МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕУЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ДИСЦИПЛИНЕ ТЕСТИРОВАНИЕ И ОТЛАДКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Направление подготовки 09.03.04 Программная инженерия
Направленность (профиль) Инженерия сложных программных систем

Ставрополь

2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Методические указания к выполнению лабораторных работ составлены в соответствии с рабочей программой дисциплины «Тестирование и отладка программного обеспечения» и предназначены для студентов направления подготовки 09.03.04 Программная инженерия.

Целью освоения дисциплины «Тестирование и отладка программного обеспечения» является формирование у студентов профессиональных компетенций, в области методов, инструментов и процессов тестирования и отладки программного обеспечения. Эта дисциплина направлена на подготовку специалистов, способных решать задачи, связанные с на обеспечение качества, надежности и соответствия разрабатываемых программных продуктов заданным требованиям в различных сферах деятельности, а также компетенций будущего бакалавра по направлению подготовки 09.03.04 Программная инженерия.

1. Перечень планируемых результатов обучения по дисциплин (модулю), соотнесенных с планируемыми результатами освоения образовательной программы

Код, формулировка компетенции	Код, формулировка индикатора	Планируемые результаты обучения по дисциплине (модулю), характеризующие этапы формирования компетенций, индикаторов
ПК-14 – Способность разрабатывать компоненты системных программных продуктов и автоматизированных систем	ИД-2$_{ПК-14}$ Применяет методы и алгоритмы оптимизации исполняемого кода. ИД-4$_{ПК-14}$ Способен разрабатывать программное обеспечение для автоматизированных систем. ИД-6$_{ПК-14}$ Способен разрабатывать программное обеспечение для	Умеет анализировать требования к ПО и проектировать тестовые сценарии. Определяет приоритеты тестирования на основе анализа требований. Рассчитывает и интерпретирует метрики тестирования. Понимает особенности разработки и тестирования компонентов роботизированных систем, систем автономного управления, в том числе для БПЛА, контроллеров и систем связи БАС.

	роботизированных систем, систем автономного управления, в том числе для БПЛА, контроллеров и систем связи БАС.	
ПК-15 – Способность проводить отладку и тестирование компонентов программного обеспечения и конечных программных продуктов и ИС	ИД-3$_{ПК-15}$ Применяет модели тестирования ПО и методики тестирования разрабатываемого программного обеспечения ИД-4$_{ПК-15}$ Применяет методики тестирования разрабатываемого программного обеспечения ИД-6$_{ПК-15}$ Осуществляет отладку программных продуктов для целевой операционной системы ИД-7$_{ПК-15}$ Проводить тестирование компонентов программного обеспечения ИС по заданным сценариям ИД-8$_{ПК-15}$ Оценивает важность различных тестов на основе приоритетов пользователя, проектных задач и рисков возникновения ошибки	Понимает принципы тестирования и отладки ПО. Знает виды тестирования (модульное, интеграционное, системное, нагрузочное). Знает методы отладки (трассировка, логирование, профилирование). Владеет основами тестовой документации (Test Plan, Test Case, Bug Report). Умеет разрабатывать тест-кейсы и чек-листы.

ЛАБОРАТОРНАЯ РАБОТА № 1.

Тестирование программного обеспечения: разработка тестов

Цель: разработать рабочую тестовую документацию для тестирования web приложения.

План занятия:

1. Изучить теоретические сведения.

2. Выполнить практическое задание по лабораторной работе.

3. Оформить отчёт и ответить на контрольные вопросы.

Теоретические сведения

Рабочая тестовая документация значительно улучшает качество последующего тестирования за счет анализа и детального планирования тестов. После завершения тестирования наличие тестовой документации позволяет оценить, насколько успешно были проведены все этапы тестирования, а для заказчика является подтверждением реального объема работ.

Рабочую тестовую документацию тестировщик может разрабатывать исключительно на основе спецификации еще до поставки программного обеспечения. В этом случае после поставки на тестирование версии программного продукта специалист по тестированию может сразу приступить к поиску дефектов.

Существуют следующие виды рабочей тестовой документации (таблица 1):

1. Check List.

2. Acceptance Sheet.

3. Test Survey.

4. Test Cases.

Основные факторы выбора тестовой документации – сложность бизнес- логики проекта, сроки проекта, размер команды и объем проекта.

На одном проекте могут комбинироваться несколько типов тестовой документации. Например, для всего проекта составлен Acceptance Sheet, но для наиболее сложных частей составлены Test Cases. Если какие-либо модули программного продукта будут подвергаться автоматизированному тестированию, то для таких модулей в обязательном порядке составляются Test Cases.

Таблица 1 – Виды рабочей тестовой документации и их характеристика

Тип документации	Что описывают	Когда используют
Checklist	Вспомогательный тип документации, содержащий список основных проверок.	Для типовой функциональности.
Acceptance Sheet	Перечень всех модулей и функций приложения, подлежащих проверке.	Небольшие (до 3 месяцев), простые по бизнес-логике проекты.
Test Survey	Перечень всех модулей и функций, а также конкретные проверки для них. Может содержать ожидаемый результат.	Средние или большие проекты с понятной бизнес- логикой.
Test Cases	Набор входных значений, предусловий, пошаговое описание и постусловия для каждой проверки. Всегда содержит ожидаемый результат.	Большие и долгосрочные проекты, проекты со сложной бизнес-логикой, проекты с большой командой.

Примеры фрагментов рабочей тестовой документации приведены в таблице 2.

При составлении рабочей тестовой документации необходимо указать номер тестируемой сборки, тип выполняемой тестовой активности, период времени тестирования, ФИО тестировщика, тестовое окружение (операционная система, браузер, др.).

Рабочая тестовая документация представляет собой перечень всех проверок для модулей/подмодулей приложения. В качестве одного модуля как правило выступает рабочее окно приложения, в качестве подмодулей – логически завершенные блоки этого окна.

Для каждого модуля в обязательном порядке выполняется тестирование GUI, а также общие функциональные проверки (General). Далее в рамках модуля в качестве функциональных проверок выступают действия над активными элементами пользовательского интерфейса (полями, кнопками, чекбоксами и т.д.). Степень детализации каждой из таких функциональных проверок зависит от выбранного типа тестовой документации (Acceptance Sheet, Test Survey, Test Cases). В частности, для Acceptance Sheet все активные элементы пользовательского интерфейса только перечисляют. Для Test Survey для каждого элемента приводят позитивные и негативные проверки, источником которых являются базовые проверки (в виде чеклиста) для соответствующих элементов GUI. Для Test Cases каждую из позитивных и негативных проверок описывают в виде последовательности шагов с указанием ожидаемого результата.

Для Test Survey, Test Cases напротив каждой проверки указывается глубина тестирования: Smoke, MAT, AT. Для Acceptance Sheet в качестве глубины тестирования всегда указывается AT.

Таблица 2 – Примеры рабочей тестовой документации

Checklist	Acceptance Sheet	Test Survey	Test Cases
Протестировать форму авторизации	Форма авторизации: 1. GUI. 2. General. 3. Поле «Эл.адрес или телефон». 4. Поле «Пароль». 5. Кнопка «Войти». 6.Чекбокс «Не выходить из системы». 7. Ссылка «Забыли пароль».	Форма авторизации: 1. GUI. 2. General. 3. Валидный эл.адрес + валидный пароль. 4. Валидный телефон + валидный пароль. 5. Валидный эл.адрес + невалидный пароль. 6. Валидный телефон + невалидный пароль. 7. Невалидный эл.адрес или телефон + валидный пароль. 8. Невалидный эл.адрес или телефон + невалидный пароль. 9. Запомнить данные: выйти из системы и зайти обратно. 10.Ссылка «Забыли пароль».	Авторизация с помощью e-mail: 1. Открыть страницу abc.com. 2. Ввести в поле «Эл.адрес или телефон» e-mail abc@mail.ru. 3. Ввести в поле «Пароль» пароль qwerty. 4.Нажать на кнопку «Войти». Ожидаемый результат: пользователь переходит на свою домашнюю страницу.

Далее рассмотрим подробно базовые проверки графического интерфейса пользователя и функциональности web, desktop и mobile приложений.

Для любого приложения выполняется тестирование графического интерфейса пользователя (таблица 3).

Таблица 3 – Перечень основных GUI проверок для всего приложения

Название проверки	Описание проверки
1. Правописание	Лексические, грамматические и пунктуационные ошибки
2. Расположение и выравнивание	Выравнивание по левому или правому краю (в зависимости от требований приложения), отступы, идентичность расстояний между названием и полем. Корректное расположение текста, длинный текст не выходит за границы поля при вводе.
3. Длинные названия	Длинные названия корректно обрезаются с помощью многоточия в конце, при наведении возникают хинты с полнотекстовым вариантом.
4. Соответствие названий форм / элементов GUI их назначению	Проверка названий форм / элементов GUI с точки зрения их смысловой нагрузки.
5. Унификация (стиля, цвета, шрифта, названий)	Единообразие цвета, шрифта, размеров (высоты/ширины), выравнивания полей, названий полей, категорий меню и др. в рамках всего приложения.
6. Эффект «нажатия»	Изменение вида ссылок, кнопок, позиций меню и др. при наведении курсора. Изменение вида курсора при наведении на ссылки, кнопки, позиции меню и др.
7. Хинты	Проверка всплывающих подсказок с точки зрения правописания, выравнивания, соответствия назначению и др.
8. Сообщения об успешном / неуспешном завершении действия,	Проверка верхней панели (логотипа и названия) формы с сообщением.

о подтверждении действия	Если присутствует кнопка «Отмена», то в правом верхнем углу формы с сообщением присутствует «крестик» для альтернативной возможности закрыть форму. Сообщения о подтверждении удаления по умолчанию активированы на кнопку «нет».
9. Изменение размеров окна, изменение масштаба страницы	Появление скроллинга при уменьшении размера окна. Сохранение взаимного расположения элементов при уменьшении окна, изменении масштаба). Перераспределение элементов с сохранением пропорций при изменении масштаба страницы.

Общие проверки функциональности для всех типов приложений, а также общие проверки для web приложений приведены в таблицах 4, 5.

Таблица 4 – Перечень общих проверок функциональности для любого типа приложения

Название проверки	Описание проверки
1. Табуляция	Перемещение с помощью клавиатуры должно осуществляться сверху вниз слева направо. Недоступные поля должны пропускаться.
2. «Хлебные крошки»	«Хлебные крошки» – элемент навигации, являющийся признаком удобства пользования приложением в целом и перемещением по его структуре.
3. Скроллинг	Отсутствие скроллинга в случае, если текст вмещается на странице без прокрутки. Соответствующее изменения текста при использовании скроллинга. Возможность изменения положения скроллинга при помощи мыши, кнопок Page up/down, Home/End.
4. Взаимосвязь компонентов	Поведение одного компонента при изменении/удалении другого (например, при удалении категории товара не должны удаляться все товары в этой категории).

5. Фокус на кнопке для исполнения действий	Ввод данных → нажатие Enter → действие осуществилось.

Таблица 5–Перечень общих проверок функциональности для web приложений

Название проверки	Описание проверки
1. Подготовка к тестированию	Перед тестированием каждой новой сборки необходимо осуществить очистку кэша и cookies. Для этого можно воспользоваться приложением CCleaner.
2. 404 Error	Переход по некорректному адресу должен вести на страницу с Error 404, а не на страницу Page cannot be found, например. Страница Error 404 должна быть реализована в общем дизайне тестируемого приложения.
3. Логотип	Логотип должен быть ссылкой на главную страницу.
4. Email нотификации	Проверка работоспособности отправки email нотификаций (как администратору, так и пользователю), если только отсутствие писем не является спецификой проекта.
5. Проверка работоспособности приложения при отключенном JavaScript	Основная функциональность и навигация должна работать.

Для любого приложения проверяют функциональность элементов пользовательского интерфейса: поле для ввода данных, поле для загрузки файлов, поле для ввода даты, поле со списком/выпадающий список, кнопка, радиобаттон, чекбокс, меню, таблица, календарь, ссылка, сообщения, поп-ап (всплывающие окна). В таблицах 6-15 приведены основные проверки для указанных элементов пользовательского интерфейса.

Таблица 6 – Перечень основных проверок для поля ввода данных

Functional Test	GUI Test

1. Обязательность ввода. 2. Обработка только пробелов. 3. Использование пробелов в тексте (3.1. пробелы в начале и в конце строки должны отсекаться при сохранении, 3.2. пробелы внутри текста отсекаться не должны). 4. Минимально/максимально допустимое количество символов. 5. Формат данных (исходя из его логического назначения и требований приложения). 6. Формат числовых данных (если допускаются): негативные, дробные с точкой и запятой. 7. Использование специальных символов (введенные символы должны отобразиться в том же виде, в котором они были введены, если только ввод спец. символов не запрещен требованиями приложения). 8. Возможность редактирования введенных значений. 9. Корректное распределение текста по строкам (переход на новую строку автоматически). 10. Уникальные данные(например, уникальность логина, email). 11. Автоматическая постановка курсора в первое поле для ввода при открытии формы. 12. Ввод тегов и скриптов (введенные теги и скрипты должны отобразиться в том же виде, в котором они были введены).	1. Название поля (правописание, соответствие названия тематике модуля/страницы). 2. Выравнивание названий полей (выравнивание по левому или правому краю в зависимости от требований приложения, отступы, идентичность расстояний между названием и полем). 3. Корректное расположение текста, длинный текст не выходит за границы поля при вводе. 4. Унификация дизайна по отношению ко всему приложению (цвет, шрифт, размер (высота/ширина), выравнивание полей). 5. Расположение вводимого текста внутри поля (унификация, выравнивание).

Таблица 7 – Перечень основных проверок для поля загрузки файлов

Functional Test	GUI Test
1. Обязательность выбора файла. 2. Форматы: корректные/некорректные. 3. Корректный формат, но отсутствует/модифицировано расширение. 4. Ограничения на размер (включая загрузку файлов нулевого размера, большого размера).	1. Унификация дизайна по отношению ко всему приложению (цвет, шрифт, высота/ширина). 2. Выравнивание названий загруженных файлов.

5. Загрузка исполняемых файлов (EXE, PHP, JSP др.). 6. Загрузка переименованного EXE-файла. 7. Путь к файлу меньше 259 символов. 8. Путь к файлу равен 260 символов. 9. Путь к файлу больше 260 символов. 10. Корректный путь введен с клавиатуры. 11. Имитировать сбой загрузки (например, с использованием флешки). 12. Одновременная загрузка нескольких файлов.	

Таблица 8 – Перечень основных проверок для радиобаттона

Functional Test	GUI Test
1. Функциональность: включение/выключение. 2. Не может быть меньше двух радиокнопок. 3. По умолчанию одна радиокнопка должна быть включена. 4. Не может быть включено более одной радиокнопки. 5. При переходе на следующую страницу и возвращении назад выбранная радиокнопка не должна сбрасываться. 6. Активация путем нажатия как на символ, так и на текст.	1.Унификация дизайна по отношению ко всему приложению. 2. Выравнивание расположения радиобаттона с соответствующим названием. 3. Выравнивание расположений радиобаттонов. 4.Изменение радиобаттона при наведении курсора. 5.Изменение курсора при наведении на радиобаттон.

Таблица 9 – Перечень основных проверок для чекбоксов

Functional Test	GUI Test
1. Функциональность: включение/выключение. 2. Наличие дополнительного чекбокса, выставляющего/снимающего все чекбоксы при наличии больше 10 чекбоксов.	1. Унификация дизайна по отношению ко всему приложению. 2. Выравнивание расположения чекбокса с соответствующим названием.

3. При переходе на следующую страницу и возвращении назад выбранный чекбокс не должен сбрасываться. 4. Активация путем нажатия как на символ, так и на текст.	3. Изменение чекбокса при наведении курсора. 4. Изменение курсора при наведении на чекбокс.

Таблица 10 – Перечень основных проверок для поля со списком

Functional Test	GUI Test
1. Сортировка по алфавиту или по смыслу. 2. В случае, если значения выходят за границы списка, и нет возможности увеличения размера списка, то необходимо отображение хинтов (всплывающих подсказок). 3. Выбор пункта списка по нажатию соответствующей первой буквы на клавиатуре. 4. Возможность введения значений вручную (если это позволяет приложение). 5. Возможность выбора значения из списка как с помощью мыши, так и с клавиатуры.	1. Правописание. 2. Подсветка при выборе каждого из значений, при выборе нескольких значений одновременно. 3. Унификация дизайна (цвет, шрифт, размер (высота/ширина), цвет подсветки, выравнивание).

Таблица 11 – Перечень основных проверок для меню

Functional Test	GUI Test
1. Осуществление соответствующего перехода при выборе пункта меню. 2. Визуальное различие в момент работы на определенной вкладке (подсветка, подчеркивание).	1. Подсветка категории меню при наведении курсора. 2. Изменение курсора при наведении на категорию меню. 3. Если в данный момент выполняется работа в выбранной вкладке, то в меню она отличается визуально и является некликабельной. 4. Совпадение названий категорий меню в случае, если меню дублируется в нескольких местах.

Таблица 12 – Перечень основных проверок для ссылки

Functional Test	GUI Test

1. Функционирование ссылки (долженосуществиться переход на соответствующую страницу). 2. Переход по загруженной ссылкедолжен осуществляться в новой вкладке или в всплывающем окне. 3. Форматы ссылок и префиксов. 4. Срабатывание ссылки только при клике на саму ссылку, а не на пустую область возле нее.	1.Унификация стилей (в соответствие с дизайном сайта). 2.Расположение ссылок (в соответствие с дизайном сайта). Например, расположение всех ссылок слева или справа от элементов. 3.Названия (унификация, идентичность названий ссылок одинакового назначения, спеллинг, соответствие с открытым модулем или страницей, вместимость названия ссылки в отведенном блоке). 4.Измененение вида курсора при наведении на ссылку. 5.Изменение вида ссылки при наведении курсора (подчеркивание).

Таблица 13 – Перечень основных проверок для таблицы

Functional Test	GUI Test
1. При появлении нескольких страниц есть кпопки Вперед, Назад, На первую, На последнюю страницу (пагинация). 2. Проверка сортировок, в том числе сортировки по умолчанию 3. Обновление значений таблицы после добавления, изменения, удаления данных. 4. Единичное/множественное выделение нескольких значений.	1. Унификация дизайна для всего приложения (цвет, шрифт, размер (высота/ширина), выравнивание). 2. Название (соответствие с текущим модулем, спеллинг). 3. Выравнивание иконок сортировки в названии колонок. 4. Выравнивание названий колонок, значений внутри таблицы. 5. Корректное отображение длинных названий (соответствующие переходы на новые строки, сокращение названий (появление многоточия либо сокращение по слову). 6. Корректное отображение данных после использования сортировки (размеры колонок и столбцов фиксированы, текст не разбивает структуру таблицы).

Таблица 14 – Перечень основных проверок для календаря

Functional Test	GUI Test
1. Ввод даты с помощью календаря. 2. Ввод даты вручную: проверить разные форматы, номер месяца: > 12, день: > 31 (+ для февраля). 3. Логика работы поля (например, подсчет возраста после ввода даты рождения; невозможность ввести дату рождения свыше текущего дня и т.д.).	1. Унификация дизайна для всего приложения (цвет, шрифт, размер (высота/ширина), выравнивание). 2. Отображение календаря рядом с полем. 3. Корректное выравнивание всех элементов и ссылок в календаре.

Таблица 15 – Перечень основных проверок для сообщений

Functional Test	GUI Test
1. Пользователь должен быть информирован о действиях, происходящих в системе посредством сообщений об успешном завершении операции. 2. На необратимые действия, такие как удаление, должны быть подтверждающие сообщения. 3. Введенные в форму данные не должны сбрасываться после появления сообщения.	1. Правописание сообщений. 2. Соответствие сообщений смыслу выполняемого действия. 3. Соответствие названий полей в сообщениях названиям полей, форм, таблиц, кнопок, и т.д. 4. Унификация стилей (цвет, размер) для всего приложения. 5. Если присутствует кнопка «Отмена», то в правом верхнем углу формы с сообщением присутствует «крестик» для альтернативной возможности закрыть форму. 6. Сообщения о подтверждении удаления по умолчанию активированы на кнопку «нет». 7. Соответствие цвета типу сообщения (красный для сообщений об ошибках, зеленый для сообщений об успешном завершении операции), если это не противоречит специфичным требованиям приложения. 8. Невалидное значение не должно отображаться в сообщении об ошибке (неправильно: "Email 2309234@@mail.ru не соответствует допустимому формату”). 9. Согласование числительного и связанного с ним существительного должно соблюдаться (например, "1 день", "2 дня", "5 дней").

Практическое задание:

1. Осуществить тестирование web-приложения https://playground.learnqa.ru/puzzle/triangle

2. По ходу выполнения тестирования сформировать Checklist, Acceptance Sheet, Test Cases.

3. Описать разницу в полученных документах.

4. Оформить отчет и защитить лабораторную работу.

Содержание отчета:

1. Цель работы.

2. Рабочая тестовая документация.

3. Выводы по работе.

4. Ответы на контрольные вопросы.

Контрольные вопросы:

1. Какие существуют разновидности рабочей тестовой документации?

2. Check List: что описывают и когда используют?

3. Acceptance Sheet: что описывают и когда используют?

4. Test Survey: что описывают и когда используют?

5. Test Cases: что описывают и когда используют?

6. От чего зависит степень детализации каждой функциональной проверки?

7. Какая глубина тестрирования указывается для проверок в Acceptance Sheet?

8. Какая глубина тестрирования указывается для проверок в Test Survey, Test Cases?

9. Какие проверки выполняют при тестировании GUI?

10. Какие общие функциональные проверки выполняют для всего приложения?

11. Какие общие функциональные проверки выполняют для web приложения?

12. Перечислите базовые проверки для поля ввода данных.

13. Перечислите базовые проверки для поля загрузки файлов.

14. Перечислите базовые проверки для ввода даты.

15. Перечислите базовые проверки для поля со списком.

16. Перечислите базовые проверки для радиобаттона.

17. Перечислите базовые проверки для чекбокса.

18. Перечислите базовые проверки для меню.

19. Перечислите базовые проверки для таблиц.

20. Перечислите базовые проверки для ссылок.

21. Перечислите базовые проверки для сообщений.

ЛАБОРАТОРНАЯ РАБОТА № 2

Тестирование программного обеспечения: разработка тестов

Цель: протестировать web-форму и описать найденные дефекты.

План занятия:

1. Изучить теоретические сведения.

2. Выполнить практическое задание по лабораторной работе.

3. Оформить отчёт и ответить на контрольные вопросы.

Теоретические сведения

Дефекты, обнаруженные тестировщиком, должны быть корректно и понятно описаны, чтобы разработчик смог воспроизвести данный дефект и устранить его. Описание каждого дефекта сохраняется в специализированной – багтрэкинговой – системе (например, JIRA, Bugzilla, Mantis, Redmine и др.) или в предварительно созданном в программной среде Microsoft Excel файле (пример приведен в таблице 1).

Таблица 1 – Пример описания дефекта

№	Название дефекта	Важ-ность	Алгоритм воспроизведения	Фактич е- ский результат	Ожида-емый результат	Прил о-жени е	Прим е-чание
39	Администраторская часть: Файлы: Выбор файла: Ссылка "Скачать файл": Администра-тор не имеет возможности скачать загруженные студентом файлы.	Critical	Шаги по воспроизведению: 1.Входим на web сайт … . 2.Проходим авторизацию: admin/admin. 3.Выбираем в меню категорию "Файлы".	Переход к странице "HTTP Status 404".	Происх о- дит скачива- ние выбран-ного файла.	39.png	REQ-26

			4.Выбираем подкатегорию меню "Выбор файла". 5.Выбираем в поле "Обзор файла" любой доступный файл. 6.Нажимаем на ссылку "Скачать файл".				

Описание дефекта включает следующие обязательные поля:

1. Headline – название дефекта.

2. Severity – степень критичности (важность дефекта).

3. Description – алгоритм воспроизведения.

4. Result – фактический результат.

5. Expected result – ожидаемый результат.

6. Attachment – прикреплённые файлы (приложение).

В багтрэкинговых системах для каждого дефекта автоматически генерируется его уникальный номер, в случае использования Microsoft Excel номер дефекту необходимо присваивать вручную.

Требование спецификации, которое нарушает обнаруженный дефект, можно дополнительно вынести в примечание.

Дополнительно в описании дефекта может быть указана Priority – степень срочности исправления дефекта разработчиком.

Рассмотрим подробно каждую категорию описания дефекта.

Headline (название дефекта). Цель составления заголовка дефекта – предоставить краткую и в тоже время понятную информацию о том, где, что и в результате чего произошло. Характеристиками качественного заголовка являются краткость, информативность, точная идентификация проблемы.

Заголовок дефекта должен отвечать на три вопроса:

1. Где? В каком месте интерфейса пользователя находится проблема.

В данной части заголовка следует также дополнительно указать особенности теста, если это поможет разобраться в проблеме (версия операционной системы, браузера, сторонних приложений, которые имеют отношение к тестируемому программному средству).

2. Что? Что происходит или не происходит согласно спецификации или представлению о нормальной работе программного продукта. При этом необходимо указывать на наличие проблемы, а не на ее содержание (его указывают в описании). Если содержание проблемы варьируется, все известные варианты указываются в описании.

3. Когда (при каких условиях)? В какой момент работы программы или по наступлению какого события проблема проявляется.

Пример: в приложении есть диалог «Преобразовать данные» с кнопкой

«Преобразовать». При нажатии этой кнопки появляется сообщение об ошибке

«Ошибка 315». Заголовок дефекта по описанной методике составляется так: Где?: Диалог «Преобразовать данные».

Что?: Показывается сообщение об ошибке.

Когда?: При нажатии кнопки «Преобразовать».

Итоговый заголовок будет иметь следующий вид: Диалог "Преобразовать данные" показывает сообщение об ошибке при нажатии кнопки "Преобразовать". Уберём лишние слова, добавим код ошибки для удобства поиска: Диалог

«Преобразовать данные»:сообщение об ошибке 315при нажатии кнопки «Преобразовать».

Если сформулировать заголовок по формуле, Где? Что? Когда (при каких условиях)? трудно, то можно воспользоваться следующим алгоритмом действий:

1. Пропустите заголовок дефекта.

2. Напишите описание дефекта, фактический и ожидаемый результаты.

3. Выделите ключевые моменты, руководствуясь формулой Где? Что? Когда (при каких условиях)?

4. Сложите эти ключевые моменты вместе.

5. То, что получится в итоге, и будет составлять заголовок дефекта.

Severity (степень критичности). Степень критичности (серьезности, важности) показывает степень ущерба, который наносится проекту существованием дефекта. В общем случае выделяют следующие градации критичности дефектов (таблица 2): Critical (критический), Major (значительный), Average (средней значимости), Minor (незначительный), Enhancement (предложение по улучшению). Description(алгоритм воспроизведения). Цель составления алгоритма воспроизведения дефекта – последовательно описать шаги для повторения

дефекта. Description должен быть оформлен в виде списка перечисления действий:

1.Шаг #1.

2.Шаг #2.

…

n. Шаг #n.

В случае, если для воспроизведения дефекта требуется ряд начальных условий (например, должен быть создан определенный набор документов, пользователь или группа пользователей с особыми правами), то эти предусловия должны быть вынесены в начало описания:

Предусловия.

1.Шаг #1.

2.Шаг #2.

…

n. Шаг #n.

Таблица 2 – Степени критичности дефектов

Severity	Описание	Примеры
Critical (критический)	Функциональная ошибка, которая блокирует работу части функционала или всего приложения. Функциональная ошибка, которая нарушает ключевую (с точки зрения конечного пользователя или бизнеса заказчика) функциональность приложения.	Заблокированавкладка категория меню). Неправильно подсчитывается итоговая сумма при вводе скидочного промокода. Раскрывается конфиденциальная информация.
Major (значительный)	Функциональная ошибка, которая нарушает нормальную работу приложения, но не блокирует работу части функционала в целом.	Невозможно загрузить видео-файлы на персональной страничке.

		Не работает система e-mail нотификации. Не работает интеграция с социальными сетями. Необходимо перезапускать приложение при выполнении типичных сценариев работы
Average (средней значимости)	Не очень важная функциональная ошибка. Критичные дефекты GUI.	Не работает сортировка. «Уехал» текст за пределы окна.
Minor (незначительный)	Редко встречающиеся незначительные функциональные дефекты. 90 % дефектов GUI.	Введены необязательные для заполнения поля, которых нет в спецификации. Грамматические, пунктуационные ошибки.
Enhancement (предложение по улучшению)	Функциональные предложения, советы по улучшению дизайна (оформления), навигации и др. Такой дефект является необязательным для исправления.	Добавить кнопку «Наверх» на длинных формах. Увеличить размер шрифта.

При составлении Description необходимо следовать приведенным ниже рекомендациям.

Description – это четкий алгоритм, в котором приветствуются короткие, понятные фразы и нумерация.

Нельзя использовать личные предложения формата «Я думаю, что так будет лучше», «Я зашел на страницу…» и т.д.

Можно использовать специальные символы «+», «=», «<>» которые помогут сделать подобие навигации: File > Open, DOC + XLS. Однако не рекомендуется писать шаги в строку через символы перехода: это затрудняет восприятие дефекта. Следует указывать специфичные условия воспроизведения дефекта,

если таковые имеются. Например, под каким пользователем вы работаете (если это важно).

Описание предложений по улучшению должно быть максимально полным и аргументированным.

Result (фактический результат). Цель написания Result – четко описать полученный результат.

Expected result (ожидаемый результат). Цель написания Expected result – привести аргументы разработчикам, как именно должно работать приложение.

В Expected result должно быть четкое обоснование, почему именно так должно работать приложение. Лучше всего, если в нем приведена ссылка на конкретный пункт спецификации.

Если в Expected result приводится ссылка на спецификацию, сам тестировщик дополнительно цитирует текст спецификации, чтобы сократить время разработчика на анализ документа и однозначно указать на способ решения проблемы.

Если функция работает, но некорректно, то вExpected Result обязательно должно быть описание того, как она должна работать корректно.

Если сделана ошибка в надписи или интерфейсе проекта, необходимо грамотно и полностью указать, как она должна быть исправлена – написать надпись без ошибки или описать требуемые изменения интерфейса.

Expected Result всегда следует заполнять. Не стоит полагаться на очевидность представлений о правильном поведении приложения.

Текст Expected result рекомендуется писать законченными полными безличными предложениями.

Attachment (приложения). Attachment – это прикрепленный к дефекту файл, дополняющий описание: скриншот, файлы, необходимые для воспроизведения дефекта, логи программы, видео ошибки и т.д. Attachment является вспомогательным средством передачи информации о проблеме. Для всех GUI дефектов attachment обязателен.

Если к дефекту прикрепляется файл, об этом обязательно должно быть указано в описании дефекта («See the file/screenshot/log/video attached»). Прикрепленный файл не должен быть слишком большим по размеру (особенно это касается видео: до 10 мегабайт), а также должен относиться именно к описанной ошибке (например, из лога приложения стоит скопировать в прикрепляемый файл только данные об ошибке). Формат файла скриншота – PNG. Имя файла скриншота рекомендуется делать числовым, нейтральным – 1.png, 25.png и т.п.

Скриншот должен содержать следующие элементы: сама ошибка, выделение места ошибки, стрелка к прямоугольнику, описание ошибки: «Наблюдаемый результат» и/или «Ожидаемый результат». Текст на скриншоте также необходимо выделить: обвести в прямоугольник и набрать шрифтом, заметно отличающимся

от шрифта программы. Качественно подготовленный скриншот должен давать возможность понять смысл дефекта без необходимости читать его описание.

Делать снимок всего окна программы необязательно: на снимке должна быть видна ошибка и место, в котором она находится. Если для понимания дефекта необходим контекст, то помимо собственно ошибки фиксируют необходимую информацию (браузер, в котором открыто web приложение, всё окно программы в фоне с названием диалогового окна, где эта ошибка появилась). Если необходимо привести снимки нескольких страниц проекта, связанных между собой, лучше сделать это на одном скриншоте, совместив изображения по горизонтали и при необходимости отметив стрелками переходы значений, полей и т.п.

Далее приведены рекомендации по описанию дефектов.

Часто сообщение об ошибке превращается в сокращённую запись только основных действий, необходимых для воспроизведения ошибки, опуская все несущественные. Но, будучи незнакомым с внутренней структурой приложения, тестировщик не может знать, какие из выполненных им действий наиболее существенны для диагностирования данной ошибки. Пренебрегая действиями, которые кажутся незначительным, повышается риск потери важной информации. Лучший способ избежать этой проблемы состоит в том, чтобы просто перечислить все действия, которые необходимы для воспроизведения ошибочного поведения, начиная с открытия нужной формы в проекте.

Если есть подозрение на повторение дефекта в нескольких модулях проекта, этот факт нужно исследовать еще до внесения дефекта и при его описании указать все места, где дефект воспроизводится.

Дефект не должен содержать фразу: «Это не работает», дефект должен показать, что и при каких условиях не работает.

Чтобы внести дефект, его следует воспроизвести минимум два раза, причем начиная с самых нейтральных условий воспроизведения, и только после гарантированного повторения описать последовательность действий.

Нельзя не описывать дефекты только потому, что их не получается воспроизвести. Факт невозможности выяснить причину дефекта в таком случае обязательно должен быть указан в описании дефекта.

Дефекты целесообразно группировать, однако делать это необходимо в соответствии с приведенными ниже правилами.

GUI дефекты могут группироваться в один по признаку формы, на которой они находятся, т.е. если одна форма содержит несколько GUI дефектов с одинаковым уровнем Severity, то их можно объединить.

Функциональные дефекты группируются в том случае, если речь идет об однотипных дефектах, которые воспроизводятся в различных модулях, страницах или полях (например, динамическое обновление не работает в модулях 1, 2 и 4 или отсутствует валидация на спецсимволы на всех полях страницы).

Группировка функциональных дефектов по признаку формы, на которой они найдены, не применяется.

Недопустимо объединять в один дефекты разного типа, например, функциональные и GUI. В таком случае пишутся несколько дефектов на каждый тип.

Рекомендации по хорошему описанию дефектов:

1.Шаги воспроизведения, фактический и ожидаемый результаты должны быть подробно описаны.

2.Дефект должен быть понятно описан (с использованием общеупотребимой лексики, точных названий программных средств).

3.Необходимо давать ссылку на соответствующее требование, к нарушению которого приводит фактический результат работы программного средства.

4.Если существует какая-либо информация, которая поможет быстрее обнаружить или исправить дефект, необходимо сообщить эту информацию.

5.Окружение (ОС, браузер, настройки и т.п.), под которым возникла ошибка, должно быть четко указано.

6.Создавать дефект и описывать его необходимо сразу же, как только он был обнаружен. Откладывание «на потом» приводит к риску забыть о дефекте или и каких-либо деталях его воспроизведения. Несвоевременное создание дефекта не позволяет проектной команде реагировать на ее обнаружение в реальном времени.

7.После описания дефекта необходимо еще раз перечитать его: убедится, что все необходимые поля заполнены, и все написано верно.

Помимо, собственно, описания дефектов результаты тестирования вносят в рабочую тестовую документацию (Acceptance Sheet, Test Survey, Test Cases). Для этого напротив выполненной проверки указывают степень критичности обнаруженного дефекта, его номер и заголовок. Если по результатам конкретной проверки выявлено несколько дефектов, то перечисляют номера всех дефектов, а в качестве степени критичности и заголовка указывают наиболее серьезный дефект (рисунок 5.2).

Практическое задание:

1. Перейти на сайт http://testingchallenges.thetestingmap.org/index.php

2. Подготовить Check-list, acceptance sheet или test-case template для проведения тестирования.

3. Подготовить план тестирования формы ввода данных (согласно материалу прошлой лабораторной работы).

4. Провести тестирование в разделе Testing Challenge #1 - web testing. Классифицировать найденные дефекты (Critical, Major, Average, Minor, Enhancement).

5. Протестировать web приложение в соответствии с составленной тестовой документацией.

6. Описать все найденные дефекты в отчете (Check-list, acceptance sheet или test-case template).

7. В отчете о дефектах указать номер тестируемой сборки (v 1.0), название сайта, период времени тестирования, ФИО тестировщика, тестовое окружение (операционная система, браузер).

8. Для каждого дефекта указать его порядовый номер, заголовок, важность, алгоритм воспроизведения, фактический результат, ожидаемый результат, приложение, примечание.

9. Для каждого дефекта обязательно сделать скриншоты.

10.В рабочую тестовую документацию внести результаты тестирования с указанием напротив соответствующей проверки степени критичности обнаруженного дефекта, его номера и заголовка.

Содержание отчета:

1.Цель работы.

2.Отчет о результатах тестирования с перечислением тестовых проверок, сформулированных дефектов на каждый уровень Severity, описания дефектов.

3.Ответы на контрольные вопросы.

4.Выводы по работе.

Контрольные вопросы:

1. Что такое дефект?

2. Какие характеристики необходимо указать при описании дефекта?

3. Что такое Headline/Summary в описании дефекта?

4. На какие три вопроса должен отвечать Headline/Summary?

5. Что такое Severity в описании дефекта?

6. Какие существуют степени Severity? Приведите примеры.

7. Что такое Description в описании дефекта?

8. Что такое Expected result в описании дефекта?

9. Зачем нужен Attachment при описании дефекта?

10. Какие существуют рекомендации по описанию дефектов?

ЛАБОРАТОРНАЯ РАБОТА № 3

Требования к идеальному критерию тестирования

Цель работы: освоить требования к идеальному критерию тестирования

Теоретическое обоснование

Для нетривиальных классов программ в общем случае не существует полного и надежного критерия, зависящего от программ или спецификаций**.**

Поэтому мы стремимся к идеальному общему критерию через реальные частные.

Структурные критерии (класс I) — используют модель программы в виде "белого ящика", что предполагает знание исходного текста программы или спецификации программы в виде потокового графа управления. Структурная информация понятна и доступна разработчикам подсистем и модулей приложения, поэтому данный класс критериев часто используется на этапах модульного и интеграционного тестирования (Unit testing, Integration testing).

Структурные критерии базируются на основных элементах УГП, операторах, ветвях и путях.

Условие *критерия тестирования команд* (критерий C0) — набор тестов в совокупности должен обеспечить прохождение каждой команды не менее одного раза. Это слабый критерий, он, как правило, используется в больших программных системах, где другие критерии применить невозможно.

Условие *критерия тестирования ветвей* (критерий C1) — набор тестов в совокупности должен обеспечить прохождение каждой ветви не менее одного раза. Это достаточно сильный и при этом экономичный критерий, поскольку множество ветвей в тестируемом приложении конечно и не так уж велико. Данный критерий часто используется в системах автоматизации тестирования.

Условие *критерия тестирования путей* (критерий C2) — набор тестов в совокупности должен обеспечить прохождение каждого пути не менее 1 раз. Если программа содержит цикл (в особенности с неявно заданным числом итераций), то число итераций ограничивается константой (часто — 2, или числом классов выходных путей).

На пример 1 приведен пример простой программы. Рассмотрим условия ее тестирования в соответствии со структурными критериями.

```
1 public void Method (ref int x)

{
2 if (x>17)
3 x = 17-x;
4 if (x==-13)
5 x = 0;
6 }
```

Пример 2. Пример простой программы, для тестирования по структурным критериям.

```
1 void Method (int *x)

{
2 if (*x>17)
3 *x = 17-*x;
4 if (*x==-13)
5 *x = 0;
6 }
```

Пример 3. Пример простой программы, для тестирования по структурным критериям (html, txt)

Тестовый набор из одного теста, удовлетворяет критерию команд (C0): (X,Y)={(хвх=30, хвых=0)} покрывает все операторы трассы 1-2-3-4-5-6 Тестовый набор из двух тестов, удовлетворяет критерию ветвей (C1):

(X,Y)={(30,0), (17,17)} добавляет 1 тест к множеству тестов для C0 и трассу 1-2-4-6. Трасса 1-2-3-4-5-6 проходит через все ветви достижимые в операторах if при условии true, а трасса 1-2-4-6 через все ветви, достижимые в операторах if при условии false.

Тестовый набор из четырех тестов, удовлетворяет критерию путей (C2): (X,Y)={(30,0), (17,17), (-13,0), (21,-4)}

Набор условий для двух операторов if с метками 2 и 4 приведен в таблица 1

Таблица 1 – Условие if

	(30,0)	(17,17)	(-13,0)	(21,-4)
2 if (x>17)	>	≤	≤	>
4 if (x==-13)	=	≠	=	≠

Критерий ветвей C2 проверяет программу более тщательно, чем критерии — C1, однако даже если он удовлетворен, нет оснований утверждать, что программа реализована в соответствии со спецификацией.

Например, если спецификация задает условие, что|x|100, невыполнимость которого можно подтвердить на тесте (-177,-177). Действительно, операторы 3 и 4 на тесте (-177,- 177) не изменят величину x=-177 и результат не будет соответствовать спецификации. Структурные критерии не проверяют соответствие спецификации, если оно не отражено в структуре программы. Поэтому при успешном тестировании программы по критерию C2 мы можем не заметить ошибку, связанную с невыполнением некоторых условий спецификации требований.

ЛАБОРАТОРНАЯ РАБОТА № 4

Мутационный критерий (класс IV).

Постулируется, что профессиональные программисты пишут сразу почти правильные программы, отличающиеся от правильных мелкими ошибками или описками типа — перестановка местами максимальных значений индексов в описании массивов, ошибки в знаках арифметических операций, занижение или завышение границы цикла на 1 и т.п. Предлагается подход, позволяющий на основе мелких ошибок оценить общее число ошибок, оставшихся в программе.

Подход базируется на следующих понятиях:

Мутации — мелкие ошибки в программе.

Мутанты — программы, отличающиеся друг от друга мутациями.

Метод мутационного тестирования — в разрабатываемую программу P вносят мутации, т.е. искусственно создают программы-мутанты P1, P2... Затем программа P и ее мутанты тестируются на одном и том же наборе тестов (X,Y).

Если на наборе (X,Y) подтверждается правильность программы P и, кроме того, выявляются все внесенные в программы-мутанты ошибки, то набор тестов (X,Y) соответствует мутационному критерию, а тестируемая программа объявляется правильной.

Если некоторые мутанты не выявили всех мутаций, то надо расширять набор тестов (X,Y) и продолжать тестирование.

Пример применения мутационного критерия

Тестируемая программа P приведена на пример 1. Для нее создается две программы- мутанта P1 и P2.

В P1 изменено начальное значение переменной z с 1 на 2.

В P2 изменено начальное значение переменной i с 1 на 0 и граничное значение индекса цикла с n на n-1 .

При запуске тестов (X,Y) = {(x=2,n=3,y=8),(x=999,n=1,y=999), (x=0,n=100,y=0 } выявляются все ошибки в программах-мутантах и ошибка в основной программе, где в условии цикла вместо n стоит n-1:

```
// Метод вычисляет неотрицательную
// степень n числа x
static public double PowerNonNeg( double x, int n)
{
double z=1; if (n>0)
{
for (int i=1;n-1>=i;i++)
{
z = z*x;
}
}
else Console.WriteLine(
"Ошибка ! Степень числа n должна быть больше 0."); return z;
}
```

Пример 2. Основная программа P double PowerNonNeg(double x, int n)

```
{
double z=1; int i;
if (n>0)
{
for (i=1;n-1>=i;i++)
{
z = z*x;
тоном:
}
}
else printf(
"Ошибка ! Степень числа n должна быть больше 0.\n"); return z;
```

```
}
```

Пример 3. Основная программа P

Измененное начальное значение переменной z в мутанте P1 помечено светлым

```
// Метод вычисляет неотрицательную
// степень n числа x
static public double PowerMutant1( double x, int n)
{
double z=2; if (n>0)
{
for (int i=1;n>=i;i++)
{
z = z*x;
}
}
else Console.WriteLine(
"Ошибка ! Степень числа n должна быть больше 0."); return z;
}
```

Пример 4. Программа мутант P1. (html, txt) double PowerMutant1(double x, int n)

```
{
double z=2; int i;
if (n>0)
{
for (i=1;n>=i;i++)
{
z = z*x;
}
}
else printf(
```

"Ошибка ! Степень числа n должна быть больше 0.\n"); return z;

}

Пример 5. Программа мутант P1.

Измененное начальное значение переменной i и границы цикла в мутанте P2 помечено светлым тоном:

// Метод вычисляет неотрицательную

// степень n числа x

static public double PowerMutant2(double x, int n)

{

double z=1; if (n>0)

{

for (int i=0;n-1>=i;i++)

{

z = z*x;

}

}

else Console.WriteLine(

"Ошибка ! Степень числа n должна быть больше 0"); return z;

}

Пример 6. Программа-мутант P2. (html, txt) double PowerMutant2(double x, int n)

{

double z=1; int i;

if (n>0)

{

for (i=0;n-1>=i;i++)

{

z = z*x;

}

}

```
else printf(
"Ошибка! Степень числа n должна быть больше 0.\n"); return z;
}
```

ЛАБОРАТОРНАЯ РАБОТА № 5

Цель работы: научиться проводить оценку покрытия программы и проекта

Теоретическое обоснование

Тестирование программы P по некоторому критерию C означает покрытие множества компонентов программы P M = {m1...mk} по элементам или по связям

T = {t1...tn} — кортеж неизбыточных тестов ti.

Тест ti неизбыточен, если существует покрытый им компонент mi из M(P,C), не покрытый ни одним из предыдущих тестов t1...ti-1. Каждому ti соответствует неизбыточный путь pi — последовательность вершин от входа до выхода.

V(P,C) — сложность тестирования P по критерию C — измеряется max числом неизбыточных тестов, покрывающих все элементы множества M(P,C)

DV(P,C,T) — остаточная сложность тестирования P по критерию C — измеряется max числом неизбыточных тестов, покрывающих элементы множества M(P,C), оставшиеся непокрытыми, после прогона набора тестов T. Величина DV строго и монотонно убывает от V до 0.

TV(P,C,T) = (V-DV)/V — оценка степени тестированности P по критерию C.

Критерий окончания тестирования TV(P,C,T) L, где (0 L 1). L — уровень оттестированности, заданный в требованиях к программному продукту.

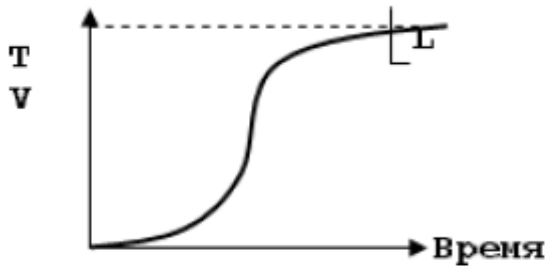

Рисунок 1 - Метрика оттестированности приложения

Рассмотрим две модели программного обеспечения, используемые при оценке оттестированности.

Для оценки степени оттестированности часто используется УГП — управляющий граф программы. УГП многокомпонентного объекта G (Рисунок 2, Пример 1), содержит внутри себя два компонента G1 и G2, УГП которых раскрыты

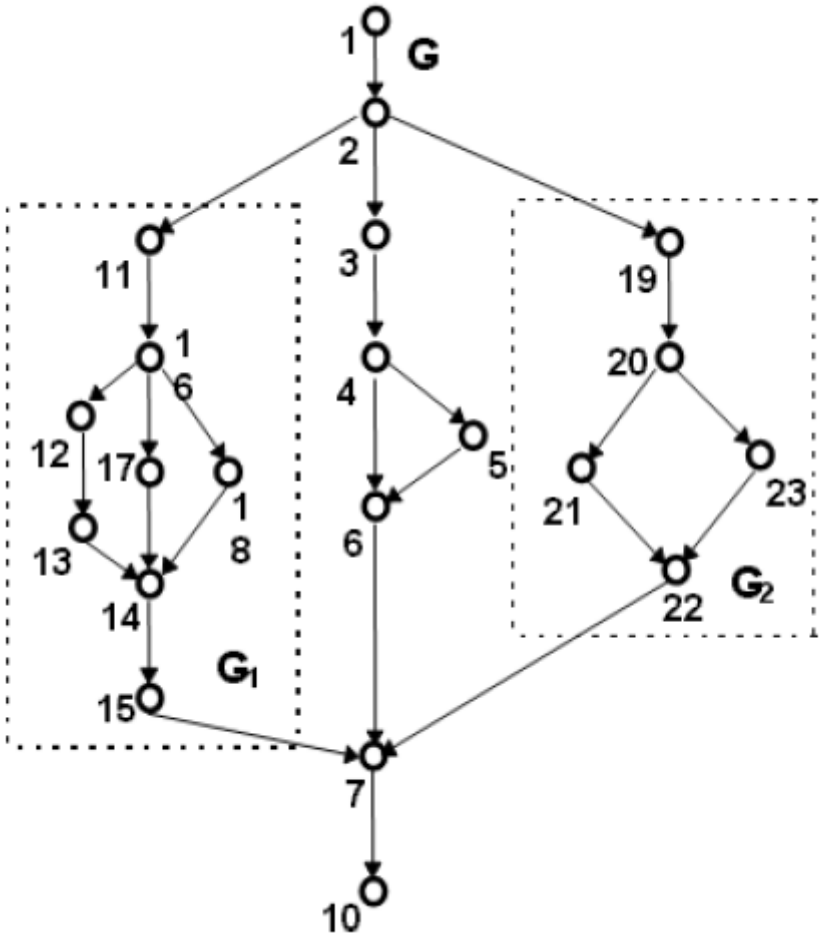

Рисунок 2 - Плоская модель УГП компонента G

В результате УГП компонента G имеет такой вид, как если бы компоненты G1 и G2 в его структуре специально не выделялись, а УГП компонентов G1 и G2 были вставлены в УГП G. Для тестирования компонента G в соответствии с критерием путей потребуется прогнать тестовый набор, покрывающий следующий набор трасс графа G (Пример 1):

P1(G) = 1-2-3-4-5-6-7-10;

P2(G) = 1-2-3-4-6-7-10;

P3(G) = 1-2-11-16-18-14-15-7-10;

P4(G) = 1-2-11-16-17-14-15-7-10;

P5(G) = 1-2-11-16-12-13-14-15-7-10;

P6(G) = 1-2-19-20-23-22-7-10;

P7(G) = 1-2-19-20-21-22-7-10;

Пример 1. Набор трасс, необходимых для покрытия плоской модели УГП компонента G

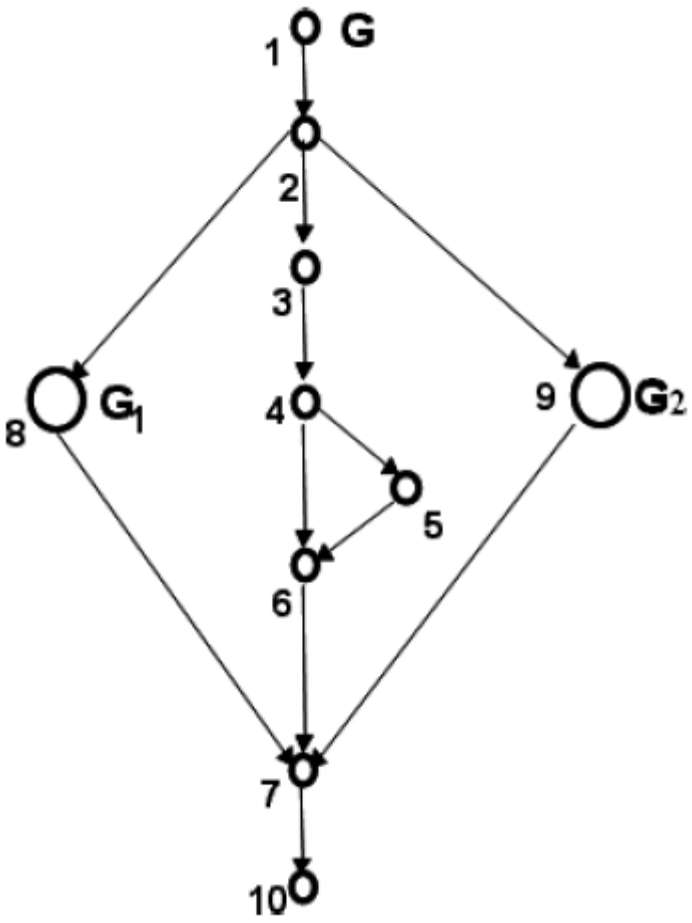

Рисунок 3 - Иерархическая модель УГП компонента G

УГП компонента G, представленный в виде иерархической модели, приведен на рисунок 3, Пример 5. В иерархическом УГП G входящие в его состав компоненты представлены ссылками на свои УГП G1 и G2 (Рисунок 4, Пример 5)

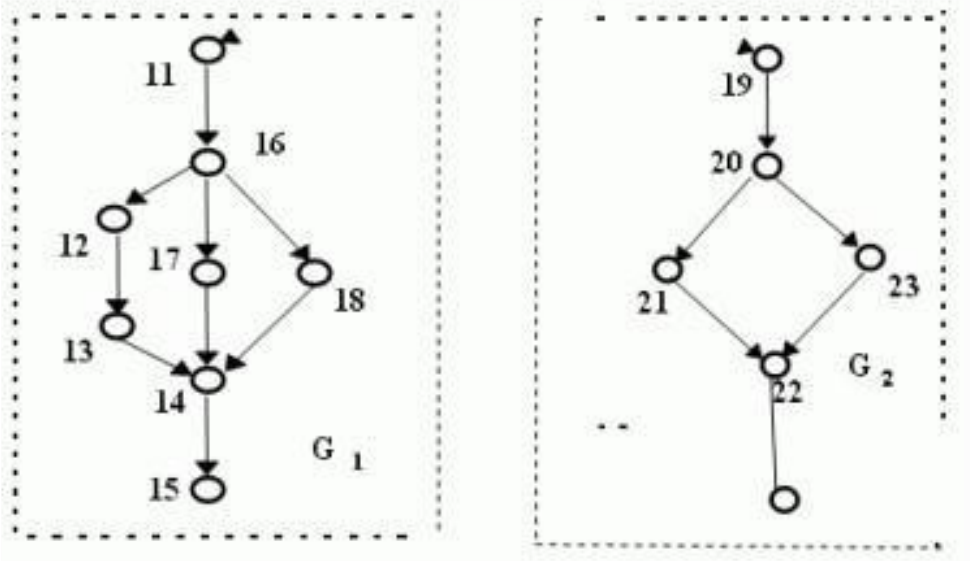

Рисунок 4 - Иерархическая модель: УГП компонент G1 и G2

Для исчерпывающего тестирования иерархической модели компонента G в соответствии с критерием путей требуется прогнать следующий набор трасс (Пример 2):

P1(G) = 1-2-3-4-5-6-7-10;

P2(G) = 1-2-3-4-6-7-10;

P3(G) = 1-2-8-7-10;

P4(G) = 1-2-9-7-10.

Пример 2. Набор трасс, необходимых для покрытия иерархической модели УГП компонента G

Приведенный набор трасс достаточен при условии, что компоненты G1 и G2 в свою очередь исчерпывающе протестированы. Чтобы обеспечить выполнение этого условия в соответствии с критерием путей, надо прогнать все трассы Пример 3.

P11(G1)=11-16-12-13-14-15; P21(G2)=19-20-21-22; P12(G1)=11-16-17-14-15;

P22(G2)=11-16-18-14-15; P13(G1)=19-20-23-22.

Пример 3. Набор трасс иерархической модели УГП, необходимых для покрытия УГП компонентов G1 и G2

ЛАБОРАТОРНАЯ РАБОТА № 6

Цель работы: освоить модульное тестирование

Теоретическое обоснование

Модульное тестирование — это тестирование программы на уровне отдельно взятых модулей, функций или классов. Цель модульного тестирования состоит в выявлении локализованных в модуле ошибок в реализации алгоритмов, а также в определении степени готовности системы к переходу на следующий уровень разработки и тестирования. Модульное тестирование проводится по принципу "белого ящика", то есть основывается на знании внутренней структуры программы, и часто включает те или иные методы анализа покрытия кода.

Модульное тестирование обычно подразумевает создание вокруг каждого модуля определенной среды, включающей заглушки для всех интерфейсов тестируемого модуля. Некоторые из них могут использоваться для подачи входных значений, другие для анализа результатов, присутствие третьих может быть продиктовано требованиями, накладываемыми компилятором и сборщиком.

На уровне модульного тестирования проще всего обнаружить дефекты, связанные с алгоритмическими ошибками и ошибками кодирования алгоритмов, типа работы с условиями и счетчиками циклов, а также с использованием локальных переменных и ресурсов. Ошибки, связанные с неверной трактовкой данных, некорректной реализацией интерфейсов, совместимостью, производительностью и т.п. обычно пропускаются на уровне модульного тестирования и выявляются на более поздних стадиях тестирования.

Именно эффективность обнаружения тех или иных типов дефектов должна определять стратегию модульного тестирования, то есть расстановку акцентов при определении набора входных значений. У организации, занимающейся разработкой программного обеспечения, как правило, имеется историческая база данных (Repository) разработок, хранящая конкретные

сведения о разработке предыдущих проектов: о версиях и сборках кода (build) зафиксированных в процессе разработки продукта, о принятых решениях, допущенных просчетах, ошибках, успехах и т.п. Проведя анализ характеристик прежних проектов, подобных заказанному организации, можно предохранить новую разработку от старых ошибок, например, определив типы дефектов, поиск которых наиболее эффективен на различных этапах тестирования.

В данном случае анализируется этап модульного тестирования. Если анализ не дал нужной информации, например, в случае проектов, в которых соответствующие данные не собирались, то основным правилом становится поиск локальных дефектов, у которых код, ресурсы и информация, вовлеченные в дефект, характерны именно для данного модуля. В этом случае на модульном уровне ошибки, связанные, например, с неверным порядком или форматом параметров модуля, могут быть пропущены, поскольку они вовлекают информацию, затрагивающую другие модули (а именно, спецификацию интерфейса), в то время как ошибки в алгоритме обработки параметров довольно легко обнаруживаются.

Являясь по способу исполнения структурным тестированием или тестированием "белого ящика", модульное тестирование характеризуется степенью, в которой тесты выполняют или покрывают логику программы (исходный текст). Тесты, связанные со структурным тестированием, строятся по следующим принципам:

На основе анализа потока управления. В этом случае элементы, которые должны быть покрыты при прохождении тестов, определяются на основе структурных критериев тестирования С0, С1,С2. К ним относятся вершины, дуги, пути управляющего графа программы (УГП), условия, комбинации условий и т. п.

На основе анализа потока данных, когда элементы, которые должны быть покрыты, определяются на основе потока данных, т. е. информационного графа программы.

Тестирование на основе потока управления. Особенности использования структурных критериев тестирования C0,C1,C2 были рассмотрены в разделе 2. К ним следует добавить критерий покрытия условий, заключающийся в покрытии всех логических (булевских) условий в программе. Критерии покрытия решений (ветвей — C1) и условий не заменяют друг друга, поэтому на практике используется комбинированный критерий покрытия условий/решений, совмещающий требования по покрытию и решений, и условий.

К популярным критериям относятся критерий покрытия функций программы, согласно которому каждая функция программы должна быть вызвана хотя бы один раз, и критерий покрытия вызовов, согласно которому каждый вызов каждой функции в программе должен быть осуществлен хотя бы один раз. Критерий покрытия вызовов известен также как критерий покрытия пар вызовов (call pair coverage).

Тестирование на основе потока данных. Этот вид тестирования направлен на выявление ссылок на неинициализированные переменные и избыточные присваивания (аномалий потока данных). Предложенная стратегия требовала тестирования всех взаимосвязей, включающих в себя ссылку (использование) и определение переменной, на которую указывает ссылка (т. е. требуется покрытие дуг информационного графа программы). Недостаток стратегии в том, что она не включает критерий C1, и не гарантирует покрытия решений.

Стратегия требуемых пар также тестирует упомянутые взаимосвязи. Использование переменной в предикате дублируется в соответствии с числом выходов решения, и каждая из таких требуемых взаимосвязей должна быть протестирована. К популярным критериям принадлежит критерий CP, заключающийся в покрытии всех таких пар дуг v и w, что из дуги v достижима дуга w, поскольку именно на дуге может произойти потеря значения переменной, которая в дальнейшем уже не должна использоваться. Для "покрытия" еще одного популярного критерия Cdu достаточно тестировать пары (вершина, дуга), поскольку определение переменной происходит в

вершине УГП, а ее использование — на дугах, исходящих из решений, или в вычислительных вершинах.

Методы проектирования тестовых путей для достижения заданной степени тестированности в структурном тестировании. Процесс построения набора тестов при структурном тестировании принято делить на три фазы:

1. Конструирование УГП.
2. Выбор тестовых путей.
3. Генерация тестов, соответствующих тестовым путям.

Первая фаза соответствует статическому анализу программы, задача которого состоит в получении графа программы и зависящего от него и от критерия тестирования множества элементов, которые необходимо покрыть тестами.

На третьей фазе по известным путям тестирования осуществляется поиск подходящих тестов, реализующих прохождение этих путей.

Вторая фаза обеспечивает выбор тестовых путей. Выделяют три подхода к построению тестовых путей:

- Статические методы.
- Динамические методы.
- Методы реализуемых путей.

Статические методы. Самое простое и легко реализуемое решение — построение каждого пути посредством постепенного его удлинения за счет добавления дуг, пока не будет достигнута выходная вершина управляющего графа программы. Эта идея может быть усилена в так называемых адаптивных методах, которые каждый раз добавляют только один тестовый путь (входной тест), используя предыдущие пути (тесты) как руководство для выбора последующих путей в соответствии с некоторой стратегией. Чаще всего адаптивные стратегии применяются по отношению к критерию С1. Основной недостаток статических методов заключается в том, что не учитывается возможная нереализуемость построенных путей тестирования.

Динамические методы. Такие методы предполагают построение полной системы тестов, удовлетворяющих заданному критерию, путем одновременного решения задачи построения покрывающего множества путей и тестовых данных. При этом можно автоматически учитывать реализуемость или нереализуемость ранее рассмотренных путей или их частей. Основной идеей динамических методов является подсоединение к начальным реализуемым отрезкам путей дальнейших их частей так, чтобы: 1) не терять при этом реализуемости вновь полученных путей; 2) покрыть требуемые элементы структуры программы.

Методы реализуемых путей. Данная методика заключается в выделении из множества путей подмножества всех реализуемых путей. После чего покрывающее множество путей строится из полученного подмножества реализуемых путей.

Достоинство статических методов состоит в сравнительно небольшом количестве необходимых ресурсов, как при использовании, так и при разработке. Однако их реализация может содержать непредсказуемый процент брака (нереализуемых путей). Кроме того, в этих системах переход от покрывающего множества путей к полной системе тестов пользователь должен осуществить вручную, а эта работа достаточно трудоемкая. Динамические методы требуют значительно больших ресурсов как при разработке, так и при эксплуатации, однако увеличение затрат происходит, в основном, за счет разработки и эксплуатации аппарата определения реализуемости пути (символический интерпретатор, решатель неравенств). Достоинство этих методов заключается в том, что их продукция имеет некоторый качественный уровень — реализуемость путей. Методы реализуемых путей дают самый лучший результат.

Пример модульного тестирования

Предлагается протестировать класс TCommand, который реализует команду для склада. Этот класс содержит единственный метод

TCommand.GetFullName(), спецификация которого описана следующим образом:

...

Операция GetFullName() возвращает полное имя команды, соответствующее ее допустимому коду, указанному в поле NameCommand. В противном случает возвращается сообщение "ОШИБКА : Неверный код команды". Операция может быть применена в любой момент.

...

Разработаем спецификацию тестового случая для тестирования метода GetFullName на основе приведенной спецификации класса (Таблица1):

Название класса: TCommand	**Название тестового случая:** TCommandTest1
Описание тестового случая: Тест проверяет правильность работы метода GetFullName - получения полного названия команды на основе кода команды. В тесте подаются следующие значения кодов команд (входные значения): -1, 1, 2, 4, 6, 20, (причем -1 - запрещенное значение).	
Начальные условия: Нет.	
Ожидаемый результат:	
Перечисленным входным значениям должны соответствовать следующие выходные:	
Коду команды -1 должно соответствовать сообщение "ОШИБКА: Неверный код команды"	
Коду команды 1 должно соответствовать полное название команды "ПОЛУЧИТЬ ИЗ ВХОДНОЙ ЯЧЕЙКИ"	
Коду команды 2 должно соответствовать полное название команды "ОТПРАВИТЬ ИЗ ЯЧЕЙКИ В ВЫХОДНУЮ ЯЧЕЙКУ"	
Коду команды 4 должно соответствовать полное название команды "ПОЛОЖИТЬ В РЕЗЕРВ"	
Коду команды 6 должно соответствовать полное название команды "ПРОИЗВЕСТИ ЗАНУЛЕНИЕ"	
Коду команды 20 должно соответствовать полное название команды "ЗАВЕРШЕНИЕ КОМАНД ВЫДАЧИ"	

Для тестирования метода класса TCommand.GetFullName() был создан тестовый драйвер — класс TCommandTester. Класс TCommandTester содержит метод TCommandTest1(), в котором реализована вся функциональность теста. В данном случае для покрытия спецификации достаточно перебрать следующие значения кодов команд: -1, 1, 2, 4, 6, 20, (-1 — запрещенное значение) и получить соответствующее им полное название команды с помощью метода GetFullName() (Пример 5.1). Пары значений (X, Yв) при исполнении теста заносятся в log-файл для последующей проверки на соответствие спецификации.

После завершения теста следует просмотреть журнал теста, чтобы сравнить полученные результаты с ожидаемыми, заданными в спецификации тестового случая TCommandTest1 (Пример 1).

```
class TCommandTester:Tester // Тестовый драйвер
{
...
```

```
TCommand OUT;
public TCommandTester()
{
OUT=new TCommand(); Run();
}
private void Run()
{
TCommandTest1();
}
private void TCommandTest1()
{
int[] commands = {-1, 1, 2, 4, 6, 20}; for(int i=0;i<=5;i++)
{
OUT.NameCommand=commands[i];  LogMessage(commands[i].ToString()+
" : "+OUT.GetFullName());
}
}
...
}
```

Пример 2. Тестовый драйвер

-1 : ОШИБКА : Неверный код команды

1 : ПОЛУЧИТЬ ИЗ ВХОДНОЙ ЯЧЕЙКИ

2 : ОТПРАВИТЬ ИЗ ЯЧЕЙКИ В ВЫХОДНУЮ ЯЧЕЙКУ

4 : ПОЛОЖИТЬ В РЕЗЕРВ

6 : ПРОИЗВЕСТИ ОБНУЛЕНИЕ

20 : ЗАВЕРШЕНИЕ КОМАНД ВЫДАЧИ

ЛАБОРАТОРНАЯ РАБОТА № 7

Цель работы: освоить интеграционное тестирование

Теоретическое обоснование

Продемонстрируем тестирование взаимодействий на примере взаимодействия класса TCommandQueue и класса TCommand, а также, как и при модульном тестировании, разработаем спецификацию тестового случая (таблица 1):

Названия взаимодействующих классов	TCommandQueue, TCommand
Название теста	TCommandQueueTest1
Описание теста	тест проверяет возможность создания объекта типа TCommand и добавления его в очередь при вызове метода AddCommand
Начальные условия	очередь команд пуста
Ожидаемый результат	в очередь будет добавлена одна команда

На основе этой спецификации разработан тестовый драйвер пример 1 — класс TCommandQueueTester, который наследуется от класса Tester.

Класс содержит:

конструктор, в котором создаются объекты классов TStore, TTerminalBearing и объект типа TcommandQueue

Методы, реализующие тесты. Каждый тест реализован в отдельном методе. Метод Run, в котором вызываются методы тестов.

Метод dump, который сохраняет в Log-файле теста информацию обо всех командах, находящихся в очереди в формате — Номер позиции в очереди: полное название команды Точку входа в программу — метод Main, в котором происходит создание экземпляра

класса TCommandQueueTester.

```
public TCommandQueueTester()
{
```

TB = new TTerminalBearing(); S = new TStore();

CommandQueue=new TCommandQueue(S,TB); S.CommandQueue=CommandQueue;

...

}

Пример 6.1. Объект типа TcommandQueue TCommandQueueTester::TCommandQueueTester()

{

TB = new TTerminalBearing(); S = new TStore();

CommandQueue=new TCommandQueue(S,TB); S->CommandQueue=CommandQueue;

}

Пример 1. Объект типа TcommandQueue (C++)

Теперь создадим тест, который проверяет, создается ли объект типа TCommand, и добавляется ли команда в конец очереди.

private void TCommandQueueTest1()

{

LogMessage("///// TCommandQueue Test1 /////");

LogMessage("Проверяем, создается ли объект типа TCommand");

// В очереди нет команд dump();

// Добавляем команду

// параметр = -1 означает, что команда

// должна быть добавлена в конец очереди CommandQueue.AddCommand(TCommand.GetR,0,0,0, new TBearingParam(),new TAxleParam(),-1); LogMessage("Command added");

// В очереди одна команда dump();

}

Пример 2. Тест

void TCommandQueueTester::TCommandQueueTest1()

{

LogMessage("///// TCommandQueue Test1 /////");

LogMessage("Проверяем, создается ли объект типа TCommand");

// В очереди нет команд dump();

// Добавляем команду

// параметр = -1 означает, что команда

// должна быть добавлена в конец очереди CommandQueue.AddCommand(GetR,0,0,0, new TBearingParam(),

new TAxleParam(),-1); LogMessage("Command added");

// В очереди одна команда

dump();

}

Пример 3. Тест (C++)

В класс включены еще два два разработанных теста.

После завершения теста следует просмотреть текстовый журнал теста, чтобы сравнить полученные результаты с ожидаемыми результатами, заданными в спецификации тестового случая TCommandQueueTest1 пример 3.

///// TCommandQueue Test1 /////

Проверяем, создается ли объект типа TCommand 0 commands in command queue

Command added

1 commands in command queue

0: ПОЛУЧИТЬ ИЗ ВХОДНОЙ ЯЧЕЙКИ

ЛАБОРАТОРНАЯ РАБОТА № 8

«Тестирование и оценка качества программного проекта»

1.1. Цель работы

- Изучить методы подготовки и проведения тестирования.
- Получить навыки создания и выполнения тестов для приложений и их компонентов.

1.2. Средства выполнения

- MS Visual Studio (по варианту) или фреймворк Jest (по варианту),
- Раздел Performance в Chrome DevTools,
- Утилита wrk,
- Библиотека NightWatch.

1.3. Пункты задания для выполнения

1. (базовое) Открыть исходный код тестируемого приложения (собственное или выданный преподавателем). Добавить Unit-тест для одной из функций. Запустить тест и просмотреть результаты. Создать несколько разных тестов для проверки значений и перехвата исключений.
2. (базовое) Установить параметры сбора статистики покрытия кода. Повторить модульные тесты и просмотреть данные о покрытии кода.
3. (расширенное) Создать тестовый проект по веб-тестам производительности (для своего сайта или любого стандартного). При этом записать сценарий работы с сайтом. Настроить параметры нагрузки (частота запросов и т.д.) Выполнить тест и просмотреть результаты.
4. (расширенное) Для тестируемого приложения (собственное или выданный преподавателем) провести профайлинг (оценку производительности). Выполнить тест и просмотреть результаты.

1.4. Дополнительное задание

5. Создать тестовый проект закодированного тестирования пользовательского интерфейса (например, для калькулятора). Наполнить тест действиями по вводу данных и проверки полученного результата. Выполнить тест и просмотреть результаты.

1.5. Содержание отчета

- Титульный лист;
- Цель работы;
- Задание;
- Описание проделанных шагов;
- Скриншоты с результатами выполнение;
- Вывод;

2. ТЕОРЕТИКО-ПРАКТИЧЕСКИЙ МАТЕРИАЛ

2.1. Тестирование

Тестирование программного обеспечения — проверка соответствия между реальным и ожидаемым поведением программы, осуществляемая на конечном наборе тестов, выбранном определенным образом. В более широком смысле, тестирование — это одна из техник контроля качества, включающая в себя активности по планированию работ (Test Management), проектированию тестов (Test Design), выполнению тестирования (Test Execution) и анализу полученных результатов (Test Analysis).

2.1.1. Краткий список терминов и определений

Качество программного обеспечения — это совокупность характеристик программного обеспечения, относящихся к его способности удовлетворять установленные и предполагаемые потребности.

Верификация — это процесс оценки системы или её компонентов с целью определения удовлетворяют ли результаты текущего этапа разработки условиям, сформированным в начале этого этапа. Т.е. выполняются ли наши цели, сроки, задачи по разработке проекта, определенные в начале текущей фазы.

Валидация — это оценка соответствия продукта ожиданиям и требованиям пользователей.

План тестирования — 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.

Отвечает на вопросы:

- Что надо тестировать?
- Что будете тестировать?
- Как будете тестировать?
- Когда будете тестировать?
- Критерии начала тестирования.
- Критерии окончания тестирования.

Разработка тестов – это этап процесса тестирования ПО, на котором проектируются и создаются тестовые сценарии (тест кейсы), в соответствии с определёнными ранее критериями качества и целями тестирования.

Error — ошибка пользователя, то есть он пытается использовать программу иным способом. Пример: вводит буквы в поля, где требуется вводить цифры (возраст, количество товара и т.п.). В качественной программе предусмотрены такие ситуации и выдаются сообщение об ошибке (error message).

Bug (defect) — ошибка программиста (или дизайнера или ещё кого, кто принимает участие в разработке), то есть, когда в программе, что-то идёт не так как планировалось и программа выходит из-под контроля. Например, когда никак не контролируется ввод пользователя, в результате неверные данные вызывают краш программы. Либо внутри программа построена так, что изначально не соответствует тому, что от неё ожидается.

Failure — сбой (причём не обязательно аппаратный) в работе компонента, всей программы или системы. То есть, существуют такие дефекты, которые приводят к сбоям и существуют такие, которые не приводят. UI-дефекты, например. Но аппаратный сбой, никак не связанный с software, тоже является failure.

Отчет об ошибках (Bug Report) — это документ, описывающий ситуацию или последовательность действий, приведшую к некорректной работе объекта тестирования, с указанием причин и ожидаемого результата.

Отчёт об ошибке содержит:

- 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
- Приоритет (Priority) Приоритет дефекта
- Статус (Status) Статус ошибки. Зависит от используемой процедуры и жизненного цикла бага
- Автор (Author) Создатель отчета

- 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.
- 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

2.1.2. Этапы тестирования:

Процесс тестирования состоит из следующих этапов:

1. Анализ продукта
2. Работа с требованиями
3. Разработка стратегии тестирования и планирование процедур контроля качества
4. Создание тестовой документации
5. Тестирование прототипа
6. Основное тестирование
7. Стабилизация
8. Эксплуатация

2.1.3. Методы тестирования

Эквивалентное Разделение (Equivalence Partitioning — EP). Как пример, у вас есть диапазон допустимых значений от 1 до 10, вы должны выбрать одно верное значение внутри интервала, скажем, 5, и одно неверное значение вне интервала — 0.

Анализ Граничных Значений (Boundary Value Analysis — BVA). Если взять пример выше, в качестве значений для позитивного тестирования выберем минимальную и максимальную границы (1 и 10), и значения больше и меньше границ (0 и 11). Анализ Граничный значений может быть применен к полям, записям, файлам, или к любого рода сущностям, имеющим ограничения.

Причина / Следствие (Cause/Effect — CE). Это, как правило, ввод комбинаций условий (причин), для получения ответа от системы (Следствие). Например, вы проверяете возможность добавлять клиента, используя определенную экранную форму. Для этого вам необходимо будет ввести несколько полей, таких как «Имя», «Адрес», «Номер Телефона» а затем, нажать кнопку «Добавить» — это «Причина». После нажатия кнопки «Добавить», система добавляет клиента в базу данных и показывает его номер на экране — это «Следствие».

Предугадывание ошибки (Error Guessing — EG). Это когда тестировщик использует свои знания системы и способность к интерпретации спецификации на предмет того, чтобы «предугадать» при каких входных условиях система может выдать ошибку. Например, спецификация говорит: «пользователь должен ввести код». Тестировщик будет думать: «Что, если я не введу код?», «Что, если я введу неправильный код?», и так далее.

Исчерпывающее тестирование (Exhaustive Testing — ET) — это крайний случай. В пределах этой техники вы должны проверить все возможные комбинации входных значений, и в принципе, это должно найти все проблемы. На практике применение этого метода не представляется возможным, из-за огромного количества входных значений.

Попарное тестирование (Pairwise Testing) — это техника формирования наборов тестовых данных. Сформулировать суть можно, например, вот так: формирование таких наборов данных, в которых каждое тестируемое значение каждого из проверяемых параметров хотя бы единожды сочетается с каждым тестируемым значением всех остальных проверяемых параметров.

Матрица соответствия требований (Traceability matrix) — это двумерная таблица, содержащая соответствие функциональных требований (functional

requirements) продукта и подготовленных тестовых сценариев (test cases). В заголовках колонок таблицы расположены требования, а в заголовках строк — тестовые сценарии. На пересечении — отметка, означающая, что требование текущей колонки покрыто тестовым сценарием текущей строки. Матрица соответствия требований используется QA-инженерами для валидации покрытия продукта тестами. МСТ является неотъемлемой частью тест-плана.

Тестовый сценарий, тестовый вариант, тест (Test Case) — это артефакт, содержащий исходные данные и ожидаемый результат при тестировании. Он также описывает совокупность шагов, конкретных условий и параметров, необходимых для проверки реализации тестируемой функции или её части.

Каждый тестовый сценарий должен иметь 3 части:

- Предусловие - PreConditions - 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 состояния.
- Описание тестового сценария - 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.
- Постусловия - PostConditions - Список действий, переводящих систему в первоначальное состояние (состояние до проведения теста — initial state)

Виды Тестовых Сценариев. Тесты разделяются по ожидаемому результату на положительные и отрицательные:

- Положительный тест использует только корректные данные и проверяет, что приложение правильно выполнило вызываемую функцию.
- Отрицательный тест 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.

Чек-лист (check list) — это документ, описывающий что должно быть протестировано. При этом он может быть абсолютно разного уровня детализации. На сколько детальным он будет зависит от требований к отчетности, уровня знания продукта сотрудниками и сложности продукта.

Как правило, список содержит только действия (шаги), без ожидаемого результата. Он менее формализован, чем тестовый сценарий. Его уместно использовать тогда, когда тестовые сценарии будут избыточны. Также чек-лист ассоциируются с гибкими подходами в тестировании.

Дефект (баг) – это несоответствие фактического результата выполнения программы ожидаемому результату. Дефекты обнаруживаются на этапе тестирования программного обеспечения (ПО), когда тестировщик проводит сравнение полученных результатов работы программы (компонента или дизайна) с ожидаемым результатом, описанным в спецификации требований.

2.1.4. Виды и типы тестирования

Тестирование программных продуктов можно разбить на виды (см. Рисунок 1)

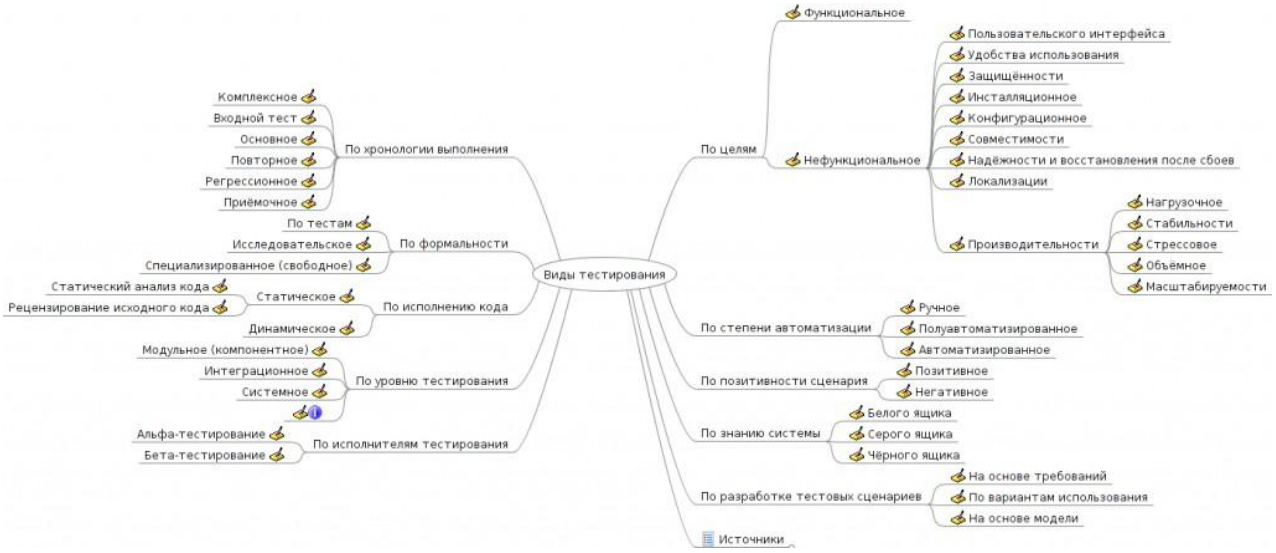

Рисунок 1. Виды тестирования

2.1.4.1. Виды Тестирования (уровни)

Модульное тестирование (Unit Testing). Компонентное (модульное) тестирование проверяет функциональность и ищет ошибки в частях приложения, которые доступны и могут быть протестированы по-отдельности (модули программ, объекты, классы, функции и т.д.).

Интеграционное тестирование (Integration Testing). Проверяется взаимодействие между компонентами системы после проведения компонентного тестирования.

Системное тестирование (System Testing). Основной задачей системного тестирования является проверка как функциональных, так и не функциональных

требований в системе в целом. При этом выявляются дефекты, такие как неверное использование ресурсов системы, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство использования и т.д.

Операционное тестирование (Release Testing). Даже если система удовлетворяет всем требованиям, важно убедиться в том, что она удовлетворяет нуждам пользователя и выполняет свою роль в среде своей эксплуатации, как это было определено в бизнес моделе системы. Следует учесть, что и бизнес-модель может содержать ошибки. Поэтому так важно провести операционное тестирование как финальный шаг валидации. Кроме этого, тестирование в среде эксплуатации позволяет выявить и нефункциональные проблемы, такие как: конфликт с другими системами, смежными в области бизнеса или в программных и электронных окружениях; недостаточная производительность системы в среде эксплуатации и др. Очевидно, что нахождение подобных вещей на стадии внедрения — критичная и дорогостоящая проблема. Поэтому так важно проведение не только верификации, но и валидации, с самых ранних этапов разработки ПО.

Приемочное тестирование (Acceptance Testing). Формальный процесс тестирования, который проверяет соответствие системы требованиям и проводится с целью: определения удовлетворяет ли система приемочным критериям и вынесения решения заказчиком или другим уполномоченным лицом принимается приложение или нет.

2.1.4.2. Типы тестирования

Функциональные виды тестирования.

- Функциональное тестирование (Functional testing). Проверка корректности и полноты выполнемых системой функций в соответствии с требованиеми к ней. Рассматривает заранее указанное функцинное поведение и основывается на анализе спецификаций функциональности компонента или системы в целом.
- Тестирование пользовательского интерфейса (GUI Testing). Функциональная проверка интерфейса на соответствие требованиям — размер, шрифт, цвет, consistent behavior.
- Тестирование безопасности (Security and Access Control Testing). Это стратегия тестирования, используемая для проверки безопасности системы, а

также для анализа рисков, связанных с обеспечением целостного подхода к защите приложения, атак хакеров, вирусов, несанкционированного доступа к конфиденциальным данным.

- Тестирование взаимодействия (Interoperability Testing). Это тестирование, проверяющее способность приложения взаимодействовать с одним и более компонентами или системами и включающее в себя тестирование совместимости (compatibility testing), и интеграционное тестирование.

Нефункциональные виды тестирования.

- Все виды тестирования производительности:
    - нагрузочное тестирование (Performance and Load Testing). Это автоматизированное тестирование, имитирующее работу определенного количества бизнес-пользователей на каком-либо общем (разделяемом ими) ресурсе.
    - стрессовое тестирование (Stress Testing). Оно позволяет проверить насколько приложение и система в целом работоспособны в условиях стресса и также оценить способность системы к регенерации, т.е. к возвращению к нормальному состоянию после прекращения воздействия стресса. Стрессом в данном контексте может быть повышение интенсивности выполнения операций до очень высоких значений или аварийное изменение конфигурации сервера.
    - тестирование стабильности или надежности (Stability / Reliability Testing). Задачей этого вида тестирования является проверка работоспособности приложения при длительном (многочасовом) тестировании со средним уровнем нагрузки.
    - объемное тестирование (Volume Testing) Задачей является получение оценки производительности при увеличении объемов данных в базе данных приложения.
- Тестирование установки (Installation testing) Оно направленно на проверку успешной инсталляции и настройки, а также обновления или удаления программного обеспечения.
- Тестирование удобства пользования (Usability Testing) Это метод тестирования, направленный на установление степени удобства использования, обучаемости, понятности и привлекательности для пользователей разрабатываемого продукта в контексте заданных условий.

- Тестирование на отказ и восстановление (Failover and Recovery Testing) Оно проверяет тестируемый продукт с точки зрения способности противостоять и успешно восстанавливаться после возможных сбоев, возникших в связи с ошибками программного обеспечения, отказами оборудования или проблемами связи (например, отказ сети). Целью данного вида тестирования является проверка систем восстановления (или дублирующих основной функционал систем), которые, в случае возникновения сбоев, обеспечат сохранность и целостность данных тестируемого продукта.
- Конфигурационное тестирование (Configuration Testing) Специальный вид тестирования, направленный на проверку работы программного обеспечения при различных конфигурациях системы (заявленных платформах, поддерживаемых драйверах, при различных конфигурациях компьютеров и т.д.).

Связанные с изменениями виды тестирования

- Дымовое тестирование (Smoke Testing) Оно рассматривается как короткий цикл тестов, выполняемый для подтверждения того, что после сборки кода (нового или исправленного) устанавливаемое приложение стартует и выполняет основные функции.
- Регрессионное тестирование (Regression Testing) Это вид тестирования направленный на проверку изменений, сделанных в приложении или окружающей среде (устранение дефекта, слияние кода, миграция на другую операционную систему, базу данных, веб сервер или сервер приложения), для подтверждения того факта, что существующая ранее функциональность работает как и прежде. Регрессионными могут быть как функциональные, так и нефункциональные тесты.
- Повторное тестирование (Re-testing) Тестирование, во время которого исполняются тестовые сценарии, выявившие ошибки во время последнего запуска, для подтверждения успешности исправления этих ошибок.
- Тестирование сборки (Build Verification Test). Тестирование, направленное на определение соответствия, выпущенной версии, критериям качества для начала тестирования. По своим целям является аналогом Дымового Тестирования, направленного на приемку новой версии в дальнейшее тестирование или эксплуатацию. Вглубь оно может проникать дальше, в зависимости от требований к качеству выпущенной версии.

- Санитарное тестирование или проверка согласованности/исправности (Sanity Testing) Узконаправленное тестирование достаточное для доказательства того, что конкретная функция работает согласно заявленным в спецификации требованиям. Является подмножеством регрессионного тестирования. Используется для определения работоспособности определенной части приложения после изменений, произведенных в ней или окружающей среде. Обычно выполняется вручную.

2.1.5. Дефекты

Дефекту или багу, выявленному в ходе тестирования, присваивается степень серьезности в зависимости от того, насколько баг влияет на работу программного продукта. Всего их пять:

- 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 посредствам пользовательского интерфейса, проблема сторонних библиотек или сервисов, проблема, не оказывающая никакого влияния на общее качество продукта.

Также ошибке выставляют приоритет. Он нужен для того, чтобы разработчику было проще определить с исправление какого бага ему начать. Приоритеты бывают:

- 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.

2.1.6. Принципы тестирования

Принцип 1. Тестирование демонстрирует наличие дефектов. Тестирование может показать, что дефекты присутствуют, но не может доказать, что их нет. Тестирование снижает вероятность наличия дефектов, находящихся в программном обеспечении, но, даже если дефекты не были обнаружены, это не доказывает его корректности.

Принцип 2. Исчерпывающее тестирование недостижимо. Полное тестирование с использованием всех комбинаций вводов и предусловий физически невыполнимо, за исключением тривиальных случаев. Вместо исчерпывающего тестирования должны использоваться анализ рисков и расстановка приоритетов, чтобы более точно сфокусировать усилия по тестированию.

Принцип 3. Раннее тестирование. Чтобы найти дефекты как можно раньше, активности по тестированию должны быть начаты как можно раньше в жизненном цикле разработки программного обеспечения или системы, и должны быть сфокусированы на определенных целях.

Принцип 4. Скопление дефектов. Усилия тестирования должны быть сосредоточены пропорционально ожидаемой, а позже реальной плотности дефектов по модулям. Как правило, большая часть дефектов, обнаруженных при тестировании или повлекших за собой основное количество сбоев системы, содержится в небольшом количестве модулей.

Принцип 5. Парадокс пестицида. Если одни и те же тесты будут прогоняться много раз, в конечном счете этот набор тестовых сценариев больше не будет находить новых дефектов. Чтобы преодолеть этот “парадокс пестицида”, тестовые сценарии должны регулярно рецензироваться и корректироваться, новые тесты должны быть разносторонними, чтобы охватить все компоненты программного обеспечения, или системы, и найти как можно больше дефектов.

Принцип 6. Тестирование зависит от контекста. Тестирование выполняется по-разному в зависимости от контекста. Например, программное обеспечение, в котором критически важна безопасность, тестируется иначе, чем сайт электронной коммерции.

Принцип 7. Заблуждение об отсутствии ошибок. Обнаружение и исправление дефектов не помогут, если созданная система не подходит пользователю и не удовлетворяет его ожиданиям и потребностям.

2.2. Модульное тестирование

Модульное тестирование получило такое название, так как функции программы разбиваются на отдельные тестируемые участки поведения, которые можно протестировать в качестве отдельных модулей. Обозреватель тестов Visual Studio предоставляет гибкий и эффективный способ запуска модульных тестов и просмотра результатов в Visual Studio. Visual Studio устанавливает платформы модульного тестирования Microsoft для управляемого и машинного кода. Платформа модульного тестирования используется для создания модульных тестов, их запуска и создания отчетов о результатах таких тестов.

Модульное тестирование максимально влияет на качество кода, когда оно является неотъемлемой частью рабочего процесса разработки ПО. После написания функции или другого блока кода приложения создаются модульные тесты, которые проверяют поведение кода в ответ на стандартные, граничные и некорректные случаи ввода данных; также проверяются любые явные или предполагаемые допущения, сделанные кодом. При разработке, управляемой тестами, модульные тесты создаются перед написанием кода, поэтому модульные тесты используются в качестве технической документации и спецификации функциональности.

2.2.1. Покрытие кода тестами

Чтобы определить, какая часть кода проекта в действительности тестируется закодированными тестами, такими как модульные тесты, можно воспользоваться возможностью покрытия кода в Visual Studio. Для обеспечения эффективной защиты от ошибок тесты должны выполнять ("покрывать") большую часть кода.

Покрытие кода возможно при выполнении методов тестов с помощью обозревателя тестов. В таблице результатов отображается процент кода, который был выполнен в каждой сборке, классе и методе. Кроме того, редактор исходного кода показывает, какой код был протестирован. Пример отчета о покрытии исходного кода изображен на Рисунке 2.

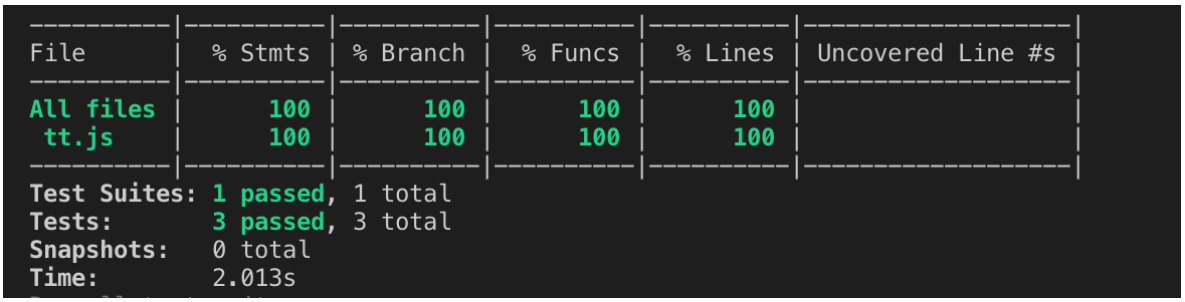

Рисунок 2. Покрытие кода

Данный вид теста запускается с помощью команды `npm run test -collectCoverage` в командной строке.

Во время работы каждого тестового примера выполняется некоторый участок программного кода системы, при выполнении всей системы тестов выполняются все участки программного кода, которые задействует эта система тестов. В случае, если существуют участки программного кода, не выполненные при выполнении системы тестов, система тестов потенциально неполна (т.е. не проверяет всю функциональность системы), либо система содержит участки защитного кода или неиспользуемый код (например, "закладки" или задел на будущее использование системы). Таким образом, отсутствие покрытия каких-либо участков кода является сигналом к переработке тестов или кода (а иногда – и требований).

К анализу покрытия программного кода можно приступать только после полного покрытия требований. Полное покрытие программного кода не гарантирует того, что тесты проверяют все требования к системе. Одна из типичных ошибок начинающего тестировщика – начинать с покрытия кода, забывая про покрытие требований.

2.2.2. Фреймворк Jest

Для покрытия кода тестами предлагается использовать Jest (https://jestjs.io/ru/).

Jest — это фреймворк для тестирования, разработанный Facebook. Он основан на Jasmine. По состоянию на сегодняшний день компания Facebook переработала большую часть его функционала и создала на его основе множество новых возможностей.

Среди особенностей Jest можно отметить следующие:

- Производительность. Фреймворк Jest признан самым быстрым в применении к большим проектам со множеством файлов тестов благодаря реализации интеллектуального механизма параллельного тестирования.

- Пользовательский интерфейс. Интерфейс Jest отличается понятностью и удобством.
- Наличие всего необходимого для начала работы. Jest поставляется со средствами создания утверждений, функций-шпионов и имитаций, которые эквивалентны отдельным библиотекам, вроде Sinon, выполняющим те же функции.
- Глобальные переменные. Как и в случае с Jasmine, Jest, по умолчанию, создаёт глобальные переменные тестирования, поэтому их не нужно явно подключать. Эту особенность можно воспринимать и как недостаток, так подобный подход вредит гибкости тестов и возможностям по управлению ими, но в большинстве случаев это всего лишь упрощает работу:

```
// "describe" уже в глобальной области видимости
// поэтому данные команды импорта не требуются:
// import { describe } from 'jest'
// import { describe } from 'jasmine'


describe('calculator', function() {
  ...
```

- Анализ покрытия кода тестами. В Jest имеются мощные и быстрые встроенные средства для анализа покрытия кода тестами.

2.2.3. Модульные тесты для функции multiplay

Имеется функция multiplay (см. Рисунок 3). Написана на языке JavaScript. Данная функция принимает 2 числа на вход и выводит их произведение. Тест написан с использованием библиотеки Jest. Jest можно установить, выполнив команду:

```
npm i jest
```

Тест запускается с помощь команды `npm run test` в командной строке.

```
function multiplay(a, b) {
    return a * b;
};

module.exports = { multiplay };
```

Рисунок 3. Функция multiplay

В начале каждого теста вызывается функция `test` принимающая на вход комментарий к тесту и другую функцию. В теле другой функции необходимо вызвать функцию `expect`, которой в качестве параметров мы передадим тестируемую функцию и ее параметры (входные данные для тестирования). После вызывается метод `toBe` в который передается ожидаемый результат выполнения тестируемой функции.

В данном примере (см. Рисунок 4) описано 3 unit теста каждый из которых принимает на вход пары чисел: (1,3), (2,3), (4,10) соответственно. После вызывается метод `toBe` в который передается значение, которое мы ожидаем после выполнения данной функции 3, 6, 40 соответственно.

```
const { multiplay } = require("./multiplay");

test('multiplay: adds 1 * 3 to equal 3', () => {
    expect(multiplay(1, 3)).toBe(3);
});
test('multiplay: adds 2 * 3 to equal 3', () => {
    expect(multiplay(2, 3)).toBe(6);
});
test('multiplay: adds 4 * 10 to equal 3', () => {
    expect(multiplay(4, 10)).toBe(40);
});
```

Рисунок 4. Unit-тесты

Функция `test` возвращает либо успешное сведения о результатах выполнения тестов. Если тесты выполнены успешно и полученный результат равен ожидаемому, то возвращается `PASS`, в противно случае возвращается `ERROR`.

Результаты тестирование приведены на Рисунке 5. По ним понятно, что проверялась одна функция (`Test Suites`), по 3 тестам (`Tests`). Все тесты прошли успешно, время занятое тестирование 1.228sec (`Time`)

```
> jest

 PASS  src/core/tt.test.js
  ✓ 1 arg (4ms)
  ✓ 2 arg
  ✓ 3 arg (1ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        1.228s
Ran all test suites.
```

Рисунок 5. Результат тестирования

2.2.4. Анализ покрытия

Целью анализа полноты покрытия кода является выявление участков кода, которые не выполняются при выполнении тестовых примеров. Тестовые примеры, основанные на требованиях, могут не обеспечивать полного выполнения всей структуры кода. Поэтому для улучшения покрытия проводится анализ полноты покрытия кода тестами и, при необходимости, дополнительные проверки, направленные на выяснение причины недостаточного покрытия, а также определение необходимых действий по его устранению. Обычно анализ покрытия выполняется с учетом следующих соглашений.

Анализ должен подтвердить, что полнота покрытия тестами структуры кода соответствует требуемому виду покрытия и заданному минимально допустимому проценту покрытия.

Анализ полноты покрытия тестами может выявить часть исходного кода, которая не исполнялась в ходе тестирования. Для разрешения этого обстоятельства могут потребоваться дополнительные действия в процессе проверки программного обеспечения.

3. Практика - инструменты тестирования

3.1. Веб-тесты производительности

Для веб-тестов производительности предлагается использовать вкладку Performance в Chrome DevTools (см. рисунок 15).

Панель отображает линию времени использования сети, выполнения JavaScript кода и уровень загрузки памяти. После первоначального построения графиков линии времени, будут доступны подробные данные о выполнение кода и всем жизненном цикле страницы. Будет возможность ознакомится с временем исполнения отдельных частей кода, появится возможность выбрать отдельный промежуток на временной шкале и ознакомится с тем какие процессы происходили в этот момент.

Инструмент применяется для улучшения производительности работы Вашей страницы в целом.

Ключевые возможности:

- Возможность сделать запись чтобы проанализировать каждое событие, которое произошло после загрузки страницы или взаимодействия с пользователем.

- Возможность просмотреть FPS, загрузку CPU и сетевые запросы в области Overview. Щелкните по событию в диаграмме, чтобы посмотреть детали об этом.
- Возможность изменить масштаб линию времени, чтобы сделать анализ проще.

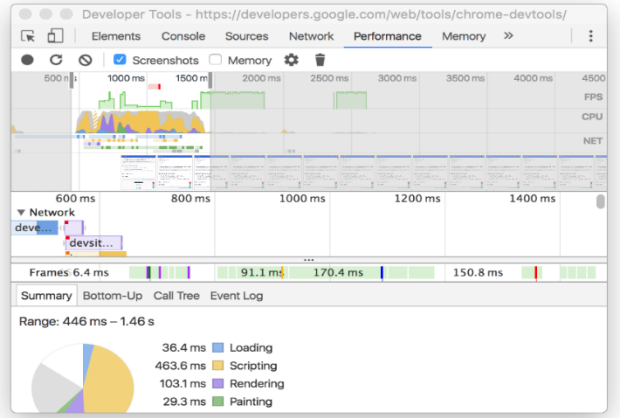

Рисунок 15. Performance вкладка в DevTools

3.2. Нагрузочное тестирование

Для проведения нагрузочного тестирования API предлагается использовать wrk. Ссылка на Github проекта: https://github.com/wg/wrk

3.3. GUI- тестирование

Для проведения GUI (интеграционного тестирования) предлагается использовать NightWatch (https://nightwatchjs.org/).

Пример как можно настроить NightWatch https://habr.com/ru/post/329660/

4. Практика - примеры

4.1. Нагрузочное тестирование API

Нагрузочное тестирование API выполняется с помощью команды wrk -t12 -c400 -d30s http://127.0.0.1:8080/index.html

wrk - это современный инструмент нагрузочного тестирования HTTP, способный генерировать значительную нагрузку при работе на одном многоядерном процессоре.

Опишем команды:

- t - количество используемых потоков для проведения теста
- c – количество соединений, используемых для проведения теста. Количество соединений по протоколу HTTP к вашему API
- d – время нагрузочного тестирования.

Нагрузочное тестирование (англ. load testing) — вид тестирования производительности, сбор показателей и определение производительности и времени отклика программно-технической системы или устройства в ответ на внешний запрос с целью установления соответствия требованиям, предъявляемым к данной системе (устройству).

Для исследования времени отклика системы на высоких или пиковых нагрузках производится стресс-тестирование, при котором создаваемая на систему нагрузка превышает нормальные сценарии её использования. Не существует чёткой границы между нагрузочным и стресс-тестированием, однако эти понятия не стоит смешивать, так как эти виды тестирования отвечают на разные бизнес-вопросы и используют различную методологию.

```
Running 30s test @ https://velox-server-usa.herokuapp.com/getuser
  12 threads and 400 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   216.14ms  142.53ms   1.60s    94.81%
    Req/Sec   112.96     71.76   320.00     57.87%
  32101 requests in 30.10s, 9.58MB read
  Socket errors: connect 167, read 0, write 0, timeout 4
  Non-2xx or 3xx responses: 32101
Requests/sec:   1066.47
Transfer/sec:    325.98KB
```

Рисунок 16. Проведение нагрузочного тестирования с помощью wrk

Мы провели нагрузочное тестирование (см. рисунок 16) с данными параметрами wrk -t12 -c400 -d30s и ниже описаны наши результаты.

Мы видим, что AVG Latency (средняя задержка ответа) у нас составляет 216.14 мс. Среднее количество запросов в секунду 112.96 (AVG Req/Sec). Максимальная задержка ответа составляет 1.6 сек (Max Latency), а Количество запросов 320 запросов/сек (Max Req/sec). По данным результатам можно сделать вывод о том, что наша система пригодна для довольно большого количества пользователей, и отвечает довольно быстро.

Показатель Запросов в секунду = 1066.47 (Requests/sec)

В течении теста было проведено 32101 (Non-2xx or 3xx responses) запрос в течении 30.1 сек (Requests in). И прочитано 9.58 Мбайт данных.

Среднее количество переданных данных в секунду = 325.98Кбайт (Transfer/sec)

4.2. Тестирование приложения (профайлинг)

4.2.1. Панель Performance (Скорость работы)

Панель отображает линию времени использования сети, выполнения JavaScript кода и загрузки памяти. После первоначального построения графиков линии времени, будут доступны подробные данные о выполнение кода и всем жизненном цикле страницы. Будет возможно ознакомится с временем исполнения отдельных частей кода, появится возможность выбрать отдельный промежуток на временной шкале и ознакомится с тем какие процессы происходили в этот момент.

Инструмент применяется для улучшения производительности работы Вашей страницы в целом.

4.2.2. Ключевые возможности:

- Возможность сделать запись чтобы проанализировать каждое событие, которое произошло после загрузки страницы или взаимодействия с пользователем.
- Возможность просмотреть FPS (Количество кадров в секунду), загрузку CPU (Процессора) и сетевые запросы в области Overview.
- Щелкните по событию в диаграмме, чтобы посмотреть детали об этом.
- Возможность изменить масштаб временную линию, чтобы сделать анализ проще.

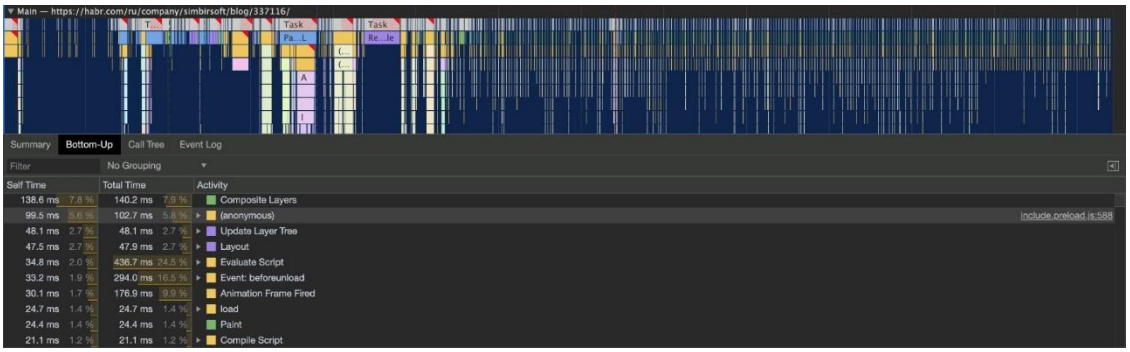

Рисунок 17. Вкладка Bottom-up

Перейдя на кладку «Bottom-up» или «Call Tree» (см. рисунок 17) можно увидеть дерево вызова функций (call tree (дерево вызовов)). Каждый из узлов можно раскрыть и увидеть более подробную информацию. Также сверху отображается графическое дерево вызовов, в котором можно легко увидеть «бутылочное горлышко» вашей системы.

Красные «треугольники» в углах узлов означают неоптимизированное время выполнение на которые и нужно обращать внимание. Во вкладке «Event log» (см. рисунок 18) можно увидеть подробное описание, всех событий, которые описаны ниже во вкладке «Summary», а именно Loading, Scipting, Rendering, Painting, System и idle (отрисовка страницы, выполнения кода Javascript, прорисовка стилей страницы, системные браузерные события и время простоя). Также можно раскрыть узлы и увидеть более подробную информацию.

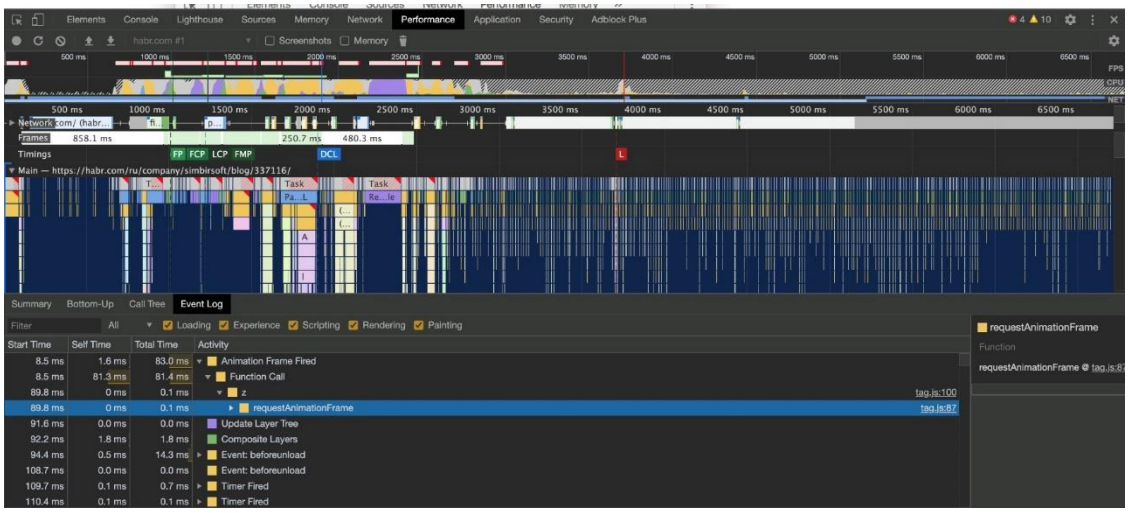

Рисунок 18. Вкладка Event log

4.2.3. Подробнее про графическое отображение

Отображение «Network» иллюстрирует работу с сетью - запрос/ответ, время запроса и размер данных. (см. рисунок 19)

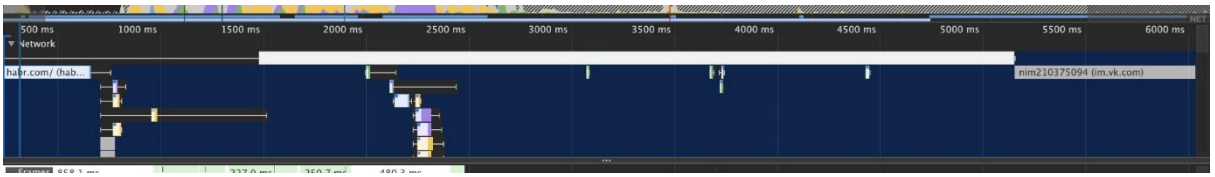

Рисунок 19. Отображение Network

Отображение «Main» иллюстрирует работу главного потока исполнения(интерпритации) Javascript. Отображение является деревом, в каждом узле которого можно увидеть более подробный список вызванных функций или проведенных операций.Также при наведении можно увидеть время выполнения конкретного узла в миллисекундах. (см. рисунок 20)

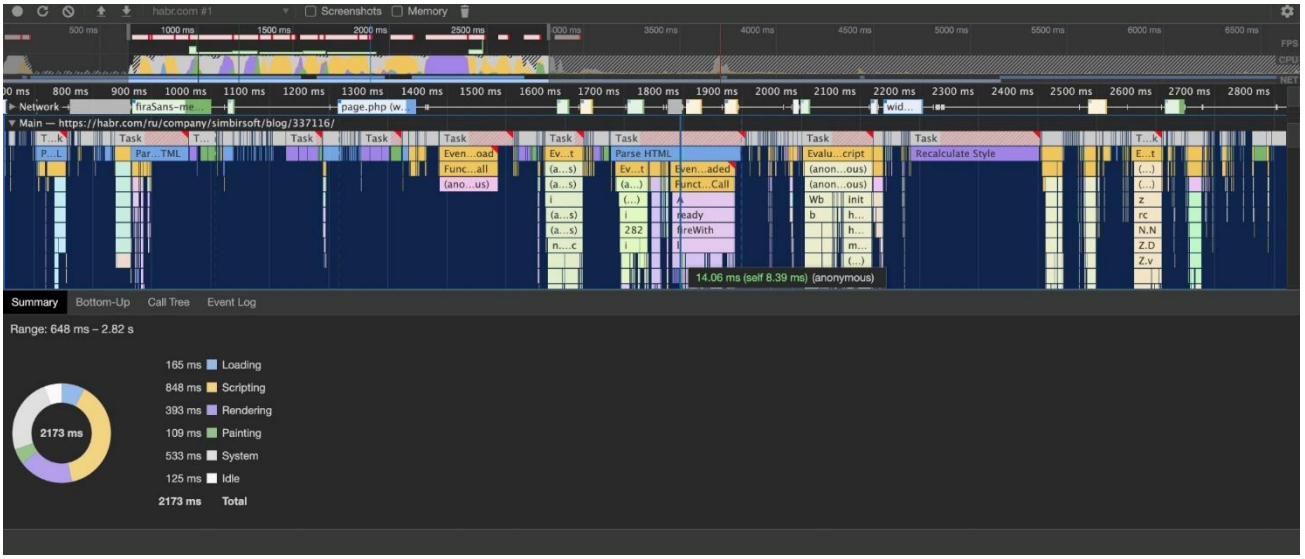

Рисунок 20. Отображение Main и общий график

Отображение «GPU» иллюстрирует загрузку графического процессора для отрисовки страницы. (см. рисунок 21)

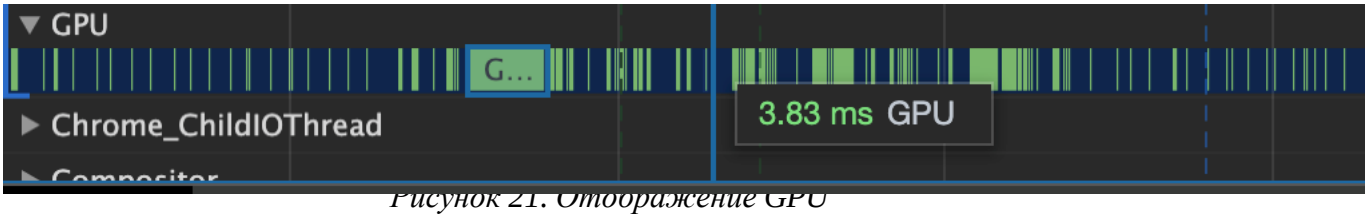

Рисунок 21. Отображение GPU

На общем графике (см. рисунок 20) можно выделить интересующий интервал, для подробного исследования показателей. Каждый цвет на графике соответствует цвету и описанию в диаграмме «Summary». В зависимости от цветовой темы браузера они

могут отличаться. Например, на рисунке выше видно, что желтая линия на общем графике соответствует метрике Scripting (время выполнение Javascript).

4.2.4. Запуск тестов

Откроем необходимый сайт. Откроем DevTools а именно инструменты разработчика в вашем браузере. Сверху обнаружите вкладку Preformance и кнопку Record Button. Нажмите на эту кнопку и запустите тестирование. (см. рисунок 22)

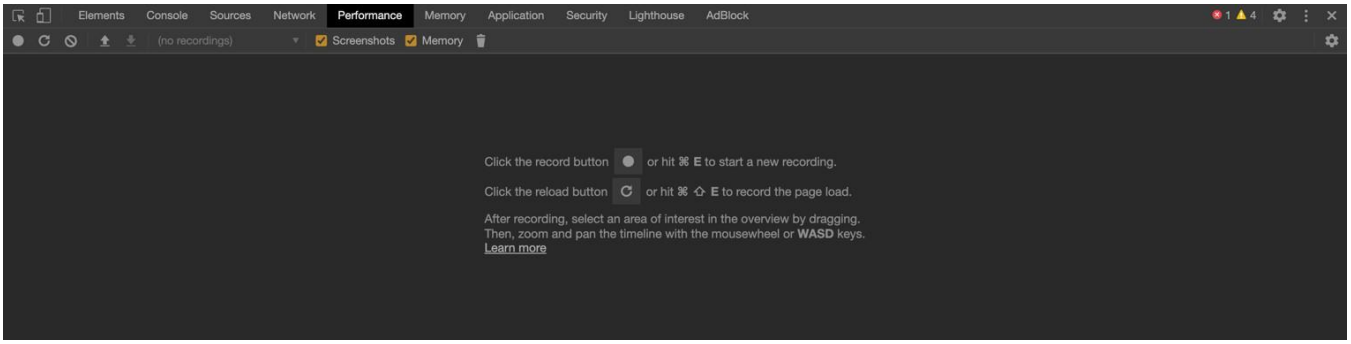

Рисунок 22. Запуск тестирования

Результаты тестирования будут представлены в виде графиков. (см. рисунок 23)

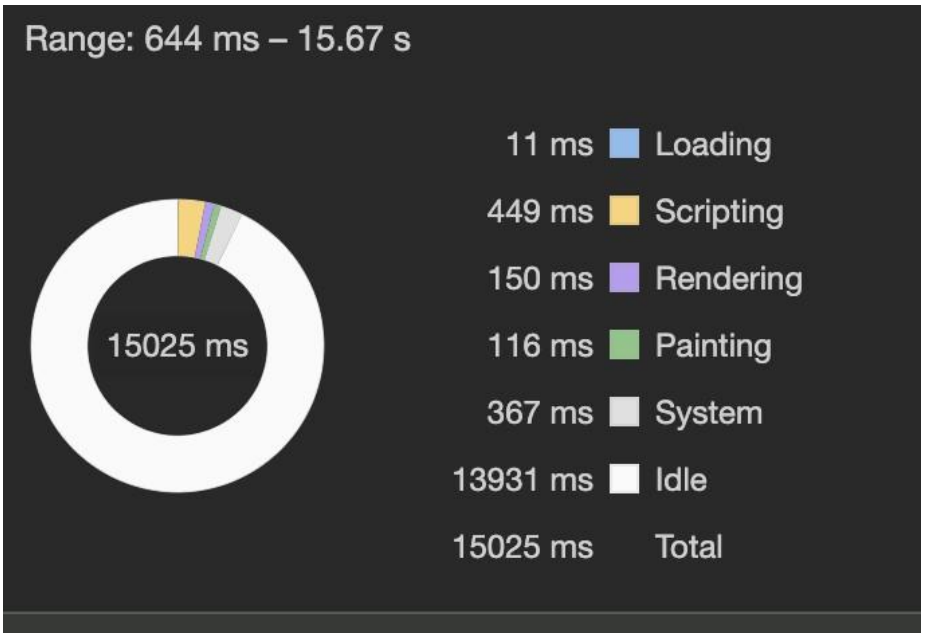

Рисунок 23. Результаты тестирования

Рассмотрим полученные результаты.

- Загрузка (Loading) страницы заняла 11 мс
- Выполнение и загрузка скрипта (Scripting) страницы заняло 449 мс
- Рендеринг, а именно отрисовка страницы (Rendering) страницы занял 150 мс
- Погрузка css - стилей страницы (Painting) страницы занял 116 мс
- Погрузка системных браузерных настроек (System) страницы занял 367 мс

- Время простоя системы после выполнения всех процессов до окончания нагрузочного тестирования приложения (Idle) страницы занял 13931 мс

Исходя из результатов, полученных нами, мы можем судить, что приложение довольно быстрое.

4.3. UI тестирование

Ниже вы видите пример теста написанного на языке JavaScript с использованием библиотеки NightWatch.js .

Для установки данной библиотеки выполните в командной строке команду

```
$ npm install nightwatch --save-dev
```

Тест вы можете запустить с помощью команды

```
$ npm test
```

Репозиторий данной библиотеки вы найдете здесь: https://github.com/nightwatchjs/nightwatch

Рассмотрим написанный (см. рисунок 24) тест более подробно. В данном тесте исследуется приложение reactjs.org

Ключевым словом browser мы говорим системе начать эмуляцию браузера и далее с помощью команды .url() показываем на какую страницу ей необходимо перейти.

Далее с помощью команды waitForElementVisible() мы дожидаемся загрузки интересующего нас элемента. Путь к данному элементы необходимо указать с помощью css-селекторов, вторым аргументом передается время ожидания данного элемента.

После чего с помощью команды clickLinkContaingText() мы говорим браузеру нажать на элемент сайта с текстом ‘Get Started’. Далее с помощью уже известной нам команды waitForElementVisible() ожидаем загрузки интересующего нас элемента. После чего с помощью функции assert.ContainsText() проверяем текст указанный в интересующем нас элементе. Адресация к элементу идет с помощью css-селекторов.

Далее показываем что необходимо закончить данный тест с помощью команды end()

Такой тест можно написать довольно большой, и содержать в себе множество вариаций и сценариев взаимодействия с веб приложением.

```
module.exports = {
    'https://reactjs.org/': function (browser) {
        browser
            .url('https://reactjs.org/')
            .waitForElementVisible('.css-1053yfl', 1000)
            .clickLinkContainingText('Get Started')
            .waitForElementPresent('.css-1rwyxsf', 1000)
            .assert.containsText('.css-1rwyxsf', 'Getting Started')
            .end();
    }
};
```

Рисунок 24. Код теста

Результаты выполнения данного теста вы можете наблюдать ниже. (см. рисунок 25) Здесь описана информация о времени которое занял тест, об успешном или неуспешном результате выполнения. А также обо всех этапах, выполненных в результате данного теста.

```
Starting selenium server... started - PID:  3418

[First Test] Test Suite
===========================
Executing the global `beforeEach`

[Fourth Test] Test Suite
============================
Executing the global `beforeEach`

Running:  https://reactjs.org/
  Warn: WaitForElement found 2 elements for selector ".css-1053yfl". Only the first one will be checked.
 ✔ Element <.css-1053yfl> was visible after 64 milliseconds.
 ✔ Element <.css-1rwyxsf> was present after 536 milliseconds.
 ✔ Testing if element <.css-1rwyxsf> contains text: "Getting Started".

OK. 3 assertions passed. (4.182s)

[Second Test] Test Suite
============================
Executing the global `beforeEach`

[Third Test] Test Suite
===========================
Executing the global `beforeEach`

OK. 3  total assertions passed. (4.299s)
```

Рисунок 25. Выполнение теста

5. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое тестирование, как его выполняют? Какие виды тестирования бывают?
2. Что содержит тестовый вариант?
3. Что такое модульное тестирование? Его цели и выявляемые ошибки.
4. Каков алгоритм модульного тестирования в общем виде? Возможности и средства проведения модульного тестирования?
5. Зачем и как проводят автоматизацию тестирования? Что такое тестовый драйвер и заглушка?
6. Что такое нагрузочное тестирование? Его цели?
7. Каковы параметры нагрузочного тестирования? Какую информацию можно получить по его результатам?
8. Исходные данные и результаты веб-тестов производительности?
9. Что такое профайлинг? Его цели и результаты?
10. Что такое тестирование пользовательского интерфейса? Его цели?
11. Что такое функциональное тестирование? Его цели и выявляемые ошибки. Основные методы.
12. Что такое структурное тестирование? Его цели и выявляемые ошибки. Основные методы.
13. Как определить покрытие кода? Зачем оно нужно?
14. Основные типы и виды тестирования?
15. Что такое регрессионное, повторное, системное и стрессовое тестирование? Какие виды тестирования еще бывают?
16. Каковы принципы составления тестовых вариантов?

ЛАБОРАТОРНАЯ РАБОТА № 9

Цель работы: освоить системное тестирование

Теоретическое обоснование

Системное тестирование качественно отличается от интеграционного и модульного уровней. Системное тестирование рассматривает тестируемую систему в целом и оперирует на уровне пользовательских интерфейсов, в отличие от последних фаз интеграционного тестирования, которое оперирует на уровне интерфейсов модулей. Различны и цели этих уровней тестирования. На уровне системы часто сложно и малоэффективно анализировать прохождение тестовых траекторий внутри программы или отслеживать правильность работы конкретных функций. Основная задача системного тестирования — в выявлении дефектов, связанных с работой системы в целом, таких как неверное использование ресурсов системы, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство в применении и тому подобное.

Системное тестирование производится над проектом в целом с помощью метода

«черного ящика». Структура программы не имеет никакого значения, для проверки доступны только входы и выходы, видимые пользователю. Тестированию подлежат коды и пользовательская документация.

Категории тестов системного тестирования:

Поскольку системное тестирование проводится на пользовательских интерфейсах, создается иллюзия того, что построение специальной системы автоматизации тестирования не всегда необходимо. Однако объемы данных на этом уровне таковы, что обычно более эффективным подходом является полная или частичная автоматизация тестирования, что приводит к созданию

тестовой системы гораздо более сложной, чем система тестирования, применяемая на уровне тестирования модулей или их комбинаций.

Пример системного тестирования приложения «Поступление подшипника на склад» В спецификации тестового случая задано состояние окружения (входные данные) и ожидаемая последовательность событий в системе (ожидаемый результат). После прогона тестового случая мы получаем реальную последовательность событий в системе (пример 1, пример 3) при заданном состоянии окружения. Сравнивая фактический результат с ожидаемым, можно сделать вывод о том, прошла или не прошла тестируемая система испытание на заданном тестовом случае. В качестве ожидаемого результата будем использовать спецификацию тестового случая, поскольку она определяет, как, для заданного состояния окружения, система должна функционировать.

Рисунок 1 - Краткое описание тестируемой системы 'Поступление подшипника на

Спецификация тестового случая №1:

Состояние окружения (входные данные — X):

Статус склада — 32. Пришел подшипник.

Статус обмена с терминалом подшипника (0 — есть подшипник) и его параметры —

"Статус=0 Диаметр=12".

Статус обмена с терминалом оси (1 — нет оси) и ее параметры — "Статус=1 Диаметр=12".

"Статус=1 Диаметр=12".

Статус команды — 0. Команда успешно принята. Сообщение от склада — 1. Команда успешно выполнена.

Ожидаемая последовательность событий (выходные данные – Y):

Система запрашивает статус склада (вызов функции GetStoreStat) и получает 32 Система запрашивает параметры подшипника (вызов функции GetRollerPar) и

получает Статус = 0 Диаметр=12

Система запрашивает параметры оси (вызов функции GetAxlePar) и получает Статус

= 1 Диаметр=0

Система добавляет в очередь команд склада на последнее место команду SendR (получить из приемника в ячейку) (вызов функции SendStoreCom) и получает сообщение о том, что команда успешно принята – статус = 0

Система запрашивает склад о результатах выполнения команды (вызов функции GetStoreMessage) и получает сообщение о том, что команда успешно выполнена — статус

= 1

Выходные данные (результаты выполнения Yв) – зафиксированы в журнале теста (пример 1)

ВЫЗОВ: GetStoreStat РЕЗУЛЬТАТ: 32

ВЫЗОВ: GetRollerPar

РЕЗУЛЬТАТ: Статус = 0 Диаметр = 12 ВЫЗОВ: GetAxlePar

РЕЗУЛЬТАТ: Статус = 1 Диаметр = 0 ВЫЗОВ: SendStoreCom РЕЗУЛЬТАТ: 0

ВЫЗОВ: GetStoreMessage РЕЗУЛЬТАТ: 1

Пример 2. Журнал теста

Приведенный на примере 22 тест был разработан в соответствии со спецификацией тестового случая №1. Детальная спецификация приведена в FS (Практикум, Приложение 1), результаты прогона показаны на примере 3.

Пример 2. Тест для системного тестирования Пример 2.1. Тест для системного тестирования (C++)

После завершения теста следует просмотреть текстовый журнал теста, чтобы выяснить, какая последовательность событий в системе была реально зафиксирована (выходные данные) и сравнить их с ожидаемыми результатами, заданными в спецификации тестового случая1. Пример журнала теста (пример 1):

Test started

CALL:GetStoreStat 0 RETURN:32

CALL:GetRollerPar

RETURN:0 NewUser Depot1 123456 1 12 1 1

CALL:GetAxlePar

RETURN:1 NewUser Depot1 123456 1 0 12 12

CALL:SendStoreCom 1 0 0 1 0 0 0 RETURN:0

CALL:GetStoreMessage RETURN:1

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕУЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ

по организации самостоятельной работы обучающихся
по дисциплине **«Администрирование баз данных»**
Направление подготовки 09.03.04 Программная инженерия
Направленность (профиль) «Разработка и сопровождение
программного обеспечения»

Ставрополь

2025

Введение

Подготовка бакалавра – сложный, комплексный процесс, в котором органически сочетается решение всех основных задач – образовательных, воспитательных и развивающих. Деятельность преподавателя в учебном процессе – ведущая. Однако, как бы ни были совершенны применяемые преподавателем в учебном процессе форм и методов обучения, без активной познавательной деятельности самих обучаемых они не могут давать желаемых результатов. Поэтому особое значение приобретает проблема сознательного отношения самих студентов к приобретению знаний, умений и навыков в результате самостоятельной работы.

Самостоятельная работа - это познавательная деятельность, выполняемая студентами самостоятельно, при частичном руководстве преподавателя. Самостоятельная работа предусмотрена учебной программой. Самостоятельная работа является управляемым процессом. Благодаря этому познавательно-практическая деятельность студентов становится целенаправленной, творчески осмысленной, содержательной.

Процесс управления самостоятельной работы служит главным целям обучения:

- усвоение, закрепление, совершенствование знаний в объеме вузовских программ;
- приобретение умений и навыков, составляющих содержание подготовки выпускника высшей школы;
- максимальная активность каждого обучающегося в овладении профессиональных компетенций (ПК-14, ПК-15), формируемых в результате освоения дисциплины.

1. Самостоятельная работа студентов на лабораторных и практических занятиях

1.1. Аудиторные занятия

Задачи:

- • расширение, углубление и детализация знаний, полученных на лекциях;
- • повышение уровня усвоения учебного материала (от уровня информации,
- полученной на лекциях, до уровня знаний за счет привития умений и навыков);
- • развитие научного мышления и речи студентов;
- • проверка и учет знаний;
- • развитие познавательной активности и привитие навыков самостоятельной
- работы, особенно с дополнительной и специальной литературой;
- • привитие навыков ведения коллективной беседы, участия в творческой
- дискуссии, умения аргументированно отстаивать свои взгляды.

Основное требование к подготовке и участию студентов в занятиях заключается в том, что студент должен самостоятельно готовиться к занятиям и творчески в них участвовать.

Этапы подготовки к занятиям включают:

- повторение уже имеющихся знаний по конспекту, а затем по учебнику;
- углубление знаний по теме с использованием рекомендованной
- литературы;
- выполнение конкретного задания (решение задач, составление отчетов и т.п.).

Особое место занимают лабораторные работы. Главная цель которых – быть связующим звеном теории изучаемой дисциплины с практической реализацией, что позволяет углублять и закреплять теоретические положения, получаемые студентами на лекции, проверять их применение в практике экспериментальным путем, знакомить студентов с оборудованием, приборами, вычислительной техникой, изучать на практике методы научных исследований.

Студенты обеспечиваются методическими указаниями к лабораторным работам, содержащими теоретическую информацию и конкретные практические задания.

Для студентов младших курсов указания даются подробно и детализировано. Для студентов старших курсов предусматривается большая самостоятельность при выполнении заданий. Т.е. они должны уже демонстрировать переход от уровня умений к уровню навыков.

Оформление лабораторных работ должно быть максимально приближено к уровню, на котором ведется экспериментальная научно-исследовательская работа в конкретной предметной области.

1.2. Самостоятельная работа студентов во внеаудиторное время.

Кроме наиболее распространенных в вузах форм самостоятельной работы во внеаудиторное время (написание рефератов, контрольных работ, курсовых работ/проектов и т.д.) учебные планы включают такие виды и соответственно такой компонент фонда оценочных средств, как собеседование (по конкретным вопросам) и выполнение предлагаемых тем для самостоятельной учебно-исследовательской работы.

2. Указания студентам по подготовке к собеседованию и материалов по темам самостоятельной работы

2.1. Указания представлены в виде рекомендаций студентам в ходе изучения вопросов, выносимых на собеседование и материалов тем самостоятельной работы.

Своевременное и качественное выполнение самостоятельной работы базируется на соблюдении настоящих рекомендаций и изучении рекомендованной литературы. Студент может дополнить список использованной литературы современными источниками, не представленными в списке рекомендованной литературы, и в дальнейшем использовать собственные подготовленные учебные материалы при написании рефератов, курсовых и выпускных квалификационных работ.

2.2. Рекомендации

2.2.1. Прежде чем приступить к изучению вопросов для собеседования и предложенной темы самостоятельной работы, необходимо.

Во-первых:

- уяснить суть вопроса (темы) на самостоятельную работу;
- провести подбор рекомендованной литературы;
- составить план работы, в котором определяются основные пункты предстоящей подготовки.

Составление плана дисциплинирует и повышает организованность в работе.

Во-вторых:

- начинать надо с изучения рекомендованной литературы.

Необходимо помнить, что на лекции обычно рассматривается не весь материал, а только его часть. Остальная его часть восполняется в процессе самостоятельной работы. В связи с этим работа с рекомендованной литературой обязательна (см. Приложение 1). Особое внимание при этом необходимо обратить на содержание основных положений и выводов, объяснение явлений и фактов, уяснение практического приложения рассматриваемых теоретических вопросов.

В процессе этой работы студент должен стремиться понять и запомнить основные положения рассматриваемого материала, примеры, поясняющие его, а также разобраться в иллюстративном материале.

Заканчивать подготовку следует составлением плана (оглавления) по изучаемому материалу. Это позволит дать целостный, сжатый ответ при собеседовании и представить целостную картину по предложенной тематике.

В процессе самостоятельной работы рекомендуется взаимное обсуждение материала (между студентами), во время которого закрепляются знания, а также приобретается практика в изложении и разъяснении полученных знаний, развивается речь.

При необходимости студент может обращаться за консультацией к преподавателю. Идя на консультацию, студенту необходимо хорошо продумать вопросы, которые требуют разъяснения.

При подготовке к собеседованию и выполнению самостоятельной работы по теме студент должен вести записи. В результате у него создается свой индивидуальный фонд дополнительных материалов для быстрого повторения прочитанного, для накопления знаний. Особенно важны и полезны записи тогда, когда в них находят отражение мысли, возникшие при самостоятельной работе.

Основные формы записи: план (простой и развернутый), выписки, тезисы.

Результаты этих записей могут быть представлены в различных формах.

План – это схема прочитанного материала, краткий (или подробный)

перечень вопросов, отражающих структуру и последовательность материала:

краткий

- воспроизведение наиболее важных положений и фактов источника;

подробный (развернутый)

- детализированный план, в котором достаточно подробные записи приводятся по тем пунктам плана, которые нуждаются в пояснении;
- это четко и кратко изложенные (сформулированные) основные положения в результате глубокого осмысливания материала;
- в нем могут присутствовать выписки, цитаты, тезисы.

Подробно составленный план позволяет эффективно использовать материал и получить желаемый результат.

2.2.2. При отчете студента о проделанной самостоятельной работе по предложенной теме, преподаватель может предложить студенту алгоритм действий, рекомендовать еще раз внимательно прочитать весь материал и все записи по теме самостоятельной работы, тщательно продумать свое устное выступление.

2.2.3. Устный отчет должен строиться свободно, убедительно и аргументированно.

2.2.4. Преподаватель следит, чтобы отчет не сводился к простому воспроизведению текста, не допускается и простое чтение. Необходимо, чтобы студент проявлял собственное отношение к тому, о чем он говорит, высказывал свое личное мнение, понимание, обосновывал его и мог сделать правильные выводы из сказанного. При этом студент может обращаться к записям собранного материала, использовать знания факты и наблюдения современной жизни и т. д.

2.2.5. В заключение преподаватель оценивает результаты работы студента по данной теме.

3. Критерий оценки самостоятельной работы студента

Оценка «зачтено» при ответах на вопросы при собеседовании и по итогам устной защиты выполненной самостоятельной работы выставляется студенту, если студент показал знание учебного материала. При этом возможно допустил неточные формулировки основных понятий и терминов или ряд незначительных неточностей, не исказивших существенно суть ответа.

Оценка «не зачтено» при ответах на вопросы при собеседовании и по итогам устной защиты выполненной самостоятельной работ выставляется студенту, если студент не знает значительной части учебного материала, допускает неправильную формулировку основных понятий и терминов, существенно исказившие суть вопроса.

Оценка «не зачтено» выставляется также, если студент не отвечает на вопросы при собеседовании, отсутствует план выполненной работы, нет объяснений основных понятий и терминов.

4. Заключение

Успешное освоение курса предполагает активное, творческое участие студента путем планомерной, повседневной работы как на лекциях, лабораторных (практических) занятиях, так и к подготовке ответов на вопросы к собеседованию и выполнению самостоятельной работы по предлагаемым темам.

Приложение 1

Рекомендации студентам по изучению рекомендованной литературы

Изучение дисциплины следует начинать с проработки рабочей программы по изучаемой дисциплине, особое внимание, уделяя целям и задачам, структуре и содержанию курса.

Студентам рекомендуется получить учебную литературу по дисциплине, необходимую для эффективной работы на всех видах аудиторных занятий, а также для самостоятельной работы по изучению дисциплины.

Приложение 2

Примерная тематика заданий для самостоятельной работы студентов

Студенты выполняют самостоятельную работу в соответствии с индивидуальным заданием, которое определяется примерными темами.

Примерные темы самостоятельной работы

1. Почему необходимо документировать false positive в баг-трекере?
2. Как пользовательские сценарии (user stories) преобразуют в тест-кейсы?
3. Какие принципы используют для группировки тест-кейсов в наборы?
4. Как избежать избыточности в наборах тест-кейсов?
5. Почему важно учитывать приоритет бизнес-логики при создании

наборов?
6. Какие типовые ошибки допускают при составлении чек-листов?
7. Как оптимизировать наборы тест-кейсов для регрессии?
8. Чем data-driven подход дополняет наборы тест-кейсов?
9. Как оценивают эффективность набора тест-кейсов?
10. Почему некоторые тест-кейсы нужно исключать из основного набора?
11. Приведите пример плохого набора тест-кейсов и предложите улучшения.
12. Как Continuous Testing интегрируется в CI/CD-пайплайн?
13. Какие тесты запускают на этапе Build в Jenkins?
14. Чем отличается Canary-развертывание от Blue-Green?
15. Как SonarQube помогает контролировать качество кода?
16. Почему Allure Report полезнее стандартных отчетов JUnit?
17. Как организовать тестирование в GitLab CI?
18. Какие тесты можно пропускать в fast CI-пайплайне?
19. Как мониторинг в Prod влияет на процесс тестирования?
20. Какие практики Chaos Engineering применимы в DevOps?
21. Как тестируют инфраструктуру как код (IaC)?
22. Какие параметры измеряют при нагрузочном тестировании?
23. Как OWASP Top 10 влияет на тестирование безопасности?
24. Какие инструменты используют для тестирования доступности?
25. Чем тестирование роботизированных систем отличается от классического ПО?
26. Как тестируют физические взаимодействия в устройствах?
27. Как симуляторы помогают в тестировании автономных систем?
28. Какие стандарты регулируют тестирование промышленных роботов?
29. Как тестируют отказоустойчивость в беспилотных автомобилях?
30. Приведите пример тест-кейса для дрона с учетом ветровой нагрузки.