

Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего
профессионального образования «Уральский федеральный университет имени первого Президента
России Б.Н.Ельцина»
Институт радиоэлектроники и информационных технологий - РтФ
Кафедра информационных технологий

ДОПУСТИТЬ К ЗАЩИТЕ
Зав. Кафедрой ИТ _Л.Г. Доросинский____
«_____» _____ 2014 г.

РАЗРАБОТКА МЕТОДИКИ ТЕСТИРОВАНИЯ ПО

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

230100.68 000000 005 ПЗ

Руководитель _____
Нормоконтролер _____
Студент гр. _____

Спиричева Н.Р.
Аксенов К.А.
Иванов Е.С.

Екатеринбург 2014 г.

РЕФЕРАТ

Актуальность. Основной всплеск интереса к теме тестирования пришёлся на 90-е годы и начался в США. Бурное развитие систем автоматизированной разработки ПО (CASE-средств) и сетевых технологий привело к росту рынка производства ПО и к пересмотру вопросов обеспечения качества и надёжности разрабатываемых программ. Резко усилившаяся конкуренция между производителями ПО потребовала особого внимания к качеству создаваемых продуктов, т.к. теперь у потребителя был выбор: многие фирмы предлагали свои продукты и услуги по достаточно приемлемым ценам, а потому можно было обратиться к тем, кто разработает программу не только быстро и дёшево, но и качественно. Ситуация осложнилась тем фактом, что в настоящее время компьютеризации подвержены практически все области человеческой жизни. И вопрос о качестве по начинает приобретать особую важность: сегодня это уже не только комфорт от работы в той или иной программе, сегодня ПО управляет оборудованием в больницах, диспетчерскими системами в аэропортах, атомными реакторами, космическими кораблями и т.д.

Осознав тот факт, что обеспечение высокого качества разрабатываемого ПО – это реальный путь «обойти» конкурентов, многие компании во всём мире вкладывают всё больше средств в обеспечение качества своих продуктов, создавая собственные группы и отделы, занимающиеся тестированием, или передавая тестирование своих продуктов сторонним организациям.

Наиболее крупные компании, заботящиеся о своей репутации и желающие пройти сертификацию на высокий уровень CMMI (Capability Maturity Model Integration) создают свои собственные системы управления качеством (Quality Management System), направленные на постоянное совершенствование производственных процессов и постоянное повышение качества ПО.

Сегодня тестирование стало обязательной частью процесса производства ПО. Оно направлено на обнаружение и устранение как можно большего числа ошибок. Следствием такой деятельности является повышение качества ПО по всем его характеристикам.

Анализ актуальности обусловили выбор темы исследования: «Разработка методики тестирования ПО».

Гипотеза исследования состоит в том, что независимо от количества методик тестирования, нету одной единой техники, с помощью которой можно было бы выявить, что ПО не содержит ошибок.

Целью исследования является изучение процесса тестирования, видов дефектов в ПО и их отслеживание, способов создания и применения тест кейсов, и, на основе полученных знаний,

разработка проекта авто-тестов для веб-сервиса "Эксперт". Дополнительной целью является проведение нагрузочного тестирования для веб-сервиса "Эксперт"

Для достижения поставленной цели необходимо решить следующие задачи:

- понять какое место занимает тестирование в разработке ПО;
- полностью ознакомиться с процессом тестирования в IT-компаниях;
- рассмотреть самые известные виды дефектов и причины их возникновения, а также системы отслеживания ошибок;
- познакомиться с техниками создания тестов и их применения;
- обзор ПО для нагрузочного тестирования;
- выбор ПО для проведения нагрузочного тестирования веб-сервиса "Эксперт";
- изучить процесс автоматизации функционального тестирования;
- разработать проект автоматизации для начинающих тестировщиков;
- разработать проект авто-тестов для веб-сервиса "Эксперт".

Объектом исследования является процесс тестирования ПО в IT-компаниях.

Предметом исследования являются техники создания тест кейсов и возможность их автоматизации.

Методы исследования включают в себя:

- методы тестирования программного обеспечения;
- методы определения качества продукта;
- методы верификации программного обеспечения;
- методы автоматизации тест-кейсов.

Теоретической основой исследования стали:

- отечественные и зарубежные исследования по обеспечению качества программного обеспечения;
- публикации на сайтах, посвященные тестированию программного обеспечения.

Теоретическая и практическая значимость работы заключается в том, что на основе полученных знаний:

- 1) можно в короткий срок ознакомиться с необходимой теорией по тестированию ПО, которая может помочь в успешном прохождении технического собеседования для устройства на работу тестировщиком;
- 2) по собранному материалу можно оформить методическое пособие для студентов и включить в программу обучения в качестве дополнительного курса "Тестирование ПО";
- 3) разработан проект авто-тестов для веб-сервиса "Эксперт" и введен в эксплуатацию.

На защиту выносятся следующие положения:

- 1) процесс тестирования ПО в IT-компаниях;
- 2) нагрузочное тестирование;
- 3) проект авто-тестов для веб-сервиса "Эксперт".

Апробация результатов исследования и публикации. Проект авто-тестов для веб-сервиса "Эксперт" введен в эксплуатацию.

Структура и объём работы. Выпускная квалификационная работа состоит из введения, 12 глав и заключения, изложенных на 106 страницах, а также списка литературы и приложений. В работе имеется 55 рисунков. Список литературы содержит 15 наименований.

СОДЕРЖАНИЕ

НОРМАТИВНЫЕ ССЫЛКИ	7
ВВЕДЕНИЕ	10
1 ОПРЕДЕЛЕНИЕ ТЕСТИРОВАНИЯ.....	12
2 ЖИЗНЕННЫЙ ЦИКЛ ПРОДУКТА И ТЕСТИРОВАНИЯ.....	13
3 МЕСТО ТЕСТИРОВАНИЯ В ПРОЦЕССЕ РАЗРАБОТКИ ПО.....	15
4 МЕТОДОЛОГИЯ ТЕСТИРОВАНИЯ	17
5 УРОВНИ ТЕСТИРОВАНИЯ	19
6 ВИДЫ ТЕСТИРОВАНИЯ.....	22
7 ПРОЦЕСС ТЕСТИРОВАНИЯ.....	26
7.1 Этапы и задачи	26
7.2 Принципы организации тестирования	27
7.3 Планирование тестирования	30
7.3.1 Вопросы, определяющие процесс планирования	30
7.3.2 Тест план.....	31
7.3.3 Виды тест планов	33
7.4 Тестовый случай (Test case). Виды. Структура.....	34
7.4.1 Виды Тестовых Случаев.....	34
7.4.2 Структура Тестовых Случаев (Test Case Structure).....	34
7.4.3 Детализация описания тест кейсов.....	36
7.4.4 Атрибуты тест кейса.....	37
8 ДЕФЕКТЫ. ПРИЧИНЫ, ОПИСАНИЕ, ОТСЛЕЖИВАНИЕ	41
8.1 Система отслеживания ошибок	42
8.2 Структура баг репорта	43
8.3 Серьезность и приоритет ошибки.....	44
8.4 Написание баг репорта	45
8.4.1 Требования к обязательным полям баг репорта	45
8.4.2 Основные ошибки при написании баг репортов.....	46
8.4.3 Заполнение полей баг репорта	47
8.4.4 Жизненный цикл бага	47
9 ТЕХНИКИ СОЗДАНИЯ ТЕСТОВ ДЛЯ ЧЕРНОГО ЯЩИКА.....	49
9.1 Эквивалентное разбиение.....	49
9.2 Анализ граничных значений	51
9.3 Анализ причинно-следственных связей.....	51
9.4 Парное тестирование	52
9.5 Предположение об ошибке	54

10 ПРИМЕНЕНИЕ ТЕХНИК.....	55
10.1 Эквивалентное разбиение и граничное условие	55
10.2 Попарное тестирование	60
10.3 Попарное тестирование с помощью РІСТ	64
11 АВТОМАТИЗАЦИЯ. НАГРУЗОЧНОЕ ТЕСТИРОВАНИЕ	69
11.1 Терминология нагрузочного тестирования	69
11.2 Цели нагрузочного тестирования	70
11.3 Этапы проведения нагрузочного тестирования	71
11.3.1 Анализ требования и сбор информации о тестируемой системе	71
11.3.2 Анализ требований в зависимости от типа проекта	72
11.3.3 Конфигурация тестового стенда для нагрузочного тестирования.....	72
11.3.4 Разработка модели нагрузки.....	74
11.4 Обзор программ нагрузочного тестирования веб-сервисов.....	75
11.5 Нагрузочное тестирование с помощью Jmeter	81
11.5.1 Подготовительные действия	81
11.5.2 Запись скрипта при помощи HTTP Proxy Server.....	82
11.5.3 Отладка скрипта	83
11.5.4 Параметризация.....	86
11.5.5 CSV Data Set Config	88
11.5.6 Создание ФА	89
12 АВТОМАТИЗИРОВАННОЕ ФУНКЦИОНАЛЬНОЕ ТЕСТИРОВАНИЕ	94
12.1 Преимущества и недостатки	94
12.2 Применение автоматизации	95
12.3 Как автоматизировать	96
12.4 Уровни автоматизации тестирования.....	97
12.5 Архитектура тестов.....	98
12.7 Проект по автоматизированному тестированию для начинающих	99
12.8 Проект авто-тестов для веб-сервиса Эксперт.....	106
12.8.1 Описание инфраструктуры страницы	106
12.8.2 Написание тест кейсов	109
ЗАКЛЮЧЕНИЕ.....	111
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	112
ПРИЛОЖЕНИЕ А.....	113
ПРИЛОЖЕНИЕ Б	116

НОРМАТИВНЫЕ ССЫЛКИ

В пояснительной записке использованы ссылки на следующие стандарты:

ГОСТ 7.32-2001	Отчет о научно-исследовательской работе. Структура и правила оформления
ГОСТ 34.602-89	Информационная технология. Комплекс стандартов на автоматизированные системы. Техническое задание на создание автоматизированной системы
ЕСПД ГОСТ 19.201-78	Техническое задание. Требования к содержанию и оформлению
СТП УГТУ-УПИ 1-96	Стандарт предприятия. Общие требования и правила оформления дипломных и курсовых проектов (работ)

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

В пояснительной записке используются следующие определения, обозначения и сокращения:

ПО	Программное обеспечение
СММІ	Capability Maturity Model Integration - комплексная модель производительности и зрелости – набор моделей (методологий) совершенствования процессов в организациях разных размеров и видов деятельности. СММІ содержит набор рекомендаций в виде практик, реализация которых, по мнению разработчиков модели, позволяет реализовать цели, необходимые для полной реализации определенных областей деятельности
IEEE	Institute of Electrical and Electronics Engineers - Институт инженеров по электротехнике и электронике международная некоммерческая ассоциация специалистов в области техники, мировой лидер в области разработки стандартов по радиоэлектронике и электротехнике
RUP	Rational Unified Process - рациональный унифицированный процесс разработки программного обеспечения, созданный компанией Rational Software
Баг	В программировании жаргонное слово, обычно обозначающее ошибку в программе или системе, которая выдает неожиданный или неправильный результат
ISO 9660	Стандарт, выпущенный Международной организацией по стандартизации, описывающий файловую систему для дисков CD-ROM
I/O	Ввод/вывод - взаимодействие между обработчиком информации (например, компьютер) и внешним миром, который может представлять как человек, так и любая другая система обработки информации. Ввод — сигнал или данные, полученные системой, а вывод — сигнал или данные, посланные ею (или из неё)
MDAC	Microsoft Data Access Components — совокупность технологий компании Microsoft, позволяющих получить унифицированный способ доступа к данным из различных реляционных и не реляционных источников. Термин MDAC является общим обозначением для всех разработанных компанией Microsoft технологий, связанных с базами данных

ООП	Объектно-ориентированное программирование
ОПФ	Организационно-правовая форма
FTP	File Transfer Protocol - стандартный протокол, предназначенный для передачи файлов по TCP-сетям
JDBC	Java DataBase Connectivity - платформенно-независимый промышленный стандарт взаимодействия Java-приложений с различными СУБД
СУБД	Система управления базами данных
ОПФ	Организационно-правовая форма
ОВД	Основной вид деятельности
GUI	Разновидность пользовательского интерфейса, в котором элементы интерфейса (меню, кнопки, значки, списки и т. п.), представленные пользователю на дисплее, исполнены в виде графических изображений
ИС	Информационная система
ИТ	Информационные технологии

ВВЕДЕНИЕ

Основной всплеск интереса к теме тестирования пришёлся на 90-е годы и начался в США. Бурное развитие систем автоматизированной разработки ПО (CASE-средств) и сетевых технологий привело к росту рынка производства ПО и к пересмотру вопросов обеспечения качества и надёжности разрабатываемых программ. Резко усилившаяся конкуренция между производителями ПО потребовала особого внимания к качеству создаваемых продуктов, т.к. теперь у потребителя был выбор: многие фирмы предлагали свои продукты и услуги по достаточно приемлемым ценам, а потому можно было обратиться к тем, кто разработает программу не только быстро и дёшево, но и качественно. Ситуация осложнилась тем фактом, что в настоящее время компьютеризации подвержены практически все области человеческой жизни. И вопрос о качестве ПО начинает приобретать особую важность: сегодня это уже не только комфорт от работы в той или иной программе, сегодня ПО управляет оборудованием в больницах, диспетчерскими системами в аэропортах, атомными реакторами, космическими кораблями и т.д.

Осознав тот факт, что обеспечение высокого качества разрабатываемого ПО – это реальный путь «обойти» конкурентов, многие компании во всём мире вкладывают всё больше средств в обеспечение качества своих продуктов, создавая собственные группы и отделы, занимающиеся тестированием, или передавая тестирование своих продуктов сторонним организациям.

Наиболее крупные компании, заботящиеся о своей репутации и желающие пройти сертификацию на высокий уровень CMMI (Capability Maturity Model Integration) создают свои собственные системы управления качеством (Quality Management System), направленные на постоянное совершенствование производственных процессов и постоянное повышение качества ПО.

Сегодня тестирование стало обязательной частью процесса производства ПО. Оно направлено на обнаружение и устранение как можно большего числа ошибок. Следствием такой деятельности является повышение качества ПО по всем его характеристикам.

Существующие на сегодняшний день методы тестирования программного обеспечения не позволяют однозначно и полностью устранить все дефекты и ошибки и установить корректность функционирования программного продукта. Поэтому, все существующие методы тестирования действуют в рамках формального процесса проверки исследуемого или разрабатываемого программного продукта.

Такой процесс формальной проверки или верификации может доказать, что дефекты отсутствуют, с точки зрения используемого метода. То есть нет никакой возможности точно

установить или гарантировать отсутствие дефектов в программном продукте с учётом человеческого фактора, присутствующего на всех этапах жизненного цикла программного обеспечения.

Существует множество подходов к решению задачи тестирования и верификации программного обеспечения, но эффективное тестирование сложных программных продуктов — это процесс в высшей степени творческий, не сводящийся к следованию строгим и чётким процедурам или созданию таковых.

Конечной целью любого процесса тестирования является обеспечение такого ёмкого (совокупного) понятия как *Качество*, с учётом всех или наиболее критичных для данного конкретного случая составляющих.

Тестирование программного обеспечения — попытка определить, выполняет ли программа то, что от неё ожидают. Как правило, никакое тестирование не может дать абсолютной гарантии работоспособности программы в будущем.

Задачи тестирования программного обеспечения — снизить стоимость разработки путем раннего обнаружения дефектов.

1 ОПРЕДЕЛЕНИЕ ТЕСТИРОВАНИЯ

В соответствие с IEEE Std 829-1983 **Тестирование** — это процесс анализа ПО, направленный на выявление отличий между его реально существующими и требуемыми свойствами (дефект) и на оценку свойств ПО [5].

По ГОСТ Р ИСО МЭК 12207-99 в жизненном цикле ПО определены среди прочих вспомогательные процессы верификации, аттестации, совместного анализа и аудита. Процесс верификации является процессом определения того, что программные продукты функционируют в полном соответствии с требованиями или условиями, реализованными в предшествующих работах. Данный процесс может включать анализ, проверку и испытание (тестирование). Процесс аттестации является процессом определения полноты соответствия установленных требований, созданной системы или программного продукта их функциональному назначению. Процесс совместного анализа является процессом оценки состояний и, при необходимости, результатов работ (продуктов) по проекту. Процесс аудита является процессом определения соответствия требованиям, планам и условиям договора. В сумме эти процессы и составляют то, что обычно называют тестированием.

Тестирование основывается на тестовых процедурах с конкретными входными данными, начальными условиями и ожидаемым результатом, разработанными для определенной цели, такой, как проверка отдельной программы или верификация соответствия на определенное требование. Тестовые процедуры могут проверять различные аспекты функционирования программы — от правильной работы отдельной функции до адекватного выполнения бизнес-требований.

При выполнении проекта необходимо учитывать, в соответствии с какими стандартами и требованиями будет проводиться тестирование продукта. Какие инструментальные средства будут (если будут) использоваться для поиска и для документирования найденных дефектов. Если помнить о тестировании с самого начала выполнения проекта, тестирование разрабатываемого продукта не доставит неприятных неожиданностей. А значит и качество продукта, скорее всего, будет достаточно высоким.

2 ЖИЗНЕННЫЙ ЦИКЛ ПРОДУКТА И ТЕСТИРОВАНИЯ

Все чаще используются итеративные процессы разработки ПО, в частности, технология RUP — Rational Unified Process. На рисунке 2.1 можно увидеть жизненный цикл продукта по RUP. При использовании такого подхода тестирование перестает быть процессом «на отшибе», который запускается после того, как программисты написали весь необходимый код. Работа над тестами начинается с самого начального этапа выявления требований к будущему продукту и тесно интегрируется с текущими задачами. И это предъявляет новые требования к тестировщикам. Их роль не сводится просто к выявлению ошибок как можно полнее и как можно раньше. Они должны участвовать в общем процессе выявления и устранения наиболее существенных рисков проекта. Для этого на каждую итерацию определяется цель тестирования и методы ее достижения. А в конце каждой итерации определяется, насколько эта цель достигнута, нужны ли дополнительные испытания, и не нужно ли изменить принципы и инструменты проведения тестов. В свою очередь, каждый обнаруженный дефект должен пройти через свой собственный жизненный цикл.

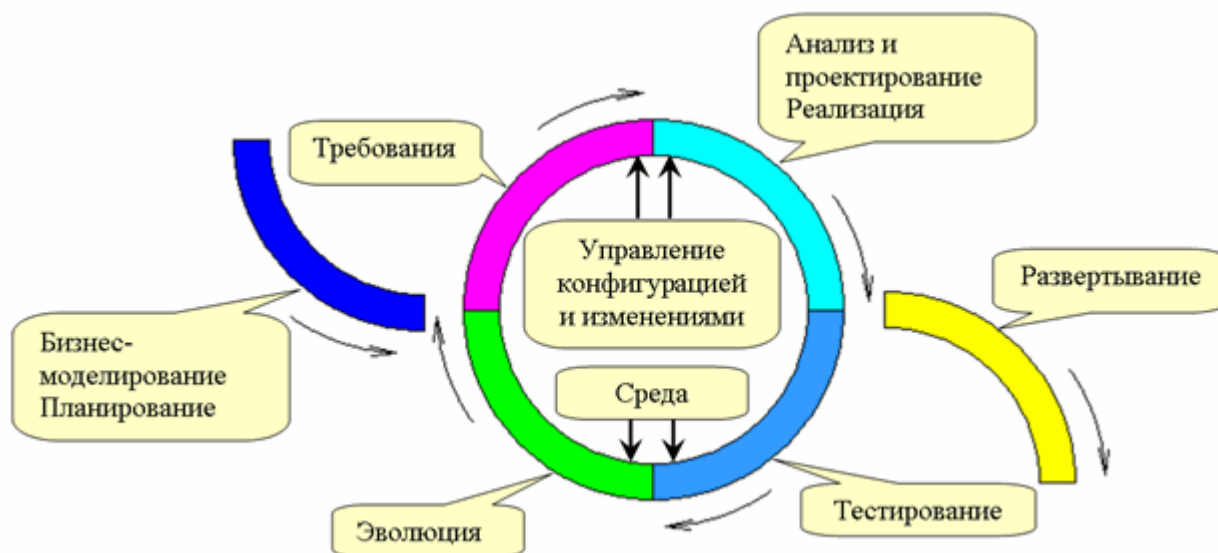


Рисунок 2.1 - Жизненный цикл продукта по RUP

Тестирование обычно проводится циклами, каждый из которых имеет конкретный список задач и целей. Цикл тестирования может совпадать с итерацией или соответствовать ее определенной части. Как правило, цикл тестирования проводится для конкретной сборки системы.

Жизненный цикл программного продукта, который изображен на рисунке 2.2 состоит из серии относительно коротких итераций. **Итерация** — это законченный цикл разработки, приводящий к выпуску конечного продукта или некоторой его сокращенной версии, которая расширяется от итерации к итерации, чтобы, в конце концов, стать законченной системой.

Каждая итерация включает, как правило, задачи планирования работ, анализа, проектирования, реализации, тестирования и оценки достигнутых результатов. Однако соотношения этих задач может существенно меняться. В соответствии с соотношением различных

задач в итерации они группируются в фазы. В первой фазе — **Начало** — основное внимание уделяется задачам анализа. В итерациях второй фазы — **Разработка** — основное внимание уделяется проектированию и опробованию ключевых проектных решений. В третьей фазе — **Построение** — наиболее велика доля задач разработки и тестирования. А в последней фазе — **Передача** — решаются в наибольшей мере задачи тестирования и передачи системы Заказчику.

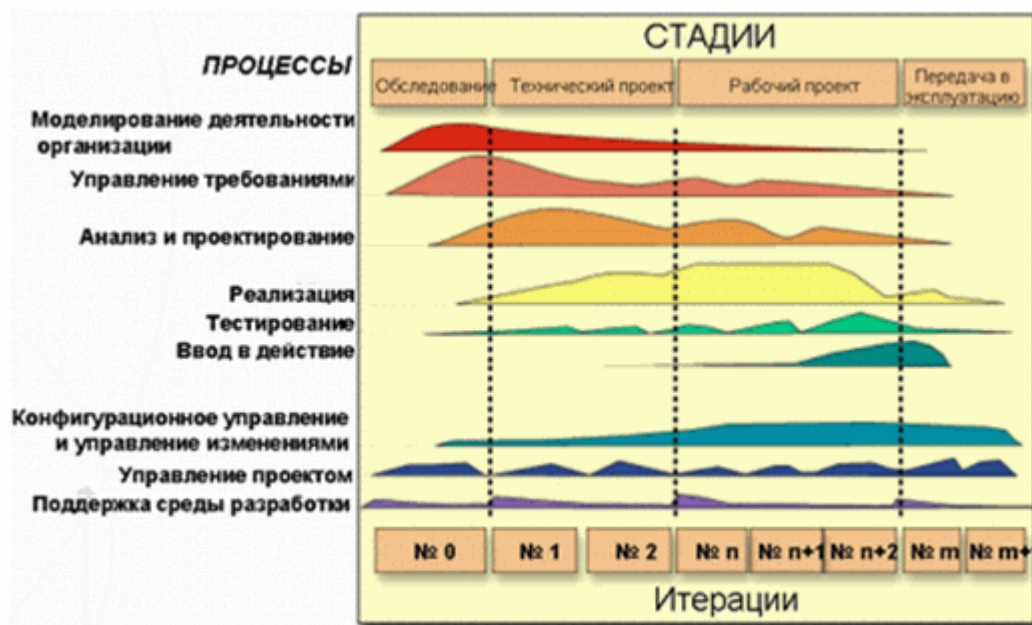


Рисунок 2.2 - Итерации жизненного цикла программного продукта

Каждая фаза имеет свои специфические цели в жизненном цикле продукта и считается выполненной, когда эти цели достигнуты. Все итерации, кроме, итераций фазы Начало, завершаются созданием функционирующей версии разрабатываемой системы [6].

3 МЕСТО ТЕСТИРОВАНИЯ В ПРОЦЕССЕ РАЗРАБОТКИ ПО

Структура фирмы — разработчика программного обеспечения отражает этапы жизненного цикла программного средства. То или иное подразделение обеспечивает выполнение работ по одному или нескольким этапам жизненного цикла программного обеспечения.

Аналитический отдел. В задачи аналитического отдела входят:

- определение концепций и функционального направления развития программного продукта;
- проведение предпроектного обследования;
- определение функциональных возможностей системы;
- определение (совместно с разработчиками) технических требований к системе;
- описание бизнес-процессов предметной области в терминах, понятных разработчикам (постановки задач и спецификации на разработку);
- написание постановок задач и спецификаций на доработку программного средства при изменении законодательства, требований клиентов, расширении функциональных возможностей продукта;
- контроль процесса реализации новых возможностей в программных продуктах компании.

Отдел документации. Часто данный отдел не выделяется в обособленную структуру, он может входить, например, в состав аналитического отдела.

В задачи отдела входят написание технической документации для конечного пользователя, отслеживание изменений в программном средстве и актуализация в документации.

Отдел разработки. Это ключевой отдел для фирмы. Если без остальных отделов зачастую можно обойтись, то без отдела разработки нельзя. В его задачи входят:

- определение (совместно с аналитиками) технических требований к системе;
- реализация базовых функций программного средства;
- расширение перечня функций программного средства (реализация доработок);
- исправление найденных ошибок;
- адаптация программного продукта для функционирования в других условиях (переход на новую СУБД, новый язык программирования и пр.);
- оптимизация программного продукта (увеличение быстродействия, надежности и пр.).

Отдел технической поддержки (горячая линия). Осуществляет консультации пользователей по вопросам, связанным с установкой и эксплуатацией программного средства по различным каналам связи (телефон, почта, электронная почта).

Отдел тестирования. В задачи отдела входят:

- комплексный контроль качества;
- подготовка тестовой документации (планы тестирования и пр.);
- обнаружение и локализация ошибок в функционировании программных продуктов;
- фиксирование и отслеживание ошибок в функционировании программных средств;
- проверка соответствия документации программного продукта стандартам и реально реализованным функциям;
- участие в разработке и внедрении системы качества;
- автоматизация тестирования;
- оценка производительности разрабатываемых программных средств на различных программно-аппаратных платформах и их специфических конфигурациях.

В некоторых компаниях на отдел тестирования возлагаются сборка и выпуск программного обеспечения (в некоторых компаниях этим занимается отдел разработки) [7].

Все отделы компании взаимодействуют между собой, данные взаимодействия упорядочены между собой и представляют производственные технологические процессы. Технологические процессы, как правило, регламентированы внутренними документами или внутрикорпоративными стандартами; в совокупности представляют собой технологический цикл производства программного средства.

Типичных технологических цепочек внутри компании — разработчика программного обеспечения большое множество. В качестве примера рассмотрим схему взаимодействия отдела тестирования программного обеспечения с другими отделами при обнаружении ошибки во время функционирования программного обеспечения у пользователя.

4 МЕТОДОЛОГИЯ ТЕСТИРОВАНИЯ

В терминологии профессионалов тестирования (программного и некоторого аппаратного обеспечения), фразы «тестирование белого ящика» и «тестирование чёрного ящика» относятся к тому, имеет ли разработчик тестов доступ к исходному коду тестируемого программного обеспечения, или же тестирование выполняется через пользовательский интерфейс либо прикладной программный интерфейс, предоставленный тестируемым модулем.

При тестировании методом **белого ящика** (*white-box testing*, также говорят — *прозрачного ящика*, оно же **структурное** тестирование), разработчик теста имеет доступ к исходному коду программ и может писать код, который связан с библиотеками тестируемого программного обеспечения. На рисунке 4.1 можно увидеть схематическое изображение данного метода.

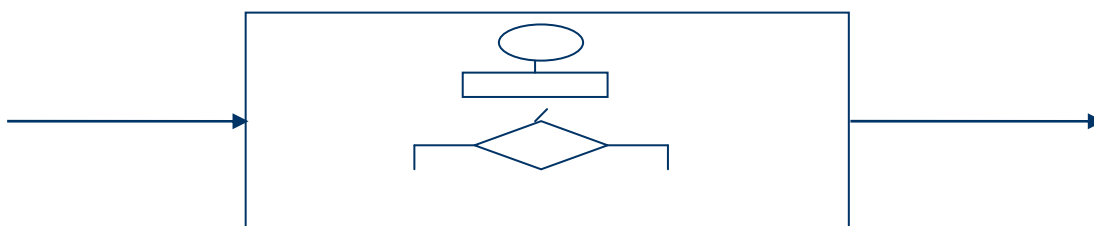


Рисунок 4.1 - Метод белого ящика

Тесты создаются на основе знаний о структуре самой системы и о том, как она работает. Критерии полноты основаны на проценте элементов кода, которые отработали в ходе выполнения тестов. Для оценки степени соответствия требованиям могут привлекаться дополнительные знания о связи требований с определенными ограничениями на значения внутренних данных системы (например, на значения параметров вызовов, результатов и локальных переменных).

При тестировании методом **чёрного ящика**, схему которого можно увидеть на рисунке 4.2, тестировщик имеет доступ к программному обеспечению только через те же интерфейсы, что и заказчик или пользователь, либо через внешние интерфейсы, позволяющие другому компьютеру либо другому процессу подключиться к системе для тестирования.

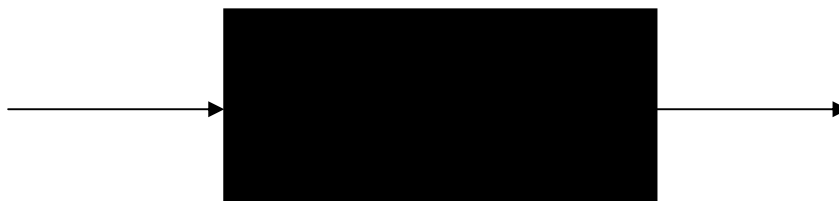


Рисунок 4.2 - Метод черного ящика

Тестирование черного ящика, нацеленное на проверку требований. Тесты для него и критерий полноты тестирования строятся на основе требований и ограничений, четко зафиксированных в спецификациях, стандартах, внутренних нормативных документах. Часто такое тестирование называется тестированием на соответствие (*conformance testing*). Частным случаем его является функциональное тестирование — тесты для него, а также используемые критерии полноты проведенного тестирования определяют на основе требований к функциональности.

Помимо методов тестирования «белый ящик» и «черный ящик» различают тестирование методом «серого ящика», схема которого изображена на рисунке 4.3.

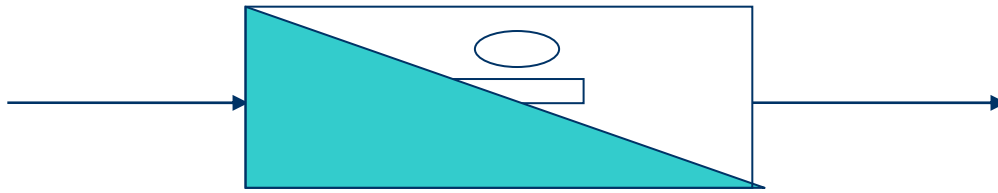


Рисунок 4.3 - Метод серого ящика

В данном случае у человека, который разрабатывает тест кейсы, есть некоторая информация о внутренней структуре приложения или о деталях реализации. Данный метод применяется в последнее время чаще предыдущих.

5 УРОВНИ ТЕСТИРОВАНИЯ

Существует классификация видов тестирования, которая основана на том уровне детализации работ проекта, на который оно нацелено. На рисунке 5.1 изображены уровни тестирования. Эти же разновидности тестирования можно связать с фазой жизненного цикла, на которой они выполняются.

Модульное тестирование (Unit-testing) — уровень тестирования, на котором тестируется минимально возможный для тестирования компонент, например, отдельный класс или функция. На этом уровне применяются методы «белого ящика». В современных проектах модульное тестирование («юнит-тестинг») осуществляется разработчиками [9].

Модульное тестирование предназначено для проверки правильности отдельных модулей, вне зависимости от их окружения. При этом проверяется, что если модуль получает на вход данные, удовлетворяющие определенным критериям корректности, то и результаты его корректны. Для описания критериев корректности входных и выходных данных часто используют программные контракты — предусловия, описывающие для каждой операции, на каких входных данных она предназначена работать, постусловия, описывающие для каждой операции, как должны соотноситься входные данные с возвращаемыми ею результатами, и инварианты, определяющие критерии целостности внутренних данных модуля.

Основной недостаток модульного тестирования состоит в том, что проводить его можно, только когда проверяемый элемент программы уже разработан. Снизить влияние этого ограничения можно, подготавливая тесты (а это — наиболее трудоемкая часть тестирования) на основе требований заранее, когда исходного кода еще нет.

Модульное тестирование является важной составной частью **отладочного тестирования**, выполняемого разработчиками для отладки написанного ими кода.

Интеграционное тестирование (Integration testing) – уровень тестирования, на котором отдельные программные модули объединяются и тестируются в группе. Обычно интеграционное тестирование проводится после модульного тестирования (юнит-тесты для модулей должны быть выполнены и найденные ошибки исправлены) [9].

Интеграционное тестирование в качестве входных данных использует модули, над которыми было проведено модульное тестирование, группирует их в более крупные множества, выполняет тесты, определённые в плане тестирования для этих множеств, и представляет их в качестве выходных данных и входных для последующего системного тестирования.

При этом проверяется, что в ходе совместной работы модули обмениваются данными и вызовами операций, не нарушая взаимных ограничений на такое взаимодействие, например, предусловий вызываемых операций.

Интеграционное тестирование выполняется разработчиками используется при отладке, но на более позднем этапе разработки.

Системное тестирование (*System testing*) - это тестирование программного обеспечения, выполняемое на полной, интегрированной системе, с целью проверки соответствия системы исходным требованиям. Системное тестирование относится к методам тестирования «чёрного ящика», и, тем самым, не требует знаний о внутреннем устройстве системы.

Системное тестирование выполняется через внешние интерфейсы программного обеспечения и тесно связано с тестированием пользовательского интерфейса (или через пользовательский интерфейс), проводимым при помощи имитации действий пользователей над элементами этого интерфейса. Частными случаями этого вида тестирования являются тестирование графического пользовательского интерфейса (Graphical User Interface, GUI) и пользовательского интерфейса Web-приложений (WebUI). Системное тестирование выполняется инженерами по тестированию.

Приемочное тестирование (*Acceptance testing*) - это тестирование готового продукта конечными пользователями на реальном окружении, в котором будет функционировать тестируемое приложение. Приемочные тесты разрабатываются пользователями, обычно, в виде сценариев. Для того, чтобы найти больше ошибок рекомендуется планировать не только системное тестирование и приемочное, но и модульное и интеграционное [5].

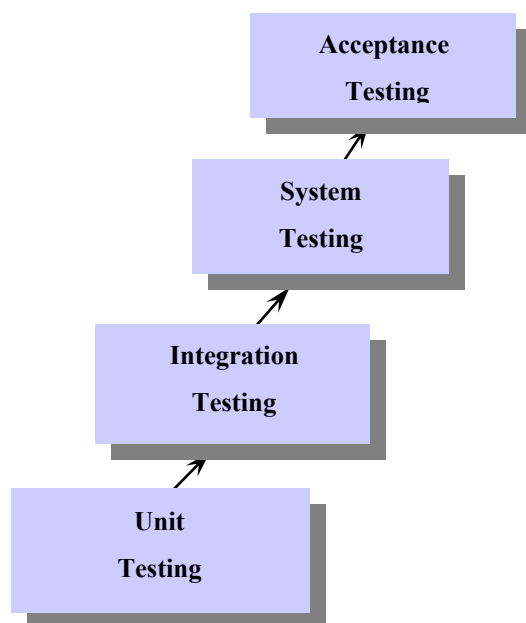


Рисунок 5.1 - Уровни тестирования

Статическое тестирование (*Static testing*) – тестирование, в ходе которого тестируемая программа (код) не выполняется (запускается). Анализ программы происходит на основе исходного кода, который вычитывается вручную, либо анализируется специальными инструментами.

Примеры статического тестирования:

- обзоры (Reviews);
- инспекции (Inspections);
- сквозные просмотры (Walkthroughs);
- аудиты (Audits).

Динамическое тестирование (Dynamic testing) – тестирование, в ходе которого тестируемая программа (код) выполняется (запускается).

Альфа-тестирование — тестирование в процессе разработки.

Бета-тестирование — тестирование выполняется пользователями (end-users).

Перед тем, как выпускается программное обеспечение, как минимум, оно должно проходить стадии альфа (внутреннее пробное использование) и бета (пробное использование с привлечением отобранных внешних пользователей) версий. Отчеты об ошибках, поступающие от пользователей этих версий продукта, обрабатываются в соответствии с определенными процедурами, включающими подтверждающие тесты (любого уровня), проводимые специалистами группы разработки. Данный вид тестирования не может быть заранее спланирован.

Регрессионное тестирование (Regression testing) – тестирование функциональности, которая была уже протестирована до внесения какого-либо изменения [5].

После внесения изменений в очередную версию программы, регрессионные тесты подтверждают, что сделанные изменения не повлияли на работоспособность остальной функциональности приложения. Регрессионное тестирование может выполняться как вручную, так и средствами автоматизации тестирования.

Определение успешности регрессионных тестов (IEEE 610-90 “Standard Glossary of Software Engineering Terminology”) гласит: “повторное выборочное тестирование системы или компонент для проверки сделанных модификаций не должно приводить к непредусмотренным эффектам”. На практике это означает, что если система успешно проходила тесты до внесения модификаций, она должна их проходить и после внесения таковых. Основная проблема регрессионного тестирования заключается в поиске компромисса между имеющимися ресурсами и необходимостью проведения таких тестов по мере внесения каждого изменения. В определенной степени, задача состоит в том, чтобы определить критерии “масштабов” изменений, с достижением которых необходимо проводить регрессионные тесты.

«Смок-тест» (Smoke Testing, «дымовое тестирование») в тестировании означает минимальный набор тестов на явные ошибки. Дымовой тест обычно выполняется самим программистом; не проходящую этот тест программу не имеет смысла отдавать на более глубокое тестирование [5].

6 ВИДЫ ТЕСТИРОВАНИЯ

Функциональное тестирование (Functional testing) - цель данного тестирования состоит в том, чтобы убедиться в надлежащем функционировании объекта тестирования:

- каждое функциональное требование транслируется в тест-кейсы (используя техники «черного ящика») для того, чтобы проверить, что система функционирует в точности, как и описано в спецификации (функциональных требованиях к системе);
- проверяются, все ли функциональные требования действительно запрограммированы/реализованы.

Тестирование производительности (Performance testing) - тестирование, которое проводится с целью определения, как быстро работает система или её часть под определённой нагрузкой. Также может служить для проверки и подтверждения других атрибутов качества системы, таких как масштабируемость, надёжность и потребление ресурсов. Существует особый подвид таких тестов, когда делается попытка достижения количественных пределов, обусловленных характеристиками самой системы и ее операционного окружения:

- продемонстрировать, что система удовлетворяет критериям производительности при заданных условиях;
- измерить, какая часть системы является причиной «плохой» производительности системы;
- измерить время реакции на действие пользователя, время отклика на запрос, и т.д [11].

Стрессовое тестирование (Stress testing) - обычно используется для понимания пределов пропускной способности приложения. Этот тип тестирования проводится для определения надёжности системы во время экстремальных или диспропорциональных нагрузок и отвечает на вопросы о достаточной производительности системы в случае, если текущая нагрузка сильно превысит ожидаемый максимум.

Нагрузочное тестирование (Load testing) - это простейшая форма тестирования производительности. Нагрузочное тестирование обычно проводится для того, чтобы оценить поведение приложения под заданной ожидаемой нагрузкой. Этой нагрузкой может быть, например, ожидаемое количество одновременно работающих пользователей приложения, совершающих заданное число транзакций за интервал времени. Такой тип тестирования обычно позволяет получить время отклика всех самых важных бизнес-транзакций. В случае наблюдения за базой данных, сервером приложений, сетью и т.д., этот тип тестирования может также идентифицировать некоторые узкие места приложения [11].

Тестирование стабильности (Stability testing) проводится с целью убедиться в том, что приложение выдерживает ожидаемую нагрузку в течение длительного времени. При проведении этого вида тестирования осуществляется наблюдение за потреблением приложением памяти,

чтобы выявить потенциальные утечки. Такое тестирование выявляет деградацию производительности, выражающуюся в снижении скорости обработки информации и/или увеличением времени ответа приложения после продолжительной работы по сравнению с началом теста.

Тестирование удобства использования (Usability testing) - исследование, выполняемое с целью определения, удобен ли некоторый искусственный объект (такой как веб-страница, пользовательский интерфейс или устройство) для его предполагаемого применения. Таким образом, проверка эргономичности измеряет эргономичность объекта или системы. Проверка эргономичности сосредоточена на определённом объекте или небольшом наборе объектов, в то время как исследования взаимодействия человек-компьютер в целом — формулируют универсальные принципы [11].

Это метод оценки удобства продукта в использовании, основанный на привлечении пользователей в качестве тестируемых, испытателей и суммировании полученных от них выводов.

При испытании многих продуктов пользователю предлагают в «лабораторных» условиях решить основные задачи, для выполнения которых этот продукт разрабатывался, и просят высказывать во время выполнения этих тестов свои замечания.

Процесс тестирования фиксируется в протоколе (логе) и/или на аудио- и видеоустройства — с целью последующего более детального анализа.

Если юзабилити-тестирование выявляет какие-либо трудности (например сложности в понимании инструкций, выполнении действий или интерпретации ответов системы), то разработчики должны доработать продукт и повторить тестирование.

Наблюдение за тем, как люди взаимодействуют с продуктом, нередко позволяет найти для него более оптимальные решения. Если при тестировании используется модератор, то его задача — держать респондента сфокусированным на задачах (но при этом не "помогать" ему решать эти задачи).

Основную трудность после проведения процедуры юзабилити-тестирования нередко представляют большие объёмы и беспорядочность полученных данных. Поэтому для последующего анализа важно зафиксировать:

- 1) речь модератора и респондента;
- 2) выражение лица респондента (снимается на видеокамеру);
- 3) изображение экрана компьютера, с которым работает респондент;
- 4) различные события, происходящие на компьютере, связанные с действиями пользователя:

- перемещение курсора и нажатия на клавиши мыши;
- использование клавиатуры;
- переходы между экранами (браузера или другой программы).

Все эти потоки данных должны быть синхронизированы по тайм-кодам, чтобы при анализе их можно было бы соотносить между собой.

Наряду с модератором в тестировании нередко участвуют наблюдатели. По мере обнаружения проблем они делают свои заметки о ходе тестирования так, чтобы после можно было синхронизировать их с основной записью. В итоге каждый значимый фрагмент записи теста оказывается прокомментирован в заметках наблюдателя. В идеале ведущий (т.е. модератор) представляет разработчика, наблюдатели — заказчика (например издателя), а испытатели — конечного пользователя (например покупателя).

Существует еще один подход к usability-тестированию: для решения задачи предложенной пользователю разрабатывается "идеальный" сценарий решения этой задачи. Как правило, это сценарий, на который ориентировался разработчик. При выполнении задачи пользователями регистрируются их отклонения от задуманного сценария для последующего анализа. После нескольких итераций доработки программы и последующего тестирования можно получить удовлетворительный интерфейс с точки зрения пользователя.

Тестирование интерфейса пользователя (UI testing) - тестирование графического интерфейса пользователя для того, чтобы убедиться, что он соответствует принятым стандартам и их требованиям. Проверяется, как приложение обрабатывает ввод с клавиатуры и «мышки» и как отображаются элементы графического интерфейса (текст, кнопки, меню, списки и прочие элементы).

Тестирование безопасности (security testing) - оценка уязвимости программного обеспечения к различным атакам. Компьютерные системы очень часто являются мишенью незаконного проникновения. Под проникновением понимается широкий диапазон действий: попытки хакеров проникнуть в систему из спортивного интереса, месть рассерженных служащих, взлом мошенниками для незаконной наживы. Тестирование безопасности проверяет фактическую реакцию защитных механизмов, встроенных в систему, на проникновение. В ходе тестирования безопасности испытатель играет роль взломщика. Ему разрешено все:

- попытки узнать пароль с помощью внешних средств;
- атака системы с помощью специальных утилит, анализирующих защиты;
- подавление, ошеломление системы (в надежде, что она откажется обслуживать других клиентов);
- целенаправленное введение ошибок в надежде проникнуть в систему в ходе восстановления;

- просмотр несекретных данных в надежде найти ключ для входа в систему.

Тестирование локализации (Localization testing) - многогранная вещь, подразумевающая проверку множества аспектов, связанных с адаптацией продукта для пользователей из других стран. Например, тестирование локализации для пользователей из Японии может заключаться в проверке того, не выдаст ли система ошибку, если этот пользователь на сайте знакомств введет рассказ о себе символами *Kanji*, а не английским шрифтом.

Тестирование совместимости (Compatibility testing) — вид нефункционального тестирования, основной целью которого является проверка корректной работы продукта в определенном окружении. Окружение может включать в себя следующие элементы:

- аппаратная платформа;
- сетевые устройства;
- периферия (принтеры, CD/DVD-приводы, веб-камеры и пр.);
- операционная система (Unix, Windows, MacOS, ...);
- базы данных (Oracle, MS SQL, MySQL, ...);
- системное программное обеспечение (веб-сервер, файрвол, антивирус, ...);
- браузеры (Internet Explorer, Firefox, Opera, Chrome, Safari).

7 ПРОЦЕСС ТЕСТИРОВАНИЯ

7.1 Этапы и задачи

Тестирование — это проверка соответствия программного обеспечения требованиям, осуществляемая с помощью наблюдения за его работой в специальных, искусственно построенных ситуациях. Такого рода ситуации называют тестовыми или просто тестами [5].

Тестирование — конечная процедура. Набор ситуаций, в которых будет проверяться тестируемое программное обеспечение, всегда конечен. Более того, он должен быть настолько мал, чтобы тестирование можно было провести в рамках проекта разработки программного обеспечения, не слишком увеличивая его бюджет и сроки. Это означает, что при тестировании всегда проверяется очень небольшая доля всех возможных ситуаций.

Тестирование может использоваться для достаточно уверенного вынесения оценок о качестве программного обеспечения. Для этого необходимо иметь **критерии полноты** тестирования, описывающие важность различных ситуаций для оценки качества, а также эквивалентности и зависимости между ними. Этот критерий может утверждать, что все равно в какой из ситуаций, А или В, проверять правильность работы программного обеспечения, или, если программа правильно работает в ситуации А, то, скорее всего, в ситуации В все тоже будет правильно. Часто критерий полноты тестирования задается при помощи определения эквивалентности ситуаций, дающей конечный набор классов ситуаций. В этом случае считают, что все равно, какую из ситуаций одного класса использовать в качестве теста. Такой критерий называют **критерием тестового покрытия**, а процент классов эквивалентности ситуаций, случившихся вовремя тестирования, — достигнутым **тестовым покрытием**.

Таким образом, основные задачи тестирования: построить такой набор ситуаций, который был бы достаточно представителен и позволял бы завершить тестирование с достаточной степенью уверенности в правильности программного обеспечения вообще, и убедиться, что в конкретной ситуации программное обеспечение работает правильно, в соответствии с требованиями.

Организация тестирования программного обеспечения регулируется следующими стандартами:

- *ieee 829-1998 Standard for Software Test Documentation*. Описывает виды документов, служащих для подготовки тестов;
- *ieee 1008-1987 (R1993, R2002) Standard for Software Unit Testing*. Описывает организацию модульного тестирования;
- *iso/iec 12119:1994 (аналог AS/NZS 4366:1996 и ГОСТ Р-2000, также принят IEEE под номером IEEE 1465-1998) Information Technology. Software packages — Quality requirements and testing*. Описывает требования к процедурам тестирования программных систем [5].

На рисунке 7.1 изображена схема процесса тестирования. Разработка тестов происходит на основе проверяемых требований и критерия полноты тестиирования. Разработанный тесты формируются в тейс-кейс (набор тестов) и выполняем на ПО, которое нужно протестировать. После прогона всех тестов анализируется результат, в результате чего можно выявить ошибки в программе.



Рисунок 7.1 - Схема процесса тестирования

Тестировать можно соблюдение любых требований, соответствие которым выявляется во время работы программного обеспечения. Из характеристик качества по ISO 9126 этим свойством не обладают только атрибуты удобства сопровождения. Поэтому выделяют виды тестирования, связанные с проверкой определенных характеристик и атрибутов качества — тестирование функциональности, надежности, удобства использования, переносимости и производительности, а также тестирование защищенности, функциональной пригодности и пр. Кроме того, особо выделяют нагрузочное или стрессовое тестирование, проверяющее работоспособность программного обеспечения и показатели его производительности в условиях повышенных нагрузок — при большом количестве пользователей, интенсивном обмене данными с другими системами, большом объеме передаваемых или используемых данных и пр [3].

7.2 Принципы организации тестирования

Тест хороший только в том случае, в котором высока вероятность обнаружить ошибку. Эта аксиома является фундаментальным принципом тестирования. Поскольку невозможно показать, что программа не имеет ошибок и, значит, все такие попытки бесплодны, процесс тестирования должен представлять собой попытки обнаружить в программе прежде не найденные ошибки.

Одна из самых сложных проблем при тестировании — решить, когда нужно его остановить. Исчерпывающее тестирование (т. е. испытание всех входных значений) невозможно. Таким образом, при тестировании идет столкновение с экономической проблемой: как выбрать конечное число тестов, которое дает максимальную отдачу (вероятность обнаружения ошибок) для

данных затрат. Известно слишком много случаев, когда написанные тесты имели крайне малую вероятность обнаружения новых ошибок, в то время как довольно очевидные хорошие тесты оставались незамеченными.

Следует избегать тестирования программы ее автором. Ни один программист не должен пытаться тестировать свою собственную программу. Это относится ко всем формам тестирования — как к тестированию системы, так и к тестированию внешних функций и даже отдельных модулей. Тестирование должно быть в высшей степени разрушительным процессом, но имеются глубокие психологические причины, по которым программист не может относиться к своей собственной программе как разрушитель. Дополнительное давление (например, жесткий график) на отдельного программиста или весь коллектив разработчиков проекта часто мешает программисту или всему коллективу выполнить адекватное тестирование. Если модуль содержит дефекты вследствие каких-то ошибок перевода, довольно высока вероятность того, что программист допустит ту же ошибку перевода (например, неправильно интерпретирует спецификации) и при подготовке тестов. Все ошибки в его понимании других модулей и их сопряжении также отразятся на тестах.

Отсюда не следует, что программист не может тестировать свою программу. Многие программисты с этим вполне успешно справляются. Здесь лишь делается вывод о том, что тестирование является более эффективным, если оно выполняется кем-либо другим. Все рассуждения не относятся к отладке, т. е. к исправлению уже известных ошибок. Эта работа эффективнее выполняется самим автором программы.

Необходимая часть любого теста — описание ожидаемых выходных данных или результатов. Одна из самых распространенных ошибок при тестировании состоит в том, что результаты каждого теста не прогнозируются до его выполнения. Ожидаемые результаты нужно определять заранее, чтобы не возникла ситуация, когда глаз видит то, что хочет увидеть. Чтобы совсем исключить такую возможность, лучше разрабатывать самопроверяющиеся тесты, либо пользоваться инструментами тестирования, способными автоматически сверять ожидаемые и фактические результаты.

Иногда, например, при тестировании математического программного обеспечения, приходится допускать исключения. Математическое программное обеспечение обладает тем свойством, что выходные данные являются только приближением правильного результата. Это происходит из-за использования конечных вычислительных процессов вместо бесконечных математических процессов, из-за ошибок округления, связанных с конечной точностью машинной арифметики и неточностью представления чисел, а также ошибок из-за конечной точности представления входных данных и констант. Поэтому во многих случаях оказывается важной не абсолютная точность, а корреляция ошибок. Например, когда математическая программа

возвращает массив чисел, может оказаться важным, чтобы вычисленное решение было точным решением для набора входных данных, аппроксимирующего реальные выходные данные. Поэтому при тестировании математического программного обеспечения прогнозирование точных выходных данных затруднительно.

Тесты для неправильных и непредусмотренных входных данных следует разрабатывать так же тщательно, как для правильных и предусмотренных. При тестировании программ имеется естественная тенденция концентрировать внимание на правильных и предусмотренных входных условиях, а неправильным и непредусмотренным входным данным не придавать значения. Например, при тестировании задачи о треугольниках, лишь немногие смогут привести в качестве теста длины сторон 1, 2 и 5, чтобы убедиться в том, что треугольник не будет ошибочно интерпретирован как неравносторонний. Многие ошибки, которые потом неожиданно обнаруживаются в работающих программах, проявляются вследствие никак не предусмотренных действий пользователя программы. Тесты, представляющие неожиданные или неправильные входные данные, часто лучше обнаруживают ошибки, чем правильные тесты.

Необходимо проверять не только, делает ли программа то, для чего она предназначена, но и не делает ли она то, что не должна делать. Это логически вытекает из предыдущего принципа. Необходимо проверить программу на нежелательные побочные эффекты. Например, программа расчета зарплаты, которая производит правильные платежные чеки, окажется неверной, если она произведет лишние чеки для работающих или дважды запишет первую запись в список личного состава.

Детально изучать результаты каждого теста. Самые изощренные тесты ничего не стоят, если их результаты удостоиваются лишь беглого взгляда. Тестирование программы означает большее, нежели выполнение достаточного количества тестов; оно также предполагает изучение результатов каждого теста.

Не следует выбрасывать тесты, даже если программа уже не нужна. Эта проблема наиболее часто возникает при использовании интерактивных систем отладки. После внесения изменений или исправления ошибок необходимо повторять тестирование, тогда приходится заново изобретать тесты. Как правило, этого стараются избегать, поскольку повторное создание тестов требует значительной работы. В результате повторное тестирование бывает менее тщательным, чем первоначальное, т. е. если модификация затронула функциональную часть программы и при этом была допущена ошибка, то она зачастую может остаться необнаруженной.

Нельзя планировать тестирование в предположении, что ошибки не будут обнаружены. Такую ошибку обычно допускают руководители проекта, использующие неверное определение тестирования как процесса демонстрации отсутствия ошибок в программе, корректного функционирования программы.

По мере того как число ошибок, обнаруженных в некотором компоненте программного обеспечения, увеличивается, растет относительная вероятность существования в нем необнаруженных ошибок. Ошибки образуют кластеры, т. е. встречаются группами. С ростом числа ошибок, обнаруженных в компоненте программы (например, в модуле, подсистеме, функции пользователя), увеличивается также вероятность существования в этом компоненте еще не обнаруженных ошибок. Если при тестировании двух модулей в них обнаружены одна и восемь ошибок соответственно, кривая показывает, что для модуля с восемью ошибками вероятность того, что в нем еще есть ошибки, выше.

Тестирование, как почти всякая другая деятельность, должно начинаться с постановки целей. Цикл тестирования подобен полному циклу разработки программного обеспечения. Тесты должны быть спроектированы, реализованы, проверены и, наконец, выполнены. Поэтому задачи тестирования должны быть сформулированы на каждом его этапе, например, для каждого конкретного типа тестирования должны быть определены ориентиры (число пройденных путей, проверенных условных переходов и т. п.) и доля ошибок, которые должны быть обнаружены на этом этапе.

Тестирование — процесс творческий. Для тестирования большой программы требуется больший творческий потенциал, чем для ее проектирования [10].

7.3 Планирование тестирования

7.3.1 Вопросы, определяющие процесс планирования

Процесс тестирования находится в прямой зависимости от процесса разработки программного обеспечения, но при этом сильно отличается от него, поскольку преследует другие цели. Разработка ориентирована на построение программного продукта, тогда как тестирование отвечает на вопрос, соответствует ли разрабатываемый программный продукт требованиям, в которых зафиксирован первоначальный замысел изделия (т.е. то, что заказал заказчик).

Вместе оба процесса охватывают виды деятельности, необходимые для получения качественного продукта. Ошибки могут быть привнесены на каждой стадии разработки. Следовательно, каждому этапу разработки должен соответствовать этап тестирования. Отношения между этими процессами таковы, что если что-то разрабатывается, то оно подвергается тестированию, а результаты тестирования используются для определения, соответствует ли это "что-то" набору предъявляемых требований. Процесс тестирования возвращает выявленные им ошибки в процесс разработки. Процесс разработки передает процессу тестирования новые и исправленные проектные версии.

Как было отмечено выше, процесс тестирования тесно связан с процессом разработки. Соответственно планирование тестирования тоже зависит от выбранной модели разработки.

Однако вне зависимости от модели разработки при планировании тестирования необходимо ответить на пять вопросов, определяющих этот процесс:

- **кто будет тестировать и на каких этапах?** Разработчики продукта, независимая группа тестировщиков или совместно;
- **какие компоненты надо тестировать?** Будут ли подвергнуты тестированию все компоненты программного продукта или только компоненты, которые угрожают наибольшими потерями для всего проекта;
- **когда надо тестировать?** Будет ли это непрерывный процесс, вид деятельности, выполняемый в специальных контрольных точках, или вид деятельности, выполняемый на завершающей стадии разработки;
- **как надо тестировать?** Будет ли тестирование сосредоточено только на проверке того, что данный продукт должен выполнять, или также на том, как это реализовано;
- **в каком объеме тестировать?** Как определить, в достаточном ли объеме выполнено тестирование, или как распределить ограниченные ресурсы, выделенные под тестирование.

Все ответы на поставленные вопросы и многое другое фиксируется в документе – **Тест план**.

7.3.2 Тест план

Тест план (Test Plan) - это документ, описывающий весь объем работ по тестированию, начиная с описания объекта, стратегии, расписания, критериев начала и окончания тестирования, до необходимого в процессе работы оборудования, специальных знаний, а также оценки рисков с вариантами их разрешения.

Тест план содержит:

- 1) перечень тестовых ресурсов;
- 2) перечень функций и подсистем, подлежащих тестированию;
- 3) тестовую стратегию:
 - анализ функций и подсистем с целью определения слабых мест, требующих исчерпывающего тестирования, то есть участков функциональности, где появление дефектов наиболее вероятно;
 - определение стратегии выбора входных данных для тестирования. Поскольку в реальных применениях множество входных данных программного продукта практически бесконечно, выбор конечного подмножества для проведения тестирования является сложной задачей. Для ее решения могут быть применены методы покрытия классов входных и выходных данных, анализ крайних значений,

покрытие случаев использования и т.п. Выбранная стратегия должна быть обоснована и задокументирована;

- определение потребности автоматизации процесса тестирования. При этом решение об использовании существующей, либо о создании новой автоматизированной системы тестирования должно быть обосновано, а также продемонстрирована оценка затрат на создание новой системы или на внедрение уже существующей.

- 4) график (расписание) тестовых циклов;
- 5) указание конкретных параметров аппаратуры и программного окружения;
- 6) определение тестовых метрик, которые необходимо собирать и анализировать, таких как покрытие набора требований, покрытие кода, количество и уровень серьезности дефектов, объем тестового кода и т.п.

Тест план должен как минимум отвечать на следующие вопросы:

- 1) **что надо тестировать?** Описание объекта тестирования: системы, приложения, оборудование;
- 2) **что будете тестировать?** Список функций и описание тестируемой системы и её компонент в отдельности;
- 3) **как будете тестировать?** стратегия тестирования, а именно: методологии, виды тестирования и их применение по отношению к тестируемому объекту, приоритеты, автоматизация и пр.;
- 4) **когда будете тестировать?** Последовательность проведения работ: фазы, циклы тестирования, процедура тестирования - подготовка (Test Preparation), тестирование (Testing), анализ результатов (Test Result Analysis) в разрезе запланированных фаз разработки;
- 5) **где будете тестировать?**
 - окружение тестируемой системы – описание;
 - необходимое для тестирования оборудование и программные средства.
- 6) **кто будет тестировать?**
 - роли и обязанности;
 - имена и фамилии.
- 7) **критерии начала тестирования:**
 - готовность окружения;
 - готовность тест кейсов;
 - законченность разработки требуемого функционала;
 - выполнение юнит-тестов;

- билд построен и удовлетворяет определенным критериям.
- 8) **критерии окончания тестирования:**
 - результаты тестирования удовлетворяют определенным критериям;
 - требования к количеству открытых багов выполнены (определяются заранее).
- 9) **критерии передачи системы для приемочного тестирования:**
 - приемочные тесты – где хранятся и когда выполняются;
 - процедура приемки.
- 10) **какие риски существуют и как мы ими будем управлять?** Риски и их разрешение
- 11) **метрики и отчеты:**
 - продуктовые метрики – кто собирает, с какой периодичностью;
 - отчеты - кто создает, кому отправляет, и т.п.
- 12) **расписание билдов.**

Ответив в тест плане на вышеперечисленные вопросы, можно считать, что у нас на руках есть хороший черновик документа по планированию тестирования. Далее, чтобы документ приобрел более менее серьезный вид, предлагается дополнить его следующими пунктами:

- окружение тестируемой системы (описание программно-аппаратных средств);
- необходимое для тестирования оборудование и программные средства (тестовый стенд и его конфигурация, программы для автоматизированного тестирования и т.д.);
- риски и пути их разрешения.

7.3.3 Виды тест планов

Чаще всего на практике приходится сталкиваться со следующими видами тест планов:

- 1) **мастер тест план (Master Plan or Master Test Plan);**
- 2) **тест план (Test Plan);**
- 3) **план приемочных испытаний (Product Acceptance Plan)** - документ, описывающий набор действий, связанных с приемочным тестированием: стратегия, дата проведения, ответственные и т.д.;
- 4) **план автоматизации (Test Automation Plan)** - документ, описывающий набор действий, связанных с автоматизацией тестированием: стратегия, правила, ответственные и т.д.

Явное отличие Master Test Plan от просто Test Plan в том, что **мастер тест план является более статичным** в силу того, что содержит в себе высокоуровневую информацию, которая не подвержена частому изменению в процессе тестирования и пересмотра требований. Сам же **детальный тест план**, который содержит более конкретную информацию по стратегии, видам

тестировании, расписанию выполнения работ, является "живым" документом, который постоянно претерпевает изменения, отражающие реальное положение вещей на проекте.

В повседневной жизни на проекте может быть один Мастер Тест План и несколько детальных тест планов, описывающих отдельные модули одного приложения [11].

7.4 Тестовый случай (Test case). Виды. Структура.

Тестовый случай (Test Case) – это

- минимальный элемент тестирования (всего одна верификация\проверка);
- совокупность шагов, конкретных условий и параметров, необходимых для проверки реализации тестируемой функции или её части;
- описание определенных действий и условий, которые необходимы для того, чтобы выявить тот или иной баг.

Под тест кейсом понимается структура вида: **Action > Expected Result > Test Result**. В таблице 7.1 показан пример test case.

Таблица 7.1 - Пример тестового случая

Action	Expecred Result	Test Result (passed/failed/blocked)
Open page "Login"	Login page is opened	Passed

7.4.1 Виды Тестовых Случаев

Тест кейсы разделяются по ожидаемому результату на позитивные и негативные:

- **позитивный тест кейс** использует только корректные данные и проверяет, что приложение правильно выполнило вызываемую функцию;
- **негативный тест кейс** оперирует как корректными так и некорректными данными (минимум 1 некорректный параметр) и ставит целью проверку исключительных ситуаций (срабатывание валидаторов), а также проверяет, что вызываемая приложением функция не выполняется при срабатывании валидатора.

7.4.2 Структура Тестовых Случаев (Test Case Structure)

Каждый тест кейс должен состоять из трех частей. В таблице 7.2 показаны эти части.

Таблица 7.2 - Структура Test case

Pre conditions	Список действий, которые приводят систему к состоянию пригодному для проведения основной проверки. Либо список условий, выполнение которых говорит о том, что система находится в пригодном для проведения основного теста состоянии.
Test case description	Список действий, переводящих систему из одного состояния в другое, для получения результата, на основании которого можно сделать вывод о удовлетворении реализации, поставленным требованиям
Post conditions	Список действий, переводящих систему в первоначальное состояние (состояние до проведения теста - initial state)

Примечание: **Post Conditions** не является обязательной частью. Эта часть актуальна при автоматизированном тестировании, когда за один прогон можно наполнить базу данных сотней или даже тысячей некорректных документов.

Пример тест кейса

do A1, verify B1

do A2, verify B2

do A3, verify B3

В приведенном примере конечная проверка - B3. Это значит, что именно она является ключевой. Значит, A1 и A2 - это действия приводящие систему в тестопригодное состояние. А B1 и B2 - условия того, что система находится в состоянии пригодном для тестирования. Таким образом в таблице 7.3 показано условие тест кейса.

Таблица 7.3 - Условие тест кейса

Action	Expected Result	Test Result (passed/failed/blocked)
Preconditions		
do A1	verify B1	
do A2	verify B2	
Test Case Description		
do A3	verify B3	
Postconditions		

PostConditions в данном примере не были описаны, но по логике вещей надо выполнить шаги, которые бы вернули систему в первоначальное состояние. (например, удалили созданную запись, или отменили бы изменения сделанные в документе)

Нужно ответить на вопрос: "Почему данное разбиение удобно использовать?"

Ответ в таблице 7.4: конечная проверка одна, т.е. в случае если тест провален (**test failed**) будет сразу ясно из-за чего. Т.к. если провальными окажутся проверки B1 и/или B2, то тест кейс будет заблокирован (**test blocked**), из-за того, что функцию не возможно привести в тестопригодное состояние (состояние пригодное для проведения тестирования), но это не значит, что тестируемая функция не работает.

Таблица 7.4 - Один из вариантов результата

Action	Expected Result	Test Result (passed/failed/blocked)
Preconditions		
do A1	verify B1	passed
do A2	verify B2	<i>failed</i>
Test Case Description		
do A3	verify B3	blocked
Postconditions		

7.4.3 Детализация описания тест кейсов

Уровень детализации тест кейсов должен быть таков, чтобы обеспечивать разумное соотношение времени прохождения к тестовому покрытию. Т.е. до тех пор пока покрытие тестами определенного функционала не меняется, можно уменьшать детализацию тест кейсов. В таблице 7.5 можно увидеть пример детализации тест кейса:

Таблица 7.5 - Пример тест кейса 1

Проверка отображения страницы		
Действие	Ожидаемый результат	Результат теста
Открыть страницу "Вход в систему"	<ul style="list-style-type: none"> - окно "Вход в систему" открыто; - название окна - Вход в систему; - логотип компании отображается в правом верхнем углу; - на форме 2 поля - Имя и Пароль; - кнопка Вход доступна; - ссылка "забыл пароль" - доступна. 	...

Пример тест кейса 2:

Название: Проверка отображения страницы

Действие: Открыть страницу "Вход в систему"

Проверка: Проверьте, что отображаемая страница соответствует странице на картинке 1 (и прилагаем изображение страницы "Вход в систему")

В примере 1 и 2 покрытие будет одинаковым, но вот время, которое потребуется для прохождения, будет разным. Второй пример будет нагляднее.

7.4.4 Атрибуты тест кейса

В таблице 7.6 представлены часто используемые атрибуты тест кейса.

Таблица 7.6 - Атрибуты тест кейса

Атрибут тест кейса	Описание
Test Case ID	Уникальное значение в пределах не только документа, но и всей фирмы □
Test Case Priority	Приоритет. Измеряется от 1 до n 1 – самый высокий n – самый низкий (для не очень больших проектов рационально выбирать n=4) □
IDEA	Описание идеи, проверяемой тест кейсом
SETUP and ADDITIONAL INFO	Входные параметры □
Revision History	История редактирования. Пример формата: <i>Created on <date> by<name></i> <i>Modified on<date>by<name></i> <i>Change</i> – какие изменения и зачем они
Execution Part □	Выполнимая часть тест кейса. Пример формата: Action – список команд EXPECTED RESULT – ожидаемый результат TEST RESULT – (passed, failed, blocked)

Пример тест кейса: пусть для тестирования возможности оплаты услуги на сайте необходимо выполнить следующий алгоритм:

- 1) открыть vk.com;
- 2) ввести в поле “Имя пользователя” значение “user”;
- 3) ввести в поле “Пароль” значение “password”;
- 4) нажать кнопку “Войти”;
- 5) ввести в поле “Искать людей” значение “Петр Петров”;
- 6) нажать кнопку “Поиск”;

- 7) нажать на страницу с Петр Петров;
- 8) в открывшейся странице нажать на ссылку “Отправить подарок”;
- 9) в открывшейся странице нажать на любую ссылку с подарком;
- 10) в открывшейся странице нажать на кнопку “Подарить”;
- 11) в открывшейся странице нажать на ссылку “Банковские карты”;
- 12) ввести в поле “Номер карты” значение карты VISA “1111-1111-1111-1111”;
- 13) ввести в поле “Действительно до” значение “07/15”;
- 14) ввести в поле “CVC” значение “111”;
- 15) нажать кнопку “Оплатить”;
- 16) записать номер заказа;
- 17) запросить базу данных “select res from payment where id = <номер заказа>”.

Ожидаемый результат: “10”

В таблице 7.7 можно увидеть как для данного примера будет выглядеть тест кейс.

Преимущество такой структуры тест кейса в отличии от первоначального списка заключается в том, что есть возможность протестировать по тому же сценарию другие данные (например: user2, password2, Джулия Робертс, 2222-2222-2222-2222).

Для выполнения одного и того же сценария на N наборах тестовых данных создается N тестов.

Этому правилу необходимо следовать. Таким образом, чтобы сделать подарок Ивану Иванову, нужно скопировать содержимое тест кейса VV12345, например, в тест кейс VV12346 и переписать только входные параметры.

Такой вид тест кейса называется управляемый данными (data-driven).

Проблемы сценария в примере:

- пункты 1-4 могут меняться в связи с миграцией сайта на новый домен, изменением интерфейса и т.д.;
- поиск друга в пунктах 5-7 “Петр Петров” может привести в никуда при удалении страницы;
- пункты 8-15 могут быть легко изменены за счет нового дизайна сайта.

Следовательно, при любом таком изменении придется переписывать весь тест кейс, чтобы заново протестировать первоначальную задачу.

Таблица 7.7 - Тест кейс для примера

Test Case ID/Priority	VV12345	1
IDEA: Оплата картой VISA “подарка другу”		
SETUP and ADDITIONAL INFO:		
Аккаунт: user/password		
Имя друга: Петр Петров		
Номер карты: 1111-1111-1111-1111		
Срок действия: 07/15		
CVC: 111		
Запрос SQL: select res from payment where id = <номер заказа> <input type="checkbox"/>		
Revision History		
Created on 23/01/2014 by Иван Иванов	Новый тест кейс	
Execution Part		
ACTION	EXPECTED RESULT	TEST RESULT
1) открыть vk.com; 2) ввести в поле “Имя пользователя” значение “user”; 3) ввести в поле “Пароль” значение “password”; 4) нажать кнопку “Войти”; 5) ввести в поле “Искать людей” значение “Петр Петров”; 6) нажать кнопку “Поиск”; 7) нажать на страницу с Петр Петров; 8) в открывшейся странице нажать на ссылку “Отправить подарок”; 9) в открывшейся странице нажать на любую ссылку с подарком; 10) в открывшейся странице нажать на кнопку “Подарить”; 11) в открывшейся странице нажать на ссылку “Банковские карты”; 12) ввести в поле “Номер карты” значение карты VISA “1111-1111-1111-1111”; 13) ввести в поле “Действительно до” значение “07/15”; 14) ввести в поле “CVC” значение “111”; 15) нажать кнопку “Оплатить”; 16) записать номер заказа; 17) запросить базу данных “select res from payment where id = <номер заказа>”.	10	

Если же разбить задачу на несколько тест кейсов, например, за пункты 1-4 будет отвечать тест кейс “Вход в систему”, за пункты 5-7 “Поиск друга”, 8-15 – “Оплата подарка” (на внутреннем уровне каждого из них возможно еще более детальное разделение), то можно переписать тест кейс так, как указано в таблице 7.8.

Возможен вариант, когда все, что нужно – это выполнение команды номер 5, при условии что другие команды выполнены заранее. В таблице 7.9 указан упрощенный вариант тест кейса.

Таблица 7.8 - Упрощение тест кейса

Test Case ID/Priority	VV12347	1
IDEA: Оплата картой VISA “подарка другу” SETUP and ADDITIONAL INFO: Аккаунт: user/password Имя друга: Петр Петров Номер карты: 1111-1111-1111-1111 Срок действия: 07/15 CVC: 111 Запрос SQL: select res from payment where id = <номер заказа> □		
Revision History		
Created on 23/01/2014 by Марина Гончарова	Новый тест кейс	
Modified on 23/01/2014 by Иванов Евгений	Упрощение шагов	
Execution Part		
ACTION	EXPECTED RESULT	TEST RESULT
1) войти в систему; 2) найти друга; 3) платить подарок; 4) записать номер заказа; 5) запросить базу данных “select res from payment where id = <номер заказа>”	10	

Таблица 7.9 - Упрощение тест кейса

Test Case ID/Priority	VV12348	1
IDEA: Подтверждение оплаты SETUP and ADDITIONAL INFO: Номер заказа: параметр		
Revision History		
Created on 23/01/2014 by Марина Гончарова	Новый тест кейс	
Modified on 23/01/2014 by Иванов Евгений	Упрощение шагов	
Modified on 24/01/2014 by Сергей Галкин	Изменение структуры	
Execution Part		
ACTION	EXPECTED RESULT	TEST RESULT
Запросить базу данных “select res from payment where id = <номер заказа>”	10	

Неизвестным параметр < Номер заказа > после завершения выполнения теста получит свое значение, которое будет задокументировано [11].

8 ДЕФЕКТЫ. ПРИЧИНЫ, ОПИСАНИЕ, ОТСЛЕЖИВАНИЕ

Ошибками в программном обеспечении являются все возможные несоответствия между демонстрируемыми характеристиками его качества и сформулированными или подразумеваемыми требованиями и ожиданиями пользователей [8].

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

- **defect** — самое общее нарушение каких-либо требований или ожиданий, не обязательно проявляющееся вовне (к дефектам относятся нарушения стандартов кодирования, недостаточная гибкость системы и пр.);
- **failure** — наблюдаемое нарушение требований, проявляющееся при каком-то реальном сценарии работы ПО. Это можно назвать проявлением ошибки;
- **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;
- **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:

- **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;
- **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;

- **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.

8.1 Система отслеживания ошибок

Система отслеживания ошибок (*bug tracking system*) — прикладная программа, разработанная с целью помочь разработчикам программного обеспечения (программистам, тестировщикам и др.) учитывать и контролировать ошибки (баги), найденные в программах,

пожелания пользователей, а также следить за процессом устранения этих ошибок и выполнения или невыполнения пожеланий [5].

Главный компонент такой системы — база данных, содержащая сведения об обнаруженных дефектах. Эти сведения фиксируются в **отчете об ошибке**.

Отчет об ошибке (баг репорт) - это документ, описывающий ситуацию или последовательность действий приведшую к некорректной работе объекта тестирования, с указанием причин и ожидаемого результата.

8.2 Структура баг репорта

Разные системы отслеживания ошибок, предлагают разные поля для заполнения и разные структуры описания ошибок. В таблице 8.1 указан **шаблона баг репорта**.

Таблица 8.1 - Шаблон баг репорта

Шапка	
Короткое описание (Summary)	Короткое описание проблемы, явно указывающее на причину и тип ошибочной ситуации.
Проект (Project)	Название тестируемого проекта
Компонент приложения (Component)	Название части или функции тестируемого продукта
Номер версии (Version)	Версия на которой была найдена ошибка
Серьезность (Severity)	Наиболее распространена пятиуровневая система градации серьезности дефекта: <ul style="list-style-type: none"> – s1 Блокирующий (Blocker); – s2 Критический (Critical); – s3 Значительный (Major); – s4 Незначительный (Minor); – s5 Тривиальный (Trivial).
Приоритет (Priority)	Приоритет дефекта: <ul style="list-style-type: none"> – p1 Высокий (High); – p2 Средний (Medium); – p3 Низкий (Low). (подробнее в разделе Серьезность и приоритет ошибки)
Статус (Status)	Статус бага. Зависит от используемой процедуры и жизненного цикла бага (bug workflow and life cycle)
Автор (Author)	Создатель баг репорта
Назначен на (Assigned To)	Имя сотрудника, назначенного на решение проблемы
Окружение	
ОС / Сервис Пак и т.д. / Браузера + версия / ...	Информация об окружении, на котором был найден баг: операционная система, сервис пак, для WEB тестирования - имя и версия браузера и т.д.

Продолжение таблицы 8.1

Описание	
Шаги воспроизведения (Steps to Reproduce)	Шаги, по которым можно легко воспроизвести ситуацию, приведшую к ошибке.
Фактический Результат (Result)	Результат, полученный после прохождения шагов к воспроизведению
Ожидаемый результат (Expected Result)	Ожидаемый правильный результат
Дополнения	
Прикрепленный файл (Attachment)	Файл с логами, скриншот или любой другой документ, который может помочь прояснить причину ошибки или указать на способ решения проблемы

8.3 Серьезность и приоритет ошибки

Разные системы отслеживания ошибок предлагают разные пути описания серьезности и приоритета баг репорта, неизменным остается лишь смысл, вкладываемый в эти поля.

- **серьезность (Severity)** - это атрибут, характеризующий влияние ошибки на работоспособность приложения;
- **приоритет (Priority)** - это атрибут, указывающий на очередность выполнения задачи или устранения ошибки. Можно сказать, что это инструмент менеджера по планированию работ. Чем выше приоритет, тем быстрее нужно исправить дефект.

Градации серьезности дефекта (severity):

- **s1 Блокирующая (Blocker).** Блокирующая ошибка, приводящая приложение в нерабочее состояние, в результате которого дальнейшая работа с тестируемой системой или ее ключевыми функциями становится невозможна. Решение проблемы необходимо для дальнейшего функционирования системы;
- **s2 Критическая (Critical).** Критическая ошибка, неправильно работающая ключевая бизнес логика, дыра в системе безопасности, проблема, приведшая к временному падению сервера или приводящая в нерабочее состояние некоторую часть системы, без возможности решения проблемы, используя другие входные точки. Решение проблемы необходимо для дальнейшей работы с ключевыми функциями тестируемой системой;
- **s3 Значительная (Major).** Значительная ошибка, часть основной бизнес логики работает некорректно. Ошибка не критична или есть возможность для работы с тестируемой функцией, используя другие входные точки;
- **s4 Незначительная (Minor).** Незначительная ошибка, не нарушающая бизнес логику тестируемой части приложения, очевидная проблема пользовательского интерфейса.

- **s5 Тривиальная (Trivial).** Тривиальная ошибка, не касающаяся бизнес логики приложения, плохо воспроизводимая проблема, малозаметная посредством пользовательского интерфейса, проблема сторонних библиотек или сервисов, проблема, не оказывающая никакого влияния на общее качество продукта.

Градация приоритета ошибки (Priority):

- **p1 Высокий (High).** Ошибка должна быть исправлена как можно быстрее, т.к. ее наличие является критической для проекта;
- **p2 Средний (Medium).** Ошибка должна быть исправлена, ее наличие не является критичной, но требует обязательного решения;
- **p3 Низкий (Low).** Ошибка должна быть исправлена, ее наличие не является критичной, и не требует срочного решения.

Порядок исправления ошибок по их приоритетам: **High > Medium > Low.**

8.4 Написание баг репорта

Баг репорт - это технический документ и в связи с этим язык описания проблемы должен быть техническим. Должна использоваться правильная терминология при использовании названий элементов пользовательского интерфейса (editbox, lightbox, combobox, link, text area, button, menu, popup menu, title bar, system tray и т.д.), действий пользователя (click link, press the button, select menu item и т.д.) и полученных результатах (window is opened, error message is displayed, system crashed и т.д.).

8.4.1 Требования к обязательным полям баг репорта

Обязательными полями баг репорта являются: короткое описание (*Bug Summary*), серьезность (*Severity*), шаги к воспроизведению (*Steps to reproduce*), результат (*Actual Result*), ожидаемый результат (*Expected Result*). Ниже приведены требования и примеры по заполнению этих полей.

Короткое описание. В одном предложении нужно уместить смысл всего баг репорта, а именно: коротко и ясно, используя правильную терминологию сказать что и где не работает. Например:

- приложение зависает, при попытке сохранения текстового файла размером больше 50Мб;
- данные на форме "Профайл" не сохраняются после нажатия кнопки "Сохранить".

Серьезность. В двух словах можно отметить, что если проблема найдена в ключевой функциональности приложения и после ее возникновения приложение становится полностью недоступно, и дальнейшая работа с ним невозможна, то она блокирующая. Обычно все

блокирующие проблемы находятся во время первичной проверки новой версии продукта (**Build Verification Test, Smoke Test**), т.к. их наличие не позволяет полноценно проводить тестирование. Если же тестирование может быть продолжено, то серьезность данного дефекта будет критическая. На счет значительных, незначительных и тривиальных ошибок вопрос достаточно прозрачный и зависит от ситуации.

Шаги к воспроизведению/Результат/Ожидаемый результат. Очень важно четко описать все шаги, с упоминанием всех вводимых данных (имени пользователя, данных для заполнения формы) и промежуточных результатов.

Например:

1) войдите в системы: Пользователь Тестер1, пароль xxxXXX:

— вход в систему осуществлен.

2) кликните линк Профайл:

— страница Профайл открылась.

3) введите Новое имя пользователя: Тестер2;

4) нажмите кнопку Сохранить.

Результат. На экране появилась ошибка. Новое имя пользователя не было сохранено

Ожидаемый результат. Страница профайл перегрузилась. Новое значение имени пользователя сохранено.

8.4.2 Основные ошибки при написании баг репортов

Недостаточность предоставленных данных. Не всегда одна и та же проблема проявляется при всех вводимых значениях и под любым вошедшим в систему пользователем, поэтому настоятельно рекомендуется вносить все необходимые данные в баг репорт.

Определение серьезности. Очень часто происходит либо завышение, либо занижение серьезности дефекта, что может привести к неправильной очередности при решении проблемы.

Язык описания. Часто при описании проблемы используются неправильная терминология или сложные речевые обороты, которые могут ввести в заблуждение человека, ответственного за решение проблемы.

Отсутствие ожидаемого результата. В случаях, если вы не указали, что же должно быть требуемым поведением системы, вы тратите время разработчика, на поиск данной информации, тем самым замедляете исправления дефекта. Вы должны указать пункт в требованиях, написанный тест кейс или же ваше личное мнение, если эта ситуация не была документирована.

8.4.3 Заполнение полей баг репорта

В таблице 8.2 представлены основные поля баг репорта и роль работника, ответственного за заполнение данного поля. Жирным курсивом выделены обязательные для заполнения поля.

Таблица 8.2 - Заполнение полей баг репорта

Поле	Ответственный за заполнение поля
<i>Короткое описание (Summary)</i>	Автор баг репорта (обычно это Тестировщик)
Проект (Project)	Автор баг репорта (обычно это Тестировщик)
Компонент приложения (Component)	Автор баг репорта (обычно это Тестировщик)
Номер версии (Version)	Автор баг репорта (обычно это Тестировщик)
<i>Серьезность (Severity)</i>	Автор баг репорта (обычно это Тестировщик), однако данный атрибут может быть изменен вышестоящим менеджером
Приоритет (Priority)	Менеджер проекта или менеджер ответственный за разработку компонента, на который написан баг репорт
Статус (Status)	Автор баг репорта (обычно это Тестировщик), но многие системы баг трекинга выставляют статус по умолчанию
Автор (Author)	Устанавливается по умолчанию, если нет, то указывается имя автора баг репорта
Назначен на (Assigned To)	Менеджер проекта или менеджер ответственный за разработку компонента, на который написан баг репорт
ОС / Сервис Пак и т.д. / Браузера + версия / ...	Автор баг репорта (обычно это Тестировщик)
<i>Шаги воспроизведения (Steps to Reproduce)</i>	Автор баг репорта (обычно это Тестировщик)
<i>Фактический Результат (Result)</i>	Автор баг репорта (обычно это Тестировщик)
<i>Ожидаемый результат (Expected Result)</i>	Автор баг репорта (обычно это Тестировщик)
Прикрепленный файл (Attachment)	Автор баг репорта (обычно это Тестировщик), а также любой член командной группы, считающий, что прикрепленные данные помогут в исправлении бага

8.4.4 Жизненный цикл бага

Рисунок 8.1 иллюстрирует жизненный цикл дефекта, принятый во многих крупных компаниях. Баг может находиться в одном из представленных на рисунке состояний.

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.

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

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

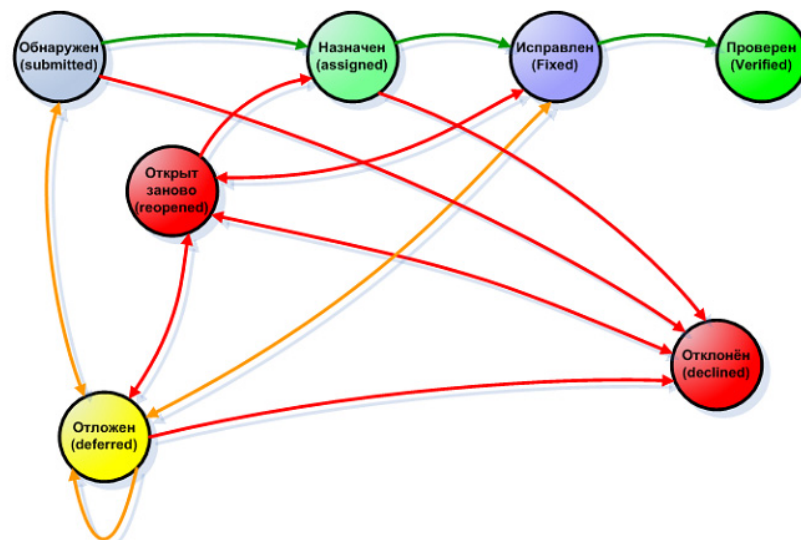


Рисунок 8.1 - Жизненный цикл бага

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*.

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

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).

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*.

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

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)*.

9 ТЕХНИКИ СОЗДАНИЯ ТЕСТОВ ДЛЯ ЧЕРНОГО ЯЩИКА

Целью тестирования ставится выяснение обстоятельств, в которых поведение программы не соответствует спецификации. Для обнаружения всех ошибок в программе необходимо выполнить *исчерпывающее тестирование*, то есть тестирование на всевозможных наборах данных. Для большинства программ такое невозможно, поэтому применяют *разумное тестирование*, при котором тестирование программы ограничивается небольшим подмножеством всевозможных наборов данных. При этом необходимо выбирать наиболее подходящие подмножества, подмножества с наивысшей вероятностью обнаружения ошибок [6].

Свойства правильно выбранного теста:

- уменьшает более, чем на одно число других тестов, которые должны быть разработаны для разумного тестирования;
- покрывает значительную часть других возможных тестов, что в некоторой степени свидетельствует о наличии или отсутствии ошибки до и после ограниченного множества тестов.

Техники черного ящика:

- эквивалентное разбиение;
- анализ граничных значений;
- анализ причинно-следственных связей;
- попарное тестирование;
- предположение об ошибке.

9.1 Эквивалентное разбиение

Основу метода эквивалентное разбиение составляют два положения:

- исходные данные необходимо разбить на конечное число классов эквивалентности. В одном классе эквивалентности содержатся такие тесты, что, если один тест из класса эквивалентности обнаруживает некоторую ошибку, то и любой другой тест из этого класса эквивалентности должен обнаруживать эту же ошибку;
- каждый тест должен включать, по возможности, максимальное количество классов эквивалентности, чтобы минимизировать общее число тестов.

Разработка тестов этим методом осуществляется в два этапа: выделение классов эквивалентности и построение теста.

Эквивалентный класс — это одно или больше значений ввода, к которым ПО применяет одинаковую логику.

Классы эквивалентности выделяются путём выбора каждого входного условия, которые берутся с помощью технического задания или спецификации и разбиваются на две и более группы.

Выделение классов эквивалентности является эвристическим способом, однако существует ряд правил:

- если входное условие описывает область значений, например «Целое число принимает значение от 0 до 999», то существует один правильный класс эквивалентности и два неправильных;
- если входное условие описывает число значений, например «Число строк во входном файле лежит в интервале (1..6)», то также существует один правильный класс и два неправильных;
- если входное условие описывает множество входных значений, то определяется количество правильных классов, равное количеству элементов в множестве входных значений. Если входное условие описывает ситуацию «должно быть», например «Первый символ должен быть заглавным», тогда один класс правильный и один неправильный;
- если есть основание считать, что элементы внутри одного класса эквивалентности могут программой трактоваться по-разному, необходимо разбить данный класс на подклассы. На этом шаге тестирующий на основе таблицы должен составить тесты, покрывающие собой все правильные и неправильные классы эквивалентности. При этом составитель должен минимизировать общее число тестов.

Несколько тестов эквивалентны, если:

- они направлены на поиск одной и той же ошибки;
- если один из тестов обнаруживает ошибку, другие ее тоже, скорее всего, обнаружат;
- если один из тестов не обнаруживает ошибку, другие ее тоже, скорее всего, не обнаружат;
- тесты используют один и те же наборы входных данных;
- для выполнения тестов приходится совершать одни и те же операции;
- тесты генерируют одинаковые выходные данные или приводят приложение в одно и то же состояние;
- все тесты приводят к срабатыванию одного и того же блока обработки ошибок ("error handling block");
- ни один из тестов не приводит к срабатыванию блока обработки ошибок ("error handling block").

Определение тестов:

- каждому классу эквивалентности присваивается уникальный номер;
- если еще остались не включенные в тесты правильные классы, то пишутся тесты, которые покрывают максимально возможное количество классов;
- если остались не включенные в тесты неправильные классы, то пишут тесты, которые покрывают только один класс.

9.2 Анализ граничных значений

Граничные условия — это ситуации, возникающие на высших и нижних границах входных классов эквивалентности [5].

Анализ граничных значений отличается от эквивалентного разбиения следующим:

- выбор любого элемента в классе эквивалентности в качестве представительного осуществляется таким образом, чтобы проверить тестом каждую границу этого класса;
- при разработке тестов рассматриваются не только входные значения (пространство входов), но и выходные (пространство выходов).

Метод требует определённой степени творчества и специализации в рассматриваемой задаче.

Существует несколько правил:

- 1) построить тесты с неправильными входными данными для ситуации незначительного выхода за границы области значений. Если входные значения должны быть в интервале $[-1.0 .. +1.0]$, нужно проверить -1.0 , 1.0 , -1.000001 , 1.000001 ;
- 2) обязательно писать тесты для минимальной и максимальной границы диапазона;
- 3) использовать первые два правила для каждого из входных значений (использовать пункт 2 для всех выходных значений);
- 4) если вход и выход программы представляет упорядоченное множество, сосредоточить внимание на первом и последнем элементах списка.

Анализ граничных значений, если он применён правильно, позволяет обнаружить большое число ошибок. Однако определение этих границ для каждой задачи может являться отдельной трудной задачей. Также этот метод не проверяет комбинации входных значений.

9.3 Анализ причинно-следственных связей

Анализ причинно-следственных связей позволяет системно выбирать высокорезультативные тесты. Метод использует алгебру логики и оперирует понятиями «причина» и «следствие». *Причиной* в данном случае называют отдельное входное условие или класс эквивалентности. *Следствием* - выходное условие или преобразование системы.

Идея метода заключается в отнесении всех следствий к причинам, т. е. в уточнении причинно-следственных связей. Данный метод дает полезный побочный эффект, позволяя обнаруживать неполноту и неоднозначность исходных спецификаций.

Построение тестов осуществляют в несколько этапов. Сначала, поскольку таблицы причинно-следственных связей при применении метода к большим спецификациям становятся громоздкими, спецификации разбивают на «рабочие» участки, стараясь по возможности выделять в отдельные таблицы независимые группы причинно-следственных связей. Затем в спецификации определяют множество причин и следствий. Далее на основе анализа семантического (смыслового) содержания спецификации строят таблицу истинности, в которой каждой возможной комбинации причин ставится в соответствие следствие. При этом целесообразно истину обозначать «I», ложь - «O», а для обозначения безразличных состояний условий применять обозначение «X», которое предполагает произвольное значение условия (0 или 1). Таблицу сопровождают примечаниями, задающими ограничения и описывающими комбинации причин и/или следствий, которые являются невозможными из-за синтаксических или внешних ограничений. При необходимости аналогично строится таблица истинности для класса эквивалентности. И, наконец, каждую строку таблицы преобразуют в тест. При этом рекомендуется по возможности совмещать тесты из независимых таблиц.

Данный метод позволяет строить высокорезультативные тесты и обнаруживать неполноту и неоднозначность исходных.

9.4 Парное тестирование

Pairwise testing - это техника формирования наборов тестовых данных, в которых каждое тестируемое значения каждого из проверяемых параметров хотя бы единожды сочетается с каждым тестируемым значением всех остальных проверяемых параметров.

Для демонстрации данной техники возьмем в качестве примера абстрактную систему с большим числом параметров, влияющих на её работу, которую нужно протестировать. Опытный тестировщик знает, что для проверки всех комбинации не хватит времени. К примеру, для проверки всех сочетаний 10 параметров с 10 значениями каждый, потребуется 10 000 000 000 тестов, в то время как метод перебора пар позволяет реализовать сравнимое по качеству тестирование (учитывая количество и критичность найденных в результате багов) используя всего 177 тестов.

Метод парного тестирования основан на довольно простой, но от того не менее эффективной идее, что подавляющее большинство ошибок выявляется тестом, проверяющим один параметр, либо сочетание двух. Ошибки, причиной которых явились комбинации трех и более параметров

как правило значительно менее критичны, чем пары параметров или одного. Если этого требует ситуация, то можно дополнить тестовое покрытие кейсами на желаемые комбинации параметров.

Перебрать все пары не сложно, трудность заключается в том, чтобы обеспечить при этом минимум тестов, комбинируя проверки нескольких пар в одном тесте. Математические методы могут обеспечить такой необходимый минимум тестов. Одним из таких методов являются **ортогональные матрицы**.

Для рассмотрения того как происходит оптимизация, возьмем для примера таблицу 9.1 параметров и значений.

Таблица 9.1 - Параметры и их значения

Параметр 1	Параметр 2	Параметр 3
Значение 1.1	Значение 2.1	Значение 3.1
Значение 1.2	Значение 2.2	Значение 3.2

Переберем значения первого параметра со вторым (строки №1-4), первого с третьим (строки №5-8) и второго с третьим (строки №9-12). Результат показан в таблице 9.2. Удалив повторяющиеся наборы параметров (выделены серым), получим следующий результат, показанный в таблице 9.3.

Таблица 9.2 - Результат перебора значений

#	Параметр 1	Параметр 2	Параметр 3
1	Значение 1.1	Значение 2.1	Значение 3.1
2	Значение 1.1	Значение 2.2	Значение 3.1
3	Значение 1.2	Значение 2.1	Значение 3.1
4	Значение 1.2	Значение 2.2	Значение 3.1
5	Значение 1.1	Значение 2.1	Значение 3.1
6	Значение 1.1	Значение 2.1	Значение 3.2
7	Значение 1.2	Значение 2.1	Значение 3.1
8	Значение 1.2	Значение 2.1	Значение 3.2
9	Значение 1.1	Значение 2.1	Значение 3.1
10	Значение 1.1	Значение 2.1	Значение 3.1
11	Значение 1.1	Значение 2.2	Значение 3.1
12	Значение 1.1	Значение 2.2	Значение 3.2

Таблица 9.3 - Сокращенный результат перебора

#	Параметр 1	Параметр 2	Параметр 3
1	Значение 1.1	Значение 2.1	Значение 3.1
2	Значение 1.1	Значение 2.2	Значение 3.1
3	Значение 1.2	Значение 2.1	Значение 3.1
4	Значение 1.2	Значение 2.2	Значение 3.1
5	Значение 1.1	Значение 2.1	Значение 3.2
6	Значение 1.2	Значение 2.1	Значение 3.2
7	Значение 1.1	Значение 2.2	Значение 3.2

Зеленым выделены уникальные пары в таблице. Значения выделенные белым не являются необходимыми для перебора всех пар в таблице, поэтому могут быть заменены на любое другое значение. Заменяв их можно оптимизировать тесты, добавив проверку пар из 5, 6 и 7 строк во вторую и третью строки. В таблице 9.4 показан результат оптимизации.

Таблица 9.4 - Результат оптимизации

#	Параметр 1	Параметр 2	Параметр 3
1	Значение 1.1	Значение 2.1	Значение 3.1
2	Значение 1.1	Значение 2.2	Значение 3.2
3	Значение 1.2	Значение 2.1	Значение 3.2
4	Значение 1.2	Значение 2.2	Значение 3.1

Оптимизация даже такого малого набора параметров не так проста. При этом сложность задачи возрастает пропорционально росту числа параметров.

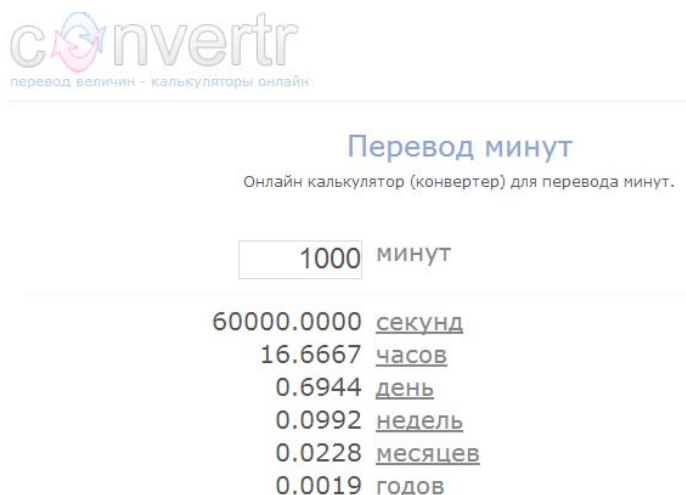
9.5 Предположение об ошибке

Программист с большим опытом выискивает ошибки без всяких методов, но при этом он подсознательно использует метод предположения об ошибке. Данный метод в значительной степени основан на интуиции. Основная идея метода состоит в том, чтобы составить список, который перечисляет возможные ошибки и ситуации, в которых эти ошибки могли проявиться. Потом на основе списка составляются тесты.

10 ПРИМЕНЕНИЕ ТЕХНИК

10.1 Эквивалентное разбиение и граничное условие

Применение данных методов будет рассмотрено на примере приложения, которое на вход принимает строку, но по смыслу данная строка должна интерпретироваться как число. На рисунке 10.1 показано это приложение, которое является онлайн-калькулятором для перевода единиц времени.



converttr
перевод величин - калькуляторы онлайн

Перевод минут

Онлайн калькулятор (конвертер) для перевода минут.

1000 минут

60000.0000 секунд
16.6667 часов
0.6944 день
0.0992 недель
0.0228 месяцев
0.0019 годов

Рисунок 10.1 - Онлайн-калькулятор

Все возможные строки можно разделить на два больших класса – “числа” и “не числа”. Но данным классам можно задать другие, более длинные, но при этом более точные названия:

- множество строк, которые программа интерпретирует как числа;
- множество строк, которые программа не может интерпретировать как числа.

В данной формулировке становится ясно, что программа на вход получает строку, которая по каким-то правилам преобразуется в число. Если это преобразование прошло успешно – полученное число используется в вычислениях. А если преобразовать строку в число не удалось – получаем информацию об этом либо в виде сообщения о возникшей проблеме, либо в виде бессмысленного результата вычислений.

Можно определить, как именно преобразователь времени ведёт себя в той и в другой ситуации, для этого достаточно подать на вход какое-нибудь значение, которое в должно интерпретироваться как число, например, “5”. На рисунке 10.2 изображен ввод входящего значения.

5 минут

300.0000 секунд
0.0833 часов
0.0035 дней
0.0005 недель
0.0001 месяцев
0.0000 годов

Рисунок 10.2 - Входящее значение 5 минут

А также значение, которое точно не является числом, например, "test". На рисунке 10.3 изображен ввод входящего нечислового значения.

test минут

NaN секунд
NaN часов
NaN дней
NaN недель
NaN месяцев
NaN годов

Рисунок 10.3 - Входящее значение не число

Наблюдается случай, когда невалидное входное значение приводит к бессмысленному результату (NaN означает “Not a Number”).

Нужно определить, какие строки программа будет интерпретировать как числа, то есть выделить в больших классах подклассы меньшего размера, но зато описанные конструктивно.

Строка, представляющая собой последовательность цифр, интерпретируется как целое число.

Требуется понять, какой длины последовательность цифр интерпретируется как число. Чтобы это сделать, необходимо применить технику разбиения на подобласти. Минимальная длина последовательности – ноль. Максимальная длина – "максимально возможная".

В данном приложении не указано никаких ограничений на размер поля ввода. Есть несколько ситуаций:

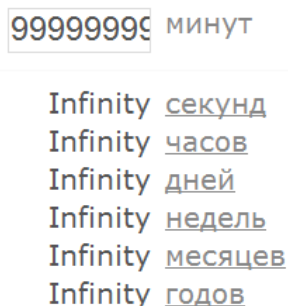
- возможно браузер накладывает какое-нибудь ограничение, однако тестировщику про это ничего не известно, если оно есть – в разных браузерах оно разное;

- если введённые данные передаются на сервер в виде GET-запроса, возможно, имеется ограничение на длину запроса – согласно стандарту RFC 2068 они должны поддерживать не менее 255 байтов, но все реально способны обрабатывать запросы большей длины, и это зависит от браузера и от веб-сервера.

Конвертер, который используется в качестве примера, реализован на языке JavaScript, на сервер никаких данных не отправляется, все вычисления производятся внутри браузера. Google Chrome успешно справился со строкой, состоящей из 10 000 000 девяток, строку из 100 000 000 девяток он обработать уже не может – появляется ошибка с предложением перезагрузить страницу. Следовательно, между этими значениями и находится максимальная длина. Поэтому нужно уточнить подкласс:

Строка, представляющая собой последовательность цифр, интерпретируется как целое число, если длина строки не превышает некоторое Максимальное Значение.

На рисунке 10.4 изображена ситуация на существенно более коротких последовательностях.

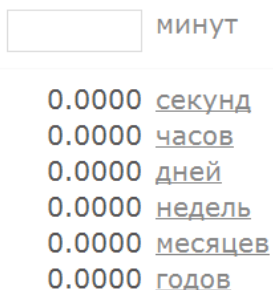


99999999 минут
Infinity секунд
Infinity часов
Infinity дней
Infinity недель
Infinity месяцев
Infinity годов

Рисунок 10.4 - Последовательность из 1000 девяток

При вычислениях возникло переполнение, однако Infinity – это не NaN, то есть согласно описанном выше считается, что такая последовательность (а также и более длинные последовательности цифр) может считаться числом.

Последовательность нулевой длины – это пустая строка. Приложение интерпретирует пустую строку как число ноль. На рисунке 10.5 изображена данная интерпретация.



минут
0.0000 секунд
0.0000 часов
0.0000 дней
0.0000 недель
0.0000 месяцев
0.0000 годов

Рисунок 10.5 - Последовательность нулевой длины

Первый подкласс, последовательность цифр, создан. Второй подкласс - последовательность не цифр. "Не цифрами" могут являться пробелы в начале и в конце, а также ведущие нули. Они обрезаются, а оставшаяся строка интерпретируется как число. Поэтому:

Строка, интерпретируемая как число, также интерпретируется как число, если добавить в начале некоторое количество нулей, при этом ведущие нули игнорируются,

Строка, интерпретируемая как число, также интерпретируется как число, если добавить в начале или в конце некоторое количество пробелов, при этом все пробелы игнорируются.

Во-первых, важно не забывать про максимальное значение длины, если добавить слишком много нулей или пробелов, строка перестанет быть числом, даже если эти пробелы добавлялись к небольшому числу. Во-вторых, добавлять сначала нули, а потом пробелы можно, а наоборот нельзя. На рисунке 10.6 изображена данная ситуация.

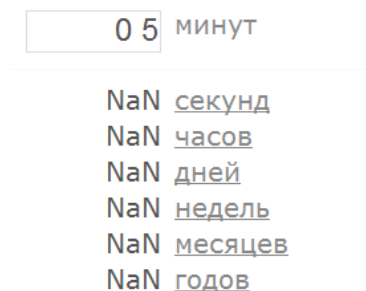


Рисунок 10.6 - Значение с пробелом

На рисунке 10.7 изображен следующий подкласс, который будет являться отрицательным числом.

Строка, интерпретируемая как число, также интерпретируется как число, если добавить в начало знак минус или плюс.

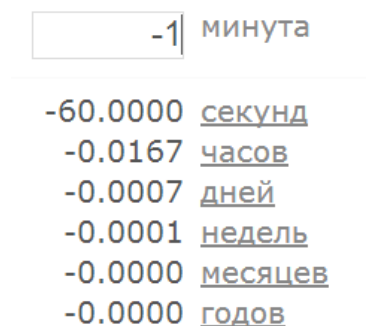


Рисунок 10.7 - Отрицательное значение

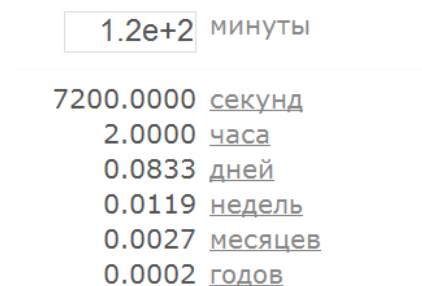
Новый подкласс - нецелое число:

Строка, состоящая из двух неразрывных цепочек цифр, разделённых десятичной точкой, интерпретируется как число.

Для нецелых чисел можно применить технику разбиения на подобласти к количеству значащих цифр, или к количеству знаков после запятой, в зависимости от того, как интерпретируется понятие точности в конкретном приложении. Но при этом следует отметить, что для чисел с плавающей точкой техника разбиения на подобласти работает плохо т.к. JavaScript реализует стандарт IEEE-754.

На рисунке 10.8 изображен следующий подкласс, который будет представлять собой не последовательность цифр, но интерпретироваться как число, например, число 120.

Строка, состоящая из числа, за которым следует символ 'e', за которым следует целое число, интерпретируется как число.



1.2e+2 минуты

7200.0000 секунд
2.0000 часа
0.0833 дней
0.0119 недель
0.0027 месяцев
0.0002 годов

Рисунок 10.8 - Число с символом 'e'

Строка, состоящая из символов '0x', за которыми следует неразрывная последовательность шестнадцатеричных цифр, интерпретируется как шестнадцатеричное целое число, которое изображено на рисунке 10.9



0xba минут

11160.0000 секунд
3.1000 часа
0.1292 дней
0.0185 недель
0.0042 месяцев
0.0004 годов

Рисунок 10.9 - Шестнадцатеричное целое число

Получены следующие подклассы, по которым можно начинать тестировать приложение:

- 1) строка, представляющая собой последовательность цифр, интерпретируется как целое число, если длина строки не превышает некоторое Максимальное Значение;
- 2) строка, интерпретируемая как число, также интерпретируется как число, если добавить в начале некоторое количество нулей, при этом ведущие нули игнорируются;
- 3) строка, интерпретируемая как число, также интерпретируется как число, если добавить в начале или в конце некоторое количество пробелов, при этом все пробелы игнорируются;
- 4) строка, интерпретируемая как число, также интерпретируется как число, если добавить в начало знак минус или плюс;
- 5) строка, состоящая из двух неразрывных цепочек цифр, разделённых десятичной точкой, интерпретируется как число;

6) строка, состоящая из числа, за которым следует символ 'e', за которым следует целое число, интерпретируется как число;

7) строка, состоящая из символов '0x', за которыми следует неразрывная последовательность шестнадцатеричных цифр, интерпретируется как шестнадцатеричное целое число.

10.2 Попарное тестирование

Метод эффективен на поздних этапах разработки, либо дополненный основными функциональными тестами. Если проводится конфигурационное тестирование, то прежде чем использовать попарное тестирование следует убедиться, что основной сценарий функционирует на всех операционных системах с параметрами по умолчанию (build verification test). Это значительно облегчит локализацию будущих багов, ведь при парном тестировании в одном тесте фигурирует множество параметров со значениями не по умолчанию, каждый из которых может стать причиной сбоя и его локализация в этом случае весьма затруднительна. В случае провала BVT следует отказаться от использования метода парного тестирования, так как многие тесты будут провальными, а исключение даже одного теста влечет за собой потерю как правило нескольких пар и смысл использования метода теряется. Поэтому метод следует использовать лишь на стабильном функционале, когда текущие тесты уже теряют свою эффективность.

Для того чтобы воспользоваться методом необходимо выполнить несколько простых шагов:

Определиться с функциональностью, которую нужно проверить

Для этого необходимо разделить функциональность на части: компоненты, функции, сценарии. Функциональность небольшой программы, например по записи дисков, упрощенно можно представить в виде всего двух сценариев: запись диска, стирание диска. Рассмотрим на примере "запись диска" и перейдем к следующему шагу.

Исследовать выбранный сценарий и выявить его параметры и их значения

На данном этапе нужно выявить параметры сценария, который могут повлиять на его работу. В качестве параметров могут выступать как настройки самой программы, так и внешние факторы.

Упрощенно, параметры и их значения при записи диска можно представить в виде таблицы 10.1.

Таблица 10.1 - Параметры их значения при записи диска

Тип носителя	Скорость записи	Файловая система	Мультисессия	Объем данных
CD	2	ISO	Нет	100 Mb
DVD	4	UDF	Начать	700 Mb
	8	UDF/ISO	Продолжить	4,7 Gb
	16			
	24			

Продолжение таблицы 10.1

Тип носителя	Скорость записи	Файловая система	Мультисессия	Объем данных
	36			
	52			

Если внимательно посмотреть на таблицу, то можно обратить внимание, что параметр «Скорость записи» имеет значения, недопустимые для DVD. Для этого нужно разделить таблицу на две: таблица 10.2 и таблица 10.3. Стоит учитывать, что на практике параметров в этом сценарии гораздо больше, и несостыковок было бы значительно больше.

Таблица 10.2 - Параметры и их значения для CD

Скорость записи	Файловая система	Мультисессия	Объем данных
2	ISO	Нет	100 Mb
4	UDF	Начать	700 Mb
8	UDF/ISO	Продолжить	
16			
24			
36			
52			

Таблица 10.3 - Параметры и их значения для DVD

Скорость записи	Файловая система	Мультисессия	Объем данных
2	ISO	Нет	100 Mb
4	UDF	Начать	4,7 Gb
8	UDF/ISO	Продолжить	
16			
24			

При выборе параметров и значений нужно помнить, что негативные тесты не стоит включать в таблицу – в одном тесте может проверяться несколько пар, а в случае негативного теста будет выполнена проверка лишь одного параметра, в результате некоторые пары могут остаться непроверенными. По этой причине в данном примере отсутствуют значения объёма данных, равные нулю и превышающие объем диска. Если их добавить, то в результате использования метода может получиться кейс в котором на нулевом объёме данных будет проверяться к примеру пара Файловой системы ISO и начала мультисессии. В результате, успешно убедившись в корректной обработке попытки записи пустого диска, мы упустим проверку пары ISO-начать мультисессию.

Если будет предпринята попытка инкрементально дополнить тестовое покрытие – количество тестов увеличиться несопоставимо больше. Забегая вперед, для данного примера с записью DVD методом перебора пар получится 17 тестов. Решив дополнить исходную таблицу всего одним значением, например скоростью записи DVD 32x, увеличится общее число тестов на 8, так как в стремление сохранить целостность метода придется вынуждено перебирать это значение со всеми значениями других параметров. Целесообразно дополнить тестовое покрытие одним-двумя тестами на проверку нового значения составленными вручную, либо заново создать таблицу, как это описано в следующем шаге.


Применить алгоритм, составляющий оптимальное число тестов с полным перебором пар.

Составлять тесты по методу парного тестирования без использования технических средств крайне сложно, поэтому чтобы упростить себе жизнь, следует воспользоваться программными решениями. В качестве примера будет использоваться *Allpairs*.

Allpairs - программа, подбирающая уникальные пары для входящего набора данных. Работает из командной строки.

В качестве входных данных для программы используется .txt файл с таблицей параметров, столбцы которой разделены табуляцией. Для создания такого файла удобнее всего использовать MS Excel в котором есть возможность сохранять “текстовые файлы с разделителями табуляции (*.txt)”.

Имея исходный файл, необходимо запустить консоль и набрать там строку, которая изображена на рисунке 10.10.



```
c:\pairs>allpairs.exe test.txt > re.txt
```

Рисунок 10.10 - Запуск Allpairs

Где:

- c:\pairs\allpairs.exe - полный путь к приложению allpairs.exe;
- test.txt – путь к исходному файлу с таблицей параметров;
- re.txt - путь и имя файла, который будет создан в результате работы программы.

Результирующий файл будет содержать в себе готовый перечень проверок. В результате, взяв таблицу с параметрами для DVD, получается следующий перечень тестов, указанный в таблице 10.4.

«~» означает, что вместо указанного значения может быть использовано любое, так как оно не составляет пары в данном тесте.

Таблица 10.4 - Результат Allpairs для DVD

#	Скорость записи	Файловая система	Мультисессия	Объем данных
1	2	ISO	Нет	100 Mb
2	2	UDF	Начать	4,7 Gb
3	4	ISO	Начать	100 Mb
4	4	UDF	Нет	4,7 Gb
5	8	ISO	Продолжить	4,7 Gb
6	8	UDF	Продолжить	100 Mb
7	16	UDF/ISO	Нет	100 Mb
8	16	UDF/ISO	Начать	4,7 Gb
9	24	UDF/ISO	Продолжить	100 Mb
10	24	ISO	Нет	4,7 Gb
11	2	UDF/ISO	Продолжить	~4,7 Gb
12	4	UDF/ISO	Продолжить	~100 Mb
13	8	UDF/ISO	Начать	~100 Mb
14	16	UDF	Продолжить	~100 Mb
15	24	UDF	Начать	~100 Mb
16	8	~ISO	Нет	~4,7 Gb
17	16	ISO	~Начать	~4,7 Gb

Если перебирать все пары последовательно: все значения скорости записи со всеми файловыми системами, плюс все значения скорости записи со всеми значениями мультисессии, и так далее до полного перебора всех пар значений получился 61 теста. Это наглядно демонстрирует возможности оптимизации

Получив такую таблицу тестировщик имеет практически готовые тесты: он знает сценарий, а также значения параметров, которые задаются при его прохождении. Поэтому уже в текущем виде тесты пригодны к выполнению.

В таблице 10.5 содержится результат для таблицы с параметрами для CD.

Таблица 10.5 - Результат Allpairs для CD

#	Скорость записи	Файловая система	Мультисессия	Объем данных
1	2	ISO	Нет	100 Mb
2	2	UDF	Начать	700 Mb
3	4	ISO	Начать	100 Mb
4	4	UDF	Нет	700 Mb
5	8	ISO	Продолжить	700 Mb
6	8	UDF	Продолжить	100 Mb
7	16	UDF/ISO	Нет	100 Mb
8	16	UDF/ISO	Начать	700 Mb
9	24	UDF/ISO	Продолжить	100 Mb
10	24	ISO	Нет	700 Mb
11	36	UDF	Начать	100 Mb
12	36	UDF/ISO	Продолжить	700 Mb
13	52	ISO	Начать	100 Mb
14	52	UDF	Нет	700 Mb
15	2	UDF/ISO	Продолжить	~100 Mb
16	4	UDF/ISO	Продолжить	~700 Mb
17	8	UDF/ISO	Нет	~100 Mb
18	16	ISO	Продолжить	~700 Mb
19	24	UDF	Начать	~100 Mb
20	36	ISO	Нет	~100 Mb
21	52	UDF/ISO	Продолжить	~100 Mb
22	8	~UDF	Начать	~700 Mb
23	16	UDF	~Продолжить	~100 Mb

Итого: 23 теста, против 87, если выполнять полный перебор параметров.

10.3 Попарное тестирование с помощью PICT

PICT программа, позволяющая генерировать компактный набор значений тестовых параметров, который представляет собой все тестовые сценарии для всестороннего комбинаторного покрытия пользовательских параметров. К примеру, у пользователя есть следующие параметры для тестирования, который указан в таблице 10.6.

Таблица 10.6 - Набор данных для тестирования

Type	Size	Format method	File system	Cluster size	Compression
Primary	10	Quick	FAT	512	On
Logical	100	Slow	FAT32	102	Off
Single	500		NTFS	2048	
Span	1000			4096	
Stripe	5000			8192	
Mirror	10000			16384	
RAID-5	40000			32768	
				65536	

Существует больше 4700 комбинаций этих значений. Будет очень сложно протестировать их за разумное время. Исследования показывают, что тестирование всех пар возможных значений обеспечивает очень хорошее покрытие и количество тест кейсов остается в пределах разумного. К примеру, {**Primary, FAT**} это одна пара и {**10, slow**} другая; один тест кейс может покрывать много пар.

Запускается PICT из командной строки, как показано на рисунке 10.11.

```

C:\Program Files (x86)\PICT>pict.exe
Pairwise Independent Combinatorial Testing
Usage: pict model [options]

Options:
/o:N - Order of combinations (default: 2)
/d:C - Separator for values (default: ,)
/a:C - Separator for aliases (default: !)
/n:C - Negative value prefix (default: ~)
/e:file - File with seeding rows
/r[:N] - Randomize generation, N - seed
/c - Case-sensitive model evaluation
/s - Show model statistics
C:\Program Files (x86)\PICT>

```

Рисунок 10.11 - Запуск PICT

На вход программа принимает простой текстовый файл с параметрами и их значениями, называемый *Моделью*, а на выход выдает сгенерированные тестовые сценарии.

Рассмотрим работу программы на примере. Имеем следующие параметры и их значения, таблица 10.7.

Таблица 10.7 - Параметры и их значения для PICT

Пол	Возраст	Наличие детей
Мужской	До 25	Да
Женский	От 25 до 60	Нет
	Более 60	

Если перебирать все возможные значения, то количество сценариев будет 12. Составим модель и посмотрим какой результат выдаст программа. Для этого создается текстовый файл, в

котором указывается параметры и их значения. Содержимое текстового файла выглядит следующим образом:

SEX: Male, Female

Age: Under 25, 25-60, Older than 60

Children: Yes, No

Где:

- sex, Age, Children - это название параметров;
- male, Female, Under 25 и т.п. - это значения параметров, которые указываются через запятую.

Чтобы PICT получила результат в командной строке нужно написать "pict" и путь к модели.

Используя модель и получится 6 тестовых сценариев, вместо 12, изображенные на рисунке 10.12.

```
C:\Program Files <x86>\PICT>pict model.txt
SEX      Age      Children
Female   25-60      No
Male     Under 25    Yes
Female   Older than 60 Yes
Female   Under 25    No
Male     25-60      Yes
Male     Older than 60 No
C:\Program Files <x86>\PICT>
```

Рисунок 10.12 - Набор тест кейсов

Можно использовать прямой вывод и сохранение тест кейсов в Excel, рисунок 10.13.

```
C:\Program Files <x86>\PICT>pict model.txt > f:\example.xls
```

Рисунок 10.13 - Сохранение результат в Excel-файл

В результате будет создан Excel файл со следующим содержанием, рисунок 10.14.

	A	B	C
1	SEX	Age	Children
2	Female	25-60	No
3	Male	Under 25	Yes
4	Female	Older than	Yes
5	Female	Under 25	No
6	Male	25-60	Yes
7	Male	Older than	No

Рисунок 10.14 - Результат сформированный в Excel

Если рассмотреть решение первого примера в данной главе, то PICT получит 60 тестовых сценариев, против 4700 при полном переборе, изображенный на рисунке 10.15.

	A	B	C	D	E	F	G
1	#	Type	Size	Format m	File syste	Cluster s	Compression
2	1	Mirror	10	quick	FAT	32768	off
3	2	RAID-5	10	slow	FAT32	512	on
4	3	Stripe	500	quick	NTFS	512	off
5	4	Span	1000	slow	NTFS	1024	on
6	5	Primary	100	quick	FAT32	16384	off
7	6	Single	1000	slow	FAT	8192	off
8	7	Primary	5000	slow	FAT	2048	on
9	8	RAID-5	40000	quick	NTFS	8192	on
10	9	Logical	10	slow	NTFS	65536	on
11	10	Span	100	quick	FAT	65536	off
12	11	Mirror	10000	slow	FAT32	65536	on
13	12	Logical	1000	quick	FAT32	512	off
14	13	Logical	40000	slow	FAT	4096	off
15	14	Single	1000	quick	NTFS	4096	on
16	15	Stripe	500	slow	FAT32	32768	on
17	16	Mirror	100	quick	NTFS	2048	off
18	17	Span	10	slow	FAT32	4096	off
19	18	Single	40000	quick	FAT32	65536	off
20	19	RAID-5	5000	quick	FAT	65536	off
21	20	Stripe	1000	slow	FAT32	2048	on
22	21	Primary	10000	quick	NTFS	8192	off
23	22	Span	10000	slow	FAT	16384	on
24	23	Primary	1000	slow	FAT32	65536	on
25	24	Single	5000	quick	FAT32	1024	off
26	25	RAID-5	100	slow	FAT	1024	on
27	26	Single	500	slow	NTFS	2048	off
28	27	Mirror	500	quick	FAT	1024	on
29	28	Stripe	100	quick	FAT	4096	on
30	29	Primary	40000	quick	FAT32	1024	off
31	30	Single	10	quick	NTFS	16384	on
32	31	Logical	5000	slow	NTFS	32768	off
33	32	Stripe	10	slow	FAT	1024	off
34	33	Primary	500	slow	NTFS	4096	off
35	34	Mirror	1000	quick	FAT	16384	on
36	35	Stripe	40000	quick	FAT	16384	off
37	36	Mirror	10	slow	FAT32	8192	on
38	37	Span	40000	quick	NTFS	32768	off
39	38	Logical	10000	slow	NTFS	1024	off
40	39	Span	5000	quick	FAT	512	on
41	40	Logical	100	slow	FAT32	8192	on
42	41	RAID-5	500	quick	NTFS	16384	on
43	42	Stripe	5000	slow	NTFS	8192	off
44	43	Mirror	5000	slow	NTFS	4096	off
45	44	Span	500	quick	FAT	65536	off
46	45	Span	10000	slow	NTFS	2048	on
47	46	Stripe	10000	quick	FAT32	65536	off
48	47	Primary	10	quick	FAT	2048	off
49	48	RAID-5	10000	slow	NTFS	4096	on
50	49	Primary	10000	quick	NTFS	32768	on
51	50	RAID-5	1000	quick	FAT32	32768	on
52	51	Primary	10000	quick	FAT	512	off
53	52	Mirror	40000	slow	FAT32	512	on
54	53	Single	100	slow	NTFS	512	off
55	54	Logical	500	quick	FAT32	16384	off
56	55	Single	100	slow	NTFS	32768	on
57	56	Mirror	5000	quick	FAT32	16384	off
58	57	Span	500	slow	FAT	8192	on
59	58	RAID-5	40000	slow	FAT	2048	off
60	59	Logical	10	quick	FAT	2048	off
61	60	Single	10000	slow	FAT32	65536	on

Рисунок 10.15 - Результат при полном переборе

На рисунке 10.16 изображен результат, полученный в программе PICT для DVD.

	A	B	C	D	E
1	#	Speed	FS	ulstisessio	Capacity
2	1	2	ISO	Begin	100Mb
3	2	2	UDF/ISO	No	4.7Gb
4	3	2	UDF	Continue	4.7Gb
5	4	4	UDF/ISO	Begin	4.7Gb
6	5	4	UDF	Continue	4.7Gb
7	6	4	ISO	No	100Mb
8	7	8	ISO	No	4.7Gb
9	8	8	UDF/ISO	Begin	100Mb
10	9	8	UDF	Continue	4.7Gb
11	10	16	ISO	Continue	100Mb
12	11	16	UDF	Begin	4.7Gb
13	12	16	UDF/ISO	No	4.7Gb
14	13	24	UDF	No	100Mb
15	14	24	ISO	Begin	4.7Gb
16	15	24	UDF/ISO	Continue	100Mb

Рисунок 10.16 - Результат для DVD в PICT

Итого 15 сценариев против 17 у Allpairs. Это значит, что в данном примере PICT способен сгенерировать меньшее количество сценариев, чем Allpairs, при одинаковом покрытии.

Дополнительные возможности PICT:

- можно указывать порядок группировки значений. По умолчанию используется порядок **2** и создаются комбинации пар значений (что и составляет попарное тестирование). Но можно указать к примеру **3** и тогда будут использоваться триплеты, а не пары. Максимальный порядок для простой модели равен количеству параметров, что создаст набор всевозможных вариантов;
- можно группировать параметры в подмодели и указывать им отдельный порядок для комбинаций. Это необходимо, если комбинации определенных параметров должны быть протестированы более тщательно или должны быть объединены по отдельности от других параметров;
- можно создавать условия и ограничения. К примеру, можно указать, что один из параметров будет принимать определенное значение только тогда, когда несколько других параметров примут нужные значения. Это позволяет отсеять создание ненужных проверок;
- можно обозначать невалидные значения для параметров, для создания комбинации, для негативных тест кейсов;
- используя весовые коэффициенты можно указать программе отдавать предпочтения определенным значениям при генерации комбинаций;
- можно использовать опцию минимизации (запускать программу несколько раз с этой опцией), чтобы получить минимальное количество тест кейсов.

11 АВТОМАТИЗАЦИЯ. НАГРУЗОЧНОЕ ТЕСТИРОВАНИЕ

11.1 Терминология нагрузочного тестирования

Чтобы обсуждать подходы к нагрузочному тестированию и проблемы, решаемые с его помощью, нужно начать с основ - терминологии [9].

- 1) **виртуальный пользователь (Virtual User)** - программный процесс, циклически выполняющий моделируемые операции. *Пример - простой пользователь, который хочет воспользоваться системой;*
- 2) **итерация (Iteration)** – это один повтор выполняемой в цикле операции. *Пример итерации - вход в систему -> проведение анализа -> получение результата анализа - > выход из системы;*
- 3) **интенсивность выполнения операции (Operation Intensity)** - частота выполнения операции в единицу времени, в тестовом скрипте задается интервалом времени между итерациями. *Пример - 3 пользователя, которые входят в систему через минуту, после того, как зашел предыдущий пользователь;*
- 4) **нагрузка (Loading)** - совокупное выполнение операций на общем ресурсе. *Пример - общее количество выполненных операций в системе тремя пользователями за определенный промежуток времени;*
- 5) **производительность (Performance)** - количество выполняемых операций за период времени. *Если система не может выполнить определенной количество операций за заданный период времени, то она начинает тормозить, что означает нехватку производительности;*
- 6) **масштабируемость приложения (Application Scalability)** - пропорциональный рост производительности при увеличении нагрузки;
- 7) **профиль нагрузки (Performance Profile)** - это набор операций с заданными интенсивностями, полученный на основе сбора статистических данных либо определенный путем анализа требований к тестируемой системе. **Также называется сценарием (скриптом);**
- 8) **нагрузочной точкой** называется рассчитанное (либо заданное Заказчиком) количество виртуальных пользователей в группах, выполняющих операции с определенными интенсивностями.

Нужно рассмотреть как эти сущности связаны между собой. Выразив интенсивность через интервал времени между итерациями, можно увидеть, что рост интенсивности выполняемых операций это сокращение интервала времени. Рост нагрузки пропорционален росту интенсивности. Также, что при увеличении интенсивности растет производительность. При этом увеличивается степень использования (загруженности) ресурсов. С какого-то момента рост производительности

прекращается (а нагрузка может продолжать расти), происходит насыщение и затем деградация системы. В дополнение можно заметить что при тестировании изменение интенсивности операций может подчиняться какому либо закону (например, Пуассона) либо быть равномерным в течении всего теста.

11.2 Цели нагрузочного тестирования

Основными целями нагрузочного тестирования являются:

- оценка производительности и работоспособности приложения на этапе разработки и передачи в эксплуатацию;
- оценка производительности и работоспособности приложения на этапе выпуска новых релизов, патчей;
- оптимизация производительности приложения, включая настройки серверов и оптимизацию кода;
- подбор соответствующей для данного приложения аппаратной (программной платформы) и конфигурации сервера.

Нужно знать, что в рамках одной цели могут использоваться разные виды нагрузочного тестирования, например, для первой, второй и третьей цели нужно производить как тестирование производительности так и тестирование стабильности. Но при планировании нагрузочного тестирования логичнее все же отталкиваться от технических целей (а не коммерческих, перечисленных выше), которые достигаются в результате тестирования и классифицировать тесты по ним:

- если интересует исследование производительности приложения, а именно время отклика для операций на разных нагрузках в довольно широких диапазонах, включая стрессовые нагрузки, то это **тестирование производительности**;
- если целью является понимание насколько приложение устойчиво в режиме длительного использования (исключение утечек памяти, некорректных конфигурационных настроек и т.д.) то проводится долгий нагрузочный тест - это **тестирование стабильности**. При этом анализ времен отклика может иметь место, но не быть первым приоритетом, главное чтобы система "не упала";
- **стресс тестирование** имеет своей целью проверить возвращается ли система после запредельной нагрузки (и как скоро) к нормальному режиму, также целями стрессового тестирования могут быть проверки поведения системы в случаях, когда один из серверов приложения в пуле перестаёт работать, аварийно изменилась аппаратная конфигурации сервера базы данных и т.д. Нужно отметить также, что при

стрессовом тестировании проверяется не производительность системы, а её способность к регенерации после сверх нагрузки.

Главное понимать цели того или иного тестирования и постараться их достигнуть.

11.3 Этапы проведения нагрузочного тестирования

Рассматривая этапы проведения нагрузочного тестирования, хотелось бы отметить следующие:

- 1) анализ требований и сбор информации о тестируемой системе;
- 2) конфигурация тестового стенда для нагрузочного тестирования;
- 3) разработка модели нагрузки;
- 4) выбор инструмента для нагрузочного тестирования;
- 5) создание и отладка тестовых скриптов;
- 6) проведение тестирования;
- 7) анализ результатов;
- 8) подготовка, отправка и публикация отчета по проведенному нагрузочному тестированию.

11.3.1 Анализ требования и сбор информации о тестируемой системе

При анализе требований основной упор необходимо сделать на определение основных критериев успешности проведенных тестов. Для этого необходимо будет выделить следующие характеристики:

- **время отклика** (время необходимое для открытия страницы или получения ожидаемого результата);
- **интенсивность** (число запросов в секунду);
- **используемые ресурсы** (загрузка процессора, кол-во используемой памяти, дисковое и сетевой I/O и т.д.);
- **максимальное количество пользователей** (определяет число пользователей, способных работать с системой в условиях заданной конфигурации).

Заданные в требованиях характеристики, будут являться базовыми нагрузочными точками тестируемого приложения. Все получаемые результаты будут сравниваться с ними для принятия решения о завершении тестирования либо дальнейшем профилировании производительности.

Примечание:

- основной проблемой при анализе требований будет являться их отсутствие. В связи с тем, что не всегда бизнес аналитики или люди ответственные за написание требований по производительности реально представляют как система должна работать под

нагрузкой, какие именно требования должны быть предоставлены, очень часто цифры берутся просто «с потолка», поэтому приходится не только выделять имеющиеся требования, но и проводить их глубокий анализ на предмет их корректности;

- характеристика "Максимальное количество пользователей" на самом деле является малоинформативной, в случае если для тестирования необходимо будет работать с несколькими группами пользователей. Поэтому крайне важно будет знать более менее точное количество пользователей в каждой группе.

11.3.2 Анализ требований в зависимости от типа проекта

При анализе требований необходимо учесть разрабатывается ли новое (startup project) или же проект направлен на профилирование нагрузки для уже находящегося в эксплуатации приложения (profiling project).

В зависимости от типа проекта разрабатываются требования по производительности.

Для startup проектов:

- анализ общепринятых критериев производительности;
- анализ производительности конкурирующих приложений;
- анализ экспертного мнения разработчиков, системных и сетевых администраторов, администраторов баз данных и инженеров по нагрузочному тестированию;
- анализ ожиданий целевых пользователей (групп пользователей).

Для profiling проектов:

- анализ общепринятых критериев производительности;
- анализ производительности конкурирующих приложений;
- анализ производительности эксплуатируемой версии приложения, с целью определения требующих профилирования функций, процессов, операций и т.д;
- анализ экспертного мнения разработчиков, системных администраторов и администраторов баз данных, инженеров по нагрузочному тестированию;
- анализ мнения целевых пользователей (групп пользователей).

И уже после получения всех данных из всех источников можно получить более менее точные требования по производительности для тестируемого приложения.

11.3.3 Конфигурация тестового стенда для нагрузочного тестирования

На результаты нагрузочного тестирования могут влиять разные факторы, такие как конфигурация тестового стенда, загруженность сети, заполненность базы данных и многие другие. Причем влияние их на производительность приложения может быть значительным и иметь нелинейную зависимость, поэтому выразить её формулой будет практически невозможно.

Следовательно, чем меньше будут разниться параметры тестовой и реальной инфраструктуры, тем меньше будет погрешность в полученных результатах.

Нужно отметить те части конфигурации, которые требуют особого внимания:

Hardware:

- процессор (тип, частота, количество ядер и т.д);
- оперативная память (тип, объем, тайминг, эффективная частота и т.д.);
- жесткие диски (тип, скорость и т.д.).

Software:

- операционная система;
- драйвера.

Network:

- топология сети;
- пропускная способность;
- протокол передачи данных.

Application:

- архитектура;
- база данных (структура + данные);
- программное обеспечение, необходимое для работы приложения (например, для Java приложений - JVM).

В самом идеальном случае тестовый стенд один к одному дублирует конфигурацию реального сервера, на котором работает или же будет работать приложение. Однако, идеальных случаев практически не бывает (то памяти мало, то процессора такой частоты нет в наличии, то операционная система не той версии, то стоимость некоторого серверного ПО не укладывается в бюджете). Перечислим основные причины, по которым не всегда получается продублировать конфигурацию системы на тестовом стенде:

- 1) сложность дублирования дорогого серверного железа для тестовых нужд;
- 2) ограничения на использование лицензий требуемого программного обеспечения;
- 3) закрытость архитектуры приложения со стороны заказчика по соображениям безопасности;
- 4) трудность воссоздания или транспортировки базы данных приложения;
- 5) сложность воссоздания требуемой архитектуры сети;
- 6) и многое другое (всё перечислить крайне сложно из-за большого количества нюансов, влияющих на конфигурацию системы).

Целесообразность же воссоздания инфраструктуры необходимо оценить с учетом выделенных ресурсов, времени и усилий, так как не всегда результат будет оправдывать средства.

11.3.4 Разработка модели нагрузки

Определившись с видами нагрузочного тестирования, целями и терминологией, нужно перейти к основной задаче нагрузочного тестирования - разработке модели нагрузки.

Для этого необходимо определить следующее:

- список тестируемых операций;
- интенсивность выполнения операций;
- зависимость изменения интенсивности выполнения операций от времени.

В список тестируемых задач должны войти операции, критичные с точки зрения бизнеса, а также с технической точки зрения:

- критичными с **точки зрения бизнеса** являются операции, скорость выполнения которых, реально влияет на производительность бизнес процесса. *Например, увеличение длительности обслуживания клиентов в банке, невозможность выполнения необходимого количества операций в течение дня и так далее;*
- критичными с **технической точки зрения** являются ресурсоемкие операции, требующие большое количество памяти, серьезно задействующие процессор, создающие значительный сетевой трафик. *Как правило, это операции выполняемые одновременно большим количеством бизнес пользователей или создание сложных отчетов, в которые входят так называемые "тяжелые" запросы к базе данных;*

Нужно подчеркнуть, что под степенью критичности операции подразумевается её влияние на бизнес процесс и работоспособность системы. *Например, создание какого-нибудь отчета, полностью загружающего сервер базы данных в ночное время, не будет носить высокий приоритет для оптимизации, а в рабочие часы будет иметь максимальный приоритет.*

Модель тестирования производительности

Постепенное увеличение нагрузки, добавляя новых пользователей с некоторым интервалом времени, позволяет определить:

- измерение времени выполнения выбранных операций при определенных интенсивностях выполнения этих операций;
- количество пользователей, способных одновременно работать с приложением;
- границы приемлемой производительности при увеличении нагрузки;
- производительность при разных нагрузках.

Модель стрессового тестирования

Увеличивая интенсивность операций выше пиковых (максимально разрешенных) значений, либо увеличивая количество пользователей до тех пор, пока нагрузка не станет выше максимально допустимых значений, проверяем, что система работоспособна в условиях стресса. Далее, опустив нагрузку до средних значений, проверяем (способность системы к регенерации), что система вернулась к нормальному состоянию (основные нагрузочные характеристики не превышают базовые).

Модель объемного тестирования

Можно использовать ту же модель что и для тестирования производительности, однако целью будет проверка работы системы с прогнозом на будущий рост объема данных. Следовательно, одним и самым важным предусловием теста будет увеличение объемов базы данных приложения до требуемых размеров. Таким образом можно проверить и оценить производительность, прогнозируя рост системы на год, два или три вперед.

Модель тестирования стабильности или надежности

Используя базовый нагрузочный профиль, запускаем тест длительностью от нескольких часов до нескольких дней, с целью выявления утечек памяти, перезапуска серверов и других аспектов влияющих на нагрузку.

11.4 Обзор программ нагрузочного тестирования веб-сервисов

Сдавая веб-сервер в повседневную эксплуатацию, нужно быть уверенным, что он выдержит планируемую нагрузку. Только создав условия, приближенные к боевым, можно оценить, достаточна ли мощность системы, правильно ли настроены приложения, участвующие в создании веб-контента, и прочие факторы, влияющие на работу веб-сервера. В этой ситуации на помощь придут специальные инструменты, которые помогут дать качественную и количественную оценку работы как веб-узла в целом, так и отдельных его компонентов.

OpenSTA, рисунок 11.1- больше чем приложение для тестов, это открытая архитектура, проектируемая вокруг открытых стандартов. Проект создан в 2001 году группой компаний CYRANO, которая поддерживала коммерческую версию продукта, но CYRANO распался, и сейчас OpenSTA распространяется как приложение с открытым кодом под лицензией GNU GPL. Для работы требует Microsoft Data Access Components (MDAC), который можно скачать с сайта корпорации [12].

Текущий инструментарий позволяет провести нагрузочное испытание HTTP/HTTPS сервисов, хотя его архитектура способна на большее. OpenSTA позволяет создавать тестовые сценарии на специализированном языке SCL (Script Control Language). Для упрощения создания и редактирования сценариев используется специальный инструмент Script Modeler. Выбирается

Tools – Canonicalize URL, поле запускается веб-браузер. Пользователь просто ходит по сайтам, собирая ссылки, которые будут сохранены в скрипт. Все параметры запроса поддаются редактированию, возможна подстановка переменных. Структура теста и заголовки будут выводиться во вкладках в панели слева. Тесты удобно объединять в наборы. Настройки прокси задаются в самом скрипте, поэтому можно указать несколько серверов. Реализована возможность организации распределенного тестирования, что повышает реалистичность, или когда с одного компьютера не получается нагрузить мощный сервер. Каждая из машин такой системы может выполнять свою группу заданий, а repository host осуществляет сбор и хранение результатов. После установки на каждой тестирующей системе запускается сервер имен, работа которого обязательна. Поддерживается аутентификация пользователей на веб-ресурсе и установление соединений по протоколу SSL. Параметры работы нагружаемой системы можно контролировать с помощью SNMP и средств Windows NT. Результаты тестирования, включающие время откликов, количество переданных байт в секунду, коды ответа для каждого запроса и количество ошибок выводятся в виде таблиц и графиков. Использование большого числа фильтров позволяет отобрать необходимые результаты. Результат можно экспортировать в CSV-файл. Возможности по выводу отчетов несколько ограничены, но по ссылкам на сайте можно найти скрипты и плагины, упрощающие, в том числе, анализ полученной информации.

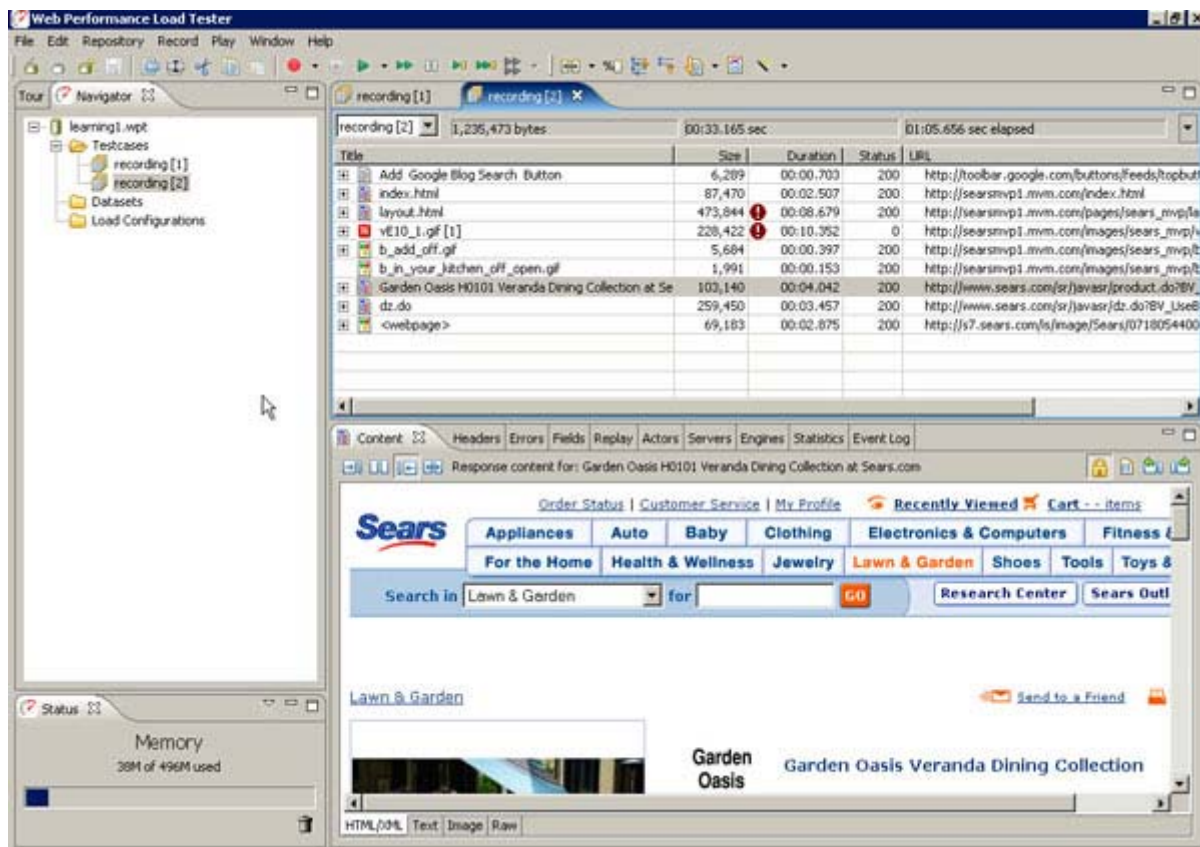


Рисунок 11.1 - OpenSTA

Apache Jmeter, рисунок 11.2 - это единственный инструмент для нагрузочного тестирования, который с одной стороны бесплатный и open source, а с другой стороны достаточно развит и имеет возможность создания тестовой нагрузки одновременно с нескольких компьютеров [12].

Тесты для JMeter создаются визуально и имеют древовидную структуру в окне редактирования теста.

Запуск тестов можно производить как из окна приложения, так и из командной строки, что в свою очередь полезно, если вы их запускаете по расписанию, например, ночью.

Для чего он лучше всего подходит

Инструмент позиционирует себя как универсальный, позволяющий работать как с http запросами, так и FTP, JDBC запросы, SOAP, Web Services, TCP, LDAP, JMS, Mail testers. Но, однозначно, он лучше всего подходит для нагрузочного тестирования веб-приложений.

Если пользователь создает многопользовательское веб-приложение, то перед его выпуском у него однозначно возникнут вопросы:

- стабильно ли работает приложение под большой нагрузкой;
- какое максимально возможное число пользователей выдержит приложение на определенной конфигурации;
- насколько быстрее стало работать приложение после улучшения архитектуры кода.

На все эти вопросы, с относительно небольшими затратами времени ответит JMeter.

Из чего состоит тест

При создании теста JMeter предлагает несколько типов компонент:

- 1) **samplers** - основные элементы, которые непосредственно общаются с тестируемым приложением, например http sampler для обращения к веб-приложению;
- 2) **logic controllers** - элементы, позволяющие группировать другие элементы в циклы, группы параллельного запуска, и т.д.;
- 3) **assertions** - элементы, выполняющие контроль. С их помощью вы можете проверить текст, который вы ожидаете на веб-странице или, например, указать, что вы ожидаете ответ от сервера не более чем 2 секунды. Если какой-либо Assert не будет удовлетворен, тест будет иметь негативный результат.

Как выглядит отчет

Что будет содержаться в отчете, пользователь настраивает сам. Удобно то, что в отчет можно включать таблицы и графики.

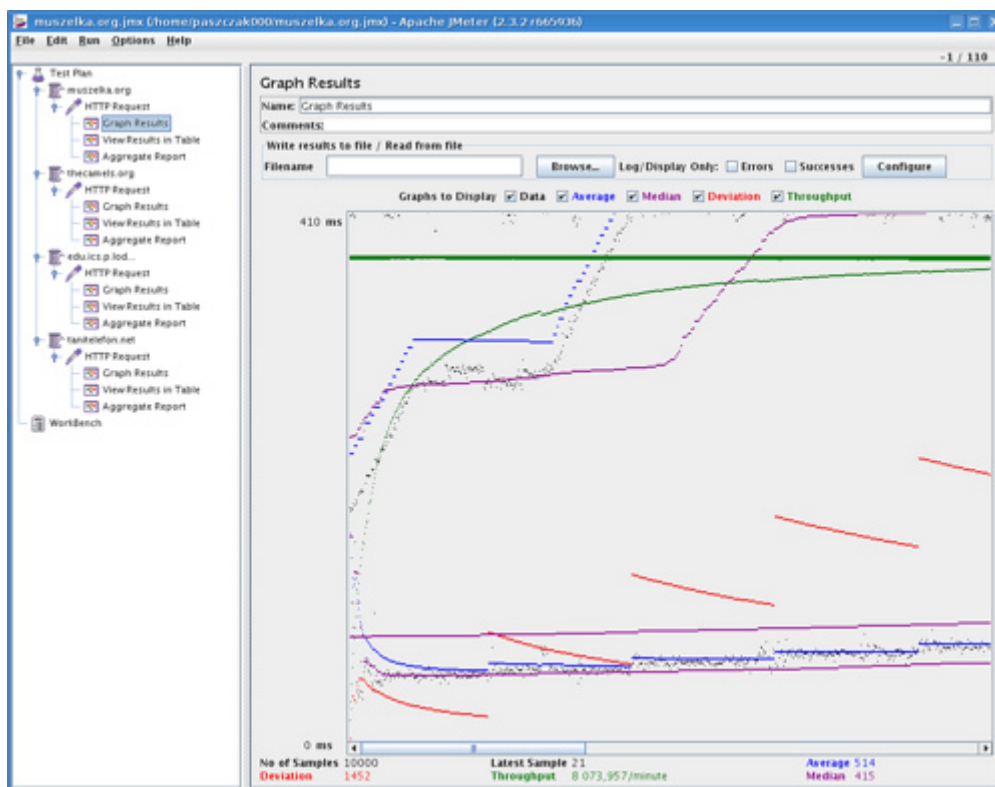


Рисунок 11.2 - Apache Jmeter

Можно ли расширить JMeter

Jmeter является удобным инструментом для запуска и мониторинга тестов, а также для просмотра отчетов. Поэтому если пользователю нужны особые виды тестов, которые он может написать на Java, то все что ему нужно, это написать код самого теста. Далее его нужно оформить как Sampler, после чего JMeter поможет ему сделать всю остальную работу по организации, конфигурированию и выполнению тестов (в том числе и с нескольких компьютеров).

WAPT (Web Application Testing), рисунок 11.3 - позволяет испытать устойчивость веб-сайта и других приложений, использующих веб-интерфейс, к реальным нагрузкам. Разрабатывается новосибирской компанией SoftLogica LLC. Это одна из самых простых в использовании программ обзора. Для проведения простого теста даже не нужно заглядывать в документацию, интерфейс прост, но не локализован. Для проверки WAPT может создавать множество виртуальных пользователей, каждый с индивидуальными параметрами. Поддерживается несколько видов аутентификации и куки. Сценарий позволяет изменять задержки между запросами и динамически генерировать некоторые испытательные параметры, максимально имитируя таким образом поведение реальных пользователей. В запрос могут быть подставлены различные варианты HTTP-заголовка, в настройках можно указать кодировку страниц. Параметры User-Agent, X-Forwarded-For, IP указываются в настройках сценария. Значения параметров запроса могут быть рассчитаны несколькими способами, в том числе, определены ответом сервера на предыдущий запрос, используя переменные и функции [12]. Поддерживается работа по

защищенному протоколу HTTPS (и все типы прокси-серверов). Созданные сценарии, сохраняемые в файле XML-формата, можно использовать повторно. Кроме стандартных Performance и Stress, в списке присутствуют несколько других тестов, позволяющих определить максимальное количество пользователей и тестировать сервер под нагрузкой в течение долгого периода.

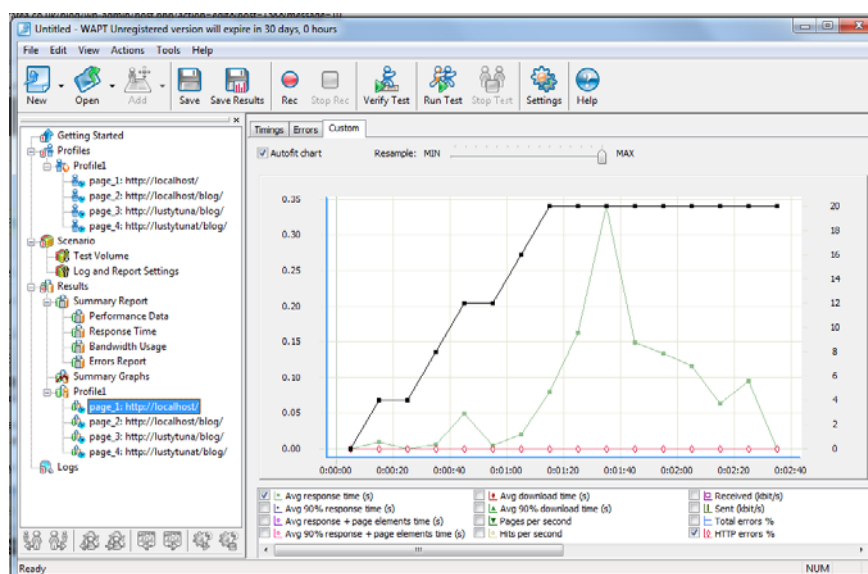


Рисунок 11.3 - WAPT

NeoLoad, рисунок 11.4 - система, позволяющая провести нагрузочное тестирование веб-приложений. Написана на Java. В отчете можно получить подробную информацию по каждому загруженному файлу. NeoLoad весьма удобен для оценки работы отдельных компонентов (AJAX, PHP, ASP, CGI, Flash, апплетов и пр.). Возможна установка времени задержки между запросами (thinktime) глобально и индивидуально для каждой страницы. Тестирование проводится как с использованием весьма удобной графической оболочки, так и с помощью командной строки (используя заранее подготовленный XML-файл). Поддерживает работу с протоколом HTTPS, с HTTP и HTTPS прокси, basic веб-аутентификацию и cookies, автоматически определяя данные во время записи сценария, и затем проигрывает во время теста. Для работы с различными профилями для регистрации пользователей могут быть использованы переменные. При проведении теста можно задействовать дополнительные мониторы (SNMP, WebLogic, WebSphere, RSTAT и Windows, Linux, Solaris), позволяющие контролировать и параметры системы, на которой работает веб-сервер [12].

При помощи **NeoLoad** можно проводить и распределенные тесты. Один из компьютеров является контролером, на остальные устанавливаются генераторы нагрузки (loadGenerator). Контролер распределяет нагрузку между loadGenerator и собирает статистику.

Очень удобно реализована работа с виртуальными пользователями. Пользователи имеют индивидуальные настройки, затем они объединяются в Populations (должна быть создана как минимум одна Populations), в Populations можно задать общее поведение (например, 40%

пользователей популяции посещают динамические ресурсы, 20% читают новости). Виртуальные пользователи могут иметь индивидуальный IP-адрес, полосу пропускания и свой сценарий теста.

Используя утилиты нагрузочного тестирования, можно получить информацию о работе веб-сервиса, принять необходимые меры по устранению выявленных недостатков и гарантировать требуемую производительность.

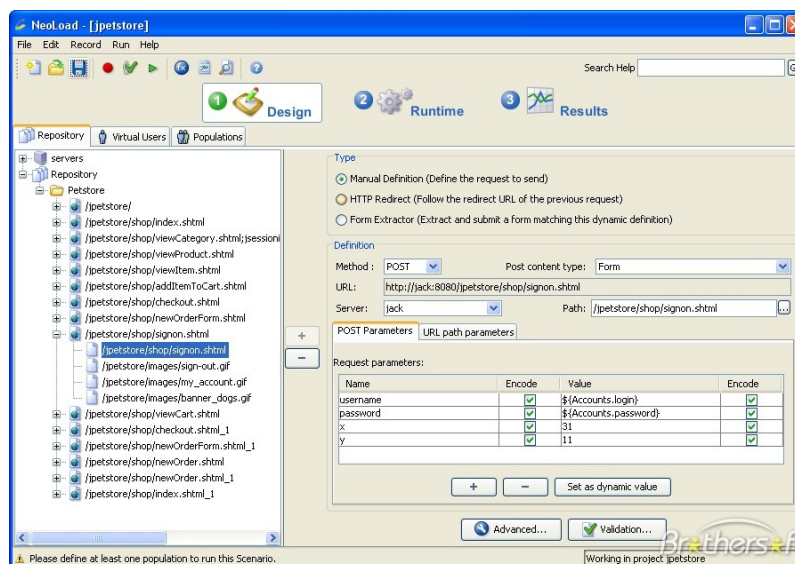


Рисунок 11.4 - NeoLoad

Продукции Microsoft. Корпорация Microsoft предлагает целых два продукта, позволяющих протестировать веб-сервер под нагрузкой. Это **Microsoft Application Stress Tool** и **Web Capacity Analysis Tool**. Первый распространяется как отдельный продукт и имеет графический интерфейс. Второй входит в состав комплекта инструментов **Internet Information Services 6.0 Resource Kit Tools**, работает из командной строки. MAST более наглядный, в создании теста поможет простой мастер создания тестов, возможна работа с cookies, регулировка нагрузки по разным URL. Сценарий тестирования может быть создан вручную или записан с помощью веб-браузера и при необходимости отредактирован. В WAST уровень нагрузки (stress level) регулируется путем задания количества нитей, осуществляющих запросы к серверу, а число виртуальных пользователей рассчитывается как произведение числа нитей на число сокетов, открытых каждой из нитей. По окончании теста получаем простой отчет в текстовой форме, в котором дана информация по числу обрабатываемых запросов в единицу времени, среднему времени задержки, скорости передачи данных на сервер и с сервера, количеству ошибок и т.д. Отчет можно экспортировать в CSV-файл. Никаких возможностей по статистической обработке не предусмотрено, то есть с его помощью можно только оценить работу при определенных условиях.

В качестве примера работы с программами был рассмотрен инструмент для нагрузочного тестирования - Jmeter.

11.5 Нагрузочное тестирование с помощью Jmeter

Работа с программой Jmeter была рассмотрена на простом примере создания скрипта для 5 пользователей, которые хотят:

- войти в систему;
- провести полный финансовый анализ;
- выйти из системы.

На первый взгляд может показаться, что сценарий слишком короткий, однако во-первых, созданный скрипт не должен содержать много действий. Если цель состоит в том, что нужно рассмотреть больше сценариев пользователя, то скрипты можно комбинировать между собой в цепочку для выявления слабых мест в системе. А во-вторых, даже такой короткий скрипт поможет разобраться в программе Jmeter и использовать полученные знания для самостоятельного изучения программы.

Одна из главных особенностей программы - автоматическая запись скрипта. Суть ее в том, что когда создается скрипт для сервиса, который тестируется методом черного ящика, тестировщик не знает всех тонкостей и особенностей его работы. Для облегчения работы существует встроенный элемент HTTP Proxy Server. Эмулируя работу прокси сервера, он будет записывать все полученные/отправляемые запросы.

11.5.1 Подготовительные действия

Перед использованием прокси нужно понять, куда записывать полученные шаги. Их можно записать прямо в "катушку" (*Thread*), на "верстак" (*Workbench*), либо в элемент "**Recording Controller**". Лучше использовать последний элемент по двум причинам: во-первых, так лучше отслеживается и формируется структура теста, а во-вторых, это позволит при необходимости отключить (*ctrl+t*) весь лог разом. Все действия по добавлению и редактированию происходят по нажатию правой кнопки мыши в контекстном меню. Созданные элементы можно просто перетаскивать (*drag and drop*). Но, для использования элемента "Recording Controller" нужно добавить хотя бы одну катушку. Для этого нужно нажать правой кнопкой на **Test Plan > Add > Threads > Thread Group**. Созданную катушку можно переименовать для удобства и понимания. Чтобы это сделать, необходимо нажать на созданный элемент, справа будет поле Name, вместо Thread Group можно написать что угодно. Катушка была названа как "*dwarf*" - имя сервера, где находится сервис, с которым была проведена работа. После добавлении катушки добавляется Recording Controller. Для этого нужно нажать правой кнопкой на созданный **Thread Group > Add > Logic Controller > Recording Controller**. Созданный элемент был назван как "*вход в систему*". На рисунке 11.5 изображен результат.

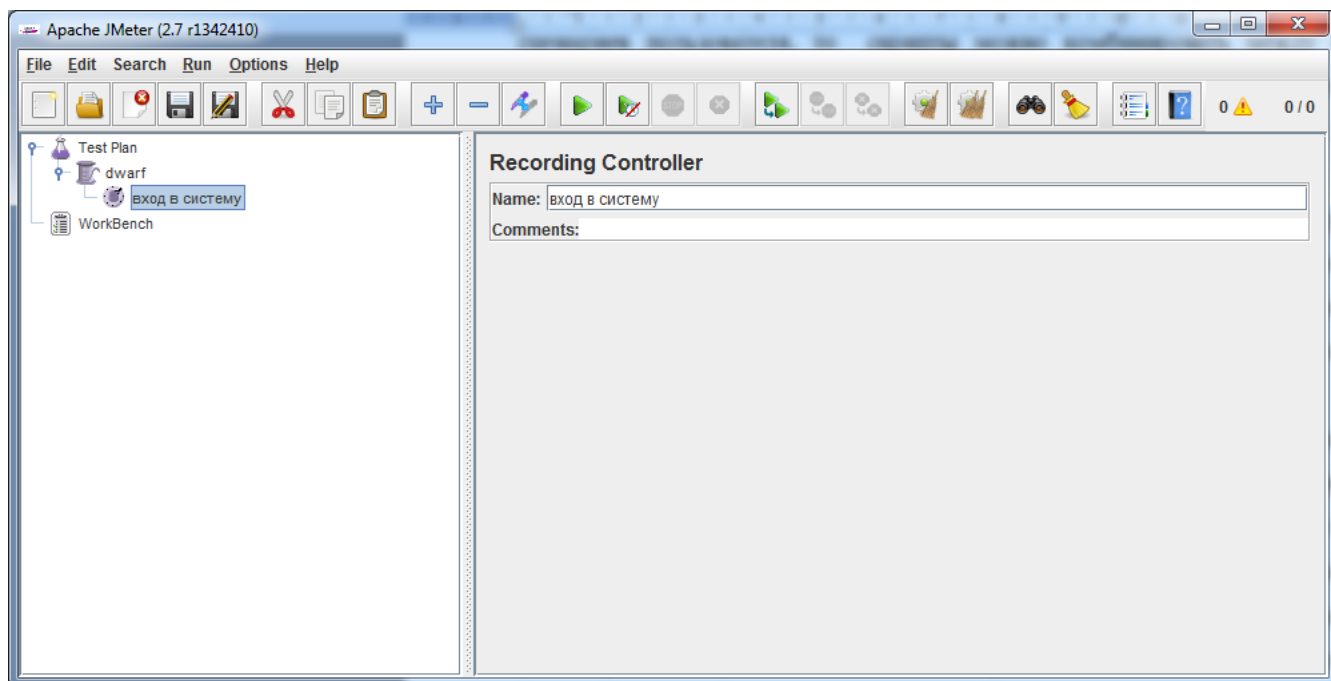


Рисунок 11.5 - Thread Group и Recording Controller

11.5.2 Запись скрипта при помощи HTTP Proxy Server

Когда подготовительные действия были завершены, то можно добавить на «верстак» прокси-элемент: **WorkBench > Add > Non-Test Elements > HTTP Proxy Server**. В нем много настроек, но важен пока только порт. Если порт 8080 занят, можно выставить, например, 18000 или любой другой. Так же можно увидеть, что по умолчанию в поле **Target Controller** стоит «*use recording controller*». На рисунке 11.6 было явно указано, куда нужно записывать скрипт, т.е. в **dwarf > вход в систему**.

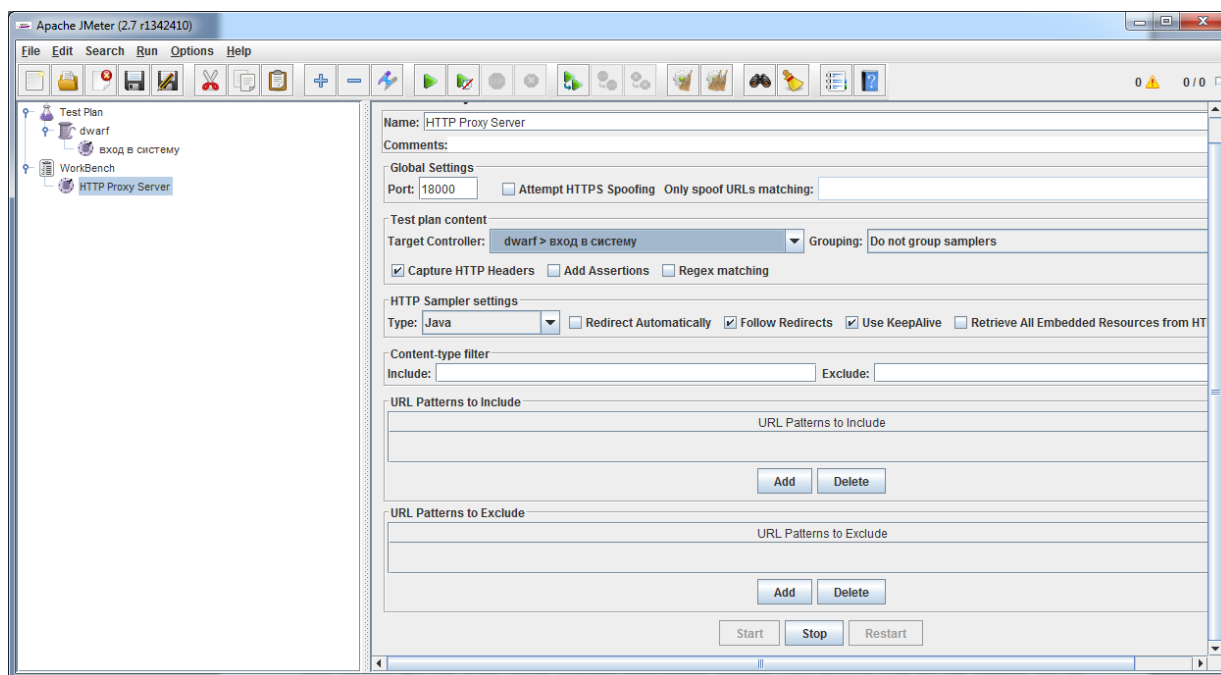


Рисунок 11.6 - Настройка HTTP Proxy Server

После того, как порт был выставлен, необходимо перейти в настройки браузера. В данном примере будет использоваться IE9. **Сервис — Свойства обозревателя — Подключения — кнопка Настройка сети**. Отмечаются галки «Использовать прокси-сервер ...», вторую галку необходимо снять. На рисунке 11.7 в поле Адрес указывается *localhost*, а порт как из Jmeter'a: *18000*.

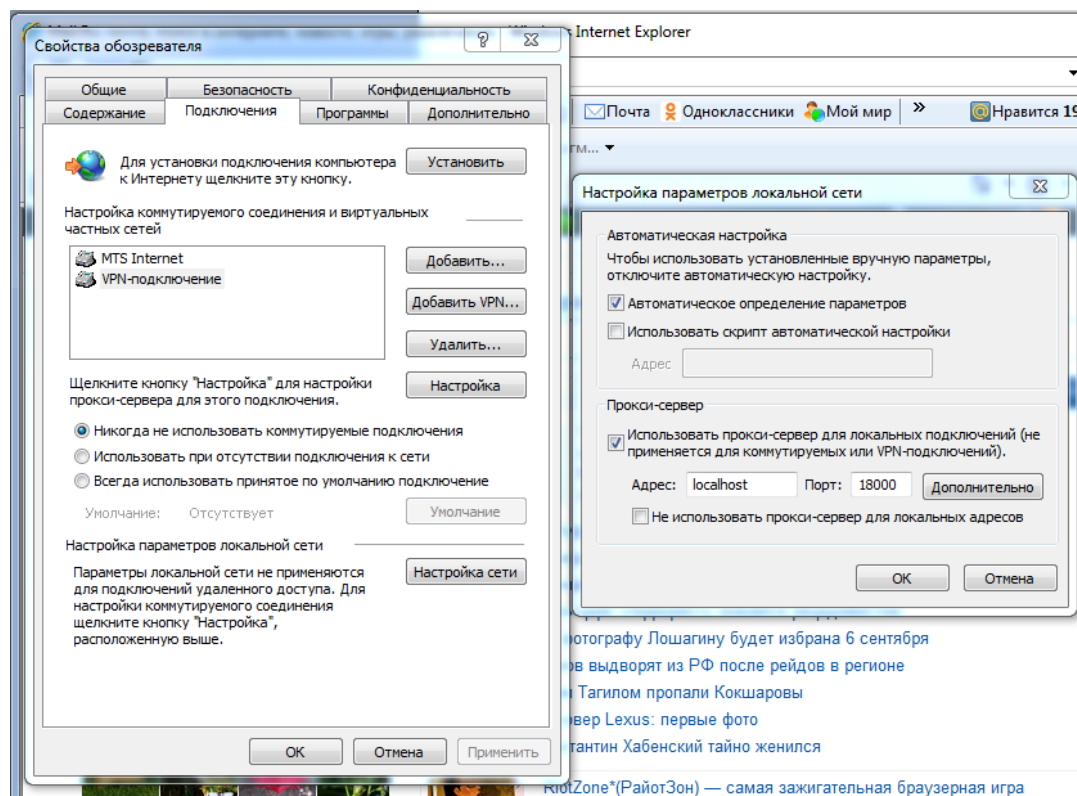


Рисунок 11.7 - Настройка браузера

Теперь все готово для записи. Нужно вернуться в Jmeter на прокси и внизу нажать кнопку Start. После нажатия все действия пользователя в браузере начнут записываться в созданный Recording Controller. Необходимо войти в систему. Записывать действия желательно в несколько шагов, т.к. скрипт может оказаться чрезмерно большим и его будет сложно редактировать. Поэтому первое действие было выбрано как авторизация на сервисе. После нажатия **Start** и прохождения процедуры авторизации на сервере и процедуры авторизации на сайте, необходимо вернуться в Jmeter и нажать на **Stop**. В записанном Recording Controller есть много дочерних элементов, которые изображены на рисунке 11.8. Это то, что было послано на сервер, а сервер ответил. Необходимо изучить скрипт убрать лишнее.

11.5.3 Отладка скрипта

Отладка скрипта представляет собой удаление различных .jpg, .png и ссылок на сторонние ресурсы. Всё это можно вычищать (клавишей delete). В целом, также можно вычистить .js. Главное найти запрос, который передает в своем теле учетные данные вашего пользователя. Для красоты найти запрос, который ведет на страницу, на которой пользователь логинится.

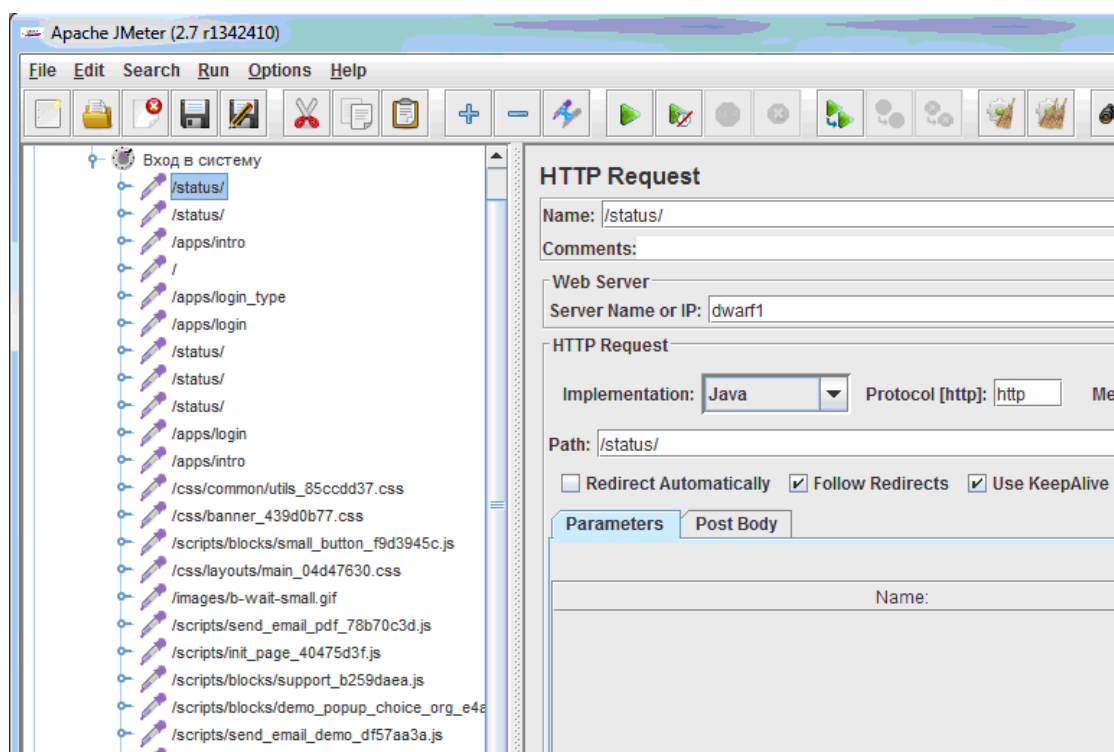


Рисунок 11.8 - Сгенерированный скрипт входа в систему

Таким образом, вместе они моделируют связку в действиях пользователя «зашел на страницу — залогинился». После того как мусор был убран, а нужные запросы были оставлены, желательно дать им читаемые имена. Чтобы проверить, то ли было удалено, скрипт надо запустить. В данном примере, если запустить скрипт, то он выполнится с ошибками. Дело в том, что для тестирования нужно авторизоваться на сервере. Для этого необходимо добавить в верхушку дерева **HTTP Authorization Manager (Thread Group > Add > Config Elements > HTTP Authorization Manager)**. Он интуитивно понятен. В нем необходимо указать адрес ресурса, на котором происходит нагрузочное тестирование, обязательно с http://, имя пользователя и пароль.

Если Менеджер Авторизации нужен для авторизации на сервере, то абсолютно точно при авторизации на сайте используются куки. Необходимо добавить **Cookie Manager** после Менеджера Авторизации (**TestPlan > Add > Config Elements > HTTP Cookie Manager**). Этот элемент нужно добавлять всегда, если речь идет о том, что пользователю нужно залогиниться. На рисунке 11.9 изображено добавление. Прежде чем запустить и проверить скрипт добавляется элемент **View Results Tree (dwarf > Add > Listener > View Results Tree)**, в котором можно будет наблюдать выполнение. Этот элемент удобно располагать всегда внизу. Еще нужно в настройках браузера убрать галку «использовать прокси».

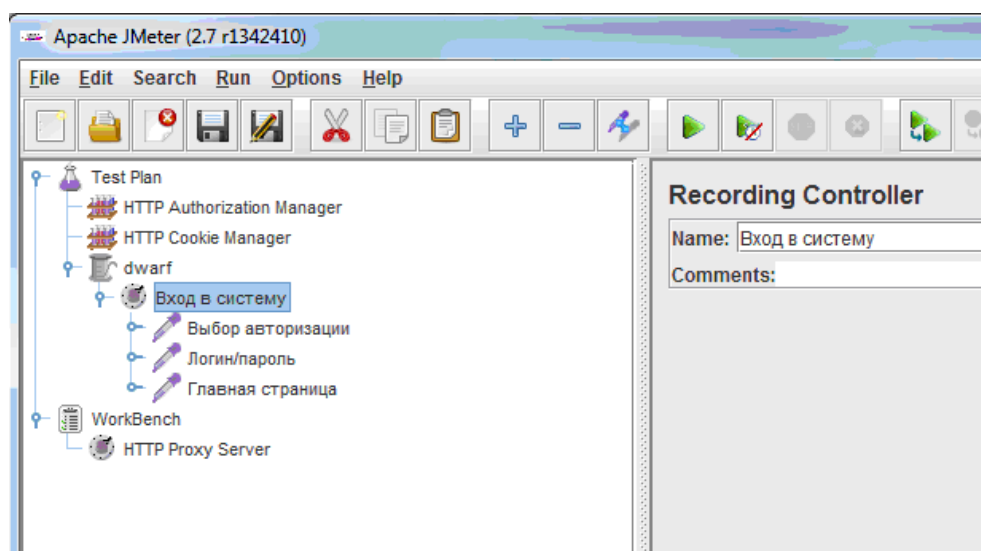


Рисунок 11.9 - Установка менеджера авторизации и куков

Для запуска используется сочетание клавиш **ctrl + r**. На рисунке 11.10 изображен результат выполнения скрипта.

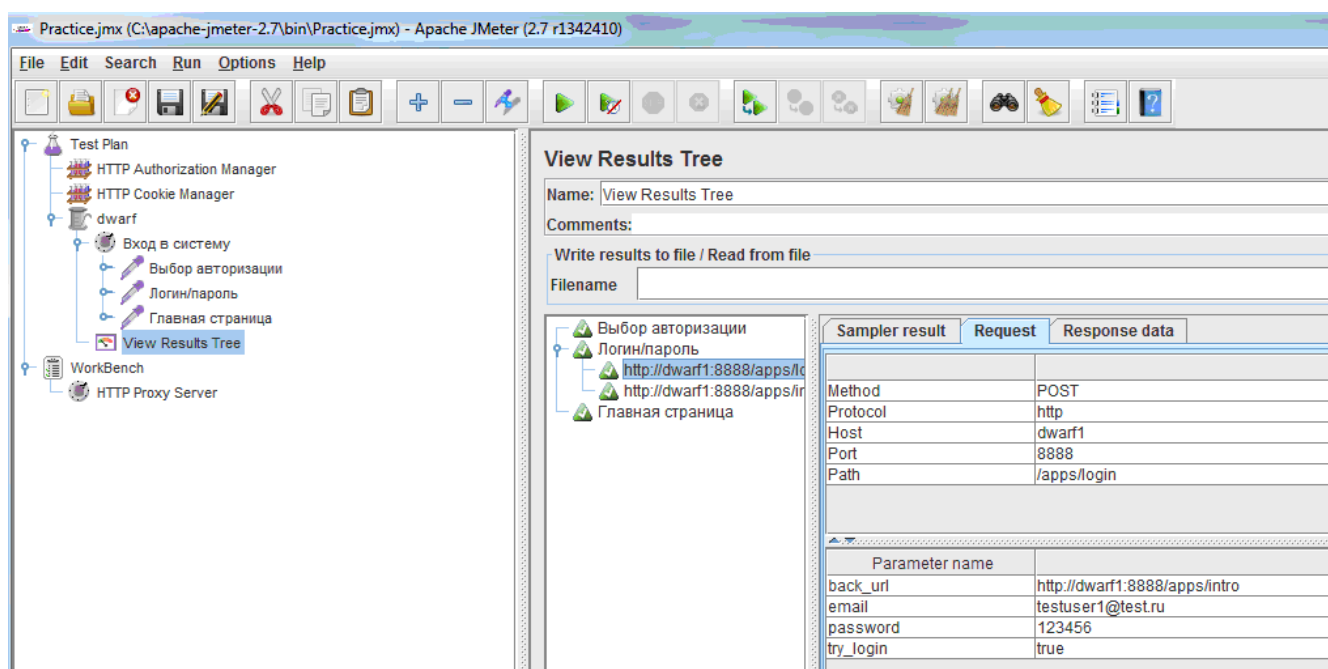


Рисунок 11.10 - Результат выполнения скрипта

Для того, чтобы убедиться, что пользователь был успешно авторизован, можно зайти в дерево результатов и посмотреть на ответ. Для этого нужно нажать на вкладку **Response Data**. В ответе сервера есть информация, которая свидетельствует об успешной авторизации пользователя. В данном примере это имя пользователя, которое было указано при регистрации email, т.к. после авторизации оно высвечивается на странице. На рисунке 11.11 можно увидеть этого пользователя в Response Data.

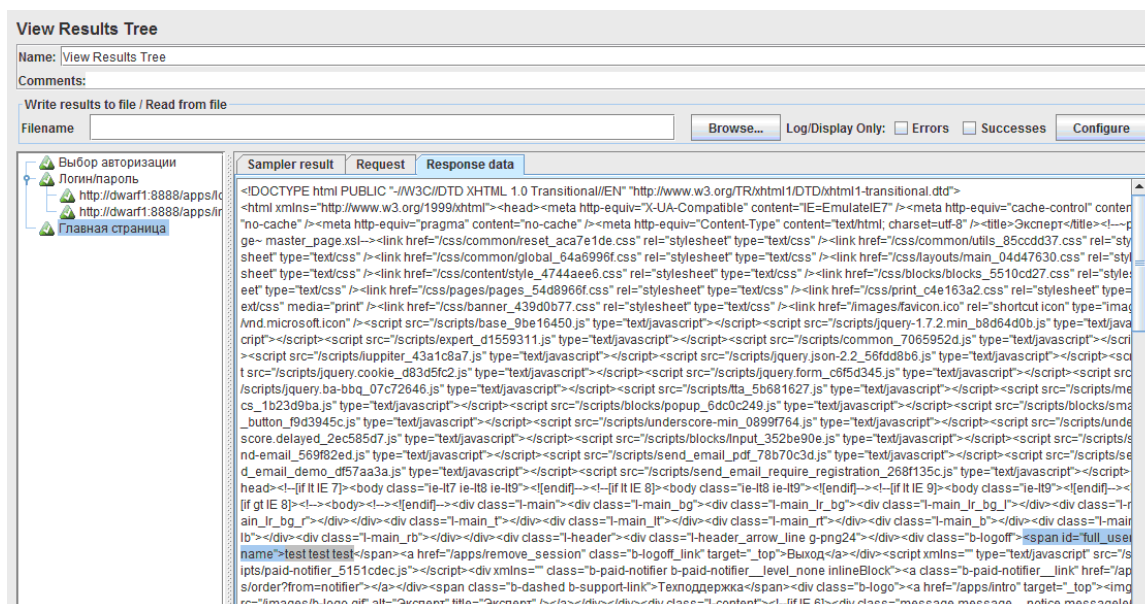


Рисунок 11.11 - Response Data

Теперь стоит задач привлечения 5 пользователей. Для этого прежде необходимо выполнить небольшой учебный пример по параметризации запросов.

11.5.4 Параметризация

Для параметризации запросов добавляется в самый верх элемент **User Define Variables**. Thread > Config Element > User Define Variables. Внизу есть кнопка Add. После ее нажатия добавляются две переменные, т.к. меняться будут только два параметра: логин и пароль. Им задаются имена: user и pswd. А в качестве значений указываются конкретные значения текущего пользователя. На рисунке 11.12 можно увидеть добавления.

User Defined Variables		
Name: User Defined Variables		
Comments:		
User Defined Variables		
Name:	Value	Description
user	testuser1@test.ru	Логин
pswd	123456	Пароль

Рисунок 11.12 - User Defined Variables

Чтобы их использовать нужно перейти в запрос Логин/пароль, и вместо конкретных значений записать имена переменных в формате ***\${имя переменной}***. В данном примере указываются в поля *Value* для логина — *\${user}*, для пароля — *\${pswd}*. На рисунке 11.13 можно увидеть добавление параметров.

HTTP Request

Name:

Comments:

Web Server

Server Name or IP: Port Number:

Timeouts (milliseconds)
Connect: Response:

HTTP Request

Implementation: Protocol [http]: Method: Content encoding:

Path:

☐ Redirect Automatically ☒ Follow Redirects ☒ Use KeepAlive ☐ Use multipart/form-data for POST ☐ Browser-compatible headers

Parameters Post Body

Send Parameters With the Request:

Name:	Value	Encode?	Include Equals?
email	\$(user)	<input type="checkbox"/>	<input checked="" type="checkbox"/>
password	\$(pswd)	<input type="checkbox"/>	<input checked="" type="checkbox"/>
back_url	http%3A%2F%2Fdwarf1%3A8888%2Fapps%2Fintro	<input type="checkbox"/>	<input checked="" type="checkbox"/>
try_login	true	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Рисунок 11.13 - Параметризация логин и пароля

Для отладки передачи параметров добавляется еще один элемент – **Debug Sampler. Thread > Add > Samplers > Debug Sampler**. В настройках элемента остается только Jmeter variables, остальные параметры в значение false. На рисунке 11.14 это изображено.

Debug Sampler

Name:

Comments:

JMeter properties:

JMeter variables:

System properties:

Рисунок 11.13 - Настройка Debug Sampler

После запуска в View Results Tree должны появиться результаты. На них можно увидеть, что в качестве переменных передались именно те значения, изображение 11.14, которые были заданы, и запросы в целом вернули то, что нужно.

Выбор авторизации
Логин/пароль
Главная страница
Debug Sampler

Sampler result Request Response data

JMeterVariables:
JMeterThread.last_sample_ok=true
JMeterThread.pack=org.apache.jmeter.threads.SamplePackage@715e8179
START.HMS=101132
START.MS=1378354292259
START.YMD=20130905
TESTSTART.MS=1378356238820
pswd=123456
user=testuser1@test.ru

Рисунок 11.14 - Просмотр параметров

Теперь можно передать данные 5 пользователей. Для этого используется *CSV Data Set Config*.

11.5.5 CSV Data Set Config

Создается .csv файл в блокноте. Для пользователей файл должен представлять собой набор из 5 строчек вида «user; pswd», как показано на рисунке 11.15

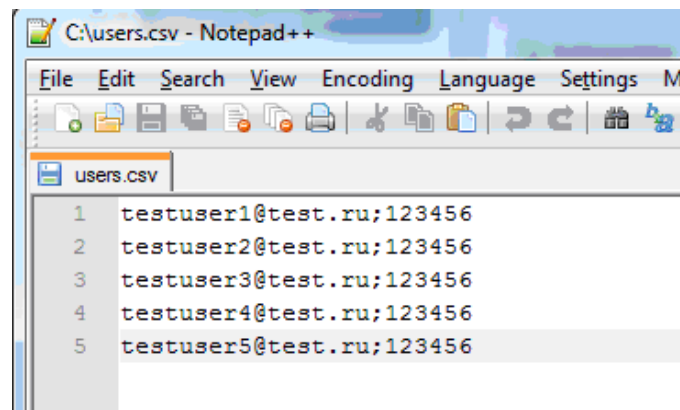


Рисунок 11.15 - CSV-файл с пользователями и паролями

Далее в дерево добавляется элемент CSV Data Set Config (**Add > Config Element > CSV Data Set Config**). Лучше его добавить не в катушку, а в тест-план, ниже пользовательских переменных. В качестве *filename* для файла нужно указать полностью путь к файлу и его имя. Ниже указать кодировку (на ваш выбор). Если все данные на английском, то поле можно оставить пустым. Ниже необходимо написать переменные, которые считаются из файла. В качестве примера, это будут user и pswd. В качестве разделителя нужно указать точку с запятой, так как в файле используется она. Нужно поставить в false повтор файла по достижении конца, а также остановку катушки. Настройка CSV Data Set изображена на рисунке 11.16

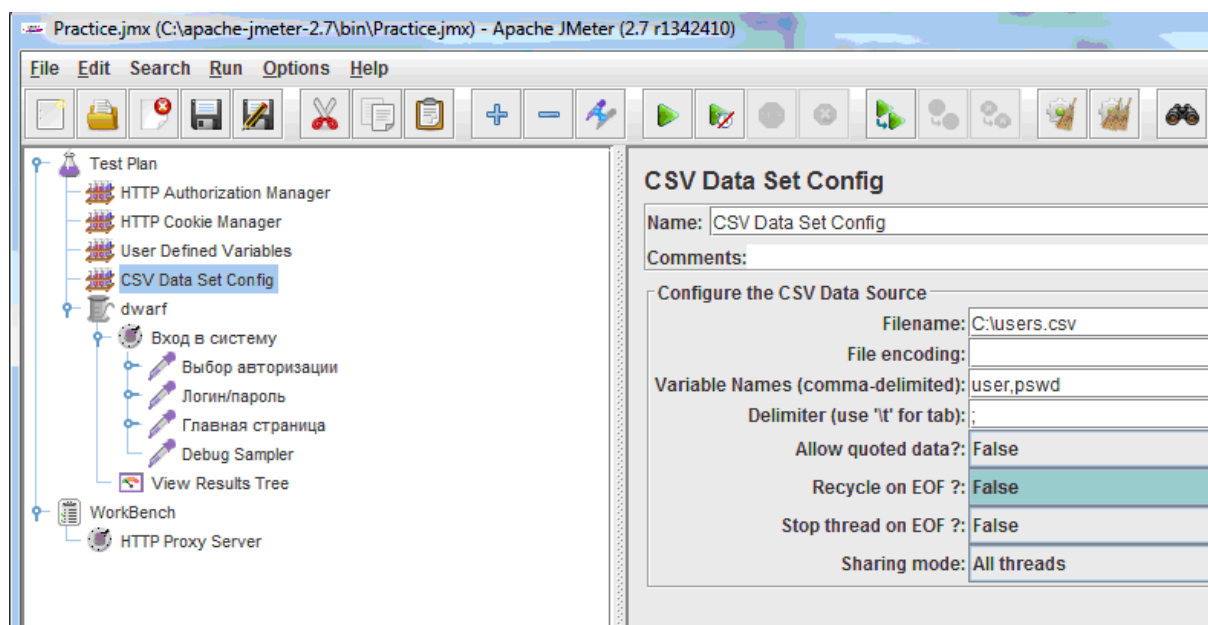


Рисунок 11.16 - Настройка CSV Data Set Config

Далее необходимо удалить подобные переменные из User Defined Variables. После этого в самой катушке количество потоков (**Number of Threrads**) устанавливается на 5. Теперь можно

запустить скрипт и наблюдать в Debug Sampler и View Results Tree результаты, рисунок 11.17. Нужно не забыть проверить, те ли данные вернул сервер, есть ли в них имена созданных пользователей.

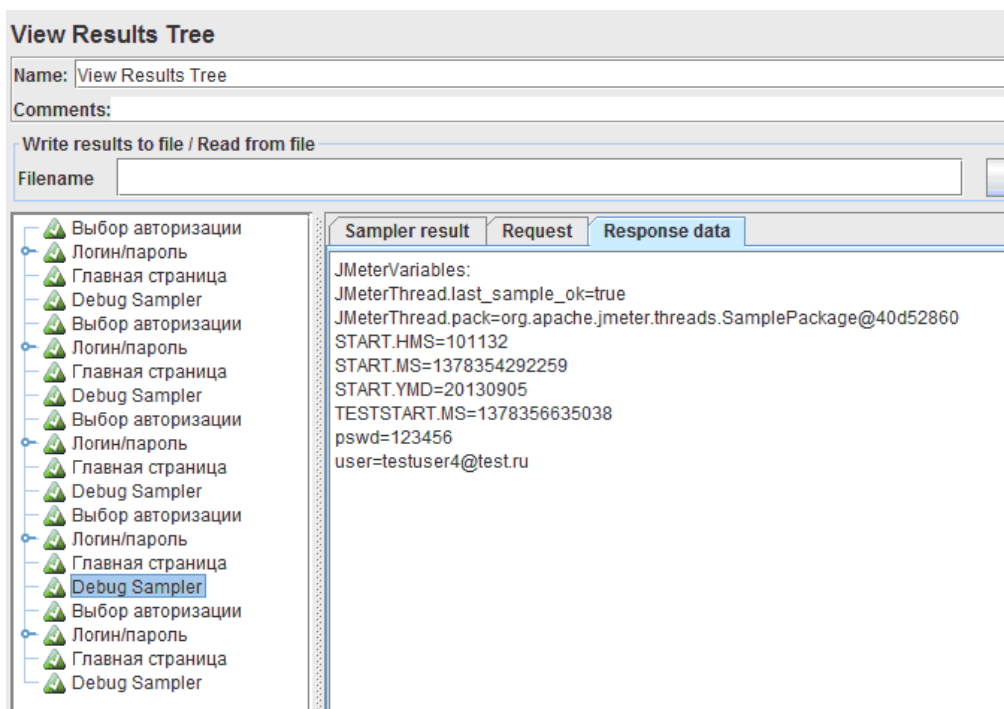


Рисунок 11.17 - Результат

11.5.6 Создание ФА

После создание скрипта на вход в систему, нужно проделать тоже самое, только для создания финансового анализа. Для этого нужно опять включить прокси-сервер и поставить галочку в браузере "использовать прокси-сервер". В Jmeter создается второй Recording Controller, который был назван "Создание ФА". Нужно не забыть поменять Target Controller в прокси на *dwarf* > *Создание ФА*, как показано на рисунке 11.18.

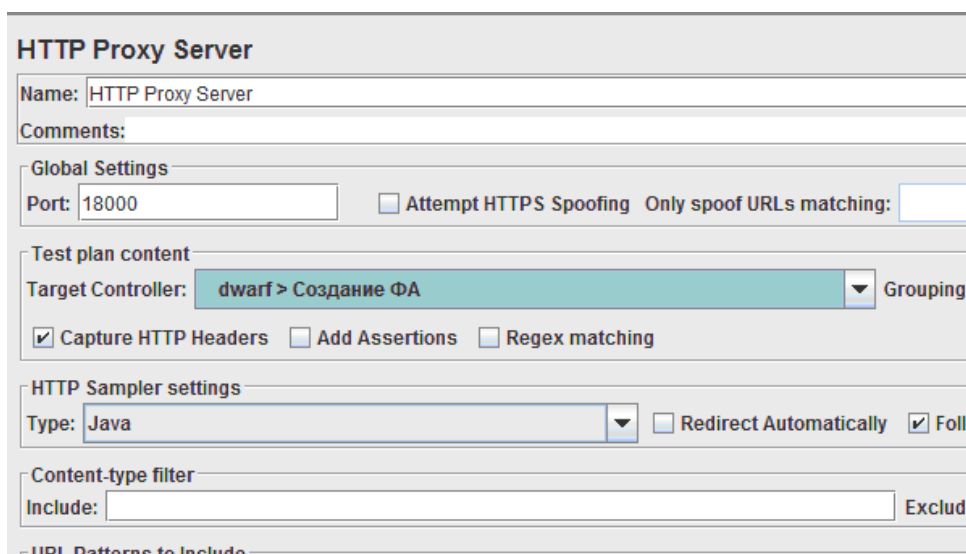


Рисунок 11.18 - Изменение цели

После записи скрипта (создание ФА) получается следующий результат, изображенный на рисунке 11.19.

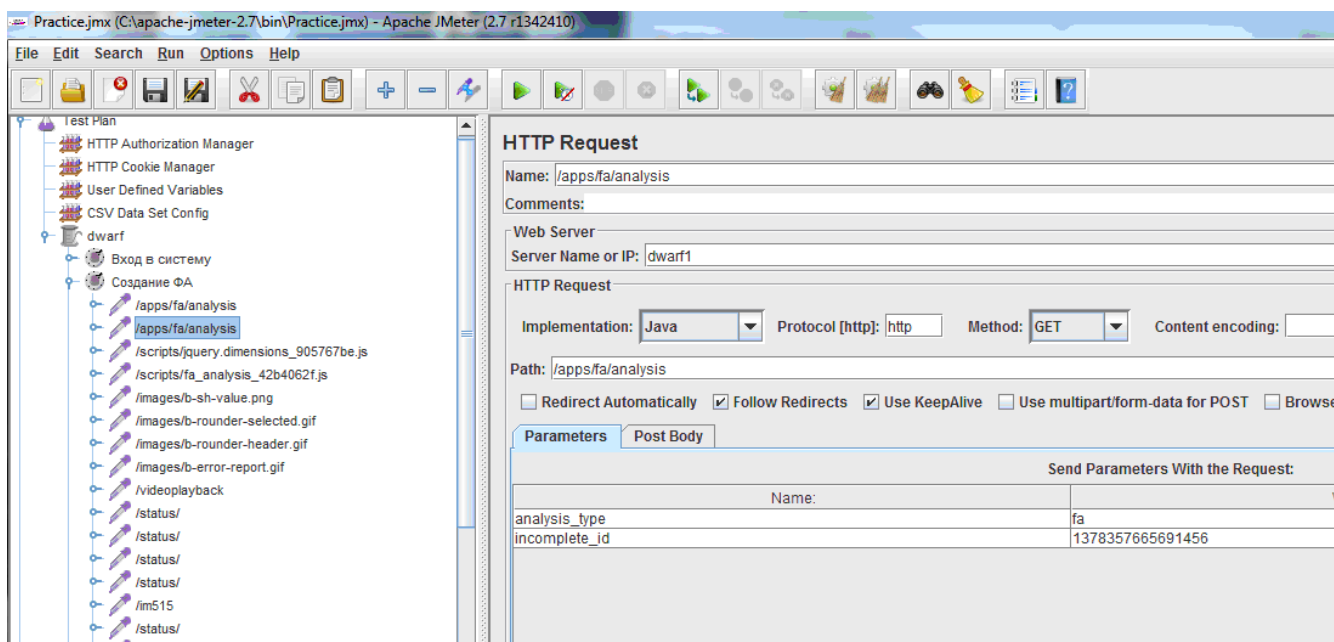


Рисунок 11.19 - Скрипт создание ФА

Данный скрипт нужно отчистить от мусора, оставить только самое нужное и поменять для удобства имена запросов.

При создании скрипта ФА нужно знать следующее:

- при создании ФА присваивается уникальный и неповторимый id;
- результат ФА также имеет уникальный скрытый id.

Следовательно, необходимо параметризовать эти 2 id.

Для параметризации скрытого id используется CSV Data Set Config еще раз, как показано на рисунке 11.20.

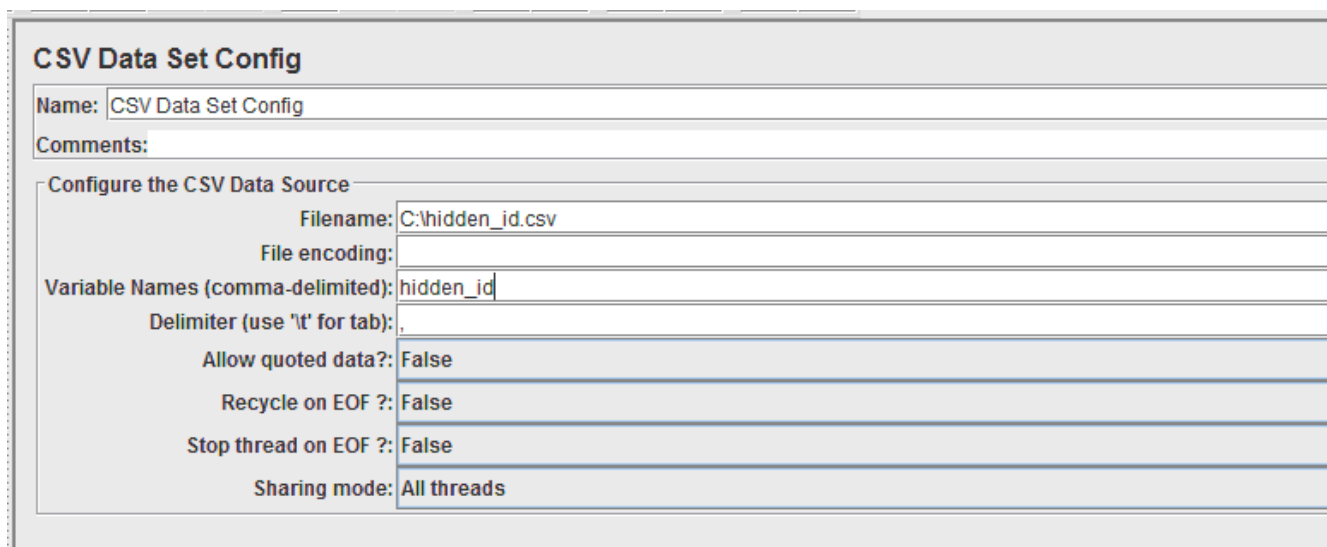


Рисунок 11.20 - Параметризация скрытого id

Для параметризации уникального id, при создании ФА, нужно его сначала достать. Для этого используется элемент **Regular Expression Extractor** (*dwarf > Add > Post Processors > Regular Expression Extractor*). Уникальный id находится в адресной строке и имеет приблизительно следующий вид: fa/analysis?analysis_type=fa&incomplete_id=1378393412734465, где incomplete_id и есть идентификатор.

В Reference Name указывается имя параметра в Regular Expression записывается регулярное выражение, которое будет доставать нужный id при создании нового ФА. \$1\$ - означает порядок расстановки регулярного выражение, в нашем случае оно одно. Если бы было их 2, то запись должна быть \$1\$2\$. На рисунке 11.21 изображены настройки Regular Expression Extractor.

Regular Expression Extractor

Name: Regular Expression Extractor

Comments:

Apply to:

☒ Main sample only ☐ Sub-samples only ☐ Main sample and sub-samples ☐ JMeter Variable

Response Field to check

☐ Body ☐ Body (unescaped) ☐ Headers ☒ URL ☐ Re:

Reference Name: id

Regular Expression: incomplete_id=([^\"]+)

Template: \$1\$

Match No. (0 for Random):

Default Value:

Рисунок 11.21 - Настройка Regular Expression Extractor

Теперь нужно заменить установленный id в скрипте на параметризованный. Вместо фиксированного идентификатора устанавливается созданный параметр, как показано на рисунке 11.22

HTTP Request

Name: /apps/fa/analysis

Comments:

Web Server

Server Name or IP: dwarf1 Port Number: 8888

HTTP Request

Implementation: Java Protocol [http]: http Method: GET Content encoding:

Path: /apps/fa/analysis

☐ Redirect Automatically ☒ Follow Redirects ☒ Use KeepAlive ☐ Use multipart/form-data for POST ☐ Browser-compatible headers

Parameters Post Body

Send Parameters With the Request:

Name:	Value
analysis_type	fa
incomplete_id	1378357665691456

Рисунок 11.22 - Замена на параметр

Все эти манипуляции с id были для того, чтобы каждый пользователь создавал новый ФА, а не пересоздавал уже созданный анализ.

Была поставлена следующая задача: нужно выяснить в каком месте, при создании ФА, серверу необходимо много времени для ответа, при высокой нагрузки на него.

Для этого была симитирована ситуация, когда в системе находится одновременно 50 пользователей и они создают финансовый анализ. Для этого в Thread Group было проставлено количество пользователей 50, а время, в течении которого эти 50 пользователей окажутся в системе, 10 секунд, как показано на рисунке 11.23. Т.е. за 10 секунд в системе будет работать ~50 пользователей. Возможно такой ситуации на самом деле маловероятно, но задача сейчас выяснить потенциально слабое место в системе.

Thread Group

Name: dwarf

Comments:

Action to be taken after a Sampler error: ☒ Continue ☐ Start Next

Thread Properties

Number of Threads (users): 50

Ramp-Up Period (in seconds): 10

Loop Count: ☐ Forever 1

☐ Scheduler

Рисунок 11.23 - Установка 50 пользователей

Для того, чтобы точно определить слабое звено был использован элемент *View Results in Table (dwarf > Add > Listener > View Results in Table)*. На рисунке 11.24 изображен результат.

Filename	Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes
	605	14:15:31.362	dwarf 1-34	Выход	13	Success	3913
	606	14:15:31.376	dwarf 1-34	Debug Sampler	0	Success	494
	607	14:15:18.858	dwarf 1-41	Результат анализа	12921	Success	78199
	608	14:15:31.780	dwarf 1-41	Главная страница	32	Success	21328
	609	14:15:31.813	dwarf 1-41	Выход	14	Success	3913
	610	14:15:31.828	dwarf 1-41	Debug Sampler	0	Success	494
	611	14:15:17.649	dwarf 1-39	Результат анализа	15477	Success	77983
	612	14:15:33.126	dwarf 1-39	Главная страница	53	Success	22782
	613	14:15:18.993	dwarf 1-50	Результат анализа	14191	Success	78155
	614	14:15:33.180	dwarf 1-39	Выход	9	Success	3913
	615	14:15:33.190	dwarf 1-39	Debug Sampler	0	Success	494
	616	14:15:33.184	dwarf 1-50	Главная страница	162	Success	21304
	617	14:15:18.126	dwarf 1-46	Результат анализа	15244	Success	78671
	618	14:15:33.347	dwarf 1-50	Выход	22	Success	3913
	619	14:15:33.371	dwarf 1-50	Debug Sampler	0	Success	492
	620	14:15:33.371	dwarf 1-46	Главная страница	20	Success	21328
	621	14:15:33.392	dwarf 1-46	Выход	8	Success	3913
	622	14:15:33.401	dwarf 1-46	Debug Sampler	0	Success	494
	623	14:15:18.524	dwarf 1-47	Результат анализа	14893	Success	78699
	624	14:15:33.417	dwarf 1-47	Главная страница	34	Success	22782
	625	14:15:33.452	dwarf 1-47	Выход	9	Success	3913
	626	14:15:33.462	dwarf 1-47	Debug Sampler	0	Success	494
	627	14:15:18.627	dwarf 1-44	Результат анализа	14869	Success	79077
	628	14:15:33.497	dwarf 1-44	Главная страница	16	Success	22786
	629	14:15:33.514	dwarf 1-44	Выход	16	Success	3913
	630	14:15:33.531	dwarf 1-44	Debug Sampler	0	Success	494
	631	14:15:18.614	dwarf 1-49	Результат анализа	19854	Success	78666
	632	14:15:19.114	dwarf 1-45	Результат анализа	19358	Success	78024
	633	14:15:38.468	dwarf 1-49	Главная страница	42	Success	22782
	634	14:15:38.472	dwarf 1-45	Главная страница	40	Success	21304
	635	14:15:38.511	dwarf 1-49	Выход	10	Success	3913
	636	14:15:38.521	dwarf 1-49	Debug Sampler	0	Success	494
	637	14:15:38.516	dwarf 1-45	Выход	16	Success	3913
	638	14:15:38.533	dwarf 1-45	Debug Sampler	0	Success	494
	639	14:15:19.378	dwarf 1-48	Результат анализа	19456	Success	78316
	640	14:15:38.834	dwarf 1-48	Главная страница	57	Success	22789
	641	14:15:38.892	dwarf 1-48	Выход	10	Success	3913
	642	14:15:38.903	dwarf 1-48	Debug Sampler	0	Success	494
	643	14:15:19.447	dwarf 1-43	Результат анализа	19477	Success	79056
	644	14:15:38.924	dwarf 1-43	Главная страница	22	Success	22788
	645	14:15:38.946	dwarf 1-43	Выход	5	Success	3913
	646	14:15:38.952	dwarf 1-43	Debug Sampler	0	Success	494
	647	14:15:19.322	dwarf 1-38	Результат анализа	19881	Success	77955
	648	14:15:39.204	dwarf 1-38	Главная страница	20	Success	22789
	649	14:15:39.224	dwarf 1-38	Выход	8	Success	3913
	650	14:15:39.233	dwarf 1-38	Debug Sampler	0	Success	494

Рисунок 11.24 - Результат

Из таблицы можно увидеть, что самое слабое звено в системе - это формирование результата анализа, т.к. время, которое нужно серверу для этого действия, с каждым разом растет, и в итоге может достигнуть критической отметки - 30 сек. В результате этого пользователь будет долго ждать свой анализ, а в веб-приложении это критический параметр. Соответственно, для того, чтобы этого не происходило нужно что-то сделать с формированием результата, чтобы это действие занимало меньше времени, а также нагрузка на сервер была низкая.

- большие затраты на разработку – разработка автоматизированных тестов это сложный процесс, так как фактически идет разработка приложения, которое тестирует другое приложение. В сложных автоматизированных тестах также есть фреймворки, утилиты, библиотеки и прочее. Естественно, все это нужно тестировать и отлаживать, а это требует времени;
- 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;
- пропуск мелких ошибок - автоматический 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 контролов и форм с которыми не осуществляется взаимодействие во время выполнения скрипта.

Для того чтобы принять решение о целесообразности автоматизации приложения нужно ответить на вопрос «перевешивают ли в нашем случае преимущества?» - хотя бы для некоторой функциональности нашего приложения. Если нельзя найти таких частей, либо недостатки в вашем случае неприемлемы – от автоматизации стоит воздержаться.

При принятии решения стоит помнить, что альтернатива – это ручное тестирование, у которого есть свои недостатки.

12.2 Применение автоматизации

Автоматизацию нужно применять в следующих случаях:

- 1) труднодоступные места в системе (бэкенд процессы, логирование файлов, запись в БД);
- 2) часто используемая функциональность, риски от ошибок в которой достаточно высоки. Автоматизировав проверку критической функциональности, можно гарантировать быстрое нахождение ошибок, а значит и быстрое их решение;
- 3) рутинные операции, такие как переборы данных (формы с большим количеством вводимых полей. Автоматизировать заполнение полей различными данными и их проверку после сохранения);
- 4) валидационные сообщения (Автоматизировать заполнение полей не корректными данными и проверку на появление той или иной валидации);
- 5) длинные end-to-end сценарии;
- 6) проверка данных, требующих точных математических расчетов;

7) проверка правильности поиска данных.

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

Для более эффективного использования автоматизации тестирования лучше разработать отдельные тест кейсы проверяющие:

- базовые операции создания/чтения/изменения/удаления сущностей (так называемые CRUD операции - Create / Read / Update / Delete). **Пример:** создание, удаление, просмотр и изменение данных о пользователе;
- 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. Мы предлагаем вам использовать именно такие сценарии, так как они позволяют вернуть систему в состояние, максимально близкое к исходному, а значит – минимально влияющее на другие тесты;
- 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, от автоматизации тестирования которой, можно получить наибольшую отдачу.

12.3 Как автоматизировать

В данном разделе рассмотрены аспекты, влияющие на выбор инструмента автоматизации тестирования.

Во первых, нужно обратить внимание насколько хорошо инструмент для автоматизации распознает элементы управления в вашем приложении. В случае когда элементы не распознаются стоит поискать плагин, либо соответствующий модуль. Если такового нет – от инструмента лучше отказаться. Чем больше элементов может распознать инструмент – тем больше времени можно сэкономить на написании и поддержке скриптов.

Во-вторых, нужно обратить внимание на то сколько времени требуется на поддержку скриптов написанных с помощью выбранного инструмента. Для этого необходим простой скрипт, который выбирает пункт меню, а потом представьте, что изменился пункт меню который необходимо выбрать. Если для восстановления работоспособности сценария придется перезаписать скрипт целиком, то инструмент не оптимален, так как реальные сценарии гораздо сложнее. Лучше всего тот инструмент, который позволяет вынести название кнопки в переменную в начале скрипта и быстро заменить ее значение. В идеале – описать меню как класс.

В-третьих, насколько удобен инструмент для написания новых скриптов. Сколько требуется на это времени, насколько можно структурировать код (поддержка ООП), насколько код читаем, насколько удобна среда разработки для рефакторинга (переработки кода) и т.п.

Лучше всего для принятия правильного решения об автоматизации отвечать на вопросы «Зачем? Что? Как?» именно в таком порядке. Это поможет избежать впустую потраченное время, нервы и финансы. С другой стороны можно получить надежность, скорость и качество.

12.4 Уровни автоматизации тестирования

Условно, тестируемое приложение можно разбить на 3 уровня:

- **unit tests layer;**
- **functional tests layer (Non-UI);**
- **gui tests layer.**

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

Уровень модульного тестирования (Unit Test layer)

Под автоматизированными тестами на этом уровне понимаются Компонентные или Модульные тесты написанные разработчиками. Тестировщикам никто не запрещает писать такие 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.

Уровень функционального тестирования (Functional Test Layer non-ui)

Не всю бизнес логику приложения можно протестировать через GUI слой. Это может быть особенностью реализации, которая прячет 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.

Уровень тестирования через пользовательский интерфейс (GUI Test Layer)

На данном уровне есть возможность тестировать не только интерфейс пользователя, но также и функциональность, выполняя операции вызывающую бизнес логику приложения. С нашей точки зрения, такого рода сквозные тесты дают больший эффект нежели просто тестирование функционального слоя, так как мы тестируем функциональность, эмулируя действия конечного пользователя, через графический интерфейс.

12.5 Архитектура тестов

Для удобства наложения автоматизированных тестов, на уже имеющиеся тест кейсы, **структура тест скриптов** должна быть аналогична структуре **тестового случая** - Precondition, Steps & Post Condition.

Получаем правило, что каждый тест скрипт должен иметь:

- precondition;
- steps (Test);
- post Condition.

Перечислим основные функции скрипта:

1) precondition:

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

2) steps:

- непосредственное проведение теста;
- занесение данных о результате теста, с обязательным сохранением причин провала и шагов, по которым проходил тест.

3) post condition:

- удаление, созданных в процессе выполнения скрипта, ненужных тестовых данных;
- корректное завершение работы приложения.

Рекомендуется также создать общую библиотеку по обработке ошибок и исключительных ситуаций. Например:

- PreConditionException;
- TestCaseException;
- PostConditionException.

В итоге, воспользовавшись вышеописанными рекомендациями будет реализована общая архитектура тест скриптов и сценариев.

12.7 Проект по автоматизированному тестированию для начинающих

Рассмотрение данного проекта преследует следующую цель: принести пользу начинающему тестировщику — автоматизатору, помочь быстро создать первый авто-тест и продолжить автоматизировать.

Установка приложений и настройка проекта

Для работы потребуется:

- Git;
- Visual Studio 2010/2012;
- NUnit 2.6.1;
- Resharper (для удобства запуска тестов и дебага);
- Selenium IDE — плагин для firefox (для удобства распознавания локаторов элементов на страницах).

Чтобы загрузить проект из репозитория, необходимо в папке (где вы хотите его содержать) открыть git bash и выполнить команду: `git clone git://github.com/4gott3n/AT.git master`

Далее нужно открыть файл AT.sln в Visual Studio. Откроется Solution, содержащий проект AT (фреймворк), а также пример тестового проекта, в котором можно увидеть реализацию страниц, тестов и т.д. (он является удобным примером для создания своего проекта).

Следующий шаг — создание проекта, который будет хранить все тесты, страницы и все остальное.

Что нужно сделать:

- 1) **добавить в Solution новый проект (Class library). Это название позже нужно прописать в App.config;**
- 2) **подключить ссылки;**

В итоге должно быть как на Рисунке 12.1.

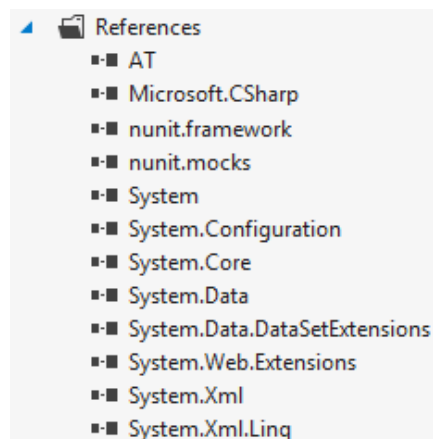


Рисунок 12.1 - Новый проект

3) создать файл конфигурации App.config;

В этом файле будут содержаться все основные параметры проекта.

Подробнее App.config:

```
<add key="project_name" value="SampleTestProject"/> <!-- название вашего проекта -->

<!-- блок настроек для рассылки отчета о запуске тестов -->

<add key="smtp_server" value="адрес smtp сервера" />
<add key="smtp_port" value="порт" />
<add key="smtp_login" value="логин" />
<add key="smtp_password" value="пароль" />
<add key="mail_from" value="ваш email" />
<add key="mail_to" value="список рассылки через запятую" />

<!-- end notif config -->
<!-- блок настроек браузера -->

<add key="ImplicitlyWait" value="время ожидания элемента (в секундах)" />
<add key="WaitForAjax" value="время ожидания выполнения ajax (в секундах)" />
<add key="browser" value="текущий браузер (firefox, chrome или iexplore)" />
<add key="browser_check_url" value="http://ya.ru" />

<!-- страница для проверка работы браузера -->
<!-- browser config end --> <!-- настройки страниц (webpages) -->

<add key="Yandex" value="http://yandex.ru/" /> <!-- соответствие названия папки
внутри WebPages главной странице тестируемой системы (в данном случае внутри папки
WebPages должна быть папка Yandex) -->

<add key="dash_prefix" value="___" />
<!-- обозначение тире в названии namespace (три нижних следа) -->

<add key="extension_prefix" value="___" /> <!-- префикс перед расширением файлов
страниц (два нижних следа), к примеру страница index-1.html должна иметь название
класса index___1_html -->

<add key="folder_index_prefix" value="_fld" />
<!-- открыть папку без имени файла страницы, это если нам нужно открыть папку, не
указывая названия файла, например страница https://yandex.ru/test/, в этом случае в
папке Yandex должна быть папка test, а в ней класс _fld -->

<!-- pages config end-->
<!-- блок настроек базы данных -->

<add key="database_name_prefix" value="db_" /> <!-- префикс строк конфига, содержащих
строки инициализации БД-->
<add key="database_selected_rows_limit" value="50" /> <!-- ограничение на выборку
(кол-во строк) -->

<add key="db_test" value="172.18.XX.XX;1521;SID;LOGIN;PASSWORD;oracle" /> <!-- пример
строки инициализации БД -->

<!-- формат: host;port;sid;login;password;type (type может быть либо oracle, либо
mssql-->

<!-- data base config end -->

<add key="test_step_prefix" value="step_" /> <!-- префикс в названиях методов для
шагов тест-кейса, к примеру step_01 -->
<add key="test_case_prefix" value="test_" /> <!-- префикс в названиях классов тест-
кейсов, к примеру test_000001 -->
```

```
<add key="datetime_string_format" value="yyyy-MMM-dd -> hh:mm:ss" /> <!-- формат даты для логирования -->  
<add key="log_file_name" value="system.log" /> <!--! название файла для логов -->
```

4) создать папку для страниц (WebPages);

В этой папке будут находиться файлы с классами страниц.

Подробнее. WebPages.

Несколько правил:

- папка со страницами должна называться WebPages;
- она должна находиться в корне проекта;
- внутри нее должны содержаться корневые папки ваших тестируемых системы (к примеру если тестировать Яндекс, то в папке WebPages должна быть папка yandex);
- страницы в папках должны находиться в той же иерархии, как они находятся в тестируемой системе.

Пример:

Страница: test.ru/step1/service/index.html

Иерархия папок: WebPages -> Test -> step1 -> service -> index.html.cs

Важно: имена папок и классов чувствительны к регистру

- Названия файлов классов страниц должны соответствовать реальным именам страниц.

Названия классов страниц:

Пример:

Страница: index-1.html

Класс: index__1__html

Здесь:

- дефис в названии страницы заменяется тремя слешами (можно изменить, параметр dash_prefix в App.config);
- точка файла заменяется двумя слешами (можно изменить, параметр extension_prefix в App.config).

Названия классов, когда нет названия файла, а есть только папка:

Пример:

Страница: test.ru/step1/service/

Класс: _fld (можно изменить, параметр folder_index_prefix в App.config)

5) создать класс Pages.cs;

```
public static class Pages { }
```

Класс будет содержать объекты классов страниц (WebPages)

Подробнее. Pages.cs

Этот класс содержит объекты классов страниц, через которые осуществляется доступ к элементам и т.д.

Несколько правил:

- в данном классе иерархия подклассов должна совпадать с иерархией в папке WebPages;
- класс и все объекты внутри него должны быть public static.

Пример класса:

```
public static class Pages
{
    public static class Test
    {
        public static index__html Index = new index__html();
        public static class step1
        {
            public static class service
            {
                public static _fld Main = new _ fld();
            }
            public static add__php Add = new add__php();
        }
    }
}
```

При такой записи страницы будут доступны в тестах примерно так:

```
Pages.Test.Index.Open();
Pages.Test.step1.service.Main.Open();
```

б) создать папку для тестов (Tests);

Здесь нет особых правил, достаточно создать в корне проекта папку Tests, внутри нее создать папки с тестами по тестируемым системам.

Подробнее. Tests

На рисунке 12.2 изображено как это сделано в проекте:

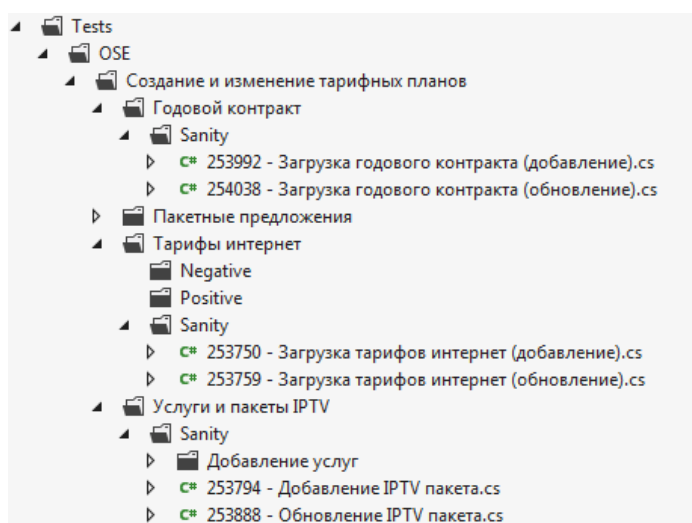


Рисунок 12.2 - Иерархия Tests

Рекомендуется создать два «служебных» теста, которые всегда будут запускаться первым и последним. В первом тесте можно выполнять различные действия по настройке среды, в

последнем, к примеру, выполнять очистку стенда и запускать нотификацию. *Nunit запускает тесты внутри категории в алфавитном порядке (по полному названию классов).*

Для запуска нотификации необходимо выполнить код:

```
AT.Service.Notifier.SendNotif();
```

Пример создания страницы (WebPages)

Все создаваемые страницы наследуются от базового класса PageBase, который содержит необходимые методы, одинаковые для всех страниц: «открыть», «открыть с параметром», «получить Url».

Пример класса страницы:

```
public class index_php : PageBase
{
    public void OpenVpdnTab()
    {
        new WebElement().ByXPath("//a[contains(@href, '#internet')]").Click();
    }
    public string VpdnAction
    {
        get
        {
            return
            WebElementSelect().ByXPath("//select[@name='action']").GetSelectedValue(); }
        set
        {
            new
            WebElementSelect().ByXPath("//select[@name='action']").SelectByValue(value); }
    }
    public string VpdnLid
    {
        set { new WebElement().ByXPath("//input[@name='lid']").SendKeys(value); }
    }
    public string VpdnTechlist
    {
        set { new WebElement().ByXPath("//input[@name='file']").SendKeys(value); }
    }
    public string VpdnStartDate
    {
        set
        {
            new
            WebElement().ByXPath("//input[@name='start_date']").SendKeys(value); }
    }
    public void VpdnSubmit()
    {
        new WebElement().ByXPath("//input[@value='Применить']").Click();
    }
}
```

Правило:

Все элементы, которые присутствуют на странице, должны инициализироваться в момент обращения к ним.

Примеры работы со страницами:

Pages.Test.Index.Open(); — открыть

Pages.Test.Index.Open("?id=1"); — открыть с параметром

var url = Pages.Test.Index.Url; — получение адреса страницы

Pages.Test.Index.VpdnSubmit(); — запуск функции, прописанной в классе страницы выше

Пример создания тест-кейса (Test)

Как уже отмечалось выше, все тест-кейсы должны лежать в папке Tests. Все тест-кейсы должны быть public и наследоваться от базового класса TestBase. Перед названием класса с тест-кейсом должен быть указан атрибут [TestFixture]. Перед каждым шагом тест-кейса должен быть указан атрибут [Test].

Подробнее об атрибутах тестов можно почитать в документации NUnit.

Пример класса с тест-кейсом:

```
namespace TestProject.Tests.OSE
{
    [TestFixture]
    [Category("OSE"), Category("OSE_Internet")] /* категории, nunit позволяет
запускать тесты выборочно по категориям */
    public class test_253750 : TestBase
    {
        [Test]
        public void step_01()
        {
            Pages.OSE.Inaclogin.Open();
            Pages.OSE.Inaclogin.Login = "user";
            Pages.OSE.Inaclogin.Password = "password";
            Pages.OSE.Inaclogin.Submit();

            Assertion("Ошибка авторизации", () => Assert.AreEqual(Pages.OSE.Inaclogin.IsAuthSuccess, true));
        }
    }
}
```

Assertion – это проверка выполнения условия.

Формат записи:

```
Assertion (текст ошибки, () => Assert.любой_assert_из_nunit() );
```

Почему не используем просто Assert?

Все Assert'ы отлавливаются специальным классом, в котором выполняются действия по регистрации ошибок, логирования и т.д.

Примеры работы с БД

Для работы с БД используется класс AT.DataBase.Executor, содержащий методы:

- выполнение запросов на выборку (select);
- выполнение запросов типа insert, update, delete (unselect);
- выполнение хранимых процедур.

Примеры:

Select:

```
var query = select coll1, col2 from table_name";  
var list = Executor.ExecuteSelect(query, Environment.Имя_БД);
```

Unselect:

```
var query = "DELETE FROM table_name";  
Executor.ExecuteUnSelect(query, Environment.Имя_БД);
```

Выполнение хранимой процедуры:

```
Executor.ProcedureParamList.Add(new ProcedureParam("varchar", "имя_параметра",  
«значение»)); /* входной параметр */  
Executor.ProcedureParamList.Add(new ProcedureParam("varchar")); /* возвращаемое  
значение */  
var res = Executor.ExecuteProcedure("имя_процедуры", Environment.Имя_БД);
```

имя_БД должно соответствовать значению sid из строки инициализации БД в App.config

```
<add key="db_test" value="172.18.XX.XX;1521;SID;LOGIN;PASSWORD;oracle" /> <!--  
пример строки инициализации БД -->
```

Сбор результатов тестирования и пример отчета:

Как уже упоминалось выше, для запуска нотификации и получения отчета на почту необходимо после запуска последнего теста выполнить код AT.Service.Notifier.SendNotif(). Логика NUnit такова, что он запускает тесты в алфавитном порядке, соответственно чтобы нужный тест запустился последним, его нужно назвать соответствующим образом. Настройки оповещения указываются в файле App.config.

На рисунке 12.3 изображен пример отчета.

Autotest Report

000001	Failed	step_01: error След. шаг: Шаг не выполнялся, ошибка на предыдущем шаге След. шаг: Шаг не выполнялся, ошибка на предыдущем шаге След. шаг: Шаг не выполнялся, ошибка на предыдущем шаге
--------	---------------	---

Рисунок 12.3 - Пример отчета

Выполнив все пункты этой инструкции юный тестировщик — автоматизатор сможет в короткие сроки создать свой проект и получить тот минимум, который необходим для начала этого пути.

12.8 Проект авто-тестов для веб-сервиса Эксперт

В заключении будет рассмотрен завершённый проект по автоматизированному тестированию для веб-сервиса Эксперт. «Эксперт» - это веб-сервис для оценки финансового состояния предприятия и оценки вероятности налоговой проверки.

Цель создания проекта: ускорить регрессионное тестирование и выпуск обновлений. Ускорить время прохождения всех тестов.

Изначально автоматизированные тесты были написаны на языке C#, но ввиду их медленного прохождения (из-за большого их количества, специфики самого веб-сервиса и то, что работать с ними можно было только под Windows) было принято решение переписать все тесты на языке Scala. Среда разработки - Eclipse. Технический драйвер, который связывает тесты и браузер - Selenium Web Driver. WebDriver – не инструмент автоматизации тестирования, а лишь средство управления браузером.

В качестве примера, для написания тестов, будет рассмотрена одна страница Эксперта, изображенная на рисунке 12.4: написание для нее инфраструктуры и тест кейсов. Настройки площадок и различных конфигурационных файлов, а также сам запуск тестов будут опущены.

Рисунок 12.4 - Данные об организации

12.8.1 Описание инфраструктуры страницы

Перед тем как начать писать тест-кейсы для страницы, ее нужно описать, т.е. создать инфраструктуру. Инфраструктура представляет собой описание тестируемых элементов рассматриваемой страницы. В данном случае это поля: названия организации, ОПФ, ОВД и денежный единицы; кнопки "назад" и "далее"; а также различные валидационные сообщения.

Чтобы описать инфраструктуру элементов страницы, нужно знать как обратиться к этому элементу. Для это может помочь консоль разработчика в браузере, изображенная на рисунке 12.5.

Создание нового элемента является одновременно проверкой того, что этот элемент есть на странице и он видим. Если элементы появляются после определённых действий или не все элементы могут одновременно присутствовать на странице, нужно использовать ленивость с помощью модификатора `lazy`. В таком случае, первое обращение к элементу будет одновременно проверкой его существования на странице.

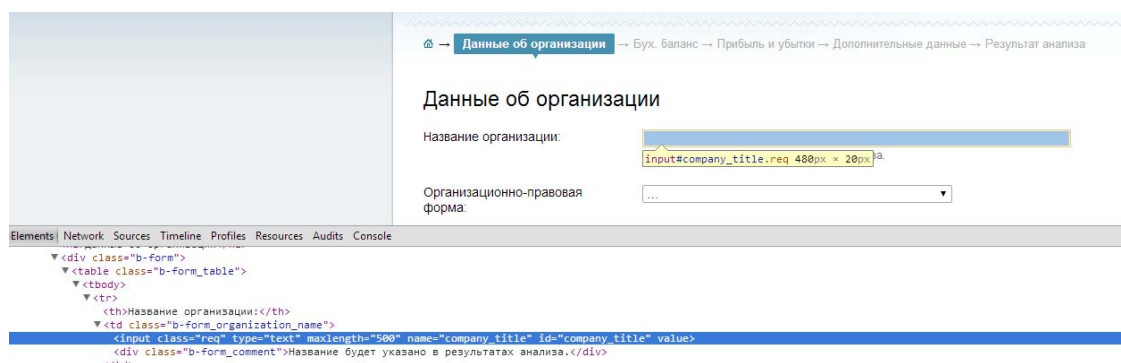


Рисунок 12.5 - Консоль разработчика

Узнав как обращаться к элементу, можно представить это в коде:

```
val companyTitle = new ExpertInput(ByJQ("input#company_title"))
```

Т.к. в поле "название организации" происходит ввод данных, то используется элемент `ExpertInput` и запрос `JQ`, чтобы можно было определить данное поле на страницы.

По тому же принципу описывается элемент `ОПФ`.

```
val orgType: ExpertSelect[OrgTypes.type] = new ExpertSelect(ByJQ("select#org_type"))
```

`ОПФ` является списком, который изображен на рисунке 12.6, поэтому используется уже не `ExpertInput`, а `ExpertSelect`. При этом нужно описать все элементы этого списка, для этого используется объект `OrgTypes`:

```
object OrgTypes extends StringEnumeration {  
  val empty = StringValue("...")  
  val OOO = StringValue("Общество с ограниченной ответственностью")  
  val OAO = StringValue("Открытое акционерное общество")  
  val ZAO = StringValue("Закрытое акционерное общество")  
  val UP = StringValue("Унитарное предприятие")  
  val ODO = StringValue("Общество с дополнительной ответственностью")  
  val PT = StringValue("Полное товарищество")  
  val TV = StringValue("Товарищество на вере")  
  val PK = StringValue("Производственный кооператив")  
  val KKH = StringValue("Крестьянское (фермерское) хозяйство")  
  val NKO = StringValue("Некоммерческая организация")  
  val IP = StringValue("Индивидуальный предприниматель")  
  val OBPUL = StringValue("Организация без прав юридического лица")  
}
```

Организационно-правовая форма:	...
Основной вид деятельности:	Общество с ограниченной ответственностью Открытое акционерное общество Закрытое акционерное общество Унитарное предприятие Общество с дополнительной ответственностью Полное товарищество Товарищество на вере
В полях с суммами использовать единицы:	

Рисунок 12.6 - Список ОПФ

Денежные единицы тоже являются списком, но т.к. он состоит всего из двух элементов, то принято решение описать выбор между элементами как 2 функции.

```
def chooseThousandsUnit : FaCompanyInfoPage[Intro, Result] = {
  unitOfMeasure.select(UnitsOfMeasure.thousands)
  new FaCompanyInfoPage(url)
}

def chooseMillionsUnit : FaCompanyInfoPage[Intro, Result] = {
  unitOfMeasure.select(UnitsOfMeasure.millions)
  new FaCompanyInfoPage(url)
}
```

На странице также есть и скрытые элементы, показанные на рисунке 12.7, которые появятся на ней только при определенных условиях. В данном случае - это валидационные сообщения. Их тоже необходимо описать, т.к. все тест кейсы будут нацелены на проверку валидации.

Данные об организации

Название организации:	<input type="text"/>	Укажите название
	Название будет указано в результатах анализа.	
Организационно-правовая форма:	...	Заполните поле
Основной вид деятельности:	Выбрать...	Укажите основной вид деятельности

Рисунок 12.7 - Скрытые элементы

Код для текста валидации:

```
val companyNameValidation = "Укажите название"
val orgTypeValidation = "Заполните поле"
val onvdCodeValidation = "Укажите основной вид деятельности"
lazy val moneyUnitValidation = "Переключение единиц измерения невозможно – значения должны быть целыми числами"
```

Функции проверки валидации:

```
def checkCompanyNameValidation() = checkText(ByJQ("label[for='company_title']"),
companyNameValidation)

def checkOrgTypeValidation() = checkText(ByJQ("label[for='org_type']"), orgTypeValidation)

def checkOnvdCodeValidation() = checkText(ByJQ("label[for='onvd_code']"),
onvdCodeValidation)

def checkMoneyValidation() {
  moneyUnitElement.text should be(moneyUnitValidation)
}
```

Полное описание страницы можно увидеть в приложении А.

12.8.2 Написание тест кейсов

Рассмотрим простой сценарий: после того как пользователь попадает на страницу "Данные об организации" он нажимает кнопку "Далее" не заполнив ни одного поля. Ожидаемый результат: сработает валидация на все поля. После этого он заполнит одно поле - валидация со всех полей исчезает.

Код в этом случае выглядит следующим образом:

Т.к. для каждого теста нужно находится на страницу "Данные об организации", то нужно повторяющийся код вывести в функцию. В данном случае это шаги выбора финансового анализа, периода и возможность заполнения полей вручную.

```
protected def setup = {  
  IntroPage().new_fa.proceed().chooseManual.selectPeriod(Years.year2013, PeriodTypes.year)
```

После идет описание шагов в код:

```
"Click next with empty fields than fill onvd code - all fields" should "be valid" in {  
  val companyInfoPage = setup  
  companyInfoPage.nextPageInactive.click()  
  companyInfoPage.checkCompanyNameValidation()  
  companyInfoPage.checkOnvdCodeValidation()  
  companyInfoPage.checkOrgTypeValidation()  
  
  companyInfoPage.onvdCodeSelectorLink.proceed().checkOnvd("15")  
  withWait() {new ExpertText(ByJQ("label[for='company_title']"))}  
  withWait() {new ExpertText(ByJQ("label[for='org_type']"))}  
  withWait() {new ExpertText(ByJQ("label[for='onvd_code']"))}  
}
```

Шаги:

- срабатывает повторяющийся код - открывается страница "Данные об организации";
- нажатие на кнопку "Далее";
- проверка, что появилась валидация для полей названия организации, ОПФ, ОВД;
- выбирается из списка ОВД значение;
- проверяется, что валидация со всех трех полей исчезает.

Рассмотрим еще один сценарий: пользователь заполняет все поля на странице "Данные об организации", нажимает кнопку "Далее" и переходит на следующую страницу. После этого он нажимает на кнопку "Назад" и попадает на предыдущую страницу, при этом все поля должны быть заполнены старыми значениями.

Код:

```
"Fill all fields, click next than return back - all fields" should "be filled" in {  
  val companyInfoPage = setup  
  companyInfoPage.companyTitle.text = OrgName.name  
  companyInfoPage.orgType.select(OrgTypes.ODO)
```

```
companyInfoPage.onvdCodeSelectorLink.proceed().checkOnvd("15").nextPage.proceed().backwardLink.proceed()
```

```
companyInfoPage.companyTitle.text should be(OrgName.name)  
companyInfoPage.orgType.getSelected should be(OrgTypes.ODO.toString())  
companyInfoPage.onvd.text should be("15. Производство пищевых продуктов, включая напитки")  
}
```

Шаги:

- срабатывает повторяющийся код - открывается страница "Данные об организации";
- заполняются поля "название организации, ОПФ, ОВД;
- нажимается кнопка "Далее";
- после перехода на новую страницу нажимается кнопка "Назад";
- проверка, что все поля заполнены старыми значениями.

Весь набор тест кейсов для данной страницы можно посмотреть в приложении Б.

В итоге, данный проект сильно ускорил регрессионное тестирование и выпуск обновлений. Скорость прохождения тестов увеличена почти в 2 раза. Перед каждым апдейтом тесты проходят на тестовой площадке и только после их успешного выполнения происходит обновление на "боевой".

ЗАКЛЮЧЕНИЕ

В ходе выполнения работы был изучен процесс тестирования программного обеспечения в IT-компаниях. Были рассмотрены большинство видов дефектов в ПО, причины возникновения и способы их отслеживания, с помощью систем отслеживания ошибок. Также изучены способы создания и применения тест кейсов для черного ящика.

Данный материал можно использовать как для учебных целей, так и для успешного прохождения технического собеседования для устройства на работу в качестве тестировщика. Также по данному материалу можно оформить методическое пособие для студентов и включить в программу обучения в качестве дополнительного курса "Тестирование ПО".

Во второй части работы рассмотрены программы для нагрузочного тестирования. С помощью программы JMeter проведено нагрузочное тестирование веб-сервиса "Эксперт", в котором было обнаружено слабое звено в системе.

Были разработаны 2 проекта по автоматизированному функциональному тестированию. Первый проект учебный и несет ознакомительный характер. Он нацелен на пользу начинающему тестировщику — автоматизатору, помочь быстро создать первый авто-тест и продолжить автоматизировать.

Второй, - проект по авто-тестам для веб-сервиса "Эксперт", который сейчас используется на боевом сервере. Он сильно ускорил регрессионное тестирование и выпуск обновлений. Перед каждым апдейтом тесты проходят на тестовой площадке и только после их успешного выполнения происходит обновление на "боевой".

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Библиотека MSDN. Источник информации для разработчиков, использующих средства, продукты, технологии и службы корпорации Майкрософт. [Электронный ресурс]. – Режим доступа: <http://msdn.microsoft.com>
2. КиберФорум. Форум начинающих и профессиональных программистов, системных администраторов, администраторов баз данных. [Электронный ресурс]. – Режим доступа: <http://cyberforum.ru>
3. ИНТУИТ. Интернет университет информационных технологий. [Электронный ресурс]. – Режим доступа: <http://intuit.ru>
4. Клуб программистов. [Электронный ресурс]. – Режим доступа: <http://www.cyberguru.ru/>
5. Википедия. Свободная энциклопедия. [Электронный ресурс]. – Режим доступа: <http://ru.wikipedia.org/wiki>
6. Тестирование и качество ПО. [Электронный ресурс]. - Режим доступа: <http://software-testing.ru/>
7. Про тестинг. [Электронный ресурс]. Режим доступа: <http://www.protesting.ru/>
8. SQA Days. [Электронный ресурс]. Режим доступа: <http://sqadays.com/>
9. Software testing training and software testing services. [Электронный ресурс]. Режим доступа: <http://www.rbc-us.com/>
10. Уиттакер Д., Арбон Д., Каролло Д. Как тестирует Google.: Пер. с англ. - Спб.: Издательский дом "Питер", 2014.-320с
11. Савин Р. Тестирование Дот Ком, или Пособие по жесткому обращению с багами в интернет-стартапах. - М.: Дело, 2007. - 312с
12. Хабрахабр. [Электронный ресурс]. Режим доступа: <http://habrahabr.ru/>
13. Code Project. Сообщество разработки программного обеспечения. [Электронный ресурс]. – Режим доступа: <http://codeproject.com>
14. Apache Software Foundation. [Электронный ресурс]. - Режим доступа: <http://jmeter.apache.org/>
15. SeleniumHQ Browser Automation. [Электронный ресурс]. - Режим доступа: <http://docs.seleniumhq.org/>

ПРИЛОЖЕНИЕ А

Программный код, описывающий инфраструктуру страницы "Данные об организации":

```
package pages.fa_pages

import pages._
import intro_pages._
import org.openqa.selenium.WebDriver
import queries._
import expert_elements._
import org.scalatest.Matchers
import common._
import popups_and_lightboxes._
import utils.StringEnumeration
import pages.pages_constants.SkipperConstants
import org.scalatest.selenium.WebBrowser
import pages.pages_constants._

object OrgTypes extends StringEnumeration {
  val empty = StringValue("...")
  val OOO = StringValue("Общество с ограниченной ответственностью")
  val OAO = StringValue("Открытое акционерное общество")
  val ZAO = StringValue("Закрытое акционерное общество")
  val UP = StringValue("Унитарное предприятие")
  val ODO = StringValue("Общество с дополнительной ответственностью")
  val PT = StringValue("Полное товарищество")
  val TV = StringValue("Товарищество на вере")
  val PK = StringValue("Производственный кооператив")
  val KKH = StringValue("Крестьянское (фермерское) хозяйство")
  val NKO = StringValue("Некоммерческая организация")
  val IP = StringValue("Индивидуальный предприниматель")
  val OBPUL = StringValue("Организация без прав юридического лица")
}

trait FaCompanyInfoPageBackground[Intro <: ExpertPage, Result <: ExpertPage] extends
BackgroundPageComponent {
  type BackgroundPage = FaCompanyInfoPage[Intro, Result]
  val backgroundPageConstructor = implicitly[Constructable[FaCompanyInfoPage[Intro,
Result]]].construct
}

trait FaCompanyInfoPageSaved[Intro <: ExpertPage, Result <: ExpertPage] extends
SavedPageComponent {
  type SavedPage = FaCompanyInfoPage[Intro, Result]
  val savedPageConstructor = new FaCompanyInfoPage[Intro, Result](_: String)
}

class FaOnvdSelector[Intro <: ExpertPage, Result <: ExpertPage]()(implicit val driver :
WebDriver)
  extends OnvdSelectorLightBoxComponent
  with FaCompanyInfoPageBackground[Intro, Result]
  with FaCompanyInfoPageSaved[Intro, Result] {
    val lightbox = new OnvdSelectorLightBox(_)
  }

/*
 * YearDependentPage - if 2010, then Fa2010BalancePage, otherwise - FaBalancePage
 */
```

```

class FaCompanyInfoPage[Intro <: ExpertPage, Result <: ExpertPage]
  (url : String, val isPeriodSelected : Boolean = false)(implicit val driver: WebDriver)
  extends FaFormPage(url) with Breadcrumbs[Intro] with Matchers
  with IncompleteSaver with ExpertEventually {
    val companyTitle = new ExpertInput(ByJQ("input#company_title"))
    val orgType: ExpertSelect[OrgTypes.type] = new ExpertSelect(ByJQ("select#org_type"))
    val onvd = new ExpertText(ByJQ("#selected_business_type_title"))

    protected lazy val faOnvdSelector = new FaOnvdSelector[Intro, Result]
    lazy val onvdCodeSelectorLink = new ExpertLink(ById("select_business_type")) with
Connector[faOnvdSelector.OnvdSelectorLightBox] {
      val construction = faOnvdSelector.lightbox
    }
    lazy val onvdSelectedCodeSelectorLink = new ExpertLink(ById("change_business_type")) with
Connector[faOnvdSelector.OnvdSelectorLightBox] {
      val construction = faOnvdSelector.lightbox
    }

    val wizardStepPage = "Данные об организации"
    val companyNameValidation = "Укажите название"
    val orgTypeValidation = "Заполните поле"
    val onvdCodeValidation = "Укажите основной вид деятельности"
    lazy val moneyUnitValidation = "Переключение единиц измерения невозможно – значения должны
быть целыми числами"

    def checkCompanyNameValidation() = checkText(ByJQ("label[for='company_title']"),
companyNameValidation)

    def checkOrgTypeValidation() = checkText(ByJQ("label[for='org_type']"), orgTypeValidation)

    def checkOnvdCodeValidation() = checkText(ByJQ("label[for='onvd_code']"),
onvdCodeValidation)

    private def checkText(textElementSelector: ExpertQuery, expectedText: String) {
      new ExpertText(textElementSelector).text should be(expectedText)
    }

    override def accept() {
      checkText(ByJQ("div#company_info h1"), wizardStepPage)
      currentStep should be("Данные об организации")
    }

    lazy val moneyUnitElement = new ExpertText(ByJQ(".units-error"))
    lazy val unitOfMeasure = new ExpertSelect[UnitsOfMeasure.type](ById("units"))

    def chooseThousandsUnit : FaCompanyInfoPage[Intro, Result] = {
      unitOfMeasure.select(UnitsOfMeasure.thousands)
      new FaCompanyInfoPage(url)
    }

    def chooseMillionsUnit : FaCompanyInfoPage[Intro, Result] = {
      unitOfMeasure.select(UnitsOfMeasure.millions)
      new FaCompanyInfoPage(url)
    }

    def checkMoneyValidation() {
      moneyUnitElement.text should be(moneyUnitValidation)
    }

    lazy val nextPageInactive = new ExpertButton(ByJQ("#buttons_pane a.b-button-small"))
  }

```

```

object FaCompanyInfoPage extends WebBrowser {
  type CompanyPageTypeConstructor[Intro <: ExpertPage, ResultPage[_ <: ExpertPage] <:
    ExpertPage] =
    FaCompanyInfoPage[Intro, ResultPage[Intro]]

  def apply[Intro <: ExpertPage, Result <: ExpertPage, YearDependentPage <: ExpertPage](url :
    String)
    (implicit driver : WebDriver) =
      new FaCompanyInfoPage[Intro, Result](url)

  implicit def FaCompanyInfoPageToConstructable[Intro <: ExpertPage, Result <: ExpertPage]
    (implicit driver : WebDriver) =
      new Constructable[FaCompanyInfoPage[Intro, Result]] {
        def construct = new FaCompanyInfoPage[Intro, Result](url : String)
      }

  def FaCompanyInfoPageDefault[Intro <: ExpertPage, Result <: ExpertPage](url : String)
    (implicit driver: WebDriver) =
      new FaCompanyInfoPage[Intro, Result](url)(driver)

  type DefaultType[Intro <: ExpertPage, Result <: ExpertPage] = FaCompanyInfoPage[Intro,
    Result]

  implicit def FaCompanyInfoPageToNextStep[Intro <: ExpertPage, Result <: ExpertPage]
    (current : FaCompanyInfoPage[Intro, Result])(implicit driver : WebDriver) =
      new FaCompanyInfoPage[Intro, Result](current.url)
      with WizardNextStep[FaCompanyInfoPage[Intro, Result], FaBalancePage[Intro, Result]] {
        override val nextPageConstructor = implicitly[Constructable[FaBalancePage[Intro,
          Result]]].construct
      }

  implicit def FaCompanyInfoPageToPrevStep[Intro <: ExpertPage : Constructable, Result <:
    ExpertPage]
    (current : FaCompanyInfoPage[Intro, Result])(implicit driver : WebDriver) =
      new FaCompanyInfoPage[Intro, Result](current.url)
      with WizardPrevStep[FaCompanyInfoPage[Intro, Result], Intro] {
        override protected val prevPageConstructor = implicitly[Constructable[Intro]].construct
      }
}

```

ПРИЛОЖЕНИЕ Б

Программный код, который содержит набор тест кейсов для страницы "Данные об организации":

```
package test_cases.fa_tests

import test_cases._
import pages.intro_pages._
import pages.fa_pages._
import expert_elements._
import queries._
import pages.pages_constants._

class FaCompanyInfoSpec extends ExpertSpec {
  protected def setup = {
    IntroPage().new_fa.proceed().chooseManual.selectPeriod(Years.year2013, PeriodTypes.year)
  }

  "Click next with empty fields than fill onvd code - all fields" should "be valid" in {
    val companyInfoPage = setup
    companyInfoPage.nextPageInactive.click()
    companyInfoPage.checkCompanyNameValidation()
    companyInfoPage.checkOnvdCodeValidation()
    companyInfoPage.checkOrgTypeValidation()

    companyInfoPage.onvdCodeSelectorLink.proceed().checkOnvd("15")
    withWait() {new ExpertText(ByJQ("label[for='company_title']"))}
    withWait() {new ExpertText(ByJQ("label[for='org_type']"))}
    withWait() {new ExpertText(ByJQ("label[for='onvd_code']"))}
  }

  "Fill all fields except onvd - validation" should "appear on onvd field" in {
    val companyInfoPage = setup
    companyInfoPage.companyTitle.text = OrgName.name
    companyInfoPage.orgType.select(OrgTypes.ODO)
    companyInfoPage.nextPageInactive.click()
    companyInfoPage.checkOnvdCodeValidation()
  }

  "Fill all fields, click next than return back - all fields" should "be filled" in {
    val companyInfoPage = setup
    companyInfoPage.companyTitle.text = OrgName.name
    companyInfoPage.orgType.select(OrgTypes.ODO)

    companyInfoPage.onvdCodeSelectorLink.proceed().checkOnvd("15").nextPage.proceed().backwardLink.proceed()

    companyInfoPage.companyTitle.text should be(OrgName.name)
    companyInfoPage.orgType.getSelected should be(OrgTypes.ODO.toString())
    companyInfoPage.onvd.text should be("15. Производство пищевых продуктов, включая напитки")
  }

  "Click next with empty fields, fill OrgType than reset and select again OrgType - all fields" should "be valid" in {
    val companyInfoPage = setup
```

```

companyInfoPage.nextPageInactive.click()
companyInfoPage.orgType.select(OrgTypes.ODO)
withWait() {new ExpertText(ByJQ("label[for='company_title']"))}
withWait() {new ExpertText(ByJQ("label[for='org_type']"))}
withWait() {new ExpertText(ByJQ("label[for='onvd_code']"))}

companyInfoPage.orgType.select(OrgTypes.empty)
companyInfoPage.nextPageInactive.click()

companyInfoPage.orgType.select(OrgTypes.ODO)
withWait() {new ExpertText(ByJQ("label[for='company_title']"))}
withWait() {new ExpertText(ByJQ("label[for='org_type']"))}
withWait() {new ExpertText(ByJQ("label[for='onvd_code']"))}
}
}

```