеспечения. Фундаментальные концепции менеджмента бизнес-приложений. — Киев: ДиаСофт, 2018. — 544 с.
11. Калбертсон Р., Браун К., Кобб Г. Быстрое тестирование. — М.: Вильямс, 2017. — 374 с.

12. Куликов С.С. Тестирование программного обеспечения. Базовый курс. [Электронный ресурс]. — FKPTM, 2017. 312c. URL: https://fktpm.ru/file/113-svjatoslav-kulikov-testirovanie-po-bazovyj-kurs.pdf 12 (дата обращения: 30.11.2019).

13. Липаев, В.В. Методы обеспечения качества крупномасштабных программных средств—М.: СИНТЕГ, 2016 – 350 с.

14. Маглинец Ю.А. Анализ требований к автоматизированным информационным системам. – М.: Интернет-университет информационных технологий, Бином. Лаборатория знаний, 2016. – 200 с.

15. Ошероув Р. Искусство автономного тестирования с примерами на С#.—ДМК Пресс, 2016. —360с.

16. Реутов А.П., Черняков М.В., Замуруев С.Н. Автоматизированные информационные системы. Методы построения и исследования. – М.: Форум, Инфра-М, 2017. – 328 с.

17. Синицын С. В., Налютин Н. Ю. Верификация программного обеспечения. — М.: БИНОМ, 2016. — 368 с.

18. Ньюмэн С. Создание микросервисов. — СПб.: Питер, 2016. — 304 с.

19. Уиттакер Дж., Арбон Дж., Каролло Дж. Как тестируют в Google. — СПб.: Питер, 2014. — 320с.

20. Фуфаев Д.Э. Разработка и эксплуатация автоматизированных информационных систем. – М.: Академия, 2017. – 304 с.

21. Хамбл Д., Фарли Д. Непрерывное развертывание ПО. Автоматизация процессов сборки, тестирования и внедрения новых версий программ. — М.: Вильямс, 2018. — 432 с.

22. Baboi M., Iftenea A., Gîfu D. Dynamic Microservices to Create Scalable and Fault Tolerance Architecture [Электронный ресурс] // Procedia Computer Science. 2019. Vol. 159. PP. 1035-1044. URL: https://www.sciencedirect.com/science/article/pii/S187705091931467X (дата обращения: 17.03.2020).

23. Banica L., Stefan C., Hagiu A. Leveraging the microservice architecture for next generation IOT applications [Электронный ресурс] // Scientific Bulletin – Economic Sciences. 2017. Vol. 16. PP. 26-32. URL: https://doaj.org/article/e72c88c1a95c4487b28a8daeb29b18c4 (дата обращения: 17.03.2020).

24. Boncea R., Zamfiroiu Alin, Bacivarov I. A scalable architecture for automated monitoring of microservice [Электронный ресурс] //Economy Informatics. 2018. Vol. 18. PP. 13-22 URL: https://doaj.org/article/4371304a09964a60b26fa20a4fa90b97 (дата обращения: 17.03.2020).

25. Divya K., Mishra K. The Impacts of Test Automation on Software's Cost, Quality and Time to Market [Электронный ресурс] // Procedia Computer Science. 2016. Vol. 79. PP. 8-15. URL: https://www.sciencedirect.com/science/article/pii/S1877050916001277?via%3Dihub (дата обращения: 29.11.2019).

26. Dani A., Hadas Schwartz C., Yaron T., Bures M., Shlomo M. Conceptual Approach for Reuse of Test Automation Artifacts on Various Architectural Levels [Электронный ресурс] // Comruter Science And Information Systems.2018. Vol. 15. PP. 449-472. URL: http://apps.webofknowledge.com/full_record.do?product=WOS&search_mode=GeneralSearch&qid=25&SID=D4YhQP7JFpNkFEdgrze&page=1&doc=10 (дата обращения: 30.11.2019).

27. Flemstrom D., Potena P. Similarity-based prioritization of test case automation [Электронный ресурс] // Software Quality Journal.2018. Vol. 26. PP. 1421-1429. URL: https://link.springer.com/article/10.1007%2Fs11219-017-9401-7 (дата обращения: 30.11.2019).

28. Sulabh T., Ritu S., Bharti S. Adopting Test Automation on Agile Development Projects: A Grounded Theory Study of Indian Software Organizations [Электронный ресурс] // Agile Processes in Software Engineering and Extreme Programming. 2017. Vol. 2. PP. 184-188. URL:

https://link.springer.com/chapter/10.1007/978-3-319-57633-6_12 (дата обращения: 30.11.2019).

29. Ulewicz S., Vogel-Heuser B. Increasing system test coverage in production automation systems [Электронный ресурс] // Control Engineering Practice.2018. Vol. 26. PP. 171-185. URL: https://www.researchgate.net/publication/323116397_Increasing_system_test_coverage_in_production_automation_systems (дата обращения: 15.11.2019).

30. Zheng L., Wei B. Application of microservice architecture in cloud environment project development [Электронный ресурс] // MATEC Web of Conferences.2018. Vol. 189. URL: https://doaj.org/article/844c18ea51ab4f499c88df2704447fa8 (дата обращения: 17.03.2020).

Приложение А

Набор тест-кейсов, созданный в библиотеке Cucumber-JVM

– Scenario: Successful order

а) Given created order from template "orderTemplate1"

б) When user fills payment data from template "paymentTemplate1"

в) When user submits order

г) Then check, that payment subsystem received request

д) Then check, that request to payment subsystem matches spec "SuccessfulPaymentSpec"

е) Then check, that notification subsystem received request

ж) Then check, that request to notification subsystem matches spec "SuccessfulPaymentNotificationSpec"

з) Then check, that delivery subsystem received request

и) Then check, that request to delivery subsystem matches spec "SuccessfulDeliverySpec"

к) Then check, that order status is "Successful"

– Scenario: Wrong payment data

а) Given created order from template "orderTemplate1"

б) When user fills payment data from template "wrongPaymentTemplate1"

в) When user submits order

г) Then check, that payment subsystem received request

д) Then check, that request to payment subsystem matches spec "WrongPaymentSpec"

е) Then check, that delivery subsystem did not receive request

ж) Then check, that order status is "In progress"

з) Then check, that order's error is "Wrong payment data"

Продолжение приложения А

– Scenario: Delivery subsystem error

а) Given created order from template "orderTemplate1"

б) When user fills payment data from template "paymentTemplate1"

в) When user submits order

г) Then check, that payment subsystem received request

д) Then check, that request to payment subsystem matches spec "SuccessfulPaymentSpec"

е) Then check, that notification subsystem received request

ж) Then check, that request to notification subsystem matches spec "SuccessfulPaymentNotificationSpec"

з) Then check, that delivery subsystem received request

и) Then check, that delivery subsystem responded with error 500

к) Then check, that order status is "Rejected"

л) Then check, that order's error is "Can't schedule the delivery"

– Scenario: Notification subsystem error

а) Given created order from template "orderTemplate1"

б) When user fills payment data from template "paymentTemplate1"

в) When user submits order

г) Then check, that payment subsystem received request

д) Then check, that request to payment subsystem matches spec "SuccessfulPaymentSpec"

е) Then check, that notification subsystem received request

ж) Then check, that notification subsystem responded with error 500

з) Then check, that order status is "Successful"

и) Then check, that there is no errors on order

к) When wait for 5 minutes

л) Then check, that notification subsystem received 5 requests total matches spec "SuccessfulPaymentSpec"