

Оглавление

1 Глава 1	3
1.1 Теоретические аспекты процесса тестирования	3
1.1.1 Определение понятия тестирования ПО	3
1.1.2 Классификация видов тестирования.....	4
1.1.2.1 По объекту тестирования	5
1.1.2.2 Тестирование, связанное с изменениями	10
1.1.2.3 По уровню тестирования	10
1.1.2.4 По исполнению кода	11
1.1.2.5 По субъекту тестирования	12
1.1.2.6 По позитивности сценария.....	13
1.1.2.7 По степени автоматизации.....	14
1.1.3 Методологии тестирования	14
1.1.3.1 Метод черного ящика.....	14
1.1.3.2 Метод белого ящика	15
1.1.3.3 Метод серого ящика	15
1.1.4 Процесс тестирования	16
1.1.4.1 Разработка тест-кейсов	17
1.1.4.2 Атрибуты тест-кейса.	17
1.1.4.3 Требования к тест-кейсу.....	18
1.1.4.4 Выполнение тест-кейсов	19
1.1.4.5 Анализ результатов тестирования	21
1.2 Описание и анализ дефектов	23
1.2.1 Жизненный цикл дефекта	25
1.3 Непрерывная интеграция.....	27
1.3.1 Организация непрерывной интеграции.....	28
1.3.2 Популярные CI-платформы.....	30
1.3.2.1 CircleCI.....	30
1.3.3 Travis CI.....	31
1.3.4 Jenkins.....	32
1.4 Фреймворки тестирования	33
1.4.1 xUnit.....	33
1.4.2 NUnit	35
1.4.3 Visual Studio Unit Testing Framework	39
1.5 Обзор систем контроля версий.....	41
1.5.1 Система управления версиями Subversion.	41

1.5.2 Система управления версиями Git.....	43
1.5.3 Система управления версиями Mercurial.	46
1.6 TestLink.....	47
1.6.1 Использование	47
6.2 Особенности	51
2 Глава 2	54
2.1 Alpha.Alarms.....	54
2.1.1 Системные требования	54
2.1.2 Варианты запуска Alpha.Alarms.....	54
2.1.2.1 Стандартный запуск.....	54
2.1.2.2 Запуск с параметрами	55
2.1.2.3 Запуск Alpha.Alarms как встраиваемого компонента	55
2.1.3 Просмотр событий.....	55
2.1.3.1 Просмотр оперативных событий.....	55
2.1.3.1.1 Квотирование событий.....	56
2.1.3.1.2 Приостановка поступления событий.....	57
2.1.3.1.3 Очистка списка событий	57
2.1.3.2 Просмотр истории событий.....	57
2.1.3.2.1 Установка временного интервала для запроса	57
2.1.4 Дополнительные возможности.....	58
2.1.4.1 Фильтрация сообщений о событиях	58
2.1.4.2 Сохранение данных в табличный файл.....	58
2.1.4.3 Импорт/экспорт предустановленных фильтров	59
2.1.4.4 Настройка папок для импорта и экспорта	60
2.1.4.5 Печать таблицы событий.....	60
2.1.4.6 Откат настроек приложения.....	61
2.2 Экономическая и временная эффективность внедрения автоматизированных тестов.....	62
2.2.1 Расчёт экономической целесообразности введения автоматизированного тестирования.....	74
2.2.2 Расчёт временной целесообразности введения автоматизированного тестирования	76
2.3 Автоматизация процесса тестирования.....	62
2.3.1 Уровни автоматизации тестирования	64
2.3.2 Архитектура тестов	66
3 Глава 3	79
3.1 Реализация приложения по тестированию	79
3.1.1 Архитектура реализованных тестов	79
Заключение	98
Список использованной литературы	99

1 Глава 1

1.1 Теоретические аспекты процесса тестирования

Данная глава посвящена решению таких задач, как выявление теоретических основ тестирования, классификация и описание видов тестирования, анализ и описание процесса тестирования, выявление критериев корректно построенного процесса.

Решение данных задач необходимо для того, чтобы лучше понимать процессы тестирования, различать их виды и применять знания при оценке целесообразности внедрения автоматизированного тестирования в компании.

1.1.1 Определение понятия тестирования ПО

Тестирование программного обеспечения - проверка соответствия между реальным и ожидаемым поведением программы, осуществляемая на конечном наборе тестов, выбранном определенным образом.

Тестирование – это одна из техник контроля качества, которая включает в себя такие процессы, как: проектирование тестов, выполнение тестирования и анализ полученных результатов.

Общая схема тестирования (рисунок 1.1.1).



Рисунок 1.1.1– Общая схема тестирования

На входе тестировщик получает программу, которую необходимо тестировать и требования. Наблюдая за программой в определенных

условиях, на выходе тестировщик получает информацию о соответствии или несоответствии программы требованиям. Данная информация используется для исправления ошибок в существующем продукте, либо для изменения требований к еще только разрабатываемому продукту.

Тест (проверка) включает в себя выбранную определенным образом искусственно созданную ситуацию и описание наблюдений, которые нужно осуществить, для проверки программы на соответствие определенным требованиям.

Тест может быть, как коротким, так и длинным, например, тест производительности, проверяющий работоспособность системы при длительной нагрузке.

Таким образом, в процессе тестирования тестировщик выполняет две основные задачи.

Первой задачей является управление выполнением программы, а также создание искусственных ситуаций, в которых и происходит проверка поведения программы.

Вторая задача состоит в наблюдении за тем, как программа ведет себя в различных созданных ситуациях, и в сравнении того, что он видит с тем, что ожидается.

Если рассматривать задачи современного тестирования, то можно прийти к заключению, что они заключаются не только в обнаружении ошибок в программах, но и в выявлении причин, по которым они возникают. Такой подход к процессу тестирования позволяет разработчикам выполнять свою работу с максимальной эффективностью, устраняя обнаруженные ошибки быстро и своевременно.

1.1.2 Классификация видов тестирования

При тестировании программного продукта применяется огромное количество различных видов тестов. Наиболее широкую и подробную классификацию предложил автор книги «Тестирование Дот Ком» Роман Савин. Он объединил виды тестирования по таким признакам, как объект,

субъект тестирования, уровень, позитивность тестирования, и степень автоматизации тестирования. Классификация была дополнена на основании таких источников, как книга Сэма Канера, «Тестирование программного обеспечения» и интернет-ресурс, посвященный тестированию, «Про Тестинг - Тестирование Программного Обеспечения» [1].

1.1.2.1 По объекту тестирования

1) Функциональное тестирование. Функциональное тестирование на сегодняшний день является одним из наиболее часто применяемых видов тестирования. Задача такого тестирования - это установить на сколько соответствует разработанное программное обеспечение (ПО) требованиям заказчика с точки зрения функционала. Иначе говоря, проведение функциональных тестов позволяет проверить способность информационной системы решать задачи пользователей.

2) Нефункциональное тестирование. Позволяет проверить соответствие свойств программного обеспечения с поставленными нефункциональными требованиями. Таким образом, нефункциональное тестирование - это тестирование всех свойств программы, не относящихся к функциональности системы. Такими свойствами могут быть предъявленные характеристики с точки зрения таких параметров как:

а) надежность (способность системы реагировать на непредвиденные ситуации).

б) производительность (способность системы работать под большими нагрузками).

с) удобство (исследование удобства работы пользователя с приложением).

д) масштабируемость (возможность масштабировать приложение как вертикально, так и горизонтально).

е) безопасность (исследование возможности нарушения работы приложения и кражи пользовательских данных злоумышленниками).

f) портируемость (возможность перенести приложение на определенный набор платформ)

И много других качеств.

1) Тестирование пользовательского интерфейса. Это тестирование корректности отображения элементов пользовательского интерфейса на различных устройствах, правильности реагирования их на совершение пользователем различных действий насколько и оценка того, насколько ожидаемо ведет себя программа в целом. Такое тестирование дает возможность оценить, насколько эффективно пользователь сможет работать с приложением и насколько внешний вид приложения соответствует утвержденным документам, созданными дизайнерами. При проведении тестирования пользовательского интерфейса основной задачей тестировщика является выявление визуальных и структурных недостатков в графическом интерфейсе приложения, проверке возможности и удобства навигации в приложении и корректность обработки приложением ввода данных с клавиатуры, мыши и других устройств ввода. Тестирование пользовательского интерфейса необходимо для того, чтобы убедиться в том, что интерфейс соответствует утвержденным требованиям и стандартам, и гарантировать возможность работы пользователя с графическим интерфейсом приложения.

2) Тестирование удобства использования. Это способ тестирования, позволяющий оценить степень удобства использования приложения, скорость обучения пользователей при работе с программой, а также насколько пользователи разрабатываемого продукта находят ее понятной и привлекательной в контексте заданных условий. Такое тестирование необходимо для обеспечения максимально положительного пользовательского опыта при работе с приложением.

3) Тестирование защищенности. Позволяет выявить главные уязвимости программного обеспечения по отношению к различным атакам со стороны злоумышленников. Компьютерные системы довольно часто

подвергаются кибер атакам с целью нарушения работоспособности информационной системы либо кражи конфиденциальных данных. Тестирование безопасности дает возможность проанализировать реальную реакцию и действенность защитных механизмов, использованных в системе, при попытке проникновения. В процессе тестирования безопасности тестировщик пытается выполнять те же действия, которые выполнял бы настоящий взломщик. При попытке тестировщиком взломать систему могут использоваться любые средства: атаки системы при помощи специальных утилит; попытки узнать логины и пароли с помощью внешних средств; DDOS атаки; целенаправленная генерация ошибок для обнаружения возможности проникновения в систему в процессе её восстановления; использование известных незакрытых уязвимостей системы.

4) Инсталляционное тестирование. Под этим термином подразумевают тестирование корректности установки (инсталляции) определенного программного продукта. Такое тестирование обычно происходит в искусственно созданных средах с целью выявить степень готовности программного обеспечения к эксплуатации. Основные причины проведения таких тестов связаны с необходимостью проверить корректность поведения программного продукта при автоматизированном развертывании либо обновлении. Обеспечение правильной и стабильной установки программного обеспечения является очень важным фактором при создании программного продукта, поскольку позволяет пользователям быстрее и с меньшими усилиями начать использовать продукт, при этом обеспечивая одинаково корректное поведение этого продукта во всех протестированных программных средах.

5) Конфигурационное тестирование. Конфигурационное тестирование предназначено для оценки работоспособности программного обеспечения при разнообразных конфигурациях системы. В зависимости от типа тестируемого программного продукта, конфигурационное тестирование может преследовать разные цели. Обычно это либо определение

оптимальной конфигурации оборудования, обеспечивающего достаточные для работы ПО параметры производительности, либо проверка определенной конфигурации оборудования (или платформы, включающей в себя помимо оборудования, стороннее ПО, необходимое для работы программы) на совместимость с тестируемым продуктом. Если речь идет о клиент-серверном программном обеспечении, то конфигурационное тестирование проводится отдельно для сервера и отдельно для клиента. Обычно при тестировании совместимости сервера с определенной конфигурацией стоит задача найти оптимальную конфигурацию, поскольку важна стабильность работы и производительность сервера. В то время как при тестировании клиента, наоборот, пытаются выявить недостатки ПО при любых конфигурациях и по возможности устранить их [7].

6) Тестирование надежности и восстановления после сбоев (стрессовое тестирование). Такой вид тестирования довольно часто проводится для программного обеспечения, работающего с ценными пользовательскими данными, бесперебойность работы и скорость восстановления после сбоев которого критичны для пользователя. Тестирование на отказ и восстановление осуществляет проверку способности программы быстро и успешно восстанавливаться после отказа оборудования, перебоев сети или критических ошибок в самом программном обеспечении. Это дает возможность оценить возможные последствия отказа и время, необходимое для последующего восстановления системы. На основе полученных в ходе тестирования данных может быть оценена надежность системы в целом, и, при условии неудовлетворительных показателей, соответствующие меры, направленные на улучшение систем восстановления, могут быть приняты

7) Тестирование локализации. Тестирование локализации дает возможность выяснить насколько хорошо приспособлен продукт для населения определенных стран и насколько он соответствует ее культурным особенностям. Обычно, рассматриваются культурный и языковой нюансы, а именно перевод пользовательского интерфейса, сопутствующей

документации и файлов на определенный язык, также тестируется правильность форматов валют, чисел, времени и телефонных номеров.

8) Нагрузочное тестирование. Нагрузочное тестирование позволяет выявить максимальное количество однотипных задач, которые программа может выполнять параллельно. Самая популярная цель нагрузочного тестирования в контексте клиент-серверных приложений - это оценить максимальное количество пользователей, которые смогут одновременно пользоваться услугами приложения.

9) Тестирование стабильности. Тестирование стабильности проверяет работоспособность приложения при длительном использовании на средних нагрузках. В зависимости от типа приложения, формируются определенные требования к длительности его бесперебойной работы. Тестирование стабильности стремится выявить такие недочеты приложения как утечки памяти, наличие ярко выраженных скачков нагрузки и прочие факторы, способные помешать работе приложения в течение длительного периода времени.

10) Тестирование масштабируемости. Это вид тестирования программного обеспечения, предназначенный для проверки способности продукта к увеличению (иногда к уменьшению) масштабов определенных нефункциональных возможностей. Некоторые виды приложений должны легко масштабироваться и, при этом, разумеется, оставаться работоспособными и выдерживать определенную пользовательскую нагрузку

11) Объемное тестирование. Задачей объемного тестирования поставлено выявление реакции приложения и оценка возможных ухудшений в работе ПО при значительном увеличении количества данных в базе данных приложения. Обычно в такое тестирование входит:

а) Замер времени выполнения операций, связанных с получением или изменением данных БД при определенной интенсивности запросов.

б) Выявление зависимости увеличения времени операций от объема данных в БД.

с) Определение максимального количества пользователей, которые имеют возможность одновременно работать с приложением без заметных задержек со стороны БД.

1.1.2.2 Тестирование, связанное с изменениями

1) Санитарное тестирование является одним из видов тестирования, целью которого служит доказательство работоспособности конкретной функции или модуля в соответствии с техническими требованиями, заявленными заказчиком. Санитарное тестирование довольно часто используется при проверке какой-то части программы или приложения при внесении в нее определенных изменений со стороны факторов окружающей среды. Данный вид тестирования обычно выполняется в ручном режиме.

2) Дымовое тестирование представляет собой короткий цикл тестов, целью которых является подтверждение факта запуска и выполнения функций устанавливаемого приложения после того как новый или редактируемый код прошел сборку. По завершении тестирования наиболее важных сегментов приложения предоставляется объективная информация о присутствии или отсутствии дефектов в работе тестируемых сегментов. По результатам дымового тестирования принимается решение об отправке приложения на доработку или о необходимости его последующего полного тестирования.

3) Регрессионное тестирование – тестирование, направленное на обнаружение ошибок в уже протестированных участках. Регрессионное тестирование проверяет продукт на ошибки, которые могли появиться в результате добавления нового участка программы или исправления других ошибок. Цель данного вида тестирования – убедиться, что обновление сборки или исправление ошибок не повлекло за собой возникновения новых багов [2].

1.1.2.3 По уровню тестирования

1) Модульное тестирование (Unit тесты). Заключается в проверке каждого отдельного модуля (самобитного элемента системы) путем запуска

автоматизированных тестов в искусственной среде. Реализации таких тестов часто используют различные заглушки и драйверы для имитации работы реальной системы. Модульное автоматизированное тестирование - это самая первая возможность запустить и проверить исходный код. Создание Unit тестов для всех модулей системы позволяет очень быстро выявлять ошибки в коде, которые могут появиться в ходе разработки.

2) Интеграционное тестирование. Это тестирование отдельных модулей системы на предмет корректного взаимодействия. Основная цель интеграционного тестирования - найти дефекты и выявить некорректное поведение, связанное с ошибками в интерпретации или реализации взаимодействия между модулями.

3) Системное тестирование. Это тестирование программы в целом, такое тестирование проверяет соответствие программы заявленным требованиям.

4) Приемочное тестирование. Это комплексное тестирование, определяющее фактический уровень готовности системы к эксплуатации конечными пользователями. Тестирование проводится на основании набора тестовых сценариев, покрывающих основные бизнес-операции системы.

1.1.2.4 По исполнению кода

1) Статическое тестирование. Это выявление артефактов, появляющихся в процессе разработки программного продукта путем анализа исходных файлов, таких как документация или программный код. Такое тестирование проводится без непосредственного запуска кода, качество исходных файлов и их соответствие требованиям оцениваются вручную, либо с использованием вспомогательных инструментов. Статическое тестирование должно проводиться до динамического тестирования, таким образом, ошибки, обнаруженные на этапе статического тестирования, обойдутся дешевле. С точки зрения исходного кода, статическое тестирование выражается в ревизии кода. Обычно ревизия кода отдельных файлов производится после каждого изменения этих файлов программистом,

сама же ревизия может проводиться как другим программистом, так и ведущим разработчиком, либо отдельным работником, занимающимся ревизией кода. Использование статического тестирования дает возможность поддерживать качество программного обеспечения на всех стадиях разработки и уменьшает время разработки продукта.

2) Динамическое тестирование. В отличие от статического тестирования, такой вид тестирования предполагает запуск исходного кода приложения. Таким образом, динамическое тестирование содержит в себе множество других типов тестирования, которые уже были описаны выше. Динамическое тестирование позволяет выявить ошибки в поведении программы с помощью анализа результатов ее выполнения. Получается, что почти все существующие типы тестирования соответствуют классу динамического тестирования.

1.1.2.5 По субъекту тестирования

1) Альфа-тестирование. Это тестирование проводится для самых ранних версий компьютерного программного обеспечения (или аппаратного устройства). Альфа-тестирование почти всегда проводится самими разработчиками ПО. В процессе альфа-тестирования разработчики приложения находят и исправляют ошибки и проблемы, имеющиеся в программе. Обычно, во время Альфа-тестирования происходит имитация работы с программой штатными разработчиками, реже имеет место реальная работа как потенциальных пользователей, так и заказчиков с продуктом. Как правило, альфа-тестирование проводится на самом раннем этапе разработки ПО, однако в отдельных случаях может быть применено для законченного или близкого к завершению продукта, например, в качестве приёмочного тестирования.

2) Бета-тестирование. Тестирование продукта, по-прежнему находящегося в стадии разработки. При бета-тестировании этот продукт предоставляется для некоторого количества пользователей, для того чтобы изучить и сообщить о возникающих проблемах, с которыми сталкиваются

пользователи. Такое тестирование необходимо чтобы найти ошибки, которые разработчики могли пропустить. Обычно бета-тестирование проводится в две фазы: закрытый бета-тест и открытое бета-тестирование. Закрытый бета-тест - это тестирование на строго ограниченном кругу избранных пользователей. Такими пользователями могут выступать знакомые разработчиков, либо их коллеги, не связанные напрямую с разработкой тестируемого продукта. Открытое бета-тестирование заключается в создании и размещении в открытом доступе публичной бета-версии. В данном случае любой пользователь может выступать бета-тестером. Обратная связь от таких бета-тестеров осуществляется с помощью отзывов на сайте и встроенных в программу систем аналитики и логирования пользовательских действий, эти системы необходимы для анализа поведения пользователей и обнаружения трудностей и ошибок, с которыми они сталкиваются.

1.1.2.6 По позитивности сценария

1) Позитивное тестирование. Тесты с позитивным сценарием проверяют способность программы выполнять заложенный в нее функционал. Как правило, для такого тестирования разрабатываются тестовые сценарии, при выполнении которых, в нормальных для ПО условиях работы, не должно возникать никаких сложностей.

2) Негативное тестирование. Негативное тестирование программного обеспечения происходит на сценариях, соответствующих нештатному поведению программы. Такие тесты проверяют корректность работы программы в экстренных ситуациях. Это позволяет удостовериться в том, что программа выдает правильные сообщения об ошибках, не повреждает пользовательские данные и ведет себя корректно в целом при ситуациях, в которых не предусмотрено штатное поведение продукта. Основная цель негативного тестирования - это проверить устойчивость системы к различным воздействиям, способность правильно валидировать входные данные и обрабатывать исключительные ситуации, возникающие как в самих программных алгоритмах, так и в бизнес-логике [3].

1.1.2.7 По степени автоматизации

1) Ручное тестирование. Ручное тестирование проводится без использования дополнительных программных средств, оно позволяет проверить программу или сайт с помощью имитации действий пользователя. В этой модели тестировщик выступает в качестве пользователя, следуя определенным сценариям, параллельно анализируя вывод программы и ее поведение в целом.

2) Автоматизированное тестирование. Такое тестирование позволяет за счет использования дополнительного программного обеспечения для автоматизации тестов значительно ускорить процесс тестирования. Такое дополнительное ПО позволяет контролировать и управлять выполнением тестов и сравнивать ожидаемый и фактический результаты работы программы. Более подробно будет рассмотрено позже.

1.1.3 Методологии тестирования

Существуют различные методологии динамического тестирования ПО. В зависимости от наличия у тестировщика доступа к исходному коду программы, выделяют следующие методы тестирования:

- 1) метод черного ящика;
- 2) метод белого ящика;
- 3) метод серого ящика.

1.1.3.1 Метод черного ящика

Впервые термин «черный ящик» упоминается психиатром У. Р. Эшби в книге "Введение в кибернетику" в 1959 г. Он писал, что метод черного ящика позволяет изучать поведение системы абстрагируясь от ее внутреннего устройства.

В области тестирования метод черного ящика - это техника тестирования, которая основана на работе с внешними интерфейсами программного обеспечения, без знания внутреннего устройства системы.

Данный метод назван “Черным ящиком”, поскольку в этом методе тестируемое программное обеспечение для тестировщика выглядит как

черный ящик, внутри которого происходят некоторые процессы, однако тестировщику о них принципиально ничего не известно. Данная техника позволяет обнаружить ошибки в следующих категориях:

- 1) ошибки интерфейса;
- 2) недостающие или неправильно реализованные функции;
- 3) недостаточная производительность или ошибки поведения системы;
- 4) некорректные структуры данных или плохая организация доступа к внешним базам данных.

Таким образом, поскольку тестировщик не имеет никакого представления о внутреннем устройстве и структуре системы, ему необходимо сконцентрироваться на том, что делает программа, а не на том, как она это делает [4].

1.1.3.2 Метод белого ящика

Как можно догадаться из названия, этот метод тестирования противоположен методу черного ящика. Данный метод тестирования основан на анализе внутренней структуры системы.

То есть в данном случае тестировщику известны все аспекты реализации тестируемого программного обеспечения. Этот метод позволяет протестировать не только корректность реакции программы на определенный ввод (как в случае с черным ящиком), но и правильную работу отдельных модулей и функций, основываясь на знании кода, который будет обрабатывать этот ввод. Знание особенностей реализации тестируемой программы – обязательное требование к тестировщику для успешного применения этой техники. Тестирование методом белого ящика позволяет углубиться во внутренне устройство ПО, за пределы его внешних интерфейсов.

1.1.3.3 Метод серого ящика

Этот метод тестирования системы предполагает комбинацию подходов Белого и Черного ящиков. Таким образом, тестировщику лишь частично известно внутреннее устройство программы. Например, предполагается,

наличие доступа к внутренней структуре программного обеспечения для разработки максимально эффективных тест-кейсов, в то время как само тестирование будет проводится методом черного ящика. Или тестировщики могут во всем следовать методу черного ящика, однако для того что бы убедиться в корректной работе отдельных алгоритмов могут смотреть информацию в логах или анализировать записи программы в базе данных.

1.1.4 Процесс тестирования

Тестирование представляет собой процесс проверки того, насколько программное обеспечение соответствует требованиям, заявленным заказчиком. Он осуществляется в специальных, искусственно создаваемых ситуациях посредством наблюдения за работой программного обеспечения. Такого рода искусственно построенные ситуации называют тестовыми или просто тестами.

Разработка тестов происходит на основе проверяемых требований и критерия полноты тестирования. Разработанные тесты формируются в тест-кейс (набор тестов) и выполняются на ПО, которое нужно протестировать. После прогона всех тестов анализируется результат, в результате чего можно выявить ошибки в программе.

Процесс тестирования схематично показан на рисунке 1.1.2.



Рисунок 1.1.2– Процесс тестирования

1.1.4.1 Разработка тест-кейсов

Тест-кейс (тестовый случай) – это минимальный элемент тестирования (одна проверка), содержащий в себе описание конкретных действий, условий и параметров, которые направлены на проверку какой-либо функциональности. Набор тест-кейсов называется тестовым набором (test suite).

Тест-кейсы позволяют тестировщикам провести проверку продукта без полного ознакомления с документацией. При условии, что созданный тест-кейс удобен в поддержке, то, написанный один раз, он позволит сэкономить большое количество времени и усилий тестировщиков. Подробные тест-кейсы также способны существенно снизить вариативность выполнения тестов различными тестировщиками, что повышает качество тестирования.

1.1.4.2 Атрибуты тест-кейса.

Тест-кейс должен включать в себя:

- 1) уникальный идентификатор тест-кейса. Этот идентификатор необходим для удобства организации и навигации по тестовым наборам;
- 2) название. В названии должна отражаться основная идея тест-кейса, цель данной проверки;
- 3) предусловия. Список шагов, не имеющих прямого отношения к проверяемому функционалу, которые необходимо выполнить до начала теста. Например, для тест-кейса «Заказ товара» предусловием может быть шаг «авторизоваться на сайте», если на данном сайте заказать товар может только авторизованный пользователь;
- 4) шаги. Описание последовательности действий, которая должна привести к ожидаемому результату;
- 5) история редактирования. Дает возможность узнать, кем и когда был изменен тест-кейс. Это удобно, поскольку позволяет более эффективно редактировать тест-кейсы;

б) ожидаемый результат. Поведение системы, предусмотренное требованиями. Один тест-кейс проверяет одну конкретную функцию, поэтому ожидаемый результат может быть только один;

7) статус кейса. Проставляется в соответствии с тем, соответствует ли фактический результат ожидаемому. Тест-кейс может иметь один из трех статусов:

а) положительный, если фактический результат совпадает с ожидаемым результатом;

б) отрицательный, если фактический результат не совпадает с ожидаемым результатом. Если статус кейса отрицательный-найдена ошибка;

с) выполнение теста заблокировано, если после одного из шагов продолжение теста невозможно. В этом случае так же, найдена ошибка.

1.1.4.3 Требования к тест-кейсу

1) Для измерения покрытия требований, требования к продукту должны быть проанализированы и впоследствии разбиты на пункты. Если тест-кейсы покрывают все требования, то может быть дан положительный или отрицательный ответ о реализации данного требования в продукте.

2) Тест является хорошим в случае, когда он может обеспечить высокую вероятность обнаружения ошибки. Показать, что в программе полностью отсутствуют ошибки невозможно, поэтому процесс тестирования должен быть направлен на выявление прежде не найденных ошибок.

3) Четкие, однозначные формулировки шагов. Описание шагов для прохождения тест-кейса должно содержать всю необходимую информацию, но при этом тест-кейс не должен быть слишком детализирован. Например, если тест-кейс содержит такие шаги, как авторизация, в описании необходимо указывать логин и пароль, но не нужно указывать в каком углу экрана находится окно авторизации.

4) Отсутствие зависимостей тест-кейсов. Если тесты связаны между собой, становится проблематичным изменение, дополнение или удаление конкретного тест-кейса, появляется необходимость изменять связанные с

ним тесты. Более того, взаимосвязанные тесты обладают конкретный сценарием от перехода одного теста к другому. Это приведет к тому что не все сценарии перехода от одного теста к другому будут протестированы и появляется вероятность пропустить баг.

5) Ожидаемый результат необходимо прогнозировать заранее и прописывать его в тест-кейсе. Если ожидаемый результат не определить заранее, может возникнуть ситуация, когда тестировщик видит то, что он хочет увидеть. При заранее определенном результате тестировщику необходимо только сравнить ожидаемый результат с фактическим.

6) Необходимо уделять внимание не только тестам, которые проверяют правильные данные, но и тем тестам, которые проверяют работу программы при неправильных, непредусмотренных данных. Большое количество ошибок связано именно с теми действиями пользователя, которые не предусмотрены программой.

7) Также необходимо проверять, не делает ли программа то, чего не должна. Нужно производить проверку на нежелательные побочные эффекты.

1.1.4.4 Выполнение тест-кейсов

Одной из особенностей процесса тестирования является необходимость проведения тестирования программы специалистом, который не является ее автором. Категорически неприемлемо тестирование продукта программистом, создавшим его. Это правило должно быть применимо ко всем, без исключения, формам тестирования, например, как к тестированию целой системы, так и к тестированию отдельных модулей, а также внешних функций. Несомненно, при проведении процесса тестирования не автором программы, он становится более точным и эффективным.

Если рассматривать суть процесса тестирования, то становится очевидным, что это процесс деструктивный. С этой его особенностью и связывают представление о тестировании как о сложной работе. Исключительно трудной эта работа представляется для программиста, создавшего продукт, так как проектирование, разработка и написание

программы является процессом конструктивным, в отличие от тестирования, в процессе которого специалисту необходим настрой на деструктивный образ мышления. Исходя из этой особенности тестирования, приступать к тщательному и беспристрастному выявлению ошибок сразу же по завершению создания программы представляется трудновыполнимым для ее автора.

Например, содержащиеся в модуле дефекты, которые являются следствием ошибок перевода (такие как, к примеру, неверная интерпретация спецификации), с высокой долей вероятности будут присутствовать и в тестах, так как оба процесса будут выполняться одним и тем же специалистом. Также могут обнаружиться в тестах и ошибки, сделанные при понимании и сопряжении других модулей.

Однако, не следует делать однозначный вывод, что тестирование программы специалистом, ее создавшим невозможно. Большое количество программистов справляются с этой задачей вполне успешно. Но, исходя из вышесказанного, можно прийти к заключению, что выполнение тестирования другим специалистом, не являющимся ее автором, гораздо эффективнее и плодотворнее. Поэтому и создаются группы тестировщиков, специально для проведения процесса тестирования с наиболее оптимальными результатами.

Принятие решения о сроках остановки тестирования является одной из сложных проблем при проведении тестирования. Это происходит, потому что невозможно провести процесс испытания всех входных значений или, другими словами, исчерпывающее тестирование. Следовательно, при процессе тестирования техническая сторона проводимых действий входит в противоречие с экономической выгодой, так как возникает вопрос выбора оптимального конечного количества тестов, которое дает максимально возможный результат (вероятность обнаружения ошибок) при определенных затратах.

Существует довольно много примеров, когда написанные тесты были малоэффективны, так как их способность обнаружения новых ошибок была крайне низкой. В то же время, хорошие тесты, обнаруживающие ошибки с высокой точностью, могли остаться без внимания специалистов.

Таким образом, принимая решение о количестве времени и проводимых тестов с программой, всегда необходимо принимать во внимание уровень риска, как технического, так и бизнес риска, для отдельно взятого продукта и для всего проекта. Более того, нужно учитывать тот факт, что проект обычно имеет ограничения по времени и бюджету. [6]

1.1.4.5 Анализ результатов тестирования

Несмотря на существование различных видов тестирования, процессы тестирования достаточно схожи. Разработкой и анализом тестов может заниматься только тестировщик. За выполнение тест-кейсов так же отвечает тестировщик, однако выполнение этих тестов может производиться как вручную, так и в автоматизированном режиме.

По результатам выполнения каждого теста, ему присваивается статус (положительный, отрицательный, блокирован). Если тест получает отрицательный статус, то в зависимости от методологии тестирования тестировщик может проводить дополнительную работу для выявления конкретной ошибки, которая была причиной некорректного поведения программы [4].

При использовании методологии черного ящика, анализ результатов теста сводится к выявлению общих закономерностей, ведущих к появлению ошибки. Однако, когда используется белый или серый ящики, тестировщик может проводить гораздо более глубокий анализ причин возникновения ошибки. В зависимости от доступных тестировщику данных (база данных, исходный код программы, логи и т.п.), он способен с некоторой точностью определить источник некорректного поведения программы.

После того, как максимально точно выявлена причина нежелательного поведения, тестировщик должен описать её вместе со способом

воспроизведения ошибки для дальнейшей передачи этой информации разработчикам. Когда источник ошибки точно определен и хорошо описан, разработчикам гораздо проще исправить эту ошибку.

1.2 Описание и анализ дефектов

Ошибками в программном обеспечении являются все возможные несоответствия между демонстрируемыми характеристиками его качества и сформулированными или подразумеваемыми требованиями и ожиданиями пользователей.

В англоязычной литературе используется несколько терминов, часто одинаково переводящихся как "ошибка" на русский язык:

1) defect — самое общее нарушение каких-либо требований или ожиданий, не обязательно проявляющееся вовне (к дефектам относятся нарушения стандартов кодирования, недостаточная гибкость системы и пр.);

2) failure — наблюдаемое нарушение требований, проявляющееся при каком-то реальном сценарии работы ПО. Это можно назвать проявлением ошибки;

3) fault — ошибка в коде программы, вызывающая нарушения требований при работе (failures), то место, которое надо исправить. Хотя это понятие используется довольно часто, оно, вообще говоря, не вполне четкое, поскольку для устранения нарушения можно исправить программу в нескольких местах. Что именно надо исправлять, зависит от дополнительных условий, выполнение которых хотим при этом обеспечить, хотя в некоторых ситуациях наложение дополнительных ограничений не устраняет неоднозначность;

4) error — используется в двух смыслах. Первый смысл — это ошибка в ментальной модели программиста, ошибка в его рассуждениях о программе, которая заставляет его делать ошибки в коде (faults). Это, собственно, ошибка, которую сделал человек в своем понимании свойств программы. Второй смысл — это некорректные значения данных (выходных или внутренних), которые возникают при ошибках в работе программы.

Эти понятия некоторым образом связаны с основными источниками ошибок. Поскольку при разработке программ необходимо сначала понять

задачу, затем придумать ее решение и закодировать его в виде программы, то, соответственно, основных источников ошибок три:

1) неправильное понимание задач. Разработчики ПО не всегда понимают, что именно нужно сделать. Другим источником непонимания служит отсутствие его у самих пользователей и заказчиков — достаточно часто они могут просить сделать несколько не то, что им действительно нужно. Для предотвращения неправильного понимания задач программной системы служит анализ предметной области;

2) неправильное решение задач. Зачастую, даже правильно поняв, что именно нужно сделать, разработчики выбирают неправильный подход к тому, как это делать. Выбираемые решения могут обеспечивать лишь некоторые из требуемых свойств, они могут хорошо подходить для данной задачи в теории, но плохо работать на практике, в конкретных обстоятельствах, в которых должно будет работать программное обеспечение. Помочь в выборе правильного решения может сопоставление альтернативных решений и тщательный анализ их на предмет соответствия всем требованиям, поддержание постоянной связи с пользователями и заказчиками, предоставление им необходимой информации о выбранных решениях, демонстрация прототипов, анализ пригодности выбираемых решений для работы в том контексте, в котором они будут использоваться;

3) неправильный перенос решений в код. Имея правильное решение правильно понятой задачи, люди, тем не менее, способны сделать достаточно много ошибок при воплощении этих решений. Корректному представлению решений в коде могут помешать как обычные опечатки, так и забывчивость программиста или его нежелание отказаться от привычных приемов, которые не дают возможности аккуратно записать принятое решение. С ошибками такого рода можно справиться при помощи инспектирования кода, взаимного контроля, при котором разработчики внимательно читают код друг друга, опережающей разработки модульных тестов и тестирования.

В программировании баг (bug — жук) — жаргонное слово, обычно обозначающее ошибку в программе или системе, которая выдает неожиданный или неправильный результат. Большинство багов возникают из-за ошибок, сделанных разработчиками программы в её исходном коде, либо в её дизайне. Также некоторые баги возникают из-за некорректной работы компилятора, вырабатывающего некорректный код. Программу, которая содержит большое число багов и/или баги, серьёзно ограничивающие её функциональность, называют нестабильной или, на жаргонном языке, «глючной», «забагованной» (unstable, buggy).

Термин «баг» обычно употребляется в отношении ошибок, проявляющих себя на стадии работы программы, в отличие, например, от ошибок проектирования или синтаксических ошибок. Отчет, содержащий информацию о баге также называют отчетом об ошибке или отчетом о проблеме (bug report). Отчет о критической проблеме (crash), вызывающей аварийное завершение программы, называют крэш репортом (crash report). «Баги» локализуются и устраняются в процессе тестирования и отладки программы.

1.2.1 Жизненный цикл дефекта

Рисунок 1.2.1 иллюстрирует жизненный цикл дефекта, принятый во многих крупных компаниях. Баг может находиться в одном из представленных на рисунке состояний.

1) Обнаружен (submitted). Тестировщик находит баг и представляет его на рассмотрение в систему отслеживания ошибок. С этого момента баг начинает свою официальную жизнь и о его существовании знают необходимые люди.

2) Изначен (assigned). Тестировщик или ведущий разработчик рассматривает баг и назначает его направление кому-то команды разработчиков.

3) Исправлен (fixed). Разработчик, которому было назначено исправление дефекта, справляет его и сообщает о том, что задание выполнено.

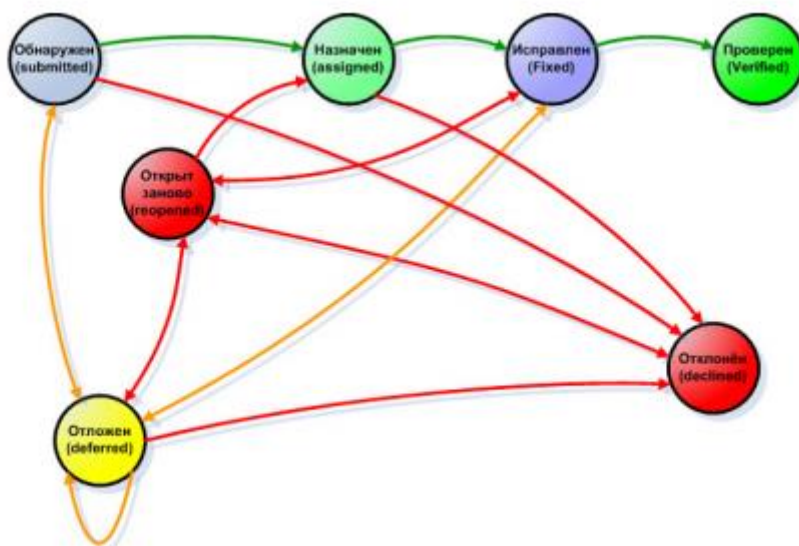


Рисунок 1.2.1 – Жизненный цикл дефекта

4) Проверен (verified). Тестировщик, который обнаружил ошибку проверяет на новом билде (в котором исправление данной ошибки заявлено), исправлен ли баг на самом деле. И только в том случае, если ошибка не проявится на новом билде, тестировщик меняет статус бага на Verified.

5) Открыт заново (reopened). Если баг проявляется на новом билде, тестировщик снова открывает этот дефект. Баг приобретает статус Reopened.

6) Отклонен (declined). Баг может быть отклонено. Во-первых, потому что для заказчика какие-то ошибки перестают быть актуальными. Во-вторых, это может случиться по вине тестировщика из-за плохого знания продукта, требований (дефекта на самом деле нет).

7) Отложен (deferred). Если исправление конкретного бага сейчас не очень важно или заказчик пока думает, или мы ждем какую-то информацию, от которой зависит исправление бага, тогда баг приобретает статус Deferred.

8) Закрытые (closed) баги. Закрытым считается баг в состояниях Проверен (verified) и Отклонен (declined).

9) Открытые (open) баги. Открытыми являются баги в состояниях Обнаружен (submitted), Назначен (assigned), Открыт заново (reopened). Иногда к открытым относят и баги в состояниях Исправлен (fixed) и Отложен (deferred).

1.3 Непрерывная интеграция

Непрерывная интеграция (CI, англ. Continuous Integration) — это практика разработки программного обеспечения, которая заключается в слиянии рабочих копий в общую основную ветвь разработки несколько раз в день и выполнении частых автоматизированных сборок проекта для скорейшего выявления потенциальных дефектов и решения интеграционных проблем. В обычном проекте, где над разными частями системы разработчики трудятся независимо, стадия интеграции является заключительной. Она может непредсказуемо задержать окончание работ. Переход к непрерывной интеграции позволяет снизить трудоёмкость интеграции и сделать её более предсказуемой за счёт наиболее раннего обнаружения и устранения ошибок и противоречий, но основным преимуществом является сокращение стоимости исправления дефекта, за счёт раннего его выявления. Непрерывная интеграция впервые названа и предложена Гради Бучем в 1991 г.

Непрерывная интеграция нацелена на ускорение и облегчение процесса выявления проблем, возникающих в процессе разработки программного обеспечения. При регулярной интеграции изменений единовременный объем проверок уменьшается. В результате на отладку тратится меньше времени, оставшееся время можно перераспределить на добавление новых функций. Также возможно добавить проверку стиля кода, цикломатической сложности (чем ниже сложность, тем легче тестировать) и другие виды контроля. Это упрощает рецензирование кода (code review), экономит время и улучшает качество кода [8].

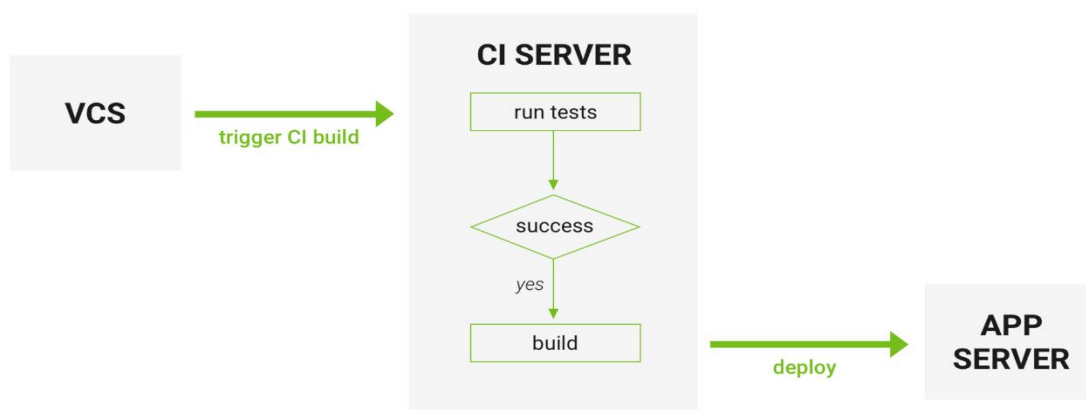


Рисунок 1.3.1– Схематичное представление непрерывной интеграции

1.3.1 Организация непрерывной интеграции

На выделенном сервере организуется служба, в задачи которой входят:

- 1) получение исходного кода из репозитория;
- 2) сборка проекта;
- 3) выполнение тестов;
- 4) развёртывание готового проекта;
- 5) отправка отчетов.

Локальная сборка может осуществляться:

- 1) по внешнему запросу;
- 2) по расписанию;
- 3) по факту обновления репозитория и по другим критериям.

Сборка по расписанию:

В случае сборки по расписанию (англ. daily build — рус. ежедневная сборка), они, как правило, проводятся каждой ночью в автоматическом режиме — ночные сборки (чтобы к началу рабочего дня были готовы результаты тестирования). Для различия дополнительно вводится система нумерации сборок — обычно, каждая сборка нумеруется натуральным числом, которое увеличивается с каждой новой сборкой. Исходные тексты и другие исходные данные при взятии их из репозитория (хранилища) системы

контроля версий помечаются номером сборки. Благодаря этому, точно такая же сборка может быть точно воспроизведена в будущем — достаточно взять исходные данные по нужной метке и запустить процесс снова. Это даёт возможность повторно выпускать даже очень старые версии программы с небольшими исправлениями.

Преимущества:

- 1) проблемы интеграции выявляются и исправляются быстро, что оказывается дешевле;
- 2) немедленный прогон модульных тестов для свежих изменений;
- 3) постоянное наличие текущей стабильной версии вместе с продуктами сборки — для тестирования, демонстрации, и т. п.;
- 4) немедленный эффект от неполного или неработающего кода приучает разработчиков к работе в итеративном режиме с более коротким циклом.

Недостатки:

- 1) затраты на поддержку работы непрерывной интеграции;
- 2) потенциальная необходимость в выделенном сервере под нужды непрерывной интеграции;
- 3) немедленный эффект от неполного или неработающего кода отучает разработчиков от выполнения периодических резервных включений кода в репозиторий.
- 4) в случае использования системы управления версиями исходного кода с поддержкой ветвления, эта проблема может решаться созданием отдельной «ветки» (англ. branch) проекта для внесения крупных изменений (код, разработка которого до работоспособного варианта займет несколько дней, но желательно более частое сохранение результата в репозиторий). По окончании разработки и индивидуального тестирования такой ветки, она может быть объединена (англ. merge) с основным кодом или «стволом» (англ. trunk) проекта [9].

1.3.2 Популярные CI-платформы.

1.3.2.1 CircleCI

Функции:

1) CircleCI — это облачная система, для которой не нужно настраивать отдельный сервер и которую не придется администрировать. Однако существует и локальная версия, которую можно развернуть в частном облаке.

2) даже для коммерческого использования существует бесплатная версия.

3) с помощью REST API можно получить доступ к проектам, сборкам и артефактам. Результатом сборки является артефакт или группа артефактов. Артефактом могут быть скомпилированное приложение или исполняемые файлы (например, APK для Android) или метаданные (например, информация об удачно завершившемся тестировании).

4) CircleCI кэширует сторонние зависимости, что позволяет избежать постоянной установки необходимых окружений.

5) существует возможность подключения к контейнеру по SSH. Это может потребоваться, если возникнут какие-то проблемы.

6) CircleCI — полностью готовое решение, требующее минимальной настройки.

Совместимость с:

Python, Node.js, Ruby, Java, Go и т. д.;

1) Ubuntu (12.04, 14.04), Mac OS X (платные аккаунты);

2) Github, Bitbucket;

3) AWS, Azure, Heroku, Docker, выделенный сервер;

4) Jira, HipChat, Slack.

Достоинства CircleCI:

1) легкое и быстрое начало работы;

2) бесплатная версия для коммерческого использования;

3) небольшие и легко читаемые файлы конфигурации в формате YAML;

4) отсутствие необходимости в выделенном сервере CircleCI.

Недостатки CircleCI:

1) CircleCI в бесплатной версии поддерживает только Ubuntu 12.04 и 14.04. Для использования MacOS придется заплатить;

2) несмотря на то что CircleCI может работать с любыми языками программирования, в базовой комплектации поддерживаются только Go (Golang), Haskell, Java, PHP, Python, Ruby/Rails, Scala;

3) При желании подстроить систему под себя в некоторых случаях могут возникнуть проблемы, и тогда для достижения цели понадобится стороннее программное обеспечение.

1.3.3 Travis CI

Travis CI:

1) использует файлы конфигурации в формате YAML;

2) развернута в облаке;

3) поддерживает Docker для запуска тестов.

Что есть в TravisCI :

1) запуск тестов одновременно под Linux и Mac OS X.

2) поддержка большого количества языков (в базовой комплектации): Android, C, C#, C++, Clojure, Crystal, D, Dart, Erlang, Elixir, F#, Go, Groovy, Haskell, Haxe, Java, JavaScript (with Node.js), Julia, Objective-C, Perl, Perl6, PHP, Python, R, Ruby, Rust, Scala, Smalltalk, Visual Basic.

3) поддержка build matrix.

Достоинства Travis CI:

1) Build matrix — это инструмент, который дает возможность выполнять тесты, используя разные версии языков и пакетов. Он обладает богатыми возможностями по настройке. Например, при неудачных сборках в некоторых окружениях система может выдать предупреждение, но сборка

целиком не будет считаться неудачной (это удобно при использовании dev-версий пакетов);

2) быстрый старт;

3) небольшие и легко читаемые файлы конфигурации в формате YAML;

4) бесплатная версия для opensource-проектов;

5) отсутствие необходимости в выделенном сервере.

Недостатки Travis CI:

1) по сравнению с CircleCI цены выше, нет бесплатной версии для коммерческого использования;

2) ограниченные возможности по настройке (для некоторых вещей может потребоваться сторонний софт).

1.3.4 Jenkins

Возможности:

1) Jenkins — это автономное приложение на Java, которое может работать под Windows, Mac OS X и другими unix-подобными операционными системами.

2) в Update Center можно найти сотни плагинов, поэтому Jenkins интегрируется практически с любым инструментом, относящимся к непрерывной интеграции и непрерывной поставке (continuous delivery).

3) возможности Jenkins могут быть практически неограниченно расширены благодаря системе подключения плагинов.

4) предусмотрены различные режимы: Freestyle project, Pipeline, External Job, Multi-configuration project, Folder, GitHub Organization, Multibranch Pipeline.

5) Jenkins Pipeline — это набор плагинов, поддерживающих создание и интеграцию в Jenkins цепочек непрерывной поставки. Pipeline предоставляет расширяемый набор инструментов по моделированию цепочек поставки типа "as code" различной степени сложности с помощью Pipeline DSL.

6) позволяет запускать сборки с различными условиями.

7) Jenkins может работать с Libvirt, Kubernetes, Docker и др.

8) используя REST API, можно контролировать количество получаемых данных, получать/обновлять config.xml, удалять задания (job), получать все сборки, получать/обновлять описание задания, выполнять сборку, включать/отключать задания.

Достоинства Jenkins:

- 1) цена (он бесплатен);
- 2) возможности по настройке;
- 3) система плагинов;
- 4) полный контроль над системой.

Недостатки Jenkins:

- 1) требуется выделенный сервер (или несколько серверов), что влечет за собой дополнительные расходы на сам сервер, DevOps и т. д.;
- 2) на настройку необходимо время [10].

1.4 Фреймворки тестирования

1.4.1 xUnit

xUnit — это собирательное название семейства фреймворков для модульного тестирования, структура и функциональность которых основана на SUnit, который предназначался для языка программирования Smalltalk. SUnit, разработанный Кентом Бекем в 1998 году, был написан в высоко структурном объектно-ориентированном стиле, получил широкую популярность и был адаптирован для множества других языков. Названия фреймворков этого семейства образованы аналогично "SUnit", обычно заменяется буква "S" на первую букву (или несколько первых) в названии предполагаемого языка ("JUnit" для Java, "NUnit" для программной платформы .NET и т. д.). Семейство таких фреймворков с общей архитектурой обычно и известно, как "xUnit".

Архитектура xUnit

Все фреймворки из семейства xUnit имеют следующие базовые компоненты архитектуры, которые в различных реализациях могут слегка варьироваться.

Модуль, выполняющий тестирование (Test runner)

Модуль представляет собой исполняемую программу, которая выполняет тесты, реализованные с помощью фреймворка, и отображает информацию о ходе их выполнения.

Тестовые сценарии (Test cases)

Варианты тестирования (тестовые сценарии/случаи) являются базовыми элементами модульных тестов.

Конфигурации тестирования (Test fixtures)

Конфигурация тестирования (также называемая контекстом) — это набор предварительно заданных условий или состояний объектов, необходимый для запуска теста. Разработчик должен задать заведомо корректную конфигурацию перед выполнением каждого теста, а затем вернуть оригинальную конфигурацию после завершения теста.

Наборы тестов (Test suites)

Тестовый набор — это несколько тестов, имеющих общую конфигурацию. Очередность выполнения тестов не должна иметь значения.

Выполнение тестов (Test execution)

Выполнение каждого теста происходит согласно следующей схеме:

```
setup(); /* Сначала подготавливается 'контекст' тестирования */  
...  
/* Тело теста - здесь указывается тестовый сценарий */  
...  
teardown(); /* После прохождения теста (независимо от его результата)  
контекст тестирования "очищается" */
```

Форматирование результатов тестирования (Test result formatter)

Модуль, выполняющий тестирование, должен вывести результаты в одном или нескольких заданных форматах. В дополнение к обычному тексту, воспринимаемому человеком, часто результаты выводятся в формате XML.

Утверждения (Assertions)

Утверждение в тесте — это функция или макрос, которая проверяет поведение или состояние тестируемого модуля. Часто утверждением является проверка равенства или неравенства некоторого параметра модуля ожидаемому результату. Неудачное прохождение проверки приводит к провалу всего тестового сценария и к исключению (если оно необходимо), которое останавливает сценарий без перехода к следующему утверждению.

Фреймворки xUnit

Фреймворки с архитектурой, характерной для xUnit, существуют для множества языков программирования и платформ разработки. Примеры:

- 1) CppUnit - фреймворк для C++;
- 2) DUnit - инструмент для среды разработки Delphi;
- 3) JUnit - библиотека для Java;
- 4) NUnit, xUnit.NET - среда юнит-тестирования для .NET;
- 5) phpUnit - библиотека для PHP. [11]

1.4.2 NUnit

NUnit — открытая среда юнит-тестирования приложений для .NET. Она была портирована с языка Java (библиотека JUnit). Первые версии NUnit были написаны на J#, но затем весь код был переписан на C# с использованием таких новшеств .NET, как атрибуты.

Существуют также известные расширения оригинального пакета NUnit, большая часть которых представлена с открытым исходным кодом. NUnit.Forms дополняет NUnit средствами тестирования элементов пользовательского интерфейса Windows Forms. NUnit.ASP выполняет ту же задачу для элементов интерфейса в ASP.NET.

Особенности

- 1) Тесты можно запускать из консоли, в Visual Studio через тестовый адаптер или через сторонние приложения;
- 2) Тесты могут выполняться параллельно;
- 3) Сильная поддержка тестов, управляемых данными;
- 4) Поддерживает несколько платформ, включая .NET Core, Xamarin Mobile, Compact Framework и Silverlight;
- 5) Каждый тестовый пример может быть добавлен в одну или несколько категорий, чтобы обеспечить избирательный запуск.

NUnit предоставляет консольное приложение (nunit3-console.exe), которое используется для пакетного выполнения тестов. Коннектор работает через NUnit Test Engine, который предоставляет ему возможность загружать, исследовать и выполнять тесты. Когда тесты должны запускаться в отдельном процессе, движок использует программу nunit-agent для их запуска.

Пример

```
using NUnit.Framework;

[TestFixture]
public class ExampleTestOfNUnit
{
    [Test]
    public void TestMultiplication()
    {
        Assert.AreEqual(4, 2*2, "Multiplication");

        // Equivalently, since version 2.4 NUnit offers a new and
        // more intuitive assertion syntax based on constraint objects
        // [http://www.nunit.org/index.php?p=constraintModel&r=2.4.7]:
        Assert.That(2*2, Is.EqualTo(4), "Multiplication constraint-based");
    }
}
```

```
}
```

// The following example shows different ways of writing the same exception test.

```
[TestFixture]
```

```
public class AssertThrowsTests
```

```
{
```

```
    [Test]
```

```
    public void Tests()
```

```
    {
```

```
        // .NET 1.x
```

```
        Assert.Throws(typeof(ArgumentException),  
            new TestDelegate(MethodThatThrows));
```

```
        // .NET 2.0
```

```
        Assert.Throws<ArgumentException>(MethodThatThrows);
```

```
        Assert.Throws<ArgumentException>(  
            delegate { throw new ArgumentException(); });
```

```
        // Using C# 3.0
```

```
        Assert.Throws<ArgumentException>(  
            () => { throw new ArgumentException(); });
```

```
    }
```

```
    void MethodThatThrows()
```

```
    {
```

```
        throw new ArgumentException();
```

```
    }
```

```
}
```

// This example shows use of the return value to perform additional verification of the exception.

```
[TestFixture]
public class UsingReturnValue
{
    [Test]
    public void TestException()
    {
        MyException ex = Assert.Throws<MyException>(
            delegate { throw new MyException("message", 42); });
        Assert.That(ex.Message, Is.EqualTo("message"));
        Assert.That(ex.MyParam, Is.EqualTo(42));
    }
}
```

// This example does the same thing using the overload that includes a constraint.

```
[TestFixture]
public class UsingConstraint
{
    [Test]
    public void TestException()
    {
        Assert.Throws(Is.TypeOf<MyException>()
            .And.Message.EqualTo("message")
            .And.Property("MyParam").EqualTo(42),
            delegate { throw new MyException("message", 42); });
    }
}
```

```
}
}
```

Расширения

1) FireBenchmarks – это расширение, способное записывать время выполнения модульных тестов и генерировать отчеты о производительности XML, CSV, XHTML с диаграммами и отслеживанием истории. Его основная цель – предоставить разработчику или команде, которая работает с гибкой методологией для интеграции показателей производительности и анализа, единую тестовую среду для легкого контроля и мониторинга эволюции программной системы с точки зрения сложности алгоритмической нагрузки и загрузки системных ресурсов.

2) NUnit.Forms – это расширение базовой структуры NUnit с открытым исходным кодом. В нем конкретно рассматривается расширение NUnit, чтобы иметь возможность обрабатывать элементы пользовательского интерфейса в Windows Forms.

3) NUnit.ASP является прекращенным расширением к основной структуре NUnit и также имеет открытый исходный код. Он специально рассматривает расширение NUnit, чтобы иметь возможность обрабатывать элементы пользовательского интерфейса в ASP.Net [12].

1.4.3 Visual Studio Unit Testing Framework

Visual Studio Unit Testing Framework интегрирован в версии Visual Studio 2005 и более поздних версий. Структура тестирования модулей определена в Microsoft.VisualStudio.QualityTools.UnitTestFramework.dll. Модульные тесты, созданные с помощью модуля тестирования, могут быть выполнены в Visual Studio или могут быть запущены с помощью MSTest.exe с параметрами из командной строки.

MSTest.exe — это запускаемая из командной строки программа, которая служит для запуска тестов. У этой программы есть несколько

параметров, с помощью которых можно настроить ее работу. Их можно указать в командной строке MSTest.exe в произвольном порядке.

Элементы тестирования:

Класс тестирования (Test class)

Тестовые классы объявляются с помощью атрибута `TestClass`. Атрибут используется для идентификации классов, содержащих методы тестирования.

Метод тестирования (Test method)

Методы тестирования объявляются атрибутом `TestMethod`. Атрибут используется для идентификации методов, которые содержат единичный тестовый код.

Утверждения (Assertions)

Утверждение представляет собой фрагмент кода, который выполняется, чтобы проверить состояние или поведение в отношении ожидаемого результата. Утверждения в модульном тестировании Visual Studio выполняются путем вызова методов в классе `Assert`.

Инициализация и методы очистки (Initialization and cleanup methods)

Методы инициализации и очистки используются для подготовки модульных тестов перед запуском и очисткой после выполнения модульных тестов. Методы инициализации объявляются путем добавления атрибута `TestInitialize`, тогда как методы очистки объявляются путем добавления атрибута `TestCleanup` [13].

1.5 Обзор систем контроля версий

Системы контроля версий стали неотъемлемой частью жизни не только разработчиков программного обеспечения, но и всех людей, столкнувшихся с проблемой управления интенсивно изменяющейся информацией, и желающих облегчить себе жизнь. Вследствие этого, появилось большое число различных продуктов, предлагающих широкие возможности и предоставляющих обширные инструменты для управления версиями. В этой статье будут кратко рассмотрены наиболее популярные из них, приведены их достоинства и недостатки.

Для сравнения были выбраны наиболее распространенные системы контроля версий: Subversion, Git, Mercurial [14].

1.5.1 Система управления версиями Subversion.

SVN создавалась как альтернатива CVS с целью исправить недостатки CVS и в то же время обеспечить высокую совместимость с ней.

Как и CVS, SVN это бесплатная система контроля версий с открытым исходным кодом. С той лишь разницей, что распространяется под лицензией Apache, а не под Открытым лицензионным соглашением GNU.

Для сохранения целостности базы данных SVN использует так называемые атомарные операции. При появлении новой версии к финальному продукту применяются либо все исправления, либо ни одно из них. Таким образом, код защищают от хаотичных частичных правок, которые не согласуются между собой и вызывают ошибки.

Многие разработчики переключились на SVN, так как новая технология унаследовала лучшие возможности CVS и в то же время расширила их.

В то время как в CVS операции с ветками кода дорогостоящие и не предусмотрены архитектурой системы, SVN создана как раз для этого. То есть, для более крупных проектов с ветвлением кода и многими направлениями разработки.

В качестве недостатков SVN упоминаются сравнительно низкая скорость и нехватка распределенного управления версиями. Распределенный контроль версий использует пиринговую модель, а не централизованный сервер для хранения обновлений программного кода. И хотя пиринговая модель работает лучше в open source проектах, она не идеальна в других случаях. Недостаток серверного подхода в том, что когда сервер падает, то у клиентов нет доступа к коду.

Достоинства:

- 1) разнообразные графические интерфейсы и удобная работа из консоли;
- 2) отслеживается история изменения файлов и каталогов даже после их переименования и перемещения;
- 3) высокая эффективность работы, как с текстовыми, так и с бинарными файлами;
- 4) встроенная поддержка во многие интегрированные средства разработки, такие как KDevelop, Zend Studio и многие другие;
- 5) возможность создания зеркальных копий репозитория;
- 6) два типа репозитория – база данных или набор обычных файлов;
- 7) возможность доступа к репозиторию через Apache с использованием протокола WebDAV;
- 8) наличие удобного механизма создания меток и ветвей проектов;
- 9) можно с каждым файлом и директорией связать определенный набор свойств, облегчающий взаимодействие с системой контроля версии;
- 10) широкое распространение позволяет быстро решить большинство возникающих проблем, обратившись к данным, накопленным Интернет-сообществом.

Недостатки:

- 1) полная копия репозитория хранится на локальном компьютере в скрытых файлах, что требует достаточно большого объема памяти;
- 2) существуют проблемы с переименованием файлов, если

переименованный локально файл одним клиентом был в это же время изменен другим клиентом и загружен в репозиторий;

3) слабо поддерживаются операции слияния веток проекта;

4) сложности с полным удалением информации о файлах, попавших в репозиторий, так как в нем всегда остается информация о предыдущих изменениях файла, и не предусмотрено никаких штатных средств для полного удаления данных о файле из репозитория.

1.5.2 Система управления версиями Git.

С февраля 2002 года для разработки ядра Linux'a большинством программистов стала использоваться система контроля версий BitKeeper. Довольно долгое время с ней не возникало проблем, но в 2005 году Лари МакВоем (разработчик BitKeeper'a) отозвал бесплатную версию программы. Разрабатывать проект масштаба Linux без мощной и надежной системы контроля версий – невозможно. Одним из кандидатов и наиболее подходящим проектом оказалась система контроля версий Monotone, но Торвальдса Линуса не устроила ее скорость работы. Так как особенности организации Monotone не позволяли значительно увеличить скорость обработки данных, то 3 апреля 2005 года Линус приступил к разработке собственной системы контроля версий – Git.

Практически одновременно с Линусом (на три дня позже), к разработке новой системы контроля версий приступил, и Мэтт Макал. Свой проект Мэтт назвал Mercurial, но об этом позже, а сейчас вернемся к распределенной системе контроля версий Git.

Git – это гибкая, распределенная (без единого сервера) система контроля версий, дающая массу возможностей не только разработчикам программных продуктов, но и писателям для изменения, дополнения и отслеживания изменения «рукописей» и сюжетных линий, и учителям для корректировки и развития курса лекций, и администраторам для ведения документации, и для многих других направлений, требующих управления историей изменений.

У каждого разработчика, использующего Git, есть свой локальный репозиторий, позволяющий локально управлять версиями. Затем, сохраненными в локальный репозиторий данными, можно обмениваться с другими пользователями.

Часто при работе с Git создают центральный репозиторий, с которым остальные разработчики синхронизируются. Пример организации системы с центральным репозиторием – это проект разработки ядра Linux’а.

В этом случае все участники проекта ведут свои локальные разработки и беспрепятственно скачивают обновления из центрального репозитория. Когда необходимые работы отдельными участниками проекта выполнены и отлажены, они, после удостоверения владельцем центрального репозитория в корректности и актуальности проделанной работы, загружают свои изменения в центральный репозиторий.

Наличие локальных репозиториями также значительно повышает надежность хранения данных, так как, если один из репозиториях выйдет из строя, данные могут быть легко восстановлены из других репозиториях. Работа над версиями проекта в Git может вестись в нескольких ветках, которые затем могут с легкостью полностью или частично объединяться, уничтожаться, откатываться и разрастаться во все новые и новые ветки проекта.

Можно долго обсуждать возможности Git’а, но для краткости и более простого восприятия приведем основные достоинства и недостатки этой системы управления версиями.

Достоинства:

- 1) надежная система сравнения ревизий и проверки корректности данных, основанные на алгоритме хеширования (Secure Hash Algorithm 1);
- 2) гибкая система ветвления проектов и слияния веток между собой;
- 3) наличие локального репозитория, содержащего полную информацию обо всех изменениях, позволяет вести полноценный локальный контроль версий и заливать в главный репозиторий только полностью прошедшие

проверку изменения;

4) высокая производительность и скорость работы;

5) удобный и интуитивно понятный набор команд;

6) множество графических оболочек, позволяющих быстро и качественно вести работы с Git'ом;

7) возможность делать контрольные точки, в которых данные сохраняются без дельты компрессии, а полностью. Это позволяет уменьшить скорость восстановления данных, так как за основу берется ближайшая контрольная точка, и восстановление идет от нее. Если бы контрольные точки отсутствовали, то восстановление больших проектов могло бы занимать часы;

8) широкая распространенность, легкая доступность и качественная документация;

9) гибкость системы позволяет удобно ее настраивать и даже создавать специализированные контроля системы или пользовательские интерфейсы на базе git;

10) универсальный сетевой доступ с использованием протоколов http, ftp, rsync, ssh и др.

Недостатки:

1) Unix – ориентированность. На данный момент отсутствует зрелая реализация Git, совместимая с другими операционными системами;

2) возможные (но чрезвычайно низкие) совпадения хеш - кода отличных по содержанию ревизий.³ Не отслеживается изменение отдельных файлов, а только всего проекта целиком, что может быть неудобно при работе с большими проектами, содержащими множество несвязных файлов;

3) при начальном (первом) создании репозитория и синхронизации его с другими разработчиками, потребуется достаточно длительное время для скачивания данных, особенно, если проект большой, так как требуется скопировать на локальный компьютер весь репозиторий;

1.5.3 Система управления версиями Mercurial.

Распределенная система контроля версий Mercurial разрабатывалась Мэттом Макалом параллельно с системой контроля версий Git, созданной Торвальдсом Линусом.

Первоначально, она была создана для эффективного управления большими проектами под Linux'ом, а поэтому была ориентирована на быструю и надежную работу с большими репозиториями. На данный момент mercurial адаптирован для работы под Windows, Mac OS X и большинство Unix систем.

Большая часть системы контроля версий написана на языке Python, и только отдельные участки программы, требующие наибольшего быстродействия, написаны на языке Си.

Идентификация ревизий происходит на основе алгоритма хеширования SHA1 (Secure Hash Algorithm 1), однако, также предусмотрена возможность присвоения ревизиям индивидуальных номеров.

Так же, как и в git'е, поддерживается возможность создания веток проекта с последующим их слиянием.

Для взаимодействия между клиентами используются протоколы HTTP, HTTPS или SSH.

Набор команд - простой и интуитивно понятный, во многом схожий с командами subversion. Так же имеется ряд графических оболочек и доступ к репозиторию через веб-интерфейс. Немаловажным является и наличие утилит, позволяющих импортировать репозитории многих других систем контроля версий.

Достоинства:

- 1) быстрая обработка данных;
- 2) кроссплатформенная поддержка;
- 3) возможность работы с несколькими ветками проекта;
- 4) простота в обращении;
- 5) возможность конвертирования репозиториях других систем

поддержки версий, таких как CVS, Subversion, Git, Darcs, GNU Arch, Bazaar и др.

Недостатки:

- 1) возможные (но чрезвычайно низкие) совпадения хеш - кода отличных по содержанию ревизий;
- 2) ориентирован на работу в консоли [15].

1.6 TestLink

TestLink - это веб - система управления тестированием, которая облегчает обеспечение качества программного продукта. TestLink разработан и поддерживается Teamtest. Платформа предлагает поддержку тестовых примеров, наборов тестов, планов тестирования, тестовых проектов и управления пользователями, а также различных отчетов и статистических данных

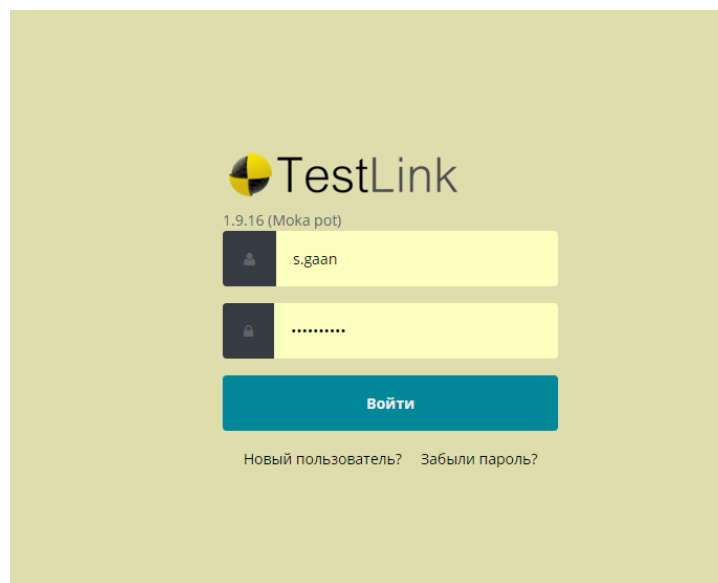


Рисунок 1.6.1 – Форма входа в TestLink

1.6.1 Использование

Основными единицами, используемыми TestLink, являются: тестовый пример, тестовый комплект, план тестирования, тестовый проект и пользователь.

План тестирования

Тестовые планы - это базовая единица для выполнения набора тестов в приложении. Планы испытаний включают сборку, назначение пользователя и результаты испытаний.

План тестирования содержит имя, описание, коллекцию выбранных тестовых случаев, сборки, результаты тестирования, назначение тестировщика и определение приоритета. Каждый план тестирования связан с текущим проектом тестирования.

Планы тестирования могут быть созданы на странице «Управление планом тестирования» пользователями с ведущими привилегиями для текущего проекта тестирования.

Определение плана тестирования состоит из заголовка, описания (html-формат) и статуса «Активный». Описание должно включать следующую информацию в отношении процессов компании:

- 1) проверяемые характеристики;
- 2) особенности, которые нельзя тестировать;
- 3) критерии испытаний (для прохождения тестируемого продукта);
- 4) тестовая среда, Инфраструктура;
- 5) инструменты тестирования;
- 6) риски.

Планы испытаний состоят из тестовых случаев, импортированных из тестовой спецификации в определенный момент времени. Планы испытаний могут быть созданы из других тестовых планов. Это позволяет пользователям создавать тестовые планы из тестовых случаев, которые существуют в нужный момент времени. Это может потребоваться при создании тестового плана для патча. Чтобы пользователь мог видеть План тестирования, они должны обладать надлежащими правами. Права могут быть назначены (по указанию) в разделе «Определить права пользователя / проекта». Это важно помнить, когда пользователи говорят, что они не могут видеть проект, над которым они работают.

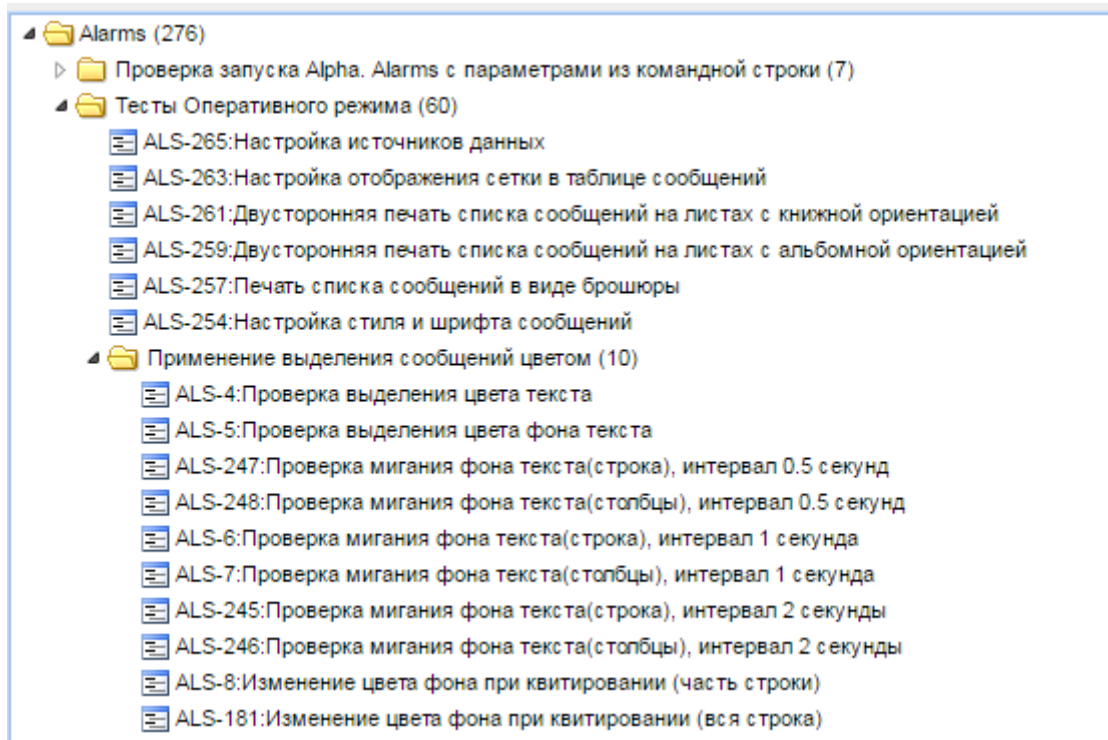


Рисунок 1.6.2 – Пример тестового плана

Тестовый пример

В тестовом примере описывается простая задача в рабочем процессе приложения. Тест является фундаментальной частью TestLink. Тестовые примеры организованы в тестовые комплекты:

1) идентификатор тестового примера автоматически назначается TestLink и не может быть изменен пользователями. Этот идентификатор состоит из префикса Test Project и счетчика, связанного с тестовым проектом, в котором создан тестовый пример;

2) название: может содержать краткое описание или аббревиатуру (например, TL-USER-LOGIN);

3) резюме: должно быть очень короткое; просто для обзора, введения и ссылок;

4) шаги: описание тестового сценария (входные действия); может также включать предварительное условие и информацию для очистки здесь;

5) ожидаемые результаты: описать контрольные точки и ожидаемое поведение тестируемого продукта или системы;

6) вложения: могут быть добавлены, если конфигурация позволяет это;

7) важность: Дизайнер тестов мог установить важность теста [HIGH, MEDIUM и LOW];

8) тип выполнения: Дизайнер тестов мог установить поддержку автоматизации теста [MANUAL / AUTOMATED];

9) пользовательские поля: администратор может определять собственные параметры для улучшения описания тестового примера или категоризации. Большие пользовательские поля (более 250 символов) невозможны. Но информация может быть добавлена в родительский тестовый набор и передаваться через настраиваемые поля. Например, вы можете описать «Стандарт», «Производительность», «Стандарт_2» и «Ссылаться через CF» на эти метки.

Версия 1			
Описание теста: цель, сценарий и исходное состояние программы			
Условия			
#	Шаги	Ожидаемая реакция	Прогон
1	Использовать конфигурацию <code>TestingCFG.xmlcfg</code> ;		Ручной
2	Воспользоваться конфигурационными файлами из папки «Проверка генерации»		Ручной
3	Запустить Alpha.Alarms		Ручной
4	Настроить генерацию событий.	В Области отображения событий сообщения отображаются со стилем Microsoft Sans Serif, начертанием обычным и 9 размером шрифта.	Ручной
5	В окне Параметры в узле Вид таблицы нажать на "..." рядом с полем Стиль шрифта	Открывается окно Шрифт	Ручной
6	Выбрать: <ul style="list-style-type: none"> шрифт Comic Sans MS начертание наклонное размер 11 и нажать на ОК		Ручной
7	Нажать на Сохранить	Текст и стиль сообщений и заголовков столбцов меняется на установленные на предыдущем шаге	Ручной
Статус : Ready for review Важность : Средняя Способ прогона : Ручной Ожидаемая продолжительность (мин) : 5.00			

Рисунок 1.6.3 – Изображение тестового примера

Пользователь

Каждый пользователь TestLink имеет назначенную роль, которая определяет доступные функции. Типы по умолчанию: Guest, Test Designer, Senior tester, Tester, Leader и Administrator, но также могут быть созданы пользовательские роли.

Тестовые проекты

Тестовые проекты - это основная организационная единица TestLink. Тестовые проекты могут быть продуктами или решениями вашей компании,

которые могут менять свои функции и функциональность с течением времени, но по большей части остаются теми же. Проект тестирования включает в себя документацию по требованиям, спецификацию тестирования, тестовые планы и конкретные права пользователей. Проекты тестирования независимы и не делят данные.

Технические характеристики теста

TestLink разбивает структуру тестовых спецификаций на тестовые объекты и тестовые случаи. Эти уровни сохраняются во всех приложениях. Один тестовый проект имеет только одну тестовую спецификацию [16].

6.2 Особенности

- 1) аоли пользователей и управление ими;
- 2) группировка тестовых примеров в тестовых спецификациях;
- 3) планы испытаний;
- 4) платформы;
- 5) требования к версии и ревизии;
- 6) поддержка тестирования различных сборок программного обеспечения;
- 7) отчеты, графики и мониторы;
- 8) настройка пользовательского интерфейса с использованием шаблонов Smarty;
- 9) интеграция с LDAP;
- 10) интеграция с другим программным обеспечением с использованием предоставленного API;
- 11) интеграция системы отслеживания ошибок (Mantis, JIRA, Bugzilla, FogBugz, Redmine и другие).

Вывод по Главе 1

В первой главе были рассмотрены теоретические аспекты процесса тестирования. Определено понятие тестирование программного продукта. Кроме того, были выделены основные классификации видов тестирования:

- 1) по объекту тестирования;
- 2) тестирование, связанное с изменениями;
- 3) по уровню тестирования;
- 4) по исполнению кода;
- 5) по субъекту тестирования;
- 6) по позитивности сценария;
- 7) по степени автоматизации.

Так же были изучены различные методологии тестирования:

- 1) метод черного ящика;
- 2) метод белого ящика;
- 3) метод серого ящика.

Были изучены аспекты разработки и выполнения различных тест-кейсов и проанализированы основные атрибуты, которые включает в себя любой тест-кейс. Также изучены требования, которые применяются при составлении тест-кейсов. Кроме того, были описаны и проанализированы различные виды дефектов, которые могут присутствовать в любом ПО и был изучен жизненный цикл любого дефекта.

Был проведен анализ популярных CI платформ, таких как CircleCI, Travis CI, Jenkins. Были выявлены достоинства и недостатки данных систем непрерывной интеграции и была представлена схема непрерывной интеграции при использовании CI платформ.

Были проанализированы различные фреймворки тестирования: xUnit, NUnit, Visual Studio Unit Testing Framework, а также были выявлены их достоинства и недостатки.

Далее были изучены системы контроля версий, такие как: Subversion, Git, Mercurial. Были проанализированы их достоинства и недостатки, а также способы работы с ними.

После этого была рассмотрена система TestLink. Так как основная часть тестов для ручного тестирования на предприятии размещена в данной системе. Были изучены способы работы с данной системой, а также ее достоинства и недостатки.

2 Глава 2

2.1 Alpha.Alarms

Приложение Alpha.Alarms используется в пунктах автоматизации и мониторинга технологических процессов. Применяется для отслеживания событий и тревог, которые появляются при изменении состояний технологических объектов. Основные функции приложения:

- 1) отображение сообщений о событиях и тревогах в режиме реального времени (оперативный режим);
- 2) отображение истории сообщений о событиях и тревогах за прошедшие периоды (исторический режим).

Alpha.Alarms может использоваться как встраиваемый компонент или работать как самостоятельное приложение. В качестве встраиваемого компонента Alpha.Alarms может использоваться в Alpha.HMI, а также в HMI SCADA систем сторонних разработчиков - Genesis, InfinitySCADA, iFix и других. [17]

2.1.1 Системные требования

1. Операционная система: MS Windows 7/2008 Server;
2. Разрядность ОС: x64 или x32;
3. Процессор: Intel Celeron 1.6 ГГц и выше;
4. Объем оперативной памяти: не менее 1 ГБ;
5. Объем дисковой памяти: не менее 500 МБ;
6. Сетевой адаптер: Ethernet 10/100/1000 Мбит/с;

2.1.2 Варианты запуска Alpha.Alarms

2.1.2.1 Стандартный запуск

Для стандартного запуска Alpha.Alarms, как отдельного приложения, воспользуйтесь командой Пуск → Все программы → Automiq → Alpha.Alarms → Alpha.Alarms или Пуск → Все программы → Automiq → Alpha.Alarms → Alpha.Alarms (x64). Аналогичное действие можно выполнить, запустив исполняемый файл Alpha.Alarms.App.exe,

расположенный в папке C:\Program Files (x86)\Automiq\Alpha.Alarms или C:\Program files\Automiq\Alpha.Alarms.

2.1.2.2 Запуск с параметрами


Использование параметров при запуске Alpha.Alarms позволяет пользователю задавать предварительные установки конфигурационных данных, заменяющие настройки по умолчанию. Для запуска приложения с параметрами используется командная строка, вызываемая командой cmd в строке поиска, либо с помощью команды Пуск → Все программы → Стандартные → Командная строка.

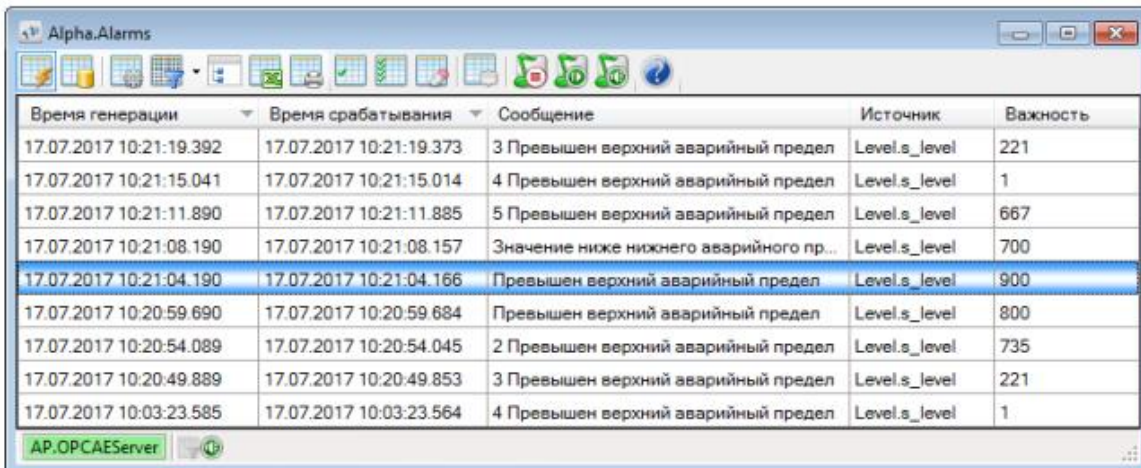
2.1.2.3 Запуск Alpha.Alarms как встраиваемого компонента

Для внедрения встраиваемого компонента Alpha.Alarms в существующий проект автоматизации необходимо добавить новый объект типа Alpha.Alarms в приложение-контейнер. Принцип добавления встраиваемого компонента может различаться в зависимости от приложения-контейнера. В качестве контейнера может выступать приложение Alpha.HMI, а также HMI SCADA систем сторонних разработчиков - Genesis, InfinitySCADA, iFix и других. Ниже приведено несколько примеров добавления компонента Alpha.Alarms.

2.1.3 Просмотр событий

2.1.3.1 Просмотр оперативных событий

Оперативный режим предназначен для поступления оповещений о событиях в режиме реального времени. Для перехода в оперативный режим предназначена кнопка  (Оперативный режим) на панели инструментов или аналогичная команда контекстного меню. Сообщения о событиях отображаются в режиме журнала, в котором каждому событию отводится отдельная строка или в режиме отображения только активных событий. Вид окна приложения, отображающего оперативные события, приведен на рисунке ниже.



Время генерации	Время срабатывания	Сообщение	Источник	Важность
17.07.2017 10:21:19.392	17.07.2017 10:21:19.373	3 Превышен верхний аварийный предел	Level.s_level	221
17.07.2017 10:21:15.041	17.07.2017 10:21:15.014	4 Превышен верхний аварийный предел	Level.s_level	1
17.07.2017 10:21:11.890	17.07.2017 10:21:11.885	5 Превышен верхний аварийный предел	Level.s_level	667
17.07.2017 10:21:08.190	17.07.2017 10:21:08.157	Значение ниже нижнего аварийного пр...	Level.s_level	700
17.07.2017 10:21:04.190	17.07.2017 10:21:04.166	Превышен верхний аварийный предел	Level.s_level	900
17.07.2017 10:20:59.690	17.07.2017 10:20:59.684	Превышен верхний аварийный предел	Level.s_level	800
17.07.2017 10:20:54.089	17.07.2017 10:20:54.045	2 Превышен верхний аварийный предел	Level.s_level	735
17.07.2017 10:20:49.889	17.07.2017 10:20:49.853	3 Превышен верхний аварийный предел	Level.s_level	221
17.07.2017 10:03:23.585	17.07.2017 10:03:23.564	4 Превышен верхний аварийный предел	Level.s_level	1

Рисунок 2.1.1– Оперативный режим

Способ отображения сообщений о событиях зависит от важности событий. Важность событий принимает значения от 1 до 1000.


Любое событие может относиться к одной из трех стандартных групп важности:

- 1) важные;
- 2) значительные (важность по умолчанию: от 334 до 666);
- 3) особой важности (важность по умолчанию: от 667 до 1000).

2.1.3.1.1 Квитирование событий


Квитированием события называется отметка о прочтении сообщения о событии, которая выставляется пользователем Alpha.Alarms и фиксируется на стороне АЕ-сервера вместе со служебной информацией о факте квитирования.

Способы квитирования:


1) выделите одно или несколько событий в таблице и нажмите кнопку  (Квитировать) на панели инструментов или в контекстном меню.

3) повторно нажмите левой кнопкой мыши по выделенному событию (способ работает, если установлен флаг квитировать события повторным кликом **по** выделенной строке в окне Параметры).


4) нажмите на кнопку квитировать в столбце Квитировать

5) квитируйте все отображаемые события кнопкой  (Квитируйте все) на панели инструментов или аналогичной командой в контекстном меню.


2.1.3.1.2 Приостановка поступления событий

Чтобы временно приостановить поступление новых событий в таблицу, включите режим Снимок кнопкой  (Снимок) на Панели инструментов или аналогичной командой контекстного меню. При переводе программы в данный режим вновь приходящие сообщения не отображаются, при этом продолжается поступление уведомлений в память приложения и проигрывание звуков поступивших сообщений. Все сообщения, пришедшие во время активности режима Снимок, отобразятся после его отключения.

2.1.3.1.3 Очистка списка событий

Чтобы очистить список событий оперативного режима, воспользуйтесь кнопкой  (Очистить список) на панели инструментов либо аналогичной командой контекстного меню.

2.1.3.2 Просмотр истории событий

Исторический режим предназначен для просмотра событий за прошедший период. Для перехода в исторический режим нажмите кнопку  (Исторический режим) на панели инструментов или аналогичную команду контекстного меню.

Для просмотра истории событий нужно установить временной интервал выборки данных, выбрать хронологию запрашиваемых данных и запросить данные у источника.

2.1.3.2.1 Установка временного интервала для запроса

Для установки временного интервала, за который требуется запросить данные, следует задать начальное и конечное значение интервала в полях ввода на панели инструментов. Формат даты DD.MM.YYYY hh:mm:ss.

Начало интервала: 31 октября 2013 г. ▼ 0:00:00 ▲ Конечн интервала: 2 ноября 2013 г. ▼ 0:00:00 ▲

Рисунок 2.1.2 – Установка временного интервала

Установка даты производится путем ввода значений с клавиатуры либо с помощью встроенного календаря. Календарь открывается при нажатии кнопок ▼, расположенных рядом с полем ввода даты.

2.1.4 Дополнительные возможности

2.1.4.1 Фильтрация сообщений о событиях

Для исключения лишних событий из таблицы воспользуйтесь фильтрацией по различным критериям.

Типы фильтрации на стороне источника данных:

1) фильтрация оперативных событий на стороне сервера применяется для ограничения объема отправляемых OPC AE-сервером оперативных событий.

2) фильтр запроса - применяется в момент запроса исторических данных, что уменьшает объем результирующей выборки, поступающей от источника исторических данных.

Типы фильтрации на стороне приложения:

1) фильтр отображения применяется для скрытия лишних событий, присутствующих в данный момент в таблице. Может применяться как в оперативном, так и историческом режиме.

2) фильтрация по объекту срабатывания применяется для отображения оперативных событий определенных технологических объектов.


По форме установки фильтры могут быть:

1) предустановленными - хранятся в приложении для многократного использования.

2) пользовательскими - задаются для использования в рамках текущего сеанса работы с приложением.

2.1.4.2 Сохранение данных в табличный файл

В приложении предусмотрена возможность сохранения сообщений, отображаемых в главном окне, в табличный файл. Для сохранения

сообщений в табличный файл предназначена кнопка  (Сохранить...), расположенная на панели инструментов, а также аналогичная команда в контекстном меню. Возможность сохранения недоступна если таблица сообщений пуста.


Сохранение таблицы сообщений производится в форматах XLSX, XML, CSV.


Для сохранения в табличный файл дополнительной информации (колоннитул, дата печати и т.д.) отметьте флагами нужные элементы в группе Содержание документа узла настроек Печать.

1	А	В	С	Д	Е	Г	Г
2	Текст верного колоннитула						
3	Время генерации	Время срабатывания	Сообщение	Источник события	Важность	Имитировано	Источник данных
4	13.02.2017 08:55:55.910	13.02.2017 08:55:55.864	Энергоснабжение. ВВ ввод №1 ЗРУ 10кВ - отключен. Команда с АРМ сеть режим Имитация логического "0"	NPS.ZRU.ABС855.Msig	3	Нет	Source1
5	13.02.2017 08:55:55.910	13.02.2017 08:55:55.864	Энергоснабжение. ВВ ввод №2 ЗРУ 10кВ - отключен. Команда с АРМ сеть режим Имитация логического "0"	NPS.ZRU.ABС856.Msig	3	Нет	Source1
6	13.02.2017 08:55:55.910	13.02.2017 08:55:55.864	Энергоснабжение. Сводный ВВ ЗРУ 10кВ - отключен. Команда с АРМ сеть режим Имитация логического "0"	NPS.ZRU.ABС861.Msig	3	Нет	Source1
7	13.02.2017 08:55:55.909	13.02.2017 08:55:55.864	Энергоснабжение. ЦСУ ЯМ. Контроль наличия напряжения на I СШ. Команда с АРМ сеть режим Имитация логического "0"	NPS.ZRU.EC851.Msig	3	Нет	Source1
8	13.02.2017 08:55:55.909	13.02.2017 08:55:55.864	Энергоснабжение. ЦСУ ЯМ. Контроль наличия напряжения на II СШ. Команда с АРМ сеть режим Имитация логического "0"	NPS.ZRU.EC852.Msig	3	Нет	Source1
9	13.02.2017 08:55:55.909	13.02.2017 08:55:55.864	Энергоснабжение. ЦСУ ЯМ. Контроль наличия напряжения на I СШ. Команда с АРМ сеть режим Имитация логического "0"	NPS.ZRU.EC853.Msig	3	Нет	Source1
10	13.02.2017 08:55:55.909	13.02.2017 08:55:55.864	Энергоснабжение. ЦСУ ЯМ. Контроль наличия напряжения на II СШ. Команда с АРМ сеть режим Имитация логического "0"	NPS.ZRU.EC854.Msig	3	Нет	Source1
11	13.02.2017 08:55:55.909	13.02.2017 08:55:55.864	Энергоснабжение. Контроль наличия напряжения НПС-2х630 на I СШ. Команда с АРМ сеть режим Имитация логического "0"	NPS.ZRU.EC871.Msig	3	Нет	Source1
12	13.02.2017 08:55:55.909	13.02.2017 08:55:55.864	Энергоснабжение. Контроль наличия напряжения НПС-2х630 на II СШ. Команда с АРМ сеть режим Имитация логического "0"	NPS.ZRU.EC872.Msig	3	Нет	Source1
13	13.02.2017 08:55:55.857	13.02.2017 08:55:55.850	САРД. Заслонка №008.P2. Изменена зона регулирования скорости по давлению на приемке МНС	NPS.URD.SARD.Valve2.Msig	4	Нет	Source1
14	13.02.2017 08:55:55.857	13.02.2017 08:55:55.849	САРД. Заслонка №008.P1. Изменена зона регулирования скорости по давлению на приемке МНС	NPS.URD.SARD.Valve1.Msig	4	Нет	Source1
15	13.02.2017 08:55:55.857	13.02.2017 08:55:55.849	САРД. Уставка давления на входе МНС. Изменено текущее значение	NPS.URD.SARD.Ust_Pln.Msig	4	Нет	Source1
16	13.02.2017 08:55:55.857	13.02.2017 08:55:55.849	САРД. Уставка положения заслонки DB.P2. Изменено текущее значение	PS.URD.SARD.Ust_Pos_1.Msig	4	Нет	Source1
17	13.02.2017 08:55:55.857	13.02.2017 08:55:55.849	САРД. Уставка положения заслонки DB.P2. Изменено текущее значение	PS.URD.SARD.Ust_Pos_2.Msig	4	Нет	Source1
18	13.02.2017 08:55:55.857	13.02.2017 08:55:55.849	САРД. Уставка давления на выходе НПС. Изменено текущее значение	PS.URD.SARD.Ust_Pout.Msig	4	Нет	Source1
19	13.02.2017 08:55:55.856	13.02.2017 08:55:55.844	УПС "Ж-Б". Задаются №015.1.4/131. Отвернется. Выполнение команды не требуется	PS.URP_X_4.VLV_015_2_4.Msig	3	Нет	Source1
20	13.02.2017 08:56						
21	Оператор (Иванов И.И.)		[подпись]				
22							

Рисунок 2.1.3 – Сохранение данных в табличный файл

2.1.4.3 Импорт/экспорт предустановленных фильтров

Чтобы сохранить предустановленные фильтры в XML-файл, перейдите в узел Предустановленные фильтры окна Параметры и нажмите кнопку  (Экспорт фильтров...).

Чтобы импортировать предустановленные фильтры из XML-файла, нажмите  (Импорт фильтров...). Импортируемый фильтр добавится в список предустановленных.

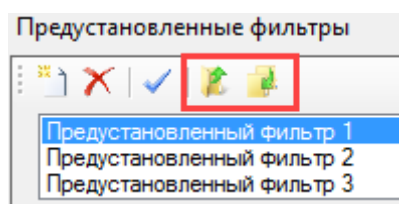


Рисунок 2.1.4– Импорт/экспорт предустановленных фильтров

2.1.4.4 Настройка папок для импорта и экспорта

Чтобы указать папки, которые будут предлагаться по умолчанию для операций импорта/экспорта, перейдите в узел Экспорт окна Параметры и укажите пути к папкам для разных категорий файлов.

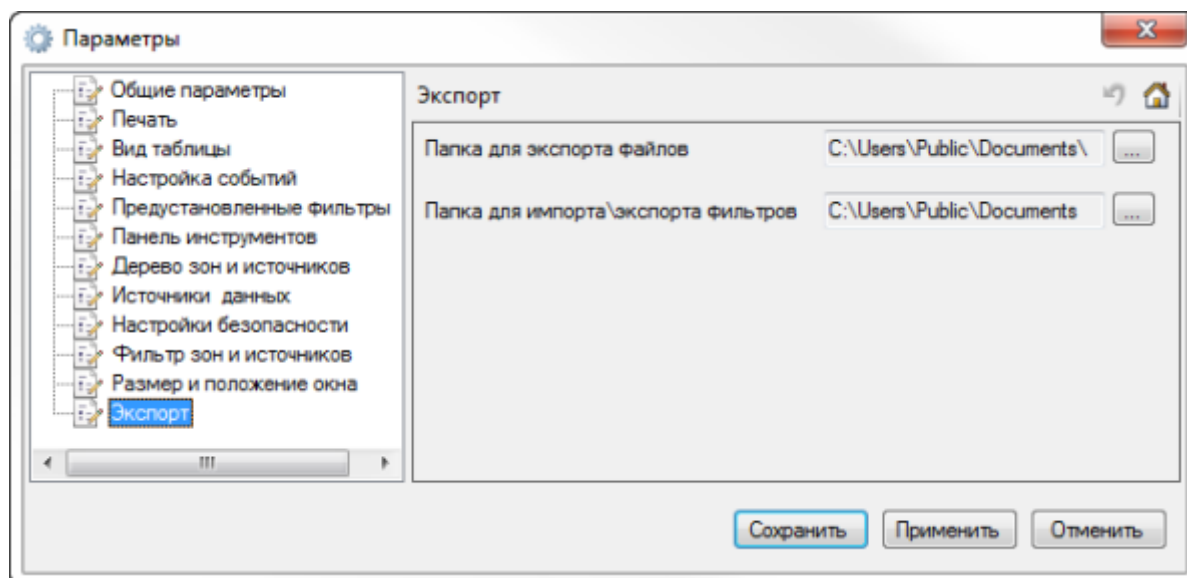



Рисунок 2.1.5– Настройка папок для импорта и экспорта

2.1.4.5 Печать таблицы событий

Для печати таблицы событий, предназначена кнопка  (Печать...), расположенная на панели инструментов или аналогичная команда в контекстном меню.

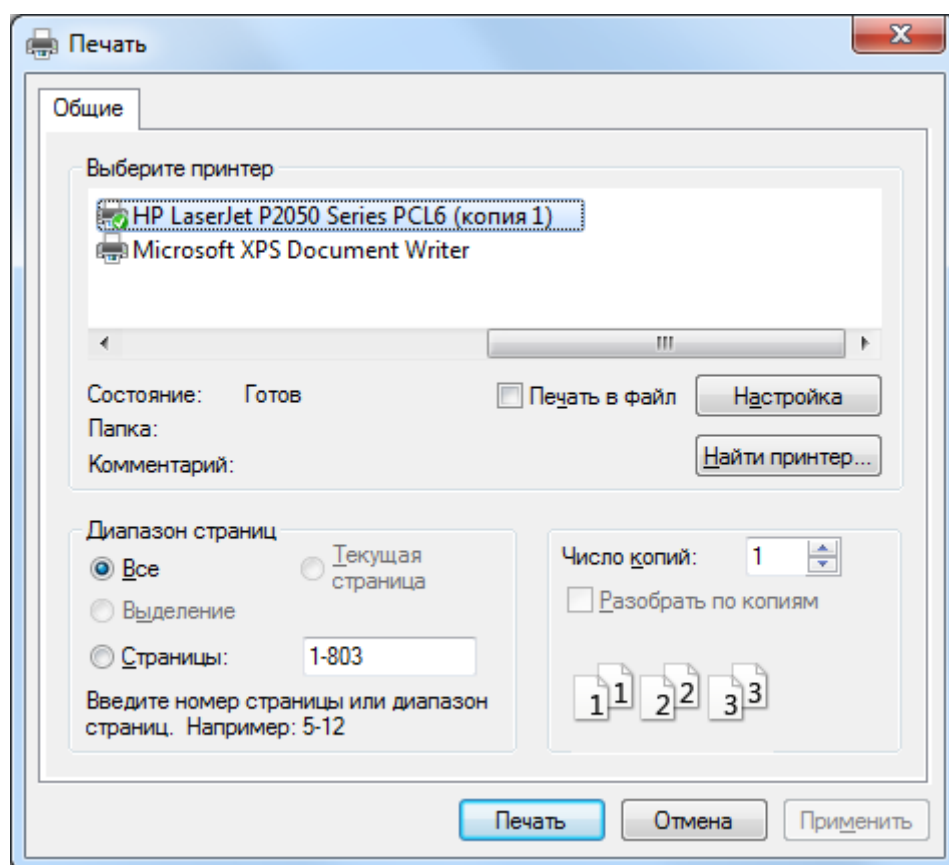


Рисунок 2.1.6 – Печать таблицы событий

2.1.4.6 Откат настроек приложения

Для отмены последних настроек, произведенных на определенном узле окна Параметры, воспользуйтесь командой отменить внесенные изменения.

Для полного восстановления настроек по умолчанию для определенного узла окна Параметры воспользуйтесь командой вернуться к значениям по умолчанию.

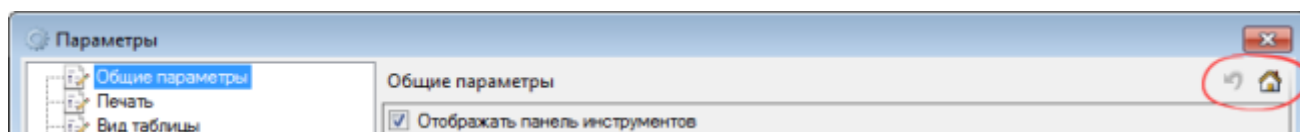


Рисунок 2.1.7– Откат настроек приложения

2.2 Автоматизация процесса тестирования

Основными задачами при внедрении автоматизированного тестирования являются:

1) повышение качества тестирования. С помощью автоматизированного тестирования можно охватить больший набор тестов, и/или тесты, которые невозможно осуществить вручную. Кроме того, автоматическое выполнение тестов позволяет свести к минимуму влияние человеческого фактора на результаты тестирования;

2) экономия времени, затрачиваемого на тестирование. На выполнение автоматизированных тестов обычно уходит значительно меньше времени, чем на выполнение аналогичных тестов вручную.

К преимуществам автоматизированного тестирования относится:

1) сокращение времени на исполнение тестов в сравнении с ручным тестированием;

2) возможность проведения тестов, которые нельзя провести вручную;

3) исключение человеческого фактора;

4) возможность проводить тестирование вне рабочих часов тестировщика.

Необходимо также отметить и недостатки внедрения автоматизации тестирования [3];

1) автоматизация тестирования зачастую требует значительных трудозатрат;

2) требуется более квалифицированный персонал в сравнении с ручным тестированием;

3) более сложный анализ результатов завершения автоматизированных скриптов;

4) любое изменение в работе системы может потребовать трудозатратных изменений в автоматизированных тестах;

5) ошибка в реакции системы на один из тестов может привести к ошибочным результатам тестовой сессии всех последующих тестов;

- 6) риск возникновения ошибок в самом автоматизированном тесте;
- 7) в некоторых случаях, не все функциональные особенности системы можно покрыть автоматизированными тестами с помощью выбранного инструментария.

Недостатки автоматизированных скриптов:

- 1) повторяемость – все написанные тесты всегда будут выполняться однообразно;
- 2) затраты на поддержку – несмотря на то, что в случае автоматизированных тестов они меньше, чем затраты на ручное тестирование того же функционала – они так же существуют. Чем чаще изменяется приложение, тем они выше;
- 3) затраты на разработку – это сложный процесс, так как фактически идет разработка приложения, которое тестирует другое приложение. В сложных автоматизированных тестах также есть фреймворки, утилиты, библиотеки и прочее. Как и основное приложение, это требует времени и сил на тестирование и отладку;
- 4) стоимость инструмента для автоматизации – в случае если используется лицензионное ПО, его стоимость может быть достаточно высока. Свободно распространяемые инструменты как правило отличаются более скромным функционалом и меньшим удобством работы;
- 5) пропуск мелких ошибок – автоматизированный скрипт может пропускать мелкие ошибки, на проверку которых он не запрограммирован. Это могут быть неточности в позиционировании окон, ошибки в надписях, которые не проверяются, ошибки элементов и форм, с которыми не осуществляется взаимодействие во время выполнения скрипта.

Автоматизацию необходимо применять при наличии в системе или программном продукте следующих категорий:

- 1) труднодоступные места (бэкенд процессы, логирование файлов, запись в БД);

2) часто используемой функциональности, риски от ошибок в которой достаточно высоки;

3) рутинные операции, такие как переборы данных, формы с большим количеством вводимых полей;

4) валидационные сообщения;

5) длинные end-to-end сценарии;

6) проверка данных, требующих точных математических расчетов.

А также, многое другое, в зависимости от требований к тестируемой системе и возможностей выбранного инструмента для тестирования.

Для более эффективного использования автоматизации тестирования необходимо разработать отдельные тест-кейсы, проверяющие:

1) базовые операции создания/чтения/изменения/удаления сущностей (так называемые CRUD операции - Create / Read / Update / Delete). Пример: создание, удаление, просмотр и изменение данных о пользователе;

2) типовые сценарии использования приложения, либо отдельные действия. Пример: пользователь заходит на почтовый сайт, листает письма, просматривает новые, пишет и отправляет письмо, выходит с сайта. Это end-to-end сценарий, который проверяет совокупность действий;

3) интерфейсы, работы с файлами и другие моменты, неудобные для тестирования вручную. Пример: система создает некоторый .xml файл, структуру которого необходимо проверить.

Это и есть функциональность, при автоматизации тестирования которой можно получить наибольшую эффективность в дальнейшей работе.

2.2.1 Уровни автоматизации тестирования

Тестируемое приложение можно условно разделить на 3 уровня:

1) unit tests layer;

2) GUI tests layer;

3) functional tests layer (Non-UI).

Для обеспечения лучшего качества продукта рекомендуется автоматизировать все 3 уровня. Рассмотрим более детально стратегию автоматизации тестирования на основе трехуровневой модели:

1) Уровень модульного тестирования (Unit Test layer)

Под автоматизированными тестами на этом уровне понимаются компонентные или модульные тесты, написанные разработчиками. Тестировщикам имеют право писать тесты, которые будут проверять код, если их квалификация позволяет это. Наличие подобных тестов на ранних стадиях проекта, а также постоянное их пополнение новыми тестами, проверяющими «баг фиксы», уберегает проект от многих серьезных проблем.

2) Уровень тестирования через пользовательский интерфейс (GUI Test Layer)

На данном уровне есть возможность тестировать не только интерфейс пользователя, но и функциональность, выполняя операции вызывающую бизнес-логику приложения. Такие сквозные тесты дают больший эффект в сравнении с обычным тестированием функционального слоя, так как здесь происходит тестирование функциональности путем эмуляции действия конечного пользователя через графический интерфейс.

3) Уровень функционального тестирования (Functional Test Layer non-UI)

Не всю бизнес-логику приложения можно протестировать через GUI слой. Так может происходить из-за особенностей реализации приложения, которые скрывают бизнес-логику от пользователей. Именно по этой причине по договоренности с разработчиками, для команды тестирования может быть реализован доступ напрямую к функциональному слою, дающий возможность тестировать непосредственно бизнес-логику приложения, минуя пользовательский интерфейс.

2.2.2 Архитектура тестов

Для удобства наложения автоматизированных тестов, на уже имеющиеся тест-кейсы, структура скриптов должна быть аналогична структуре тестового случая.

Таким образом, каждый тест-скрипт должен иметь:

- 1) precondition;
- 2) steps (Test);
- 3) post Condition.

Основными функциями скрипта являются:

- 1) precondition:

a) инициализация приложения (например, открытие главной страницы, вход под тестовым пользователем, переход в необходимую часть приложения и подведение системы к состоянию пригодному для тестирования);

- b) инициализация тестовых данных.

- 2) steps:

a) непосредственное проведение теста;

b) занесение данных о результате теста, с обязательным сохранением причин ошибочного результата и шагов, по которым проходил тест.

- 3) post condition:

a) удаление, созданных в процессе выполнения скрипта ненужных тестовых данных;

- b) корректное завершение работы приложения.

Таким образом, если тестировщик в процессе разработки автоматизированных скриптов будет придерживаться вышеописанных рекомендаций, общая архитектура тест-скриптов и тестовых сценариев будет совпадать.

2.3 Экономическая и временная эффективность внедрения автоматизированных тестов

Экономическую целесообразность автоматизации тестов не всегда возможно просчитать заранее, поскольку она зависит от целого ряда факторов:

- 1) предполагаемого жизненного цикла системы;
- 2) метода разработки системы;
- 3) методов автоматизации;
- 4) объектов автоматизации.

Существует три способа вычисления эффективности внедрения автоматизированного тестирования [18]:

- 1) прямой расчёт рентабельности инвестиций;
- 2) расчёт рентабельности инвестиций с точки зрения минимизации рисков;
- 3) расчёт рентабельности инвестиций с точки зрения эффективности использования ресурсов.

Каждый из представленных методов имеет свои достоинства и недостатки, поэтому следует комбинировать несколько методов для получения наиболее всесторонней и точной оценки прибыльности автоматизированного тестирования.

В данной работе использована следующая общая формула расчёта рентабельности инвестиций (ROI):

$$ROI = (Gain - Investments) / Gain * 100\% , \quad (2.1)$$

При прямом расчёте рентабельности инвестиций, под «Прибылью» (Gain) подразумевается расчётная стоимость ручного тестирования за три года, а под «Инвестициями» (Investments) подразумеваются расходы на создание и выполнение автоматизированной библиотеки тестов за тот же период. Данный метод вычисления рентабельности инвестиций позволяет произвести расчёт прямой выгоды относительно затраченных денежных средств.

При планировании затрат на автоматизацию тестирования, необходимо учитывать следующие факторы:

- 1) расходы на обучение персонала или привлечение более квалифицированного персонала;
- 2) стоимость программного обеспечения для автоматизации тестирования;
- 3) затраты на дополнительные аппаратные средства (в отдельных случаях);
- 4) цена разработки первоначальной библиотеки автоматизированных скриптов;
- 5) стоимость запуска каждой тестовой сессии;
- 6) затраты на анализ результатов работы автоматизированных тестов;
- 7) издержки на поддержание автоматизированных скриптов в актуальном состоянии.

Следующая формула используется для вычисления необходимых инвестиций для автоматизированного тестирования за выбранный период (I_p). При этом предполагается, что:

- 1) объектом автоматизации является существующая библиотека ручных скриптов;

- 2) автоматизацией тестирования занимаются те же работники, которые отвечали за ручное тестирование (изменения стоимости оплаты труда не происходит).

$$I_p = I_0 + C_0 + \sum_{n=1}^k (C_e + C_a + C_m), \quad (2.2)$$

где I_0 – стартовые инвестиции. Они представляют сумму затрат на лицензию программного обеспечения для автоматизации тестирования, обучения персонала и издержек на дополнительные аппаратные средства;

C_0 – стоимость разработки и отладки первоначальной библиотеки автоматизированных скриптов. Для получения данной переменной требуется умножить время, затрачиваемое на написание одного автоматизированного

скрипта одним тестирующим (в часах), на общее количество тестов и на стоимость одного рабочего часа;

k – количество планируемых тестовых сессий набора автоматизированных тестов;

C_e – стоимость однократного выполнения набора автоматизированных скриптов. Для её вычисления необходимо умножить среднее время, затрачиваемое на подготовку и выполнение одного скрипта одним тестирующим (в часах), на общее количество тестов и на стоимость одного рабочего часа. Данная переменная может принимать нулевое значение в том случае, когда происходит полностью автономное выполнение тестов, не требующее вмешательства человека ни на одной стадии исполнения теста;

C_a – стоимость анализа результатов одной тестовой сессии набора автоматизированных скриптов. Для получения данной переменной необходимо умножить предполагаемый процент неудачно проведённых тестов на общее количество скриптов, на среднее время, затрачиваемое на анализ причин неудачного завершения одного скрипта одним тестирующим (в часах), и на стоимость одного рабочего часа тестирующего;

C_m – стоимость поддержания автоматизированных скриптов в актуальном состоянии. Для её вычисления необходимо умножить коэффициент ожидаемых изменений скриптов для каждого цикла выполнения, на время, затраченное одним тестирующим на изменение одного скрипта (в часах), на общее количество скриптов и на стоимость одного рабочего часа тестирующего. Данная переменная может принимать нулевое значение в том случае, если в данной функциональной области не планируется никаких последующих изменений.

В случае привлечения более квалифицированных и дорогостоящих специалистов для автоматизации скриптов, требуется внести следующие поправки при использовании формулы расчёта необходимых инвестиций за выделенный период (I_p), приведённой выше:

1) стоимость обучения персонала следует считать равной 0 при вычислении стартовых инвестиций (I_0);

2) При расчёте стоимости разработки первоначальной библиотеки автоматизированных скриптов (C_0), расходов на одну тестовую сессию набора автоматизированных тестов (C_e), издержек на анализ результатов одной тестовой сессии набора автоматизированных скриптов (C_a) и стоимости поддержания автоматизированных тестов в актуальном состоянии (C_m) следует использовать стоимость рабочего часа квалифицированных тестировщиков, привлекаемых к автоматизации, а не тестировщиков, отвечающих за выполнение ручного тестирования.

Данная формула представляет собой расчет инвестиций для ручного тестирования за выбранный период времени (G_p):

$$G_p = G_0 + \sum_{n=1}^k (G_e + G_a + G_m), \quad (2.3)$$

где G_0 – стоимость разработки первоначальной библиотеки ручных скриптов. Данное значение равно 0 в том случае, если библиотека ручных тестов уже существует;

k – количество планируемых тестовых сессий набора ручных скриптов;

G_e – стоимость однократного выполнения набора ручных скриптов. Она вычисляется путем умножения среднего времени, затрачиваемого на подготовку и выполнение одного теста одним тестировщиком (в часах), на общее количество скриптов и на стоимость одного рабочего часа тестировщика;

G_a – стоимость анализа результатов одной тестовой сессии набора ручных скриптов. Для её получения необходимо умножить предполагаемый процент неудачно проведённых тестов на общее количество скриптов, на среднее время анализа причин неудачного выполнения одного теста одним тестировщиком (в часах) и на стоимость одного рабочего часа тестировщика. Данная величина равна нулю в большинстве случаев из-за специфики

построения ручных тестов. Они представляют собой детально описанную инструкцию для тестировщика;

G_m – стоимость поддержания ручных скриптов в актуальном состоянии. Для её получения необходимо умножить коэффициент ожидаемых изменений ручных тестов для каждого цикла выполнения, на общее количество скриптов, на среднее время, затрачиваемое на изменение одного ручного скрипта одним тестировщиком (в часах) и на стоимость одного рабочего часа. Данная переменная может принимать нулевое значение в том случае, если в данной функциональной области не планируется никаких последующих изменений.

Вычислить ожидаемую выгоду от внедрения автоматического тестирования можно и с точки зрения эффективности использования ресурсов. Данный метод основан на сравнении временных затрат, необходимых для внедрения, выполнения, анализа результатов и поддержания актуальности автоматических тестов (Инвестиции) и временных затрат на ручные тесты (Прибыль). Стоит отметить, что метод учитывает только временные затраты персонала, без учета их денежного вознаграждения. Именно поэтому он особенно полезен в том случае, когда коммерческие детали неизвестны [19].

Для расчёта временных инвестиций для автоматизирования скриптов за выделенный период (TI_p) используется следующая формула:

$$TI_p = TI_0 + TC_0 + \sum_{n=1}^k (TC_e + TC_a + TC_m), \quad (2.4)$$

где TI_0 – стартовые временные инвестиции. Они состоят из времени, затраченного на поиск программного обеспечения для автоматизации тестирования и обучение персонала. Данный параметр может принимать нулевое значение в том случае, если привлекаются уже обученные специалисты или программное обеспечение для автоматизации тестирования уже есть;

$ТС_0$ – временные затраты на разработку первоначальной библиотеки автоматических скриптов. Данная переменная вычисляется путем умножения среднего времени, затрачиваемого на написание одного скрипта одним тестировщиком (в часах), на общее количество тестов;

k – количество планируемых тестовых сессий набора автоматизированных скриптов;

$ТС_e$ – время, затраченное тестировщиком на подготовку и выполнение одного скрипта (в часах), умноженное на общее количество тестов. Данная переменная может принимать нулевое значение в случае, если происходит полностью автономное выполнение тестов, не требующее вмешательства человека ни на одной стадии исполнения скрипта;

$ТС_a$ – временные затраты на анализ результатов однократного выполнения набора автоматизированных скриптов. Для нахождения данной переменной необходимо умножить предполагаемый процент неудачно проведенных тестов на общее количество скриптов и на среднее время, затрачиваемое на анализ причин неудачного выполнения одного теста одним тестировщиком (в часах);

$ТС_m$ – временные затраты на поддержание автоматизированных скриптов в актуальном состоянии. Они высчитываются путем умножения коэффициента ожидаемых изменений скриптов для каждого цикла выполнения на среднее время, затраченное на изменение одного теста одним тестировщиком (в часах), и на общее количество скриптов. Данная переменная может принимать нулевое значение в том случае, если в данной функциональной области не планируется никаких последующих изменений.

Данная формула показывает расчет временных инвестиций для ручного тестирования за выбранный период времени (TG_p):

$$TG_p = TG_0 + \sum_{n=1}^k (TG_e + TG_a + TG_m), \quad (2.5)$$

где TG_0 – время, затраченное на разработку первоначальной библиотеки ручных скриптов одним тестировщиком (в часах). Данное

значение равно нулю в том случае, если библиотека скриптов уже существует;

k – количество планируемых тестовых сессий набора ручных скриптов;

TG_e – временные затраты на однократное выполнение набора ручных скриптов. Они вычисляются путем умножения среднего времени, затраченного на подготовку и выполнение одного скрипта одним тестировщиком (в часах), на общее количество тестов;

TG_a – временные затраты на анализ результатов выполнения одного набора ручных скриптов. Для вычисления данной переменной необходимо умножить предполагаемый процент неудачно проведённых тестов на среднее время, затрачиваемое на анализ причин неудачного выполнения одного скрипта одним тестировщиком (в часах), и на общее количество тестов. Данная величина равна нулю в большинстве случаев из-за специфики построения ручных тестов. Они представляют собой детально описанную инструкцию для тестировщика;

TG_m – временные затраты на поддержание ручных скриптов в актуальном состоянии. Для их нахождения необходимо умножить коэффициент ожидаемых изменений на среднее время, затрачиваемое на изменение одного скрипта одним тестировщиком (в часах), и на общее количество тестов. Данная переменная может принимать нулевое значение в случае, если в данной функциональной области не планируется никаких последующих изменений.

Ещё одним способом оценки рентабельности инвестиций в автоматизированное тестирование является оценка рентабельности инвестиций с точки зрения минимизации рисков. Данный метод представляет собой сравнение средств, затраченных на тестирование (Инвестиции), с убытками, которые могут возникнуть в результате ошибки функционирования готовой системы на этапе эксплуатации (Прибыль). Необходимо отметить, что часто довольно сложно точно оценить возможные убытки, а данный метод подразумевает точный анализ возможных рисков. В

данном методе предполагается, что были протестированы все аспекты функционирования системы [20].

2.3.1 Расчёт экономической целесообразности введения автоматизированного тестирования

Для проверки целесообразности автоматизации процесса тестирования в компании необходимо вычислить затраты на ручное тестирование и затраты на автоматизацию по формулам, представленным выше. Расчеты были произведены на основе из данных, полученных в ходе опроса работников отдела тестирования:

- 1) оплата тестировщика оценивается в среднем в 125 рублей в час;
- 2) данный проект рассчитан минимум на три года. Тестирование, как ручное, так и автоматизированное, проводится каждую неделю. Однако часто после исправления критических ошибок, найденных при тестировании, проверки необходимо проводить повторно. Поэтому количество тестовых сессий набора скриптов увеличивается минимум до двух единиц в одну неделю. За три года планируется совершить 312 полных тестовых сессий.
- 3) при тестировании используется 569 автоматизированных и 569 ручных тестов;
- 4) в каждой тестовой сессии примерно 1% тестов имеют отрицательные результаты;
- 5) на определение источника ошибки для каждого теста у ручного тестировщика уходит в среднем пять минут;
- 6) при автоматизированном тестировании анализ ошибки составляет в среднем 10 минут;
- 7) на подготовку к проведению набора ручных скриптов тестировщик затрачивает в среднем 30 минут;
- 8) подготовка к проведению набора автоматизированных скриптов не требуется;
- 9) среднее время, необходимое одному тестировщику на выполнение одного ручного тест-кейса, составляет две минуты;

10) вероятность появления необходимости изменения одного скрипта между тестовыми сессиями оценивается в 1%;

11) среднее время, необходимое для актуализации одного ручного теста, составляет пять минут.

12) среднее время актуализации автоматизированного теста составляет пятнадцать минут;

13) для автоматизации одного теста затрачивается в среднем 30 минут.

Рассчитаем необходимые показатели для формулы (2.2) расчета денежных инвестиций для автоматизированного тестирования на период в три года.

$$I_0 = 0 \text{ рублей};$$

$$C_0 = 569 \text{ тестов} * 0.5 \text{ часа} * 125 \text{ руб/час} = 35\,000 \text{ рублей};$$

$$k = 312 \text{ циклов тестирования};$$

$$C_e = 0 \text{ рублей};$$

$$C_a = 569 \text{ тестов} * 0.01 * 0.17 \text{ часа} * 125 \text{ руб/час} = 120,6 \text{ рублей};$$

$$C_m = 569 \text{ тестов} * 0.01 * 0.25 \text{ часа} * 125 \text{ руб/час} = 177,8 \text{ рублей};$$

Подставим полученные данные в формулу 2 и найдем денежные затраты на автоматизацию тестирования и поддержание тестов в актуальном состоянии в течение трех лет.

$$I_p = I_0 + C_0 + \sum_{n=1}^k (C_e + C_a + C_m) = 0 + 35\,000 + \sum_{n=1}^{312} (0 + 120,6 + 177,8) = 128\,100,8 \text{ рублей}.$$

Рассчитаем необходимые показатели для формулы (2.3) расчета денежных инвестиций для ручного тестирования на период в три года.

$$G_0 = 0 \text{ рублей};$$

$$k = 312 \text{ циклов тестирования};$$

$$C_e = (0.5 \text{ часа} + 569 \text{ тестов} * 0.03 \text{ часа}) * 125 \text{ руб/час} = 2196,25 \text{ рублей};$$

$$C_a = 569 \text{ тестов} * 0.01 * 0.08 \text{ часа} * 125 \text{ руб/час} = 56,9 \text{ рублей};$$

$$C_m = 569 \text{ тестов} * 0.01 * 0.08 \text{ часа} * 125 \text{ руб/час} = 56,9 \text{ рублей};$$

Подставим полученные данные в формулу 3 и найдем денежные затраты на ручное тестирования и поддержание тестов в актуальном состоянии в течение 3 лет.

$$G_p = G_0 + \sum_{n=1}^k (G_e + G_a + G_m) = 0 + \sum_{n=1}^{312} (2196,25 + 56,9 + 56,9) = 720\,735,6 \text{ рублей.}$$

2.2.2 Расчёт временной целесообразности введения автоматизированного тестирования

Наряду с экономической рентабельностью внедрения автоматизированных тестов важную роль играет и расчет временной эффективности внедрения автоматизированного тестирования. Для вычисления временной эффективности воспользуемся формулами (2.4) и (2.5) из главы 2.3.

Рассчитаем необходимые показатели для формулы (2.4) расчета временных инвестиций для автоматизированного тестирования на период в три года.

$$TI_0 = 0 \text{ часов;}$$

$$TC_0 = 569 \text{ тестов} * 0,5 \text{ часа} = 284,5 \text{ часов;}$$

$$k = 312 \text{ циклов тестирования;}$$

$$TC_e = 0 \text{ часов;}$$

$$TC_a = 569 \text{ тестов} * 0,01 * 0,17 \text{ часа} = 0,9673 \text{ часа;}$$

$$TC_m = 569 \text{ тестов} * 0,01 * 0,25 \text{ часа} = 1,4225 \text{ часа;}$$

Подставим полученные данные в формулу 4 и найдем временные затраты на автоматизацию тестирования и поддержание тестов в актуальном состоянии в течение трех лет.

$$TI_p = TI_0 + TC_0 + \sum_{n=1}^k (TC_e + TC_a + TC_m) = 0 + 284,5 + \sum_{n=1}^{312} (0 + 0,9673 + 1,4225) = 1030,11 \text{ часов.}$$

Рассчитаем необходимые показатели для формулы (2.5) расчета временных инвестиций для ручного тестирования на период в три года.

$$TG_0 = 0 \text{ часов;}$$

$$k = 312 \text{ циклов тестирования;}$$

$$TC_e = 0.5 \text{ часа} + 569 \text{ тестов} * 0.03 \text{ часа} = 17,57 \text{ часов};$$

$$TC_a = 569 \text{ тестов} * 0.01 * 0.08 \text{ часа} = 0,4552 \text{ часов};$$

$$TC_m = 569 \text{ тестов} * 0.01 * 0.08 \text{ часа} = 0,4552 \text{ часов};$$

Подставим полученные данные в формулу (2.5) и найдем денежные затраты на ручное тестирования и поддержание тестов в актуальном состоянии в течение трех лет.

$$TG_p = TG_0 + \sum_{n=1}^k (TG_e + TG_a + TG_m) = 0 + \sum_{n=1}^{312} (17,57 + 0,4552 + 0,4552) = 5765,88 \text{ часов}.$$

Вывод по Главе 2

Во данной главе была проанализирована программа Alpha.Alarms. Именно для этой программы были разработаны автоматизированные тесты, рассматриваемые в данной работе. За их основу были взяты тесты для ручного тестирования, расположенные в системе TestLink. Так же были изучены различные аспекты и режимы работы программы Alpha.Alarms, её функциональные возможности.

Так же были изучены аспекты автоматизации процесса тестирования. Были выявлены достоинства и недостатки автоматизированных тестов в сравнении с ручными скриптами. Были выявлены критерии, при которых следует применять именно автоматизированные тесты, а не ручное тестирование. Были изучены уровни автоматизации тестирования и была описана архитектура автоматизированных тестов.

Кроме того, был проведен расчет экономической рентабельности и временной эффективности замены набора ручных скриптов на автоматизированное тестирование. Данные расчёты показали, что автоматизированное тестирование в несколько раз эффективней ручного тестирования.

3 Глава 3

3.1 Реализация приложения по тестированию

В рамках данной работы была реализована программа, для тестирования программного продукта “Alpha.Alarms”. В качестве библиотеки фреймворка тестирования была выбрана библиотека NUnit. Разработка осуществлялась на объектно-ориентированном языке программирования C#. В качестве среды разработки программного обеспечения была выбрана Microsoft Visual Studio 2017 Community. Для запуска тестов использовались NUnit 3 Test Adapter и Jenkins.

3.1.1 Архитектура реализованных тестов

В главе 2.3.2 были приведены критерии для написания автоматизированных тестов. Так как в качестве основы были взяты уже готовые тесты, используемые для ручного тестирования, возникает необходимость придерживаться определённых критериев для корректной работы автоматизированных тестов. Поэтому их архитектуру можно представить следующим образом.

Инициализация всех переменных необходимых для тестирования. За инициализацию отвечает метод `ClassInitialize()`. Он показан на рисунке 3.1.1.

```

/// <summary>
/// Метод выполняющий инициализацию всех переменных. Выполняется до запуска всех тестов
/// </summary>
[OneTimeSetUp]
public static void ClassInitialize()
{
    _maxCountTest = MaxCountTests();

    TimerStatic(50);

    // Создаем экземпляр Alpha.Server
    testAlphaServer = new APServer();
    testAlphaServer.CreateServer(Resources.AlphaServerServiceName, Resources.ApIphaServerOpcDaProgId, Resources.ApIphaServerOpcAePro

    TimerStatic(50);

    testAlphaServer.SetConfiguration("Alpha.Server.cfg");

    TimerStatic(50);

    AlarmsAlarmsEXE = AppDomain.CurrentDomain.BaseDirectory + "Alpha.Alarms.App.exe";

    StaticCreatOpcDaServer();
}

```

Рисунок 3.1.1– Метод инициализации переменных

Метод `ClassInitialize()` помечен атрибутом `[OneTimeSetUp]`, это означает, что данный метод будет выполняться один раз при запуске

тестовой сессии. В данном методе происходит создание службы Alpha.Server. Она используется в тестах для проверки генерирования событий в оперативном режиме. Кроме того, данный метод производит инициализацию всех переменных перед запуском тестового прогона. Также он определяет количество тестов, которые будут выполнены в текущей тестовой сессии. Рисунок 3.1.2.

```

/// <summary>
/// Статический метод по определению максимального количества тестов в тестовом запуске
/// </summary>
static private int MaxCountTests()
{
    FieldInfo TestAdapter = typeof(TestAdapter).GetField("_test", BindingFlags.NonPublic | BindingFlags.Instance);
    var TestFixture = (NUnit.Framework.Internal.TestFixture)TestAdapter.GetValue(CurrentContext.Test);

    int CountRunState = 0;

    foreach (var CurrentParameterized in TestFixture.Tests)
    {
        if (CurrentParameterized.RunState == NUnit.Framework.Interfaces.RunState.Runnable)
        {
            CountRunState += CurrentParameterized.TestCaseCount;
        }
    }

    return CountRunState;
}

```

Рисунок 3.1.2 – Метод определяющий количество запущенных тестов

После выполнения метода ClassInitialize() начинается работа метода TestInitializeClear(). Данный метод запускает обнуление всех необходимых переменных и удаляет все настроечные конфигурации Alpha.Alarms, которые применяются при запуске приложения. Метод помечен атрибутом [SetUp], это означает, что он будет запускаться перед выполнением каждого теста. Рисунок 3.1.3.

```

/// <summary>
/// Метод выполняющий очистку всех переменных. Выполняется перед каждым тестом
/// </summary>
[SetUp]
public void TestInitializeClear()
{
    Clear();
}

```

Рисунок 3.1.3– Метод обнуления переменных перед запуском приложения

После завершения работы метода TestInitializeClear() начинается тестовая сессия.

Тесты делятся на 2 категории:

1) тесты, запускающие Alpha.Alarms как отдельный процесс. Такой запуск приложения не дает доступа к внутренней реализации, но позволяет задавать входные параметры при запуске приложения.

2) тесты, проверяющие Alpha.Alarms как встроенный компонент. При запуске приложения таким образом, отсутствует возможность передавать в него входные параметры, однако появляется доступ ко всем методам и функциям данного софта.

Рассмотрим пример теста для проверки запуска приложения с параметрами командной строки. Тест `AlphaAlarms_LaunchParam_Hist()` запускает приложение Alpha.Alarms в качестве отдельного процесса и при запуске передает приложению входные параметры. В данном случае таким параметром является строка “mode historical”. Корректным результатом для данного теста является запуск приложения Alpha.Alarms в режиме просмотра исторических событий. Листинг тестового метода представлен в Приложении к диссертации № 1.

Ко второй категории относятся тесты, где приложение Alpha.Alarms запускается в качестве встраиваемого компонента. Для этого используется специальная программа `Alarms.TestApp`. Главное окно данного приложения представлено на Рисунках 3.1.4 и 3.1.5.

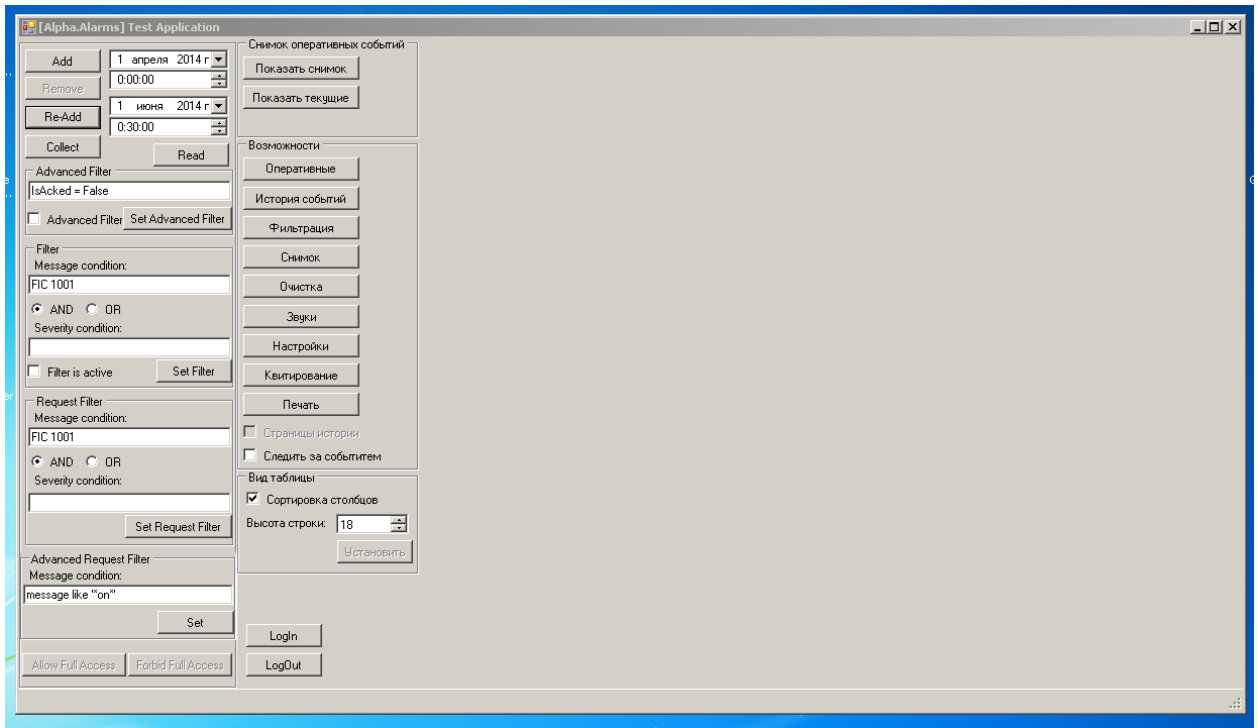


Рисунок 3.1.4 – Запуск Alarms.TestApp без запуска Alpha.Alarms

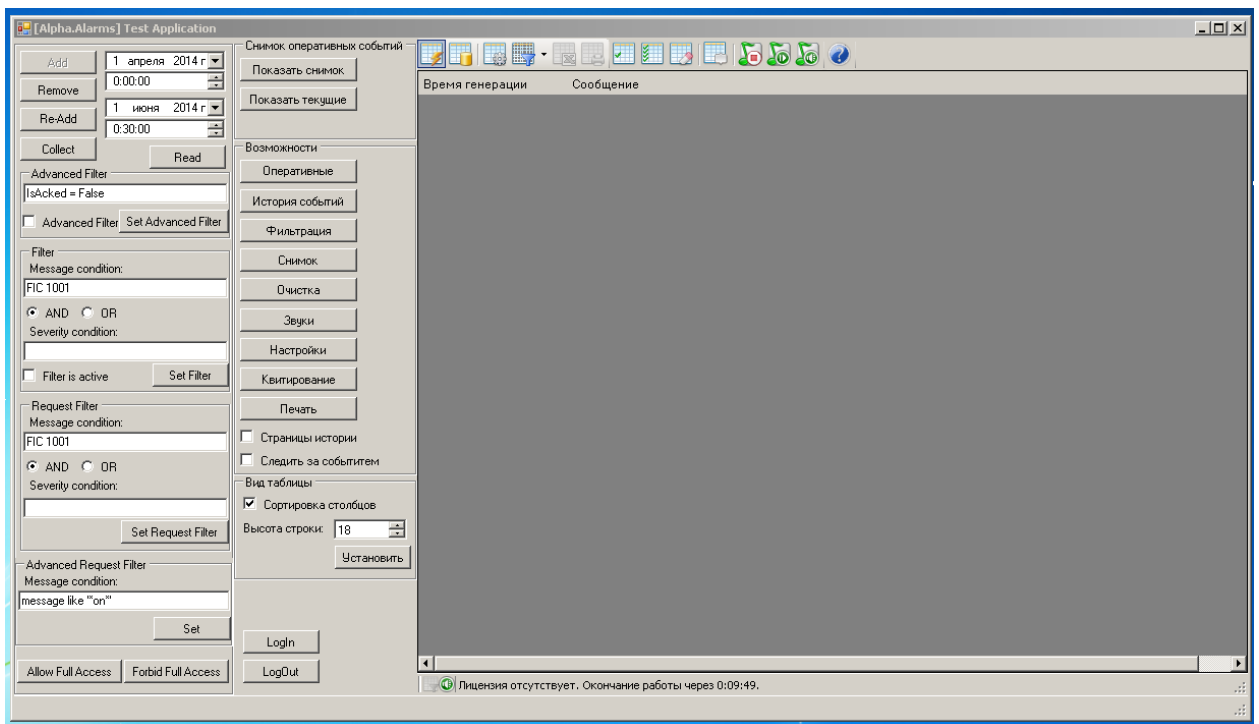


Рисунок 3.1.5 – Запуск Alarms.TestApp с запуском Alpha.Alarms

После запуска Alpha.Alarms как встраиваемого компонента появляется доступ ко всем переменным и методам данного приложения. Именно при таком запуске программного продукта появляется возможность наиболее полной и качественной его диагностики. При автоматизации тестирования

Alpha.Alarms как встраиваемого компонента были разработаны группы тестов, нацеленные на проверку:

- 1) получения оперативных событий;
- 2) получения исторических событий;
- 3) проверки корректности отображения всех элементов;
- 4) загрузки различных конфигураций;
- 5) совместимости различных конфигураций;
- 6) применения различных фильтров в историческом режиме;
- 7) применения различных фильтров в оперативном режиме
- 8) использования различных некорректных значений;
- 9) сохранения настроек после перезагрузки программы;
- 10) имитации различных действий пользователя.

На рисунке 3.1.6 представлен листинг теста на проверку получения оперативных событий `CountMessage_Oper(int CountMessage, int CountEtalon)`. С помощью метода `ReplacingConfigurations()` осуществляется копирование в папку “C:\ProgramData\AlphaPlatform\Alarms” необходимых конфигураций, после чего происходит инициализация приложения Alpha.Alarms в методе `PreparationOperTests()`. Далее в данном методе выполняется создание объекта программы Alpha.Alarms и генерирование необходимых оперативных событий (рисунок 3.1.7), которые хранятся в переменной `_grd.Rows.Count`. Затем ее значение сравнивается с эталонным значением переменной `CountEtalon` и по результатам данного сравнения выявляется успешность теста. По умолчанию в NUnit все тесты считаются успешным. Если в процессе теста не было вызвано различных исключений или ошибок, то тест считается пройденным. В противном случае, выдается ошибка и результат теста считается ошибочным.

```

/// <summary>
/// Проверяем количество событий в оперативном режиме
/// </summary>
[Test, Retry(_numberRepetition)]
[TestCase(0, 0, TestName = "CountMessage_Oper_Equally=0")]
[TestCase(10, 10, TestName = "CountMessage_Oper_Equally=010")]
[TestCase(50, 50, TestName = "CountMessage_Oper_Equally=050")]
[TestCase(250, 250, TestName = "CountMessage_Oper_Equally=250")]
[TestCase(500, 500, TestName = "CountMessage_Oper_Equally=500")]
//[Ignore("Ignore the test Jenkins")]
public void CountMessage_Oper(int CountMessage, int CountEtalon)
{
    ReplacingConfigurations(Resources.AlphaAlarmsCfg, Resources.config_new_OldEventFirst_FULL);
    ReplacingConfigurations(Resources.AlphaAlarmsCfgDataSources, Resources._datasources);

    PreparationOperTests(CountMessage, true, "Enum");

    ComparisonMethod(_grd.Rows.Count, CountEtalon, "Ожидалось = " + CountEtalon.ToString() +
        " сообщений. /nПришло = " + _grd.Rows.Count.ToString() + "сообщений");
}

```

Рисунок 3.1.6 – Тест на проверку получения оперативных событий

```

/// <summary>
/// Метод для объединения группы методов, для тестирования оперативного режима
/// </summary>
private void PreparationOperTests(int CountEvents, bool Initialize, params string[] EventType)
{
    if (Initialize)
    {
        InitializeTest();
    }

    CheckActivityOper();

    foreach (string currentEvent in EventType)
    {
        AddParametersGeneration(currentEvent, CountEvents);
    }

    Work_EventsGenerationNew();
}

```

Рисунок 3.1.7 – Инициализация Alpha.Alarms и генерация оперативных событий

После завершения выполнения каждого теста запускается метод `TestTearDown()`, он представлен на рисунке 3.1.8. Данный метод выполняет вывод в консоль имени теста, его порядковый номер в текущем тестовом запуске и количество тестов, которые учувствуют в текущей тестовой сессии (рисунок 3.1.9). Данный метод помечен атрибутом `[TearDown]`, это означает, что он будет запускаться после каждого выполненного теста.

```

/// <summary>
/// Метод выполняющийся после каждого теста. Выводи в консоль результат теста и его номер
/// </summary>
[TearDown]
public void TestTearDown()
{
    Console.Write("Test: (" + _countTest.ToString() + " in " + _maxCountTest + "). ");
    Console.WriteLine("Result: " + TestContext.CurrentContext.Result.Outcome.Status + ".");
    Console.WriteLine(" ");
}

```

Рисунок 3.1.8 – Метод выполняющий вывод информации в консоль

```

=> Alpha.Alarms.Tests.UnitTestWorkspace.AlphaAlarms_Error_ModeOperative
Test: (3 in 569). Result: Passed.

=> Alpha.Alarms.Tests.UnitTestWorkspace.AlphaAlarms_Error_IncorrectMonth
Test: (4 in 569). Result: Passed.

=> Alpha.Alarms.Tests.UnitTestWorkspace.AlphaAlarms_Error_IncorrectDate
Test: (5 in 569). Result: Passed.

=> Alpha.Alarms.Tests.UnitTestWorkspace.AlphaAlarms_Error_NotShowMilliseconds
Test: (6 in 569). Result: Passed.

=> Alpha.Alarms.Tests.UnitTestWorkspace.AlphaAlarms_Error_IncorrectBegin
Test: (7 in 569). Result: Passed.

```

Рисунок 3.1.9 – Пример вывода информации в консоль Jenkins методом
TestTearDown()

После завершения последнего в тестовой сессии теста начинается выполнение метода `ClassDestroy()`. Данный метод помечен атрибутом `[OneTimeTearDown]`, поэтому он запускается один раз после выполнения всех тестов. Данный метод выполняет обнуление всех переменных, которые использовались в тестовом запуске, и удаляет ранее установленную службу `Alpha.Server`, которая использовалась для генерирования оперативных событий и запроса исторических событий. Его действие представлено на Рисунке 3.1.10.

```

/// <summary>
/// Метод выполняющий очистку всех переменных. Выполняется после выполнения всех тестов
/// </summary>
[OneTimeTearDown]
public static void ClassDestroy()
{
    Clear();

    KillAlphaAlarms();

    TimerStatic(150);

    testAlphaServer.DestroyServer();

    TimerStatic(50);

    StaticDisconnectOpcDaServer();
}

```

Рисунок 3.1.10 – Метод, выполняющий обнуление всех значений после выполнения всех тестов

3.2 Результаты работы автоматизированных тестов

Созданное приложение для тестирования Alpha.Alarms можно запускать двумя способами: из Visual Studio и при сборке Alpha.Alarms с помощью Jenkins. У каждого способа запуска есть как свои преимущества, так и недостатки.

При запуске тестового проекта из Visual Studio запуск тестов осуществляется под текущим пользователем (Рисунок 3.2.1). Поэтому осуществлять работу с компьютером до завершения всех тестов будет невозможно. Связанно это с тем, что тесты производят эмуляцию нажатия различных клавиш и эмуляцию движения мыши, тем самым эмитируя действия обычного пользователя, который работает с программой. Так же при запуске каждого теста тестовое приложение переводит на себя активный фокус Windows.

На рисунке 3.2.2 показан пример запуска теста из Visual Studio посредством работы теста по добавлению исторического источника. Он отвечает за проверку корректности запроса на количества сохраненных событий из базы Alpha.Historian.

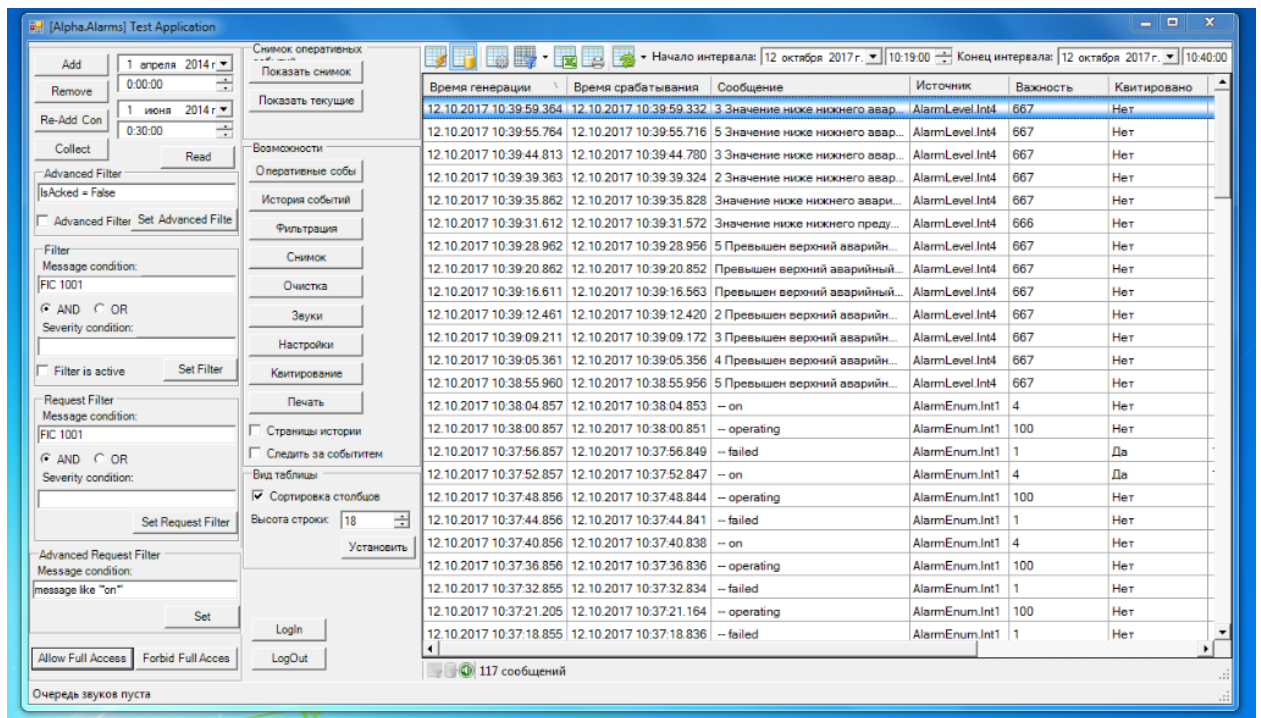


Рисунок 3.2.1 – Выполнение тестов, запущенных с помощью Visual Studio

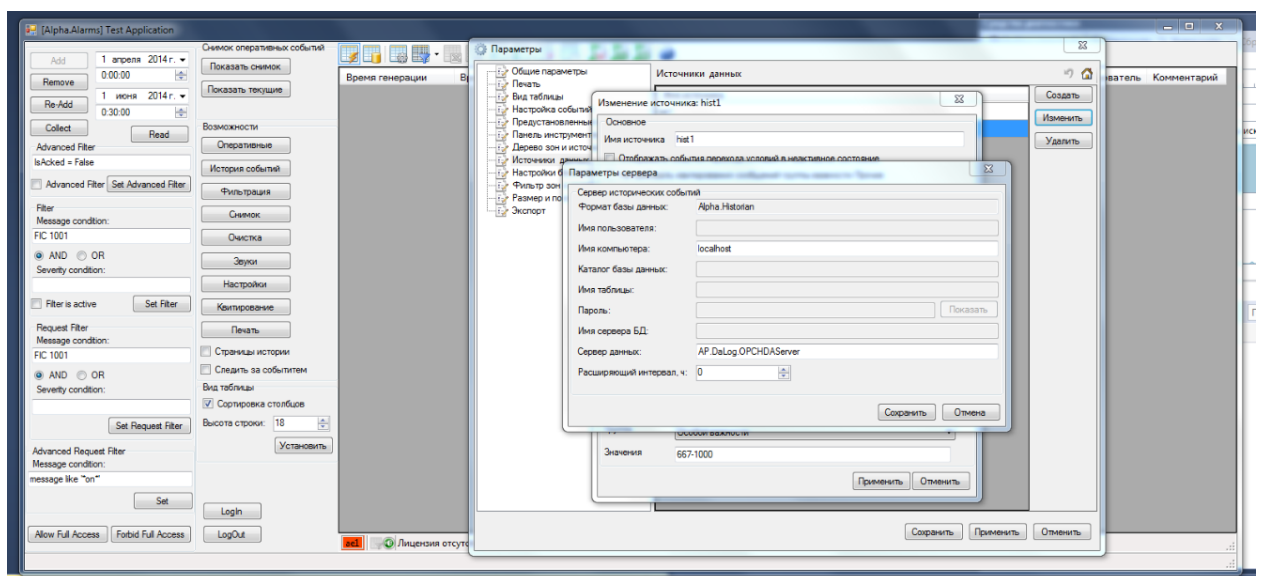


Рисунок 3.2.2 – Тест по добавлению исторического источника

Несмотря на все недостатки при запуске тестового приложения из Visual Studio, в случае «провала» теста, можно быстро найти причину его ошибочного результат.

После завершения всех тестов в тестовом запуске в обозревателе решений Visual Studio появится статистика по всем тестам (Рисунок 3.2.3). В данной статистике будут отражены такие параметры, как:

- 1) результат выполнения (Inconclusive, Passed, Warning, Failed, Error);

- 2) время выполнения каждого теста по отдельности;
- 3) общее время выполнения всех тестов.

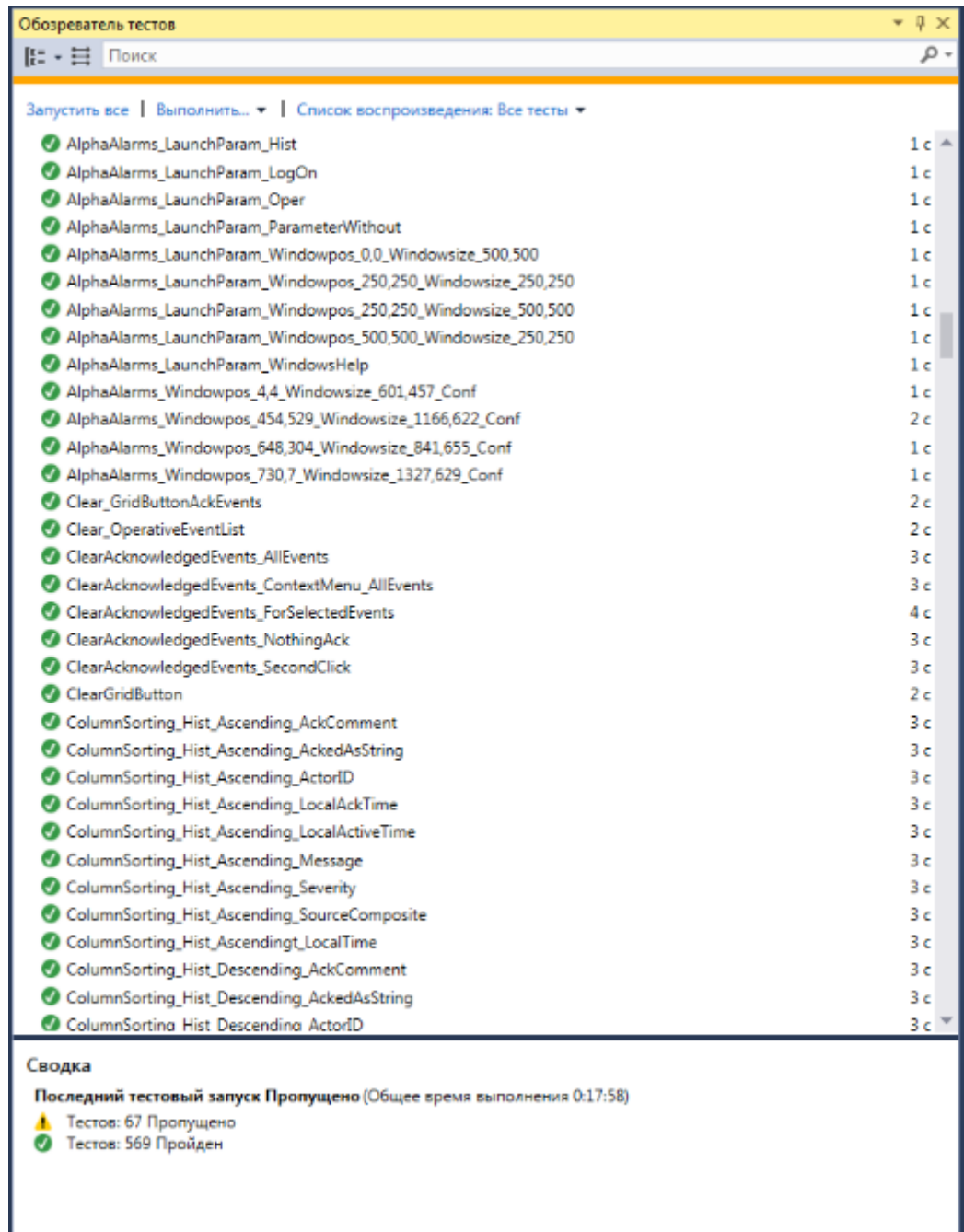


Рисунок 3.2.3– Обозреватель тестов в Visual Studio

При запуске тестов с помощью Jenkins тестовый проект запускается в интерактивной службе Jenkins, поэтому тесты никак не отражаются на работе пользователя с компьютером. Кроме того, тесты запускаются автономно при сборке проекта Alpha.Alarms. Поэтому не требуется никаких

дополнительных действий по запуску тестовой сессии. На рисунке 3.2.4 показан лог Jenkins во время выполнения тестов.

```
E:\Jenkins\jobs\Alarms\jobs\UnitTest\workspace>"C:\Program Files (x86)\NUnit.org\nunit-console\nunit3-console.exe" bina\x64\Release\Alpha.Alarms.Tests.dll --trace=Verbose
NUnit Console Runner 3.7.0
Copyright (c) 2017 Charlie Poole, Rob Prouse

Runtime Environment
  OS Version: Microsoft Windows NT 10.0.14393.0
  CLR Version: 4.0.30319.42000

Test Files
  bina\x64\Release\Alpha.Alarms.Tests.dll

=> Alpha.Alarms.Tests.UnitTestWorkSpace.AlphaAlarms_Error_Mode
Test: (1 in 569). Result: Passed.

=> Alpha.Alarms.Tests.UnitTestWorkSpace.AlphaAlarms_Error_ModeHistorical
Test: (2 in 569). Result: Passed.

=> Alpha.Alarms.Tests.UnitTestWorkSpace.AlphaAlarms_Error_ModeOperative
Test: (3 in 569). Result: Passed.

=> Alpha.Alarms.Tests.UnitTestWorkSpace.AlphaAlarms_Error_IncorrectMonth
Test: (4 in 569). Result: Passed.

=> Alpha.Alarms.Tests.UnitTestWorkSpace.AlphaAlarms_Error_IncorrectDate
Test: (5 in 569). Result: Passed.

=> Alpha.Alarms.Tests.UnitTestWorkSpace.AlphaAlarms_Error_NotShowMilliseconds
Test: (6 in 569). Result: Passed.

=> Alpha.Alarms.Tests.UnitTestWorkSpace.AlphaAlarms_Error_IncorrectBegin
Test: (7 in 569). Result: Passed.

=> Alpha.Alarms.Tests.UnitTestWorkSpace.AlphaAlarms_Error_IncorrectTimeTo
Test: (8 in 569). Result: Passed.

=> Alpha.Alarms.Tests.UnitTestWorkSpace.AlphaAlarms_Error_IncorrectTimeEnd
Test: (9 in 569). Result: Passed.
```

Рисунок 3.2.4 – Лог Jenkins во время выполнения тестов

После завершения выполнения тестов в консоль Jenkins выводится подробная информация по всем тестам, которые участвовали в тестовой сессии (Рисунок 3.2.5). В результатах отражена такая информация, как:

- 1) общее количество тестов;
- 2) количество пройденных тестов;
- 3) количество пропущенных тестов;
- 4) время начала и время завершения тестов;
- 5) общее время выполнения всех тестов.

```

Run Settings
DisposeRunners: True
InternalTraceLevel: Verbose
WorkDirectory: E:\Jenkins\jobs\Alarms\jobs\UnitTest\workspace
ImageRuntimeVersion: 4.0.30319
ImageTargetFrameworkName: .NETFramework,Version=v4.6.1
ImageRequiresX86: False
ImageRequiresDefaultAppDomainAssemblyResolver: False
NumberOfTestWorkers: 12

Test Run Summary
Overall result: Warning
Test Count: 636, Passed: 569, Failed: 0, Warnings: 0, Inconclusive: 0, Skipped: 67
Skipped Tests - Ignored: 67, Explicit: 0, Other: 0
Start time: 2018-05-15 03:24:04Z
End time: 2018-05-15 03:50:59Z
Duration: 1615.406 seconds

Results (nunit3) saved as TestResult.xml

E:\Jenkins\jobs\Alarms\jobs\UnitTest\workspace>exit 0
Set build name.
New build name is '3.9.2 #1765 [r30998] '
Recording NUnit tests results
Discard old builds...
#1753 is removed because old than numToKeep
Finished: SUCCESS

```

Рисунок 3.2.5 – Результаты тестирования на Jenkins

Кроме того, Jenkins сохраняет статистику по прошедшим тестовым запускам. Это отражено на Рисунках 3.2.6 и 3.2.7.

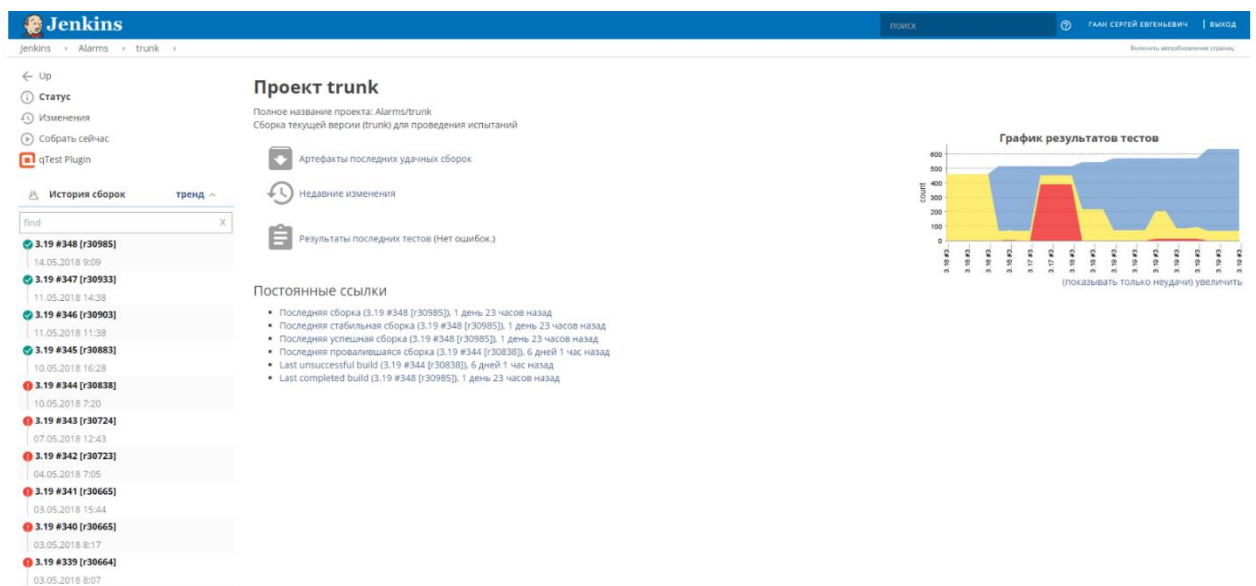
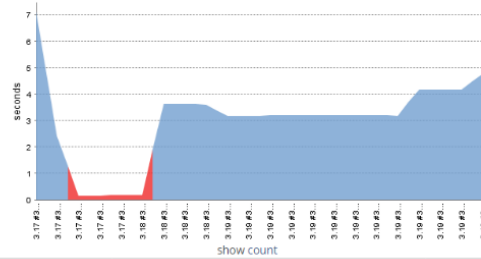


Рисунок 3.2.6 – Различные сборки ветки Trunk

История



Сборка	Описание теста	Длительность теста	Результат теста
Alarms » trunk 3.19 #348 [r30985]		4,7 секунды	Passed
Alarms » trunk 3.19 #347 [r30933]		4,1 секунды	Passed
Alarms » trunk 3.19 #346 [r30903]		4,1 секунды	Passed
Alarms » trunk 3.19 #345 [r30883]		4,1 секунды	Passed
Alarms » trunk 3.19 #344 [r30838]		3,1 секунды	Passed
Alarms » trunk 3.19 #343 [r30724]		3,1 секунды	Passed
Alarms » trunk 3.19 #342 [r30723]		3,1 секунды	Passed
Alarms » trunk 3.19 #341 [r30665]		3,1 секунды	Passed
Alarms » trunk 3.19 #340 [r30665]		3,1 секунды	Passed
Alarms » trunk 3.19 #338 [r30419]		3,1 секунды	Passed
Alarms » trunk 3.19 #337 [r30323]		3,1 секунды	Passed
Alarms » trunk 3.19 #336 [r30207]		3,1 секунды	Passed
Alarms » trunk 3.19 #335 [r30024]		3,1 секунды	Passed
Alarms » trunk 3.18 #334 [r29792]		3,6 секунды	Passed
Alarms » trunk 3.18 #333 [r29754]		3,6 секунды	Passed
Alarms » trunk 3.18 #332 [r29607]		3,6 секунды	Fixed
Alarms » trunk 3.18 #331 [r29561]		0,18 секунд	Failed
Alarms » trunk 3.17 #330 [r29439]		0,17 секунд	Failed

Рисунок 3.2.7 – Изменение времени выполнения одного из тестов в разных сборках

3.3 Оценка тестового покрытия

В данной главе будет произведена оценка общего тестового покрытия программы Alpha.Alarms автоматизированными тестами. Будут рассмотрены все доступные свойства и методы, которыми может воспользоваться обычный пользователь. Все полученные результаты представлены в таблицах 3.3.1 – 3.3.15. Общие результаты по всем категориям представлены в таблице!!!

Таблица 3.3.1 – Оценка покрытия вкладки настроек “Общие параметры”

Название параметра	Покрытие автоматизированными тестами
Отображать панель инструментов при запуске	Да
Отображать строку состояния при запуске	Да
Отображать заголовки столбцов при запуске	Да
Отображать миллисекунды	Да
Сбрасывать расширенный фильтр	Да
Запрашивать активные события при подключении к серверу	Да
Квотировать события повторным кликом	Да
Запрашивать комментарии при квотировании	Нет

Запрашивать подтверждение квитиования	Нет
Дописывать АРМ в комментарий квитиования	Нет
Способ отображения – Журнал	Да
Способ отображения –Список активных условий	Да
Максимальное количество отображаемых оперативных событий	Да
Новые события в начале списка	Да
Новые события в конце списка	Да
Хронология событий	Нет
Интервал, используемый по умолчанию	Да
Хронология событий	Нет

Таблица 3.3.2 – Оценка покрытия вкладки настроек “Печать”

Название параметра	Покрытие автоматизированными тестами
Принтер по умолчанию	Нет
Показывать диалог выбора принтера	Нет
Цветная печать	Нет
Имя принтера	Нет
Печать оперативных событий	Нет
Текст для проверки печать	Нет
Имя пользователя	Нет
Дата печати	Нет
Номера страниц	Нет
Верхний колонтитул	Нет
Текст верхнего колонтитула	Нет

Таблица 3.3.3 – Оценка покрытия вкладки настроек “Вид таблицы”

Название параметра	Покрытие автоматизированными тестами
Текст заголовка	Да
Режим отображения столбца	Да
Ширина столбца	Да
Стиль шрифта	Да
Высота строки	Да
Отображать сетку	Да

Таблица 3.3.4 – Оценка покрытия вкладки настроек “Настройка событий”

Название параметра	Покрытие автоматизированными тестами
Настройка событий	Нет
Мигание всей строки	Нет
Мигание выбранных столбцов	Нет
Цвет фона по умолчанию	Нет
Цвет текста по умолчанию	Нет
Цвет квитиованных событий	Нет
Цвет текста квитиованных событий	Нет
Цвет выделения выбранного события	Нет
Применять стиль квитиованного сообщения ко всем столбцам	Нет

Таблица 3.3.5 – Оценка покрытия вкладки настроек
“Предустановленный фильтр”

Название параметра	Покрытие автоматизированными тестами
Добавить новый фильтр	Да
Удалить фильтр	Да
Проверка фильтра	Да
Экспорт фильтров	Нет
Импорт фильтров	Нет

Таблица 3.3.6 – Оценка покрытия вкладки настроек “Панель инструментов”

Название параметра	Покрытие автоматизированными тестами
Переключение режимов	Да
Параметры	Да
Фильтр отображения	Да
Сохранить	Да
Печать	Да
Квитировать	Да
Квитировать все	Да
Показать древо сигналов	Да
Очистить список	Да
Снимок	Да
Очистить очередь звуков	Да
Пропустить звук	Да
Отключить проигрывание звуков	Да
Показать справку	Да
Элементы управления запросом истории	Да

Таблица 3.3.7 – Оценка покрытия вкладки настроек “Древо зон и источников”

Название параметра	Покрытие автоматизированными тестами
Включить подсветку веток древа зон и источников	Нет
Цвет тревог	Нет
Цвет событий	Нет

Таблица 3.3.8 – Оценка покрытия вкладки настроек “Источник данных”

Название параметра	Покрытие автоматизированными тестами
Имя источник	Да
Отображать события перехода условий в неактивное состояние	Да
Не требовать квитирования сообщений группы важности “Прочие”	Да
Оперативный источник	Да
Исторически сервер AlphaPlatform AE Log	Нет
Исторически сервер ICONICS Awx Logger	Нет
Исторически сервер InfinityOLEDBProvider	Нет

Исторически сервер Alpha.Historian	Да
Исторически сервер WinCC ODBC	Нет
Группа важности	Нет
Значение важности	Нет

Таблица 3.3.9 – Оценка покрытия вкладки настроек “Настройка безопасности”

Название параметра	Покрытие автоматизированными тестами
Использовать сервер безопасности	Нет
Название приложения в системе безопасности	Нет
Запоминать имена зарегистрированных пользователей	Нет
Использование экранной клавиатуры	Нет

Таблица 3.3.10 – Оценка покрытия вкладки настроек “Источник данных”

Название параметра	Покрытие автоматизированными тестами
Идентификатор сервера	Да
Зоны	Да
Источники	Да
Разрешенные сигналы	Да
Запрещенные сигналы	Да

Таблица 3.3.11 – Оценка покрытия вкладки настроек “Размер и положение окна”

Название параметра	Покрытие автоматизированными тестами
Определенное положение и размер	Да
Восстанавливать положение и размер предыдущего запуска	Да

Таблица 3.3.12 – Оценка покрытия вкладки настроек “Размер и положение окна”

Название параметра	Покрытие автоматизированными тестами
Папка для экспорта файлов	Да
Папка для импорта \ экспорта фильтров	Да

Таблица 3.3.13 – Оценка покрытия вкладки настроек “Фильтр пользователя”

Название параметра	Покрытие автоматизированными тестами
На основе фильтра	Да
Время генерации	Да
Время срабатывания	Да
Сообщение	Да
Важность	Да
Источник событий	Да
Источник данных	Да
Квитирование	Да

Время квити́рования	Да
Пользователь	Да
Комментарий квити́рования	Да
Необходимость выполнения всех условий	Да
Выполнение хотя бы одного условия	Да

Таблица 3.3.14 – Оценка покрытия вкладки настроек “Строка состояния”

Название параметра	Покрыв́ение автоматизированными тестами
Источник данных	Да
Фильтр пользователя	Да
Отключить проигрывание звука	Да
Лицензия	Да

Таблица 3.3.15 – Оценка покрытия вкладки настроек “Работа с файлами конфигураций”

Название параметра	Покрыв́ение автоматизированными тестами
Загрузка файла конфигураций	Да
Загрузка альтернативной конфигурации	Да
Сохранение конфигураций	Да
Проверка совместимости со старыми версиями	Да
Загрузка некорректных конфигураций	Да

Таблица 3.3.16 – Оценка общего покрытия программы Alpha.Alarms автоматизированными тестами.

Имя параметра	Общее количество свойств	Количество покрытых свойств	Количество непокрытых свойств	Процент покрытия
Общие параметры	18	13	5	72.5%
Печать	11	0	11	0 %
Вид таблицы	6	6	0	100%
Настройка событий	9	0	9	0%
Предустановленный фильтр	5	3	2	60%
Панель инструментов	15	15	0	100%
Древо зон и источников	3	0	3	0%
Источник данных	11	5	6	45.5%
Настройка безопасности	4	0	4	0%
Источник данных	5	5	0	100%
Размер и положение окна	2	2	0	100%
Размер и положение окна	2	2	0	100%
Фильтр пользователя	13	13	0	100%

Строка состояния	4	4	0	100%
Работа с файлами конфигураций	5	5	0	100%
Сумма	113	73	40	64.5 %

Вывод по Главе 3

В третьей главе рассматривается финальная версия реализованной библиотеки для тестирования программного продукта Alpha.Alarms. Был подробно описан алгоритм запуска тестов и их поведение в процессе выполнения. Также представлена Uml диаграмма последовательности, которая в графическом виде показывает жизненный цикл процесса выполнения автоматизированных тестов.

Были описаны различные группы автоматизированных тестов, для которых выполняется автоматизированное тестирование, а именно:

- 1) получение оперативных событий;
- 2) получение исторических событий;
- 3) проверка корректности отображения всех элементов;
- 4) загрузка различных конфигураций;
- 5) совместимость различных конфигураций;
- 6) применение различных фильтров в историческом режиме;
- 7) применение различных фильтров в оперативном режиме
- 8) использование различных некорректных значений;
- 9) сохранение настроек после перезагрузки программы;
- 10) имитация различных действий пользователя.

Также были описаны два способа запуска автоматизированных тестов: с помощью Microsoft Visual Studio и при сборке продукта с помощью Jenkins. Кроме того, были описаны достоинства и недостатки каждого вида запуска.

В завершение, была произведена оценка тестового покрытия приложения Alpha.Alarms на текущий момент. Из 113 элементов, доступных для работы пользователя, на текущий момент покрыто автоматизированной проверкой 73 элемента, что составляет 64.5% общего функционала приложения Alpha.Alarms.

Заключение

В ходе преддипломной практики были изучены такие аспекты процессов тестирования, как: понятие, классификация, методологии, процесс тестирования и анализ результатов тестирования. Кроме того, был проведен анализ процесса автоматизированного тестирования. Для этого был описан процесс тестирования и критерии его эффективности. В главе 3 были описаны концепция непрерывной интеграции и популярные CI-платформы.

Так же были изучены различные фреймворки тестирования и проведен обзор систем контроля версий. Была описана система TestLink которая используется для проведения ручного тестирования. Так же была описана программа Alpha.Alarms. Она используется в пунктах автоматизации и мониторинга технологических процессов. Применяется для отслеживания событий и тревог, которые появляются при изменении состояний технологических объектов.

Для Alpha.Alarms существует ряд ручных тестов в системе TestLink. В будущем планируется автоматизация ручных тестов с использованием библиотеки NUnit.

Список использованной литературы

- 1) Виды Тестирования. // материалы сайта [Электронный ресурс]. – URL: <http://www.protesting.ru/testing/types/sanity.html> (дата обращения 12.04.2018)
- 2) Certifying Software Testers Worldwide. // материалы сайта [Электронный ресурс]. – URL: <http://www.istqb.org> (дата обращения 12.04.2018)
- 3) Оценка эффективности автоматизации тестирования. // материалы сайта [Электронный ресурс]. – URL: <http://a1qa.ru/blog/otsenka-effektivnosti-avtomatizatsii-testirovaniya/> (дата обращения 17.04.2018)
- 4) Сертификация программного обеспечения ПО. // материалы сайта [Электронный ресурс]. – URL: <http://www.nspru.ru/sertsoftware/> (дата обращения 19.04.2018)
- 5) Автоматизированное тестирование. // материалы сайта [Электронный ресурс]. – URL: <https://gist.github.com/codedokode/> (дата обращения 20.04.2018)
- 6) Основные положения тестирования. // материалы сайта [Электронный ресурс]. – URL: <https://habrahabr.ru/post/110307/> (дата обращения 21.04.2018)
- 7) Что такое Конфигурационное тестирование. // материалы сайта [Электронный ресурс]. – URL: <http://software-testing.org/testing/chto-takoe-konfiguracionnoe-testirovanie-configuration-testing.html> (дата обращения 21.04.2018)
- 8) CI definition and its main goal. // материалы сайта [Электронный ресурс]. – URL: <https://djangostars.com/blog/continuous-integration-circleci-vs-travisci-vs-jenkins/> (дата обращения 26.04.2018)
- 9) Continuous Integration для самых маленьких. // материалы сайта [Электронный ресурс]. – URL: <https://habr.com/post/190412/> (дата обращения 26.04.2018)
- 10) Непрерывная интеграция. // материалы сайта [Электронный

ресурс]. – URL: https://ru.wikipedia.org/wiki/Непрерывная_интеграция (дата обращения 26.04.2018)

11) XUnit. // материалы сайта [Электронный ресурс]. – URL: <https://ru.wikipedia.org/wiki/XUnit> (дата обращения 05.05.2018)

12) NUnit. // материалы сайта [Электронный ресурс]. – URL: <https://en.wikipedia.org/wiki/NUnit> (дата обращения 05.05.2018)

13) Testing_Framework. // материалы сайта [Электронный ресурс]. – URL: https://en.wikipedia.org/wiki/Unit_Testing_Framework (дата обращения 05.05.2018)

14) Обзор систем контроля версий. // материалы сайта [Электронный ресурс]. – URL: http://all-ht.ru/inf/prog/p_0_1.html (дата обращения 05.05.2018)

15) Рейтинг систем контроля версий. // материалы сайта [Электронный ресурс]. – URL: <https://tagline.ru/version-control-systems-rating/> (дата обращения 05.05.2018)

16) TestLink. // материалы сайта [Электронный ресурс]. – URL: <https://en.wikipedia.org/wiki/TestLink/> (дата обращения 05.05.2018)

17) Документация. // материалы сайта [Электронный ресурс]. – URL: <https://www.automiq.ru/> (дата обращения 05.05.2018)

18) JOHNSON D., Test Automation ROI // материалы сайта [Электронный ресурс]. – URL: http://www.dijohnic.com/test_automation_roi.pdf (дата обращения 15.05.2018)

19) HOFFMAN D., Cost Benefits Analysis of Test Automation // материалы сайта [Электронный ресурс]. – URL: <http://www.softwarequalitymethods.com/papers/star99%20model%20paper.pdf> (дата обращения 15.05.2018)

20) 21) KANER C., Improving the Maintainability of Automated Test Suites // материалы сайта [Электронный ресурс]. – URL: <http://www.kaner.com/pdfs/autosqa.pdf> (дата обращения 15.05.2018)

Приложение 1

(справочное)

Листинг теста AlphaAlarms_LaunchParam_Hist

```

/// <summary>
/// Запуск клиента в историческом режиме
/// </summary>
[Test, Retry(_numberRepetition)]
//[Ignore("Ignore the test Jenkins")]
public void AlphaAlarms_LaunchParam_Hist()
{
    Process Process_LaunchParam_Hist = new Process();

    try
    {
        KillAlphaAlarms();

        if (FileExists())
        {
            bool IsVisible = false;

            ProcessStartInfo Alarms = new ProcessStartInfo();

            Alarms.FileName = AlarmsAlarmsEXE;
            Alarms.Arguments = UIMap.ConstaractFilter(UIMap.Mode_Hist);

            Process_LaunchParam_Hist.StartInfo = Alarms;
            Process_LaunchParam_Hist.Start();

            Timer(25);

            IntPtr WindowHandle = MainHandle(Process_LaunchParam_Hist);

            if (WindowHandle == IntPtr.Zero)
            {
                ComparisonMethod(false, true, "Не удалось найти приложение Alpha.Alarms");
            }

            IntPtr WindowHandle2 = WinAPI.GetWindow(WindowHandle,
                WinAPI.GetWindow_Cmd.GW_CHILD);

            IntPtr WindowHandle3 = WinAPI.GetWindow(WindowHandle2,
                WinAPI.GetWindow_Cmd.GW_CHILD);

            IntPtr WindowHandle4 = WinAPI.GetWindow(WindowHandle3,
                WinAPI.GetWindow_Cmd.GW_CHILD);

            IntPtr WindowHandle5 = WinAPI.GetWindow(WindowHandle4,
                WinAPI.GetWindow_Cmd.GW_HWNDNEXT);

            IntPtr WindowHandle6 = WinAPI.GetWindow(WindowHandle5,
                WinAPI.GetWindow_Cmd.GW_HWNDNEXT);

            IntPtr WindowHandle7 = WinAPI.GetWindow(WindowHandle6,
                WinAPI.GetWindow_Cmd.GW_CHILD);

            IntPtr WindowHandle8 = WinAPI.GetWindow(WindowHandle7,
                WinAPI.GetWindow_Cmd.GW_HWNDNEXT);

            IntPtr WindowHandle9 = WinAPI.GetWindow(WindowHandle8,
                WinAPI.GetWindow_Cmd.GW_HWNDNEXT);
        }
    }
}

```

```

        Timer(25);

        if ((int)WindowHandle9 != 0) // Проверка нахождения всех дескрипторов
        {
            IsVisible = WinAPI.IsWindowVisible(WindowHandle9);
        }
        ComparisonMethod(IsVisible, true, "Неудалось запустить Alpha.");
    }
}
finally
{
    if (IsRunning(Process_LaunchParam_Hist))
    {
        Process_LaunchParam_Hist.CloseMainWindow();

        Timer(30);
        Process_LaunchParam_Hist.Close();
    }
}
}

```