МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования «Уральский государственный экономический университет» (УрГЭУ)

ОТЧЕТ ПО ПРАКТИКЕ

Студента	Круглова Анастасия Александровна
Курс	4
Форма обучения	Заочная
Год набора	2019
Направление подготовки	09.03.03 Прикладная информатика
Профиль	Прикладная информатика в экономике
Вид практики	производственная
Тип практики	производственная (проектно-технологическая)
Место практики	ПАО Банк Синара
Сроки практики	13.03.2023 - 08.04.2023

Екатеринбург
2023

СОДЕРЖАНИЕ ОТЧЕТА ПО ПРАКТИКЕ

Часть 1	Информация о руководителях практики	
Часть 2	Инструктажи по созданию безопасных условий прохождения практики обучающимся, отвечающие санитарным правилам и требованиям охраны труда	
Часть 3	Краткая характеристика места практики	
Часть 4	Выполнение индивидуального задания	
Часть 5	Отчетные документы	

Часть 1
ИНФОРМАЦИЯ О РУКОВОДИТЕЛЯХ ПРАКТИКИ[1]

Руководитель практики от Университета	
ФИО	Сазанова Лариса Анатольевна
должность	Доцент
ученая степень /ученое звание	Кандидат физико-математических наук
Кафедра	Информационных технологий и статистики
Телефон	
Электронный адрес	sazanovalarisa@rambler.ru
Реквизиты распорядительного акта о прохождении практики	Приказ №20/0603-04 от 06.03.2023
Ответственное лицо от профильной организации	
Полное наименование профильной организации (по уставу)	ПАО Банк Синара
ФИО	Брусницына Ольга Викторовна
должность	Управляющий
Телефон	83433558164
Электронный адрес	BrusticunaOV@gebank.ru

1. при прохождении практики в Университете информация заполняется по открытым источникам организации, изучаемой студентом в соответствии с индивидуальным заданием.

Часть 2
ИНСТРУКТАЖИ ПО СОЗДАНИЮ БЕЗОПАСНЫХ УСЛОВИЙ ПРОХОЖДЕНИЯ ПРАКТИКИ ОБУЧАЮЩИМСЯ, ОТВЕЧАЮЩИЕ САНИТАРНЫМ ПРАВИЛАМ И ТРЕБОВАНИЯМ ОХРАНЫ ТРУДА

Дата проведения	Вид инструктажа	Ф.И.О., должность, подпись проводившего инструктаж[1]	Подпись **обучающегося**, прошедшего инструктаж
1	2	3	4
1.1. Прохождение инструктажа по технике безопасности			
13.03.2023	Первичный		
1.2. Инструктаж по охране труда			
13.03.2023	Первичный		
1.3. Инструктаж по правилам внутреннего распорядка			
13.03.2023	Первичный		
1.4. Инструктаж по санитарным правилам (при наличии требований)			
13.03.2023	Первичный		

1. При прохождении практики в Университете в столбце 3 расписывается руководитель практики от университета (**только** для подразделений ИНДО допускается вписывание ФИО руководителей)

Часть 3
КРАТКАЯ ХАРАКТЕРИСТИКА МЕСТА ПРАКТИКИ

Полное наименование места практики	ПАО Банк Синара
Адрес:	Г. Екатеринбург, ул. 8Марта, д.13
телефон:	83433558164
E-mail:	BrusticunaOV@gebank.ru
Официальный сайт	https://sinara.ru/chastnym-licam
Руководитель организации	Ошев Денис Геннадьевич Руководитель ПРЕДСЕДАТЕЛЬ ПРАВЛЕНИЯ БАНКА
Правоустанавливающие документы	Устав ПАО Банк Синара от 27.07.2022
Основные направления деятельности	-модернизация систем управления ИТ-инфраструктурой -привлечение вкладов населения - обслуживание корпоративных клиентов - работа на рынке ценных бумаг и на межбанковском рынке

Часть 4

ВЫПОЛНЕНИЕ ИНДИВИДУАЛЬНОГО ЗАДАНИЯ (ОТЧЕТНЫЕ ДОКУМЕНТЫ)

(текущий контроль)

Виды работ: из индивидуального задания	Конкретные даты выполнения из индивидуального задания	Вид отчетного документа из индивидуального задания	Краткое описание студентом алгоритма выполнения задания	Наименование отчетных документов
Виды работ: Прохождение инструктажа, ознакомление с предприятием	19.03.2023	Вид отчетного документа: Приложение 1 к отчету		Не загружено
Виды работ: Сбор данных, анализ используемых информационных систем и технологий, программных и технических средств, технологических особенностей деятельности организации. Формирование стратегии информатизации прикладных процессов и совершенствования информационной	02.04.2023	Вид отчетного документа: Приложение 2 к отчету		Не загружено
Виды работ: Подготовка отчета по практике	08.04.2023	Вид отчетного документа: Отчет		Не загружено

Заключение ответственного лица от профильной организации о результатах прохождения практики и выполнении обучающимся индивидуального задания

За время работы Круглова Анастасия Александровна проявила себя как грамотный, добросовестный, ответственный стажер, вдумчивый, самостоятельный исследователь.

Индивидуальное задание выполнено в установленные сроки.

Программа прохождения практики выполнена в полном объеме.

Профессиональные компетенции, запланированные в программе практики, в период прохождения практики освоены полностью на высоком уровне.

Замечания, высказанные в ходе индивидуальных консультаций и подготовки отчета, учтены и исправлены.

Отчет соответствует всем требованиям.

По результатам работы заслуживает положительной оценки.

Ответственное лицо от

профильной организации _____ / _____

подпись **дата**

м.п.

Ознакомлен

Круглова Анастасия Александровна

_____________________________ подпись   дата ___________________

Приложение № 1

к договору о практической подготовке обучающегося в форме практики

№ 20/0603-04 от 06.03.2023

Информация о фамилии, имени, отчестве обучающегося, группе, курсе, форме обучения, годе набора, направлении подготовки, профилю ОПОП, виде, типе, сроках практики

ФИО обучающегося	Круглова Анастасия Александровна
Группа	ЗПИЭ-19-1
Курс	4 курс
Форма обучения	Заочная
Год набора	2019
Направление подготовки	09.03.03 Прикладная информатика
Профиль ОПОП	Прикладная информатика в экономике
Вид практики	Производственная
Тип практики	проектно-технологическая
Сроки практики	13.03.2023 – 08.04.2023

Приложение № 2

к договору о практической подготовке обучающегося в форме практики

№20/0603-04 от 06.03.2023

Справка

о материально-техническом и кадровом обеспечении практики

в профильной организации

1. Перечень помещений Профильной организации, их материально-технические характеристики, оценка рабочего места.

Адрес помещений профильной организации, используемого для организации практической подготовки,

Свердловская обл., г. Екатеринбург, ул. 8 марта 13

Перечень помещений Профильной организации

ПАО Банк Синара

Проведена оценка условий труда на рабочих местах, используемых для проведения практики:

- обеспечены безопасные условия реализации компонентов образовательной программы в форме практики;

- соблюдены требования, установленные правилами техники безопасности, правилами противопожарной безопасности, правилами охраны труда, санитарно-эпидемиологическими правилами и гигиеническими нормативами;

- представлен необходимый перечень техники, оборудования, лицензионного обеспечения, необходимого для организации практической подготовки обучающегося;

Аттестация рабочих мест проведена в соответствии с требованиями законодательства РФ.

2. Кадровое обеспечение практики.

Ответственное лицо от профильной организации из числа работников, **Брусницына Ольга Викторовна Управляющий офиса ДО «Банковский»** соответствует требованиям статьи 331 Трудового кодекса Российской Федерации.

Приложение № 3

к договору о практической подготовке обучающегося в форме практики

№ 20/0603-04 от 06.03.2023

Перечень отчетных документов по результатам прохождения практики обучающимся

1. Договор о практической подготовке обучающегося в форме практики
2. Совместный рабочий график проведения практики
3. Индивидуальное задание обучающегося
4. Отчет о практике с положениями

СОГЛАСОВАНО:

Руководитель практики от Университета

_____ Сурнина Н. М.

(подпись)

СОГЛАСОВАНО:

Ответственное лицо от профильной организации

_____Кислицын Е.В.

(подпись)

СОВМЕСТНЫЙ РАБОЧИЙ ГРАФИК ПРОВЕДЕНИЯ ПРАКТИКИ

Студента	Круглова Анастасия Александровна
Курс	3
Форма обучения	Заочная
Год набора	2019
Профиль	Прикладная информатика в экономике
Наименование практики	Производственная
Тип практики	Научно-исследовательская работа

МЕРОПРИЯТИЯ

1. Создание безопасных условий прохождения практики обучающимся, отвечающие санитарным правилам и требованиям охраны труда.	
Мероприятия	Вид инструктажа
1.1. Прохождение инструктажа по технике безопасности	первичный
1.2. Инструктаж по охране труда	первичный
1.3. Инструктаж по правилам внутреннего распорядка	первичный
1.4. Инструктаж по санитарным правилам (при наличии требований)	первичный
2. Создание необходимых условий для выполнения обучающимися программы практики и индивидуальных заданий.	
2.1. Назначить руководителей практики обучающихся от профильной организации - из числа работников профильной организации	14.03.2022
2.2. Назначить руководителей практики обучающихся от УрГЭУ - из числа ППС	14.03.2022
2.3. Распределить обучающихся по рабочим местам и видам	14.03.2022

работ	
2.4. Утвердить индивидуальное задание	14.03.2022
3. Осуществлять контроль за исполнение индивидуального задания, соблюдением сроков проведения практики и соответствием ее содержания требованиями, установленными ОПОП ВО (оформление отчета)	
3.1 Текущий контроль	
Представление отчетных документов по практике	Конкретные даты и мероприятия закреплены в индивидуальном задании
3.2. Промежуточный контроль	
Защита практики	Конкретные даты и мероприятия закреплены в индивидуальном задании

Ознакомлен:

Круглова Анастасия Александровна 

_____ 14.03.2022___

(подпись) (дата)

СОГЛАСОВАНО:

Руководитель практики от Университета

_____ Сурнина Н. М.

(подпись)

СОГЛАСОВАНО:

Ответственное лицо от профильной организации

_____Кислицын Е.В.

(подпись)

ИНДИВИДУАЛЬНОЕ ЗАДАНИЕ ОБУЧАЮЩЕГОСЯ

Студента Круглова Анастасия Александровна

Часть 1. Отчета	
Информация о руководителях практик	14.03.2022
Часть 2. Отчета	
Прохождение инструктажей: - по технике безопасности	14.03.2022

- по охране труда - по правилам внутреннего распорядка - по санитарным правилам (при наличии требований)			
Часть 3. Отчета			
Краткая характеристика места практики (или той организации, деятельность которой изучает обучающийся)		19.03.2022	
Часть 4. Отчета и приложения			
Этап 1. Определение цели и задач практики	Вид отчетного документа	Приложение к отчету	14.03.2022- 19.03.2022
Этап 2. Проведение анализа предприятия. Изучение используемых технологических и информационных систем и технологий, программных и технических средств.	Вид отчетного документа	Приложение к отчету	21.03.2022- 26.03.2022
Этап 3. Формирование стратегии информатизации прикладных процессов и создания информационной системы.	Вид отчетного документа	Приложение к отчету	28.03.2022- 09.04.2022
Текущий контроль			
Загрузка в портфолио договора		14.03.2022	
Загрузка в портфолио подписанного индивидуального плана		14.03.2022	
Загрузка в портфолио подписанного совместного рабочего графика проведения практики		14.03.2022	
Сдача отчета руководителю от профильной организации		08.04.2022	
Сдача отчета руководителю от университета (загрузка в портфолио)		09.04.2022	
Промежуточный контроль			
Защита практики (защита отчета)		По расписанию	

Ознакомлен:

Круглова Анастасия Александровна  14.03.2022

Приложение № 4

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Уральский государственный экономический университет»

(УрГЭУ)

ОТЧЕТ О ПРОИЗВОДСТВЕННОЙ (ПРОЕКТНО-ТЕХНОЛОГИЧЕСКОЙ) ПРАКТИКЕ

Место практики ПАО БАНК СИНАРА

Сроки практики с 13.03.2023 по 08.04.2023

Формирующее подразделение Институт непрерывного и дистанционного образования	**Студент** Круглова Анастасия Александровна
Направление 09.03.03 Прикладная информатика	**Группа** ЗПИЭ-19-1
Направленность Прикладная информатика в экономике	**Руководитель практики от Университета** Кандидат физико-математических наук, доцент
Кафедра Кафедра информационных технологий и статистики	**Руководитель практики от Организации** Управляющий ДО «Банковский» Брусницына Ольга Викторовна

Оценка: _____

Екатеринбург

2023

СОДЕРЖАНИЕ

Введение

Тестированием программного обеспечения представляет собой исследование, исполненное для определения заинтересованным сторонам информации о качестве продукта или сервиса при их испытании. Тестирование программного обеспечения также может дать объективную, независимую от программного обеспечения оценку, что позволит бизнес - сектору оценить и понять риски, связанные с реализацией тестируемого программного обеспечения. Методы тестирования включают в себя процесс выполнения программы или приложения с целью обнаружения ошибок в программном обеспечении (ошибки или другие дефекты).

Тестирование программного обеспечения включает в себя выполнение программного компонента или компонента системы для оценки одного или нескольких необходимых свойств, представляющих интерес. В целом, эти свойства показывают уровень, где компонент или тестируемая система:

- **отвечает требованиям, которыми руководствовались при их проектировании и разработке,**
- **правильно отвечают на все виды входящих данных,**
- **выполняют свои функции в течение приемлемого времени,**
- **достаточно удобны,**
- **могут быть установлены и корректно работают в предназначенных для них средах,**
- **достигают общего результата, желаемого заинтересованными сторонами.**

По мере того как число возможных тестов, даже для самых простых программных компонентов, практически бесконечно, все тестировщики программного обеспечения используют некоторую стратегию, чтобы выбрать тесты, которые являются выполнимыми для имеющегося времени и ресурсов. В результате, тестировщики программного обеспечения пишут такие тесты, которые, как правило, пытаются выполнить программу или приложение с целью обнаружения ошибок в программном обеспечении (ошибки или другие дефекты). Задача тестирования представляет собой повторяющийся процесс, ведь когда одна ошибка исправлена, она может осветить другие, более сложные ошибки на более глубоком уровне, или вообще может создать новые.

Тестирование программного обеспечения может предоставить объективную, независимую информацию о качестве программного обеспечения и риска воспроизведения ошибок пользователей.

Тестирование программного обеспечения может быть проведено, только когда выполняется тестируемое программное обеспечение. Общий подход к разработке программного обеспечения часто определяет, когда и как проводится тестирование. Например, в поэтапном процессе, большинство тестов проводится после того, как требования к системе определены и затем реализованы в тестируемых программ. Но

зачастую все процессы по разработке - Agile подход, выставление требований, программирование и тестирование часто выполняются одновременно.

Целью данной научно-исследовательской работы является анализ основ тестирования, видов и методов тестирования ПО, функционального и структурного тестирования, организации тестирования ПО.

1 ОСНОВЫ ТЕСТИРОВАНИЯ ПО

1.1 ТЕСТИРОВАНИЕ КАК ЭЛЕМЕНТ ЖИЗНЕННОГО ЦИКЛА ПО. УРОВНИ ТЕСТИРОВАНИЯ

Тестирование программ зародилось практически одновременно с программированием. Машинное время стоило дорого, поэтому предприятия с повышенными требованиями к надежности программ (например, авиакосмическая промышленность) стали активно разрабатывать методики тестирования.

Долгое время было принято считать, что целью тестирования является доказательство отсутствия ошибок в программе. Однако этот тезис не выдерживает критики, т. к. полный перебор всех возможных вариантов выполнения программы находится за пределами вычислительных возможностей даже для очень небольших программ. Поэтому никакое тестирование не может гарантировать отсутствия ошибок.

Со временем понимание целей тестирования изменилось, и один из основоположников тестирования Гленфорд Майерс предложил следующее определение: «Тестирование – это процесс выполнения программ с целью обнаружения ошибок».

Сегодня дается следующее определение понятия "тестирование":

Тестирование программного обеспечения (Software Testing) — проверка соответствия реальных и ожидаемых результатов поведения программы, проводимая на конечном наборе тестов, выбранном определённым образом [6].

Цель тестирования — проверка соответствия ПО предъявляемым требованиям, обеспечение уверенности в качестве ПО, поиск очевидных ошибок в программном обеспечении, которые должны быть выявлены до того, как их обнаружат пользователи программы[5].

Отладка (debugging) не является разновидностью тестирования. Отладка направлена на установление точной природы известной ошибки, а затем – на исправление этой ошибки.

Баг (bug, дефект) — это отклонение фактического результата (actual result) от ожидаемого результата (expected result).

Отчет о дефекте (bug report, баг-репорт, отчет об ошибке) — это документ, содержащий отчет о любом недостатке в компоненте или системе, который может привести компонент или систему к невозможности выполнить требуемую функцию.

Система отслеживания ошибок (bug tracking system) — это программа учета и/или контроля багов. Ее обычно называют баг-трекер или баг-трекинг. Это могут быть как специально разработанные программы, так и обычные гугл-таблицы. Суть в том, что в одном месте собраны все наши отчеты об ошибках и в любое время можно получить доступ к любому отчету.

Билд (build) — это продукт или часть продукта, который можно тестировать.

Релиз или RTM (англ. Release to manufacturing — промышленное издание) — это издание продукта, готового к тиражированию.

Тест-дизайн — это этап тестирования ПО, на котором проектируются и создаются тестовые случаи (тест-кейсы).

Тест план (Test Plan) — это документ, который описывает 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.

Чек-лист (check list) — это документ, который описывает что должно быть протестировано. Чек-лист может быть абсолютно разного уровня детализации.

Чаще всего чек-лист содержит только действия, без ожидаемого результата. Чек-лист менее формализован.

Тестовый сценарий (test case) — это артефакт, описывающий совокупность шагов, конкретных условий и параметров, необходимых для проверки реализации тестируемой функции или её части.

Верификация (verification) — это процесс оценки системы, чтобы понять, удовлетворяют ли результаты текущего этапа разработки условиям, которые были сформулированы в его начале.

Валидация (validation) — это определение соответствия, разрабатываемого ПО ожиданиям и потребностям пользователя, его требованиям к системе.

Тестовое покрытие — это одна из метрик оценки качества тестирования, представляющая из себя плотность покрытия тестами требований либо исполняемого кода.

Тестирование в жизненном цикле разработки программного обеспечения

До начала 80-х годов процесс тестирования программного обеспечения был разделен с процессом разработки: вначале программисты реализовывали заданную функциональность, а затем тестировщики приступали к проверке качества созданных программ. Такая модель жизненного цикла разработки ПО называется каскадной (или водопадной) и состоит из 5 основных этапов: разработка требований к программе, проектирование, реализация, тестирование и сопровождение. Однако описанный выше подход создает множество проблем. Например, разработка программ может оказаться достаточно длительной (скажем, несколько месяцев), чем тогда в это время должны заниматься тестировщики? Другая, более серьезная проблема заключается в плохой предсказуемости результатов такого процесса разработки. Ключевым вопросом здесь может быть: какое количество времени потребуется на завершение продукта, в котором существует 500 известных ошибок? На самом деле, предугадать это совершенно невозможно, так как разные ошибки будут требовать разного количества времени на исправление, а исправление известных ошибок будет неизбежно связано с внесением новых. Существует следующая мрачная статистика: даже однострочное изменение в программе с вероятностью 55 % либо не исправляет старую ошибку, либо вносит новую. Если же учитывать изменения любого объема, то в среднем менее 20 % изменений корректны с первого раза.

В связи с этим, в 90-х годах появилась другая методика разработки, которую вслед за компанией Microsoft называют zero-defect mindset. Основная идея этой методики заключается в том, что качество программ проверяется не post factum, а постоянно в процессе разработки. Например, программист не может перейти к разработке новой функциональности, если существуют известные ошибки высокого приоритета в частях, разработанных им ранее. Так появились шарнирно-каскадная (или V-образная), а также спиралевидная модели разработки ПО.

При такой постановке вопроса тестирование становится центральной частью каждого этапа жизненного цикла разработки программ: тестированию подвергаются

требования к программному продукту, алгоритмы, исходные коды, модули, программные сборки, функциональность программ и т. д. Практика показывает, что, чем раньше найдена ошибка, тем дешевле ее исправить. Частым примером в литературе является следующий: стоимость незамеченной ошибки в документе требований, которую можно оценить в 2 $, вырастает в 200 $ на этапе сопровождения, поскольку были затрачены силы и время на все предыдущие этапы.

На разных этапах жизненного цикла ПО тестирование проводится в разных формах:

- **на этапе определения требований: их анализ и верификация также могут считаться тестированием;**
- **контроль процесса проектирования на этапе разработки дизайна системы – это тоже форма тестирования;**
- **как уже упоминалось, разработчики тоже участвуют в тестировании на уровне модульного тестирования.**

Труднее определить критерий окончания тестирования, поскольку, согласно принципам тестирования, мы никогда не можем быть уверены в том, что программа на 100% свободна от дефектов. Поэтому используются другие условия:

- **граничные сроки, установленные заранее;**
- **выполнение всех предусмотренных тест-кейсов;**
- **достижение определенного уровня тестового покрытия;**
- **когда после определенного момента, мы практически не находим новых багов или критических дефектов;**
- **решение менеджмента.**

Уровни тестирования[5]

Модульное тестирование — проводится для тестирования какого-либо одного логически выделенного и изолированного элемента (модуля) системы в коде. Проводится самими разработчиками, так как предполагает полный доступ к коду.

Цель: проверка правильности реализации функциональных / нефункциональных требований в модуле, раннее обнаружение ошибок.

Объект: модуль / компонент / unit.

Базис: дизайн системы, код, спецификация компонента.

Типичные ошибки: ошибка в реализации требований, ошибка в коде.

Ответственный: разработчик (редко тестировщик).

На этом уровне тестирования создаются модульные тесты (unit тесты), которые проверяют правильность работы модуля в тестовых условиях. Эти проверки всегда автоматизированы и выполняются очень быстро (несколько тысяч тестов в минуту).

Unit тесты, кроме поиска ошибок, также помогают оценивать качество кода, измерять покрытие кода тестами, сокращать время и затраты на тестирование.

Интеграционное тестирование — тестирование, направленное на проверку корректности взаимодействия нескольких модулей, объединенных в единое целое.

Цель: проверка правильности реализации взаимодействия между компонентами / модулями / частями системы.

Объект: модули, состоящие из нескольких компонентов; подсистемы, API, микросервисы.

Базис: дизайн системы, архитектура системы, описание связей компонентов.

Типичные ошибки: отсутствие / неправильные связи между элементами системы, неправильные передаваемые данные, отсутствие обработки ошибок, отказы и падения при обращениях к API.

Ответственный: разработчик и тестировщик.

Интеграционные тесты выполняются дольше (несколько десятков в минуту), чем модульные интеграционные тесты (несколько сотен-тысяч в минуту) и являются более творческими.

Системное тестирование — процесс тестирования системы, на котором проводится не только функциональное тестирование, но и оценка характеристик качества системы — ее устойчивости, надежности, безопасности и производительности.

Цель: проверка работы системы в целом.

Объект: система, конфигурации системы, рабочее окружение.

Базис: системные требования, бизнес требования, сценарии использования, User Stories, системные руководства, инструкции.

Типичные ошибки: невозможность выполнять функциональные задачи, для которых создавалась система, неправильная передача данных внутри системы, неспособность системы работать правильно в среде эксплуатации, нефункциональные сбои (уязвимости, зависания, выключения).

Ответственный: тестировщик.

Системное тестирование может включать в себя различные типы тестирования. Эти тесты все чаще автоматизируется и именно этот вид автоматизации сейчас очень востребован (JAVA, Python, JavaScript, C#, Selenium и т.п.)

Приёмочное тестирование — проверяет соответствие системы потребностям, требованиям и бизнес-процессам пользователя.

Существуют несколько форм приемочного тестирования:

Пользовательское приемочное тестирование (User Acceptance testing, UAT) — проверяет пригодность системы к эксплуатации конечными пользователями.

Контрактное приемочное тестирование — проводится в соответствии с критериями, указанными в контракте приемки специального ПО.

Альфа-тестирование (alpha testing) и бета-тестирование (beta-testing) — используются для получения обратной связи от потенциальных или существующих клиентов.

Альфа-тестирование проводится “внутри” компании, без участия разработчиков / тестировщиков продукта.

Бета-тестирование проводится реальными пользователями системы.

Цель: проверка готовности системы.

Объект: система, конфигурация системы, бизнес процессы, отчеты, аналитика.

Базис: системные требования, бизнес требования, сценарии использования, User Stories.

Типичные ошибки: бизнес-требования неправильно реализованы, система не соответствует требованиям контракта.

Ответственный: заказчик / клиент / бизнес-аналитик / product owner и тестировщик.

Количество тестов на приемочном уровне намного меньше, чем на других уровнях, потому что в этот момент времени вся система уже проверена. Приемочные тесты практически никогда не автоматизируются.

1.2 ВИДЫ И МЕТОДЫ ТЕСТИРОВАНИЯ

1.2.1 ВИДЫ ТЕСТИРОВАНИЯ ПО ЦЕЛИ

Все виды тестирования программного обеспечения, в зависимости от преследуемых целей, можно условно разделить на следующие группы[1]:

- **Функциональные.**
- **Нефункциональные.**
- **Связанные с изменениями.**

Функциональные виды тестирования

Функциональные тесты базируются на функциях и особенностях, а также на взаимодействии с другими системами и могут быть представлены на всех уровнях тестирования: компонентном или модульном (Component/Unit testing), интеграционном (Integration testing), системном (System testing), приемочном (Acceptance testing). Функциональные виды тестирования рассматривают внешнее поведение системы. Далее перечислены одни из самых распространенных видов функциональных тестов.

Функциональное тестирование рассматривает заранее указанное поведение и основывается на анализе спецификаций функциональности компонента или системы в целом.

1. Функциональные тесты основываются на функциях, выполняемых системой, и могут проводиться на всех уровнях тестирования (компонентном, интеграционном, системном, приемочном). Как правило, эти функции описываются в требованиях, функциональных спецификациях или в виде случаев использования системы (use cases).

Тестирование функциональности может проводиться в двух аспектах[1]:

- **Требования.**
- **Бизнес-процессы.**

Тестирование в аспекте «требования» использует спецификацию функциональных требований к системе, как основу для дизайна тестовых случаев (Test Cases). В этом случае необходимо сделать список того, что будет тестироваться, а что нет, приоритезировать требования на основе рисков (если это не сделано в документе с требованиями), а на основе этого приоритезировать тестовые сценарии (test cases). Это позволит сфокусироваться и не упустить при тестировании наиболее важный функционал.

Тестирование в аспекте «бизнес-процессы» использует знание бизнес-процессов, которые описывают сценарии ежедневного использования системы. В этом аспекте

тестовые сценарии (test scripts), как правило, основываются на случаях использования системы (use cases).

2. Тестирование безопасности (Security and Access Control Testing)

Тестирование безопасности - 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.

Общая стратегия безопасности основывается на трех основных принципах:

- **Конфиденциальность.**
- **Целостность.**
- **Доступность.**

Конфиденциальность - это сокрытие определенных ресурсов или информации. Под конфиденциальностью можно понимать ограничение доступа к ресурсу некоторой категории пользователей или, другими словами, при каких условиях пользователь авторизован получить доступ к данному ресурсу.

Существует два основных критерия при определении понятия целостности:

- **Доверие. Ожидается, что ресурс будет изменен только соответствующим способом определенной группой пользователей.**
- **Повреждение и восстановление. В случае, когда данные повреждаются или неправильно меняются авторизованным или не авторизованным пользователем, Вы должны определить, на сколько важной является процедура восстановления данных.**
- **Доступность. Доступность представляет собой требования о том, что ресурсы должны быть доступны авторизованному пользователю, внутреннему объекту или устройству. Как правило, чем более критичен ресурс, тем выше уровень доступности должен быть.**

3. Тестирование взаимодействия или Interoperability Testing

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

Программное обеспечение с хорошими характеристиками взаимодействия может быть легко интегрировано с другими системами, не требуя каких-либо серьезных модификаций. В этом случае, количество изменений и время, требуемое на их выполнение, могут быть использованы для измерения возможности взаимодействия.

Нефункциональные виды тестирования[1]

Нефункциональное тестирование описывает тесты, необходимые для определения характеристик программного обеспечения, которые могут быть измерены различными величинами. В целом, это тестирование того, как система работает.

1.Все виды тестирования производительности

Тестирование производительности (Performance testing).

Задачей тестирования производительности является определение масштабируемости приложения под нагрузкой, при этом происходит:

Измерение времени выполнения выбранных операций при определенных интенсивностях выполнения этих операций.

Определение количества пользователей, одновременно работающих с приложением.

Определение границ приемлемой производительности при увеличении нагрузки (при увеличении интенсивности выполнения этих операций).

Исследование производительности на высоких, предельных, стрессовых нагрузках.

Стрессовое тестирование (Stress Testing)

Стрессовое тестирование позволяет проверить, насколько приложение и система в целом работоспособны в условиях стресса, а 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. Также, одной из задач при стрессовом тестировании может быть оценка деградации производительности. Таким образом, цели стрессового тестирования могут пересекаться с целями тестирования производительности.

Объемное тестирование (Volume Testing)

Задачей объемного тестирования является получение оценки производительности при увеличении объемов данных в базе данных приложения, при этом происходит:

Измерение времени выполнения выбранных операций при определенных интенсивностях выполнения этих операций.

Может производиться определение количества пользователей, одновременно работающих с приложением.

Тестирование стабильности или надежности (Stability / Reliability Testing)

Задачей тестирования стабильности (надежности) является проверка работоспособности приложения при длительном (многочасовом) тестировании со средним уровнем нагрузки. Время выполнения операций может играть в данном виде тестирования второстепенную роль. При этом на первое место выходит отсутствие утечек памяти, перезапусков серверов под нагрузкой и другие аспекты влияющие именно на стабильность работы.

В англоязычной терминологии вы можете так же найти еще один вид тестирования - Load Testing - тестирование реакции системы на изменение нагрузки (в пределе допустимого). Нам показалось, что Load и Performance преследуют все же одну и ту же цель: проверка производительности (времени отклика) на разных нагрузках. Собственно, поэтому мы и не стали разделять их. В то же время кто-то может разделить. Главное все-таки понимать цели того или иного вида тестирования и постараться их достигнуть.

2. Тестирование установки (Installation Testing)

Тестирование установки направленно на проверку успешной инсталляции и настройки, а также на обновление или удаление программного обеспечения.

В настоящий момент, наиболее распространена установка ПО при помощи инсталляторов (специальных программ, которые сами по себе так же требуют надлежащего тестирования, описание которого рассмотрено в разделе "Особенности тестирования инсталляторов").

В реальных условиях инсталляторов может не быть. В этом случае придется самостоятельно выполнять установку программного обеспечения, используя документацию в виде инструкций или "read me" файлов, шаг за шагом описывающих все необходимые действия и проверки.

В распределенных системах, где приложение разворачивается на уже работающем окружении, простого набора инструкций может быть мало. Для этого часто пишется план установки (Deployment Plan), включающий не только шаги по инсталляции приложения, но и шаги отката (roll-back) к предыдущей версии (в случае неудачи). Сам по себе план установки также должен пройти процедуру тестирования для избежания проблем при выдаче в реальную эксплуатацию. Особенно это актуально, если установка выполняется на системы, где каждая минута простоя - это потеря репутации и большого количества средств. Например: банки, финансовые компании или даже баннерные сети. Поэтому тестирование установки можно назвать одной из важнейших задач по обеспечению качества программного обеспечения.

3. Тестирование удобства пользования (Usability Testing)

Иногда мы сталкиваемся с непонятными или нелогичными приложениями, многие функции и способы использования которых часто не очевидны. После такой работы редко возникает желание использовать приложение снова, и мы ищем более удобные аналоги. Для того, чтобы приложение было популярным, ему мало быть функциональным – оно должно быть еще и удобным. Если задуматься, интуитивно понятные приложения экономят нервы пользователям и затраты работодателя на обучение. Значит, они более конкурентоспособные! Поэтому тестирование удобства использования, о котором пойдет речь далее, является неотъемлемой частью тестирования любых массовых продуктов.

Тестирование удобства пользования - это метод тестирования, направленный на установление степени удобства использования, обучаемости, понятности и привлекательности для пользователей разрабатываемого продукта в контексте заданных условий.

Тестирование удобства пользования дает оценку уровня удобства использования приложения по следующим пунктам:

Производительность, эффективность (efficiency) - сколько времени и шагов понадобится пользователю для завершения основных задач приложения, например, размещение новости, регистрации, покупка и т.д. (меньше - лучше).

Правильность (accuracy) - сколько ошибок сделал пользователь во время работы с приложением (меньше - лучше).

Активизация в памяти (recall) – как много пользователь помнит о работе приложения после приостановки работы с ним на длительный период времени (повторное выполнение операций после перерыва должно проходить быстрее, чем у нового пользователя).

Эмоциональная реакция (emotional response) – как пользователь себя чувствует после завершения задачи - растерян, испытал стресс? Порекомендует ли пользователь систему своим друзьям? (положительная реакция - лучше).

Виды тестирования связанные с изменениями.

После проведения необходимых изменений, таких как исправление дефектов, программное обеспечение должно быть протестировано заново, для подтверждения того факта, что проблема была решена.

Ниже перечислены виды тестирования, которые необходимо проводить после установки программного обеспечения, для подтверждения работоспособности приложения или правильности осуществленного исправления дефекта:

1. **Дымовое тестирование (Smoke Testing)**
2. **Регрессионное тестирование (Regression Testing)**
3. **Тестирование сборки (Build Verification Test)**
4. **Санитарное тестирование или проверка согласованности/исправности (Sanity Testing)**

Дымовой тест (англ. Smoke testing или smoke test, дымовое тестирование) — в тестировании программного обеспечения означает минимальный набор тестов на явные ошибки. Дымовой тест обычно выполняется программистом; не проходившую этот тест программу не имеет смысла отдавать на более глубокое тестирование.

Регрессио́нное тести́рование (англ. regression testing, от лат. regressio — движение назад) — собирательное название для всех видов тестирования программного обеспечения, направленных на обнаружение ошибок в уже протестированных участках исходного кода.

Тестирование сборки (Build Verification Test), как и дымное тестирование, направленно для предварительной проверки разрабатываемого программного продукта перед запуском полномасштабного тестирования по всем параметрам, проводимого командой тестировщиков. Проводится оно для того, чтобы знать – готов ли релиз для такого этапа разработки ПО, как Тестирование или же он еще нуждается в доработке.

Тестирование сборки состоит из набора коротких тестов, которые и определяют готовность сборки.

Основной задачей данного вида тестирования является экономия времени команды тестировщиков, в случае, если релиз имеет серьезные проблемы со своей готовностью к полному циклу тестирования.

Санитарное тестирование - это 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.

Цель санитарного тестирования - проверить «рациональность» системы после изменений, чтобы продолжить более тщательное тестирование.

1.2.2 ВИДЫ ТЕСТИРОВАНИЯ ПО СТЕПЕНИ АВТОМАТИЗАЦИИ

При ручном тестировании (manualtesting) тестировщики вручную выполняют тесты, не используя никаких средств автоматизации. Ручное тестирование – самый низкоуровневый и простой тип тестирования, не требующих большого количества дополнительных знаний.

Тем не менее, перед тем как автоматизировать тестирование любого приложения, необходимо сначала выполнить серию тестов вручную. Мануальное тестирование требует значительных усилий, но без него мы не сможем убедиться в том, возможна ли автоматизация в принципе. Один из фундаментальных принципов тестирования гласит: 100% автоматизация невозможна. Поэтому, ручное тестирование – необходимость.

Мифы о ручном тестировании[2]:

– *кто угодно может провести ручное тестирование.*

Нет, выполнение любого вида тестирования требует специальных знаний и профессиональной подготовки.

– *автоматизированное тестирование мощнее ручного.*

Полная автоматизация невозможна. Необходимо использовать также и ручное тестирование.

– *ручное тестирование – это просто.*

Тестирование может быть очень непростым занятием. Проведение тестирования для проверки максимально возможного количества путей выполнения с использованием минимального числа тест-кейсов требует серьезных аналитических навыков.

Автоматизированное тестирование предполагает использование специального программного обеспечения (помимо тестируемого) для контроля выполнения тестов и сравнения ожидаемого фактического результата работы программы. Этот тип тестирования помогает автоматизировать часто повторяющиеся, но необходимые для максимизации тестового покрытия задачи.

Некоторые задачи тестирования, такие как низкоуровневое регрессионное тестирование, могут быть трудозатратными и требующими много времени если выполнять их вручную. Кроме того, мануальное тестирование может недостаточно эффективно находить некоторые классы ошибок. В таких случаях автоматизация может помочь сэкономить время и усилия проектной команды.

После создания автоматизированных тестов, их можно в любой момент запустить снова, причем запускаются и выполняются они быстро и точно. Таким образом, если есть необходимость частого повторного прогона тестов, значение автоматизации для упрощения сопровождения проекта и снижения его стоимости трудно переоценить. Ведь даже минимальные патчи и изменения кода могут стать причиной появления новых багов.

Существует несколько основных видов автоматизированного тестирования:

- **автоматизация тестирования кода (Code-driven testing) – тестирование на уровне программных модулей, классов и библиотек (фактически, автоматические юнит-тесты);**
- **автоматизация тестирования графического пользовательского интерфейса (Graphical user interface testing) – специальная программа (фреймворк автоматизации тестирования) позволяет генерировать пользовательские события – нажатия клавиш, клики**

мышкой, и отслеживать реакцию программы на эти действия – соответствует ли она спецификации.

- **автоматизация тестирования API (ApplicationProgrammingInterface) – программного интерфейса программы. Тестируются интерфейсы, предназначенные для взаимодействия, например, с другими программами или с пользователем. Здесь опять же, как правило, используются специальные фреймворки.**

Автоматические тесты – это полноценные программы, просто предназначенные для тестирования.

1.2.3 ВИДЫ ТЕСТИРОВАНИЯ ПО ЗАПУСКУ КОДА

Статическое тестирование (static testing) — тестирование без запуска кода на исполнение[1].

Оно представляет собой процесс или технику, которые выполняются для поиска потенциальных дефектов в программном обеспечении. Это также процесс обнаружения и устранения ошибок и дефектов в различных сопроводительных документах (например, спецификации требований к программному обеспечению).

Статическое тестирование начинается на ранних этапах жизненного цикла ПО и является, соответственно, частью процесса верификации.

Можно поделить статическое тестирование на 2 типа:

1. **Обзоры (Review).**
2. **Статический анализ (Static Analysis).**

Обзоры[1]

Обзоры (Review) – проверка обычно используется для поиска и устранения ошибок или неясностей в документах. Это могут быть требования, дизайн, тестовые случаи и так далее.

В свою очередь обзоры делятся на:

1. **Неформальные. При неофициальном рассмотрении создатель документов показывает содержание документов аудитории. Каждый присутствующий высказывает свое мнение, что позволяет выявить недостатки на ранней стадии.**
2. **Сквозные просмотры (Walkthroughs). Выполняются опытным человеком или экспертом для проверки отсутствия дефектов, с целью предупреждения возникновения проблем на этапе разработки или тестирования.**
3. **Экспертная оценка. Означает проверку документов для выявления и исправления дефектов. В основном это делается в команде.**
4. **Инспектирование ПО. Это, в большинстве, проверка документа вышестоящим органом, например, проверка требований к программному обеспечению.**

Статический анализ

***Статический анализ (Static Analysis)* – код, написанный разработчиками, анализируется на наличие структурных дефектов, которые могут привести к ошибкам.**

Статический анализ включает оценку качества кода, написанного разработчиками. Для анализа кода и сравнения его со стандартом используются разные инструменты. Статический анализ хорошо помогает найти такие ошибки, как:

- **неиспользуемые переменные,**
- **мертвый код,**
- **бесконечные циклы,**
- **переменные с неопределенными значениями,**
- **неправильный синтаксис.**

В рамках этого подхода тестированию могут подвергаться:

1. **Документы (требования, тест-кейсы, описания архитектуры приложения, схемы баз данных и т.д.).**
2. **Графические прототипы (например, эскизы пользовательского интерфейса).**
3. **Код приложения (что часто выполняется самими программистами в рамках аудита кода (code review), являющегося специфической вариацией взаимного просмотра в применении к исходному коду). Код приложения также можно проверять с использованием техник тестирования на основе структур кода.**
4. **Параметры (настройки) среды исполнения приложения.**
5. **Подготовленные тестовые данные.**

Динамическое тестирование (dynamic testing) — тестирование с запуском кода на исполнение. Запускаться на исполнение может как код всего приложения целиком (системное тестирование), так и код нескольких взаимосвязанных частей (интеграционное тестирование), отдельных частей (модульное или компонентное тестирование) и даже отдельные участки кода.

Основная идея этого вида тестирования состоит в том, что проверяется реальное поведение (части) приложения.

Проще говоря, динамическое тестирование выполняется путем фактического использования приложения и определения того, работает ли функциональность так, как ожидается.

Динамическое тестирование включает в себя тестирование ПО в режиме реального времени путем предоставления входных данных и изучения результата поведения программы. Проверка осуществляется с помощью ручного или автоматического выполнения заранее подготовленного набора тестов. Оно является частью процесса валидации программного обеспечения.

То есть любое тестирование, в котором мы начинаем взаимодействовать с приложением, является динамическим. Например, проверка авторизации на сайте, запуск приложения, посадка деревьев, смена оружия и многое другое. Наша задача — посмотреть, как продукт реагирует на наши действия. Для этого мы вводим все необходимые условия и смотрим результат.

Если рассмотреть функции, предлагаемые динамическим тестированием, можно легко понять причины его выполнения в течение жизненного цикла тестирования программного обеспечения. С помощью этого тестирования можно проверить различные критические аспекты программного обеспечения. Если оставить их без какой-либо оценки, они могут повлиять на производительность, функционирование, а также надежность программного продукта.

1.2.4 ВИДЫ ТЕСТИРОВАНИЯ ПО ВРЕМЕНИ ПРОВЕДЕНИЯ

Альфа и Бета тестирование[5].

Альфа-тестирование — имитация реальной работы с системой штатными разработчиками, либо реальная работа с системой потенциальными пользователями/заказчиком. Чаще всего альфа-тестирование проводится на ранней стадии разработки продукта, но в некоторых случаях может применяться для законченного продукта в качестве внутреннего приёмочного тестирования. Иногда альфа-тестирование выполняется под отладчиком или с использованием окружения, которое помогает быстро выявлять найденные ошибки. Обнаруженные ошибки могут быть переданы тестировщикам для дополнительного исследования в окружении, подобном тому, в котором будет использоваться ПО.

Бета-тестирование — в некоторых случаях выполняется распространение версии с ограничениями (по функциональности или времени работы) для некоторой группы лиц, с тем чтобы убедиться, что продукт содержит достаточно мало ошибок. Иногда бета-тестирование выполняется для того, чтобы получить обратную связь о продукте от его будущих пользователей.

Часто для свободного/открытого ПО стадии альфа-тестирования характеризует функциональное наполнение кода, бета-тестирования — стадию исправления ошибок. При этом как правило на каждом этапе разработки промежуточные результаты работы доступны конечным пользователям.

Дымовое тестирование или Smoke Testing.

Понятие дымовое тестирование пошло из инженерной среды:

«При вводе в эксплуатацию нового оборудования ("железа") считалось, что тестирование прошло удачно, если из установки не пошел дым.»

В области же тестирования программного обеспечения, оно применяется для поверхностной проверки всех модулей приложения на предмет работоспособности и наличия быстро находимых критических и блокирующих дефектов. Подвидом дымового тестирования являются Build Verification TestingилиAcceptance Testing, выполняемые на функциональном уровне командой тестирования, по результатам которого делается вывод о том, принимается или нет установленная версия программного обеспечения в тестирование, эксплуатацию или на поставку заказчику.

Регрессионное тестирование

Регресси́онное тести́рование (англ. regression testing, от лат. regressio — движение назад) — собирательное название для всех видов тестирования программного обеспечения, направленных на обнаружение ошибок в уже протестированных участках исходного кода. Такие ошибки — когда после внесения изменений в программу

перестает работать то, что должно было продолжать работать, — называют регрессионными ошибками (англ. regression bugs).

Регрессионное тестирование (по некоторым источникам) включает new bug-fix - проверка исправления найденного ранее дефекта, old bug-fix - проверка, что исправленный ранее и верифицированный дефект не воспроизводится в системе снова, а также side-effect - проверка того, что не нарушилась работоспособность работающей ранее функциональности, если ее код мог быть затронут при исправлении некоторых дефектов в другой функциональности. Обычно используемые методы регрессионного тестирования включают повторные прогоны предыдущих тестов, а также проверки, не попали ли регрессионные ошибки в очередную версию в результате слияния кода.

Из опыта разработки ПО известно, что повторное появление одних и тех же ошибок — случай достаточно частый. Иногда это происходит из-за слабой техники управления версиями или по причине человеческой ошибки при работе с системой управления версиями. Но настолько же часто решение проблемы бывает «недолго живущим»: после следующего изменения в программе решение перестаёт работать. И наконец, при переписывании какой-либо части кода часто всплывают те же ошибки, что были в предыдущей реализации.

Поэтому считается хорошей практикой при исправлении ошибки создать тест на неё и регулярно прогонять его при последующих изменениях программы. Хотя регрессионное тестирование может быть выполнено и вручную, но чаще всего это делается с помощью специализированных программ, позволяющих выполнять все регрессионные тесты автоматически.

В некоторых проектах даже используются инструменты для автоматического прогона регрессионных тестов через заданный интервал времени. Обычно это выполняется после каждой удачной компиляции (в небольших проектах) либо каждую ночь или каждую неделю.

Регрессионное тестирование является неотъемлемой частью экстремального программирования. В этой методологии проектная документация заменяется на расширяемое, повторяемое и автоматизированное тестирование всего программного пакета на каждой стадии цикла разработки программного обеспечения.

Регрессионное тестирование может быть использовано не только для проверки корректности программы, часто оно также используется для оценки качества полученного результата. Так, при разработке компилятора, при прогоне регрессионных тестов рассматривается размер получаемого кода, скорость его выполнения и время компиляции каждого из тестовых примеров.

Приемочное тестирование или Приемо-сдаточное испытание (User Acceptance Testing)

Формальный процесс тестирования, который проверяет соответствие системы требованиям и проводится с целью:

- **определения удовлетворяет ли система приемочным критериям;**
- **вынесения решения заказчиком или другим уполномоченным лицом принимается приложение или нет.**

Приемочное тестирование выполняется на основании набора типичных тестовых случаеви сценариев, разработанных на основании требований к данному приложению. Решение о проведении приемочного тестирования принимается, когда:

- **продукт достиг необходимого уровня качества;**

- **заказчик ознакомлен с Планом Приемочных Работ (Product Acceptance Plan) или иным документом, где описан набор действий, связанных с проведением приемочного тестирования, дата проведения, ответственные и т.д.**

Фаза приемочного тестирования длится до тех пор, пока заказчик не выносит решение об отправлении приложения на доработку или выдаче приложения.

1.2.5 ВИДЫ ТЕСТИРОВАНИЯ ПО ДОСТУПУ К КОДУ (МЕТОДЫ ТЕСТИРОВАНИЯ)

Метод белого ящика[5]

Для тестирования программного кода без его запуска применяется метод белого ящика. Тестировщик имеет доступ к исходному коду программного средства и может писать код, который связан с библиотеками тестируемого программного средства. Чаще используют для компонентного тестирования, при котором тестируются только отдельные части системы. Такие тесты основаны на знании кода и внутренних механизмах приложения. Метод белого ящика используется на стадии, когда приложение не собрано в одно целое, но необходимо проверить его компоненты, модули, процедуры и подпрограммы. Тестированием данным методом занимаются: программист, или тестировщик со знанием языка программирования.

Метод черного ящика[5]

При использовании метода черного ящика тестировщик имеет доступ к ПО только через те интерфейсы, что и заказчик, конечный пользователь. Тестировщику предоставляются спецификации или иные документы, в которых описаны требования. Тестировщик запускает приложение на выполнение и тестирует его функциональность, работает с программой как конечный пользователь и ничего не знает о внутренних механизмах и алгоритмах, по которым работает программа. Цель метода – проверить работу всех функций ПС на соответствие функциональным требованиям.

Метод серого ящика используется при тестировании веб-приложений, когда тестировщик знает принципы функционирования технологий, но может не видеть кода.

1.2.6 ВИДЫ ТЕСТИРОВАНИЯ ПО ПОЗИТИВНОСТИ СЦЕНАРИЯ

Позитивное тестирование – это тестирование с применением сценариев, которые соответствуют нормальному (штатному, ожидаемому) поведению системы. С его помощью мы можем определить, что система делает то, для чего и была создана. Например, умножение на калькуляторе цифр 3 и 5.

Негативным называют тестирование, в рамках которого применяются сценарии, которые соответствуют внештатному поведению тестируемой системы. Это могут быть, например, исключительные ситуации или неверные данные. На примере калькулятора, это умножения числа 3 на грушу. Значение “груша” не является валидным для калькулятора.

Прежде всего негативное тестирование направлено на проверку устойчивости системы к различным воздействиям, валидации неверных данных, обработки исключительных ситуаций. Сценарии позитивного тестирования, в свою очередь, направлены на проверку работы системы с теми типами данных, для которых она разрабатывалась.

Создание позитивных сценариев (тест-кейсов), как правило, предшествует созданию негативных тест-кейсов. Сначала мы проверяем работу системы, когда наш условный пользователь работает с системой “правильно”, а потом приступаем к проверке отклика системы на пользователя, который допускает различные ошибки (ввод неверных данных, например). И наша система должна быть готова ответить на неверный запрос. Это и есть цель негативного тестирования.

1.2.7 КРИТЕРИИ ВЫБОРА ТЕСТОВ И ОЦЕНКИ КАЧЕСТВА ПО

Требования к идеальному критерию тестирования[19]

1. **Критерий должен быть достаточным.**
2. **Критерий должен быть полным.**
3. **Критерий должен быть надежным.**
4. **Критерий должен быть легко проверяемым**

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.

Классы критериев[19]

1. **Структурные.**
2. **Функциональные.**
3. **Критерии стохастического тестирования формулируются в терминах проверки наличия заданных свойств у тестируемого приложения, средствами проверки некоторой статистической гипотезы.**
4. **Мутационные критерии ориентированы на проверку свойств программного изделия на основе подхода Монте-Карло.**

Структурные критерии

- используют модель программы в виде "белого ящика", что предполагает знание исходного текста программы или спецификации программы в виде потокового графа управления. Данный класс критериев часто используется на этапах модульного и интеграционного тестирования (Unit testing, Integration testing)

Структурные критерии базируются на основных элементах УГП (управляющий граф программы), операторах, ветвях и путях.

- **Условие критерия тестирования команд (критерий С0) - 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 С1) - 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 С2) - 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).**

Функциональные критерии

- важнейший для программной индустрии критерий тестирования. Он обеспечивает, прежде всего, контроль степени выполнения требований заказчика в программном продукте. Отражают взаимодействие тестируемого приложения с окружением. Используется модель "черного ящика". Проблема:трудоемкость; дело в том, что документы, фиксирующие требования к программному изделию (Software requirement specification, Functional specification и т.п.), достаточно объемны.

- **Тестирование пунктов спецификации - набор тестов в совокупности должен обеспечить проверку каждого тестируемого пункта не менее одного раза.**
- **Тестирование классов входных данных - набор тестов в совокупности должен обеспечить проверку представителя каждого класса входных данных не менее одного раза. При создании тестов классы входных данных сопоставляются с режимами использования тестируемого компонента или подсистемы приложения, что заметно сокращает варианты перебора, учитываемые при разработке тестовых наборов. Следует заметить, что мы вынуждены применять мощные тестовые наборы. Действительно, наряду с ограничениями на величины входных данных, существуют ограничения на величины входных данных во всевозможных комбинациях, в том числе проверка реакций системы на появление ошибок в значениях или структурах входных данных. Учет этого многообразия - процесс трудоемкий, что создает сложности для применения критерия**
- **Тестирование правил - набор тестов в совокупности должен обеспечить проверку каждого правила, если входные и выходные значения описываются набором правил некоторой грамматики.**
- **Тестирование классов выходных данных - набор тестов в совокупности должен обеспечить проверку представителя**

каждого выходного класса, при условии, что выходные результаты заранее расклассифицированы, причем отдельные классы результатов учитывают, в том числе, ограничения на ресурсы или на время (time out). При создании тестов классы выходных данных сопоставляются с режимами использования тестируемого компонента или подсистемы, что заметно сокращает варианты перебора, учитываемые при разработке тестовых наборов.

- **Тестирование функций - набор тестов в совокупности должен обеспечить проверку каждого действия, реализуемого тестируемым модулем, не менее одного раза. Не обеспечивает покрытия части функциональности тестируемого компонента, связанной со структурными и поведенческими свойствами, описание которых не сосредоточено в отдельных функциях (т.е. описание рассредоточено по компоненту). Критерий тестирования функций объединяет отчасти особенности структурных и функциональных критериев. Он базируется на модели "полупрозрачного ящика", где явно указаны не только входы и выходы тестируемого компонента, но также состав и структура используемых методов (функций, процедур) и классов.**
- **Комбинированные критерии для программ и спецификаций - набор тестов в совокупности должен обеспечить проверку всех комбинаций непротиворечивых условий программ и спецификаций не менее одного раза.**

Стохастические критерии

- применяется при тестировании сложных программных комплексов - когда набор детерминированных тестов имеет громадную мощность. В случаях, когда подобный набор невозможно разработать и исполнить на фазе тестирования

Необходимо разработать программы - имитаторы случайных последовательностей входных сигналов{x}. Вычислить независимым способом значения {y} для соответствующих входных сигналов {x} и получить тестовый набор (X,Y). Протестировать приложение на тестовом наборе (X,Y), используя два способа контроля результатов:

- **Детерминированный контроль - проверка соответствия вычисленного значения y значению y, полученному в результате прогона теста на наборе {x} - случайной последовательности входных сигналов, сгенерированной имитатором.**
- **Стохастический контроль - проверка соответствия множества значений {y}, полученного в результате прогона тестов на наборе входных значений {x}, заранее известному распределению результатов F(Y).**

В этом случае множество Y неизвестно (его вычисление невозможно), но известен закон распределения данного множества.

Критерии стохастического тестирования

- **Статистические методы окончания тестирования - стохастические методы принятия решений о совпадении гипотез о распределении случайных величин. К ним принадлежат широко известные: метод Стьюдента, метод Хи-квадрат.**
- **Метод оценки скорости выявления ошибок - основан на модели скорости выявления ошибок, согласно которой тестирование прекращается, если оцененный интервал времени между текущей ошибкой и следующей слишком велик для фазы тестирования приложения.**

Мутационный критерий

- 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.

Мутации **- мелкие ошибки в программе[19].**

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.

1.3 ТИПЫ И УРОВНИ ТРЕБОВАНИЙ. ВЫЯВЛЕНИЕ ТРЕБОВАНИЙ

Требование (requirement) — описание функции и/или условия, которое должно соблюдаться приложением в процессе решения пользовательской задачи[19].

Требования описывают то, что необходимо реализовать, без детализации технической стороны решения. Что, а не как.

Требования нужны в частности для того, чтобы разработчик мог определить и согласовать с заказчиком временные и финансовые перспективы проекта автоматизации. Поэтому значительная часть требований должна быть собрана и обработана на ранних этапах создания ПС. Однако собрать на ранних стадиях все данные, необходимые для реализации ПС, удается только в исключительных случаях. На практике процесс сбора, анализа и обработки растянут во времени на протяжении всего жизненного цикла ПС, зачастую нетривиален и содержит множество подводных камней.

Источники требований[19]

Все требования приходят от заказчиков, или людей связанных с ними (сотрудников, пользователей и тп.)

Для выявления требований чаще всего используются следующие техники:

- **Интервью (либо переписка) – опрос, часто в формате вопрос ответ между аналитиком и заказчиком / пользователем**
- **Фокусные группы – расширенное интервью с несколькими пользователями**
- **Мозговой штурм – позволяет за короткий промежуток времени собрать большое количество идей, которые в дальнейшем изучаются и анализируются**
- **Наблюдение – позволяет выявить процессы, о которых не упомянули в интервью, занимает много времени**
- **Прототипирование – один из лучших способов понять и уточнить требования**

Так же существуют более сложные методы, при котором аналитик должен «сам во всем разобраться», и уточнить собранную информацию у заказчиков:

- **Анализ документов**
- **Моделирование процессов**
- **Самостоятельное описание**

Классификация требований (типы и уровни требований)

Существует значительное количество различных методов классификации требований, рассмотрим требования, используя терминологию, аналогичную рассмотренной в книге Карла Вигерса "Разработка требований к программному обеспечению".

Почти в каждом проекте существует 3 заинтересованных стороны:

- **Заказчики продукта**
- **Пользователи продукта**
- **Разработчики продукта**

Все они смотрят на «продукт» со своей колокольни, следовательно, требования разделяются на соответствующие группы или уровни.

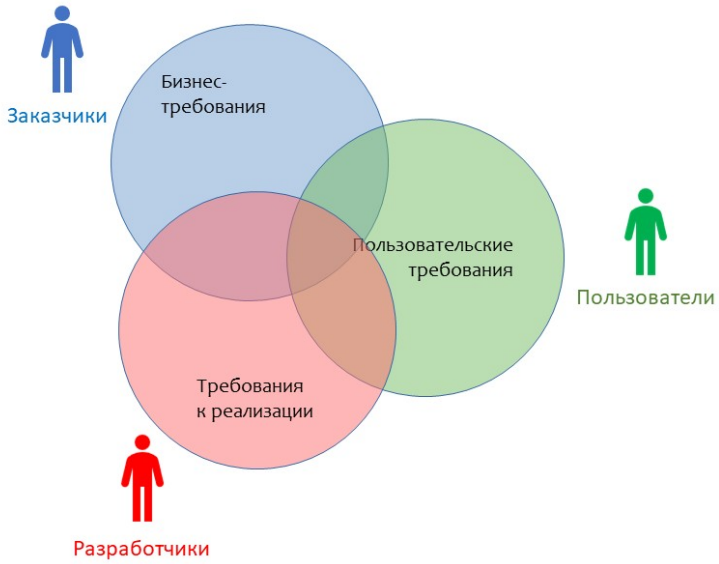

Рисунок 1 - Представление продукта разными группами

Уровень заказчика (бизнес-требований)

На этом уровне находится только один тип требований – бизнес требования (business requirements).

Они выражают цель, ради которой создается продукт (зачем он вообще нужен, каким образом он будет приносить прибыль). Они часто представлены простым текстом, без каких-либо технических подробностей.

Основываясь на этих требованиях можно получить общее видение проекта.

Уровень пользователя

Описывают “взгляд” на продукт глазами пользователя.

Включает в себя:

- **Пользовательские требования (user requirements) — описание задач, которые может выполнять пользователь при помощи системы. Оформляются в виде пользовательских историй (user stories, cases, scenarios). Эти требования могут быть использованы для оценки времени, сложности, стоимости разработки.**
- **Бизнес правила (business rules) — описывают особенности принятых в предметной области процессов, ограничений, правил.**
- **Атрибуты качества (quality attributes) — описывают ключевые показатели качества продукта.**

Уровень разработки (требований продукта)

Содержит наиболее детализированное описание функций продукта, которые должны быть реализованными.

Конечным документом, содержащим все требования уровня разработки является Спецификация требований (software requirements specification, SRS). Часто это объемный документ, содержащий сотни страниц.

К уровню разработки относятся следующие типы требований[19]:

- **Функциональные требования (functional requirements) — описывают что должна и что НЕ должна делать система.**
- **Нефункциональные требования (non-functional requirements) — требования, которые определяют критерии работы системы в целом, а не отдельные сценарии поведения. Нефункциональные требования определяют системные свойства такие как производительность, удобство сопровождения, расширяемость, надежность, средовые факторы эксплуатации.**

Примеры:

- ***«API метод должен возвращать список ресторанов в короткой форме: id, название, адрес»*** **или** ***«Никакая личная информация пользователя (логин, пароль, номера телефонов, и тд.) не должна отображаться в отчетах»*** **- это функциональные требования, они описывают поведение системы.**
- ***«Объем используемой оперативной памяти не должен превышать 256 Мб»*** **или** ***«Все данные системы должны храниться в БД под управлением СУБД MySQL»*** **- это нефункциональные требования, они описывают свойства системы.**

Разработка и управление требованиями (выявление требований)

Рисунок 2 - Разработка и управление требованиями

Область разработки технических условий разделяется на разработку требований и управление требованиями. И начинается все с выявления требований.

Выявление и сбор требований (elecitation)

Этот этап включает в себя все действия, связанные с выявлением требований, таких как интервью, совещания, анализ документов, создание прототипов и другие. К ключевым действиям относятся:

- **определение классов ожидаемых пользователей продукта и других заинтересованных лиц;**

- **понимание задач и целей, а также бизнес-целей, которым соответствуют эти задачи;**
- **изучение среды, в которой будет использоваться новый продукт;**
- **работа с отдельными людьми для понимания их потребностей и ожидания в отношении качества.**

Анализ требований[19]

Этот этап подразумевает получение более обширного и точного понимания всех требований и представление наборов требований в различном виде. Основными действиями на этом этапе будут:

- **анализ информации и отделение функциональных требований от нефункциональных, бизнес-правил, предполагаемых решений и другой информации;**
- **разложение высокоуровневых требований до нужного уровня детализации;**
- **выведение функциональных требований из информации других требований;**
- **распределение требований по компонентам ПО;**
- **согласование приоритетов реализации;**
- **понимание относительной важности атрибутов качества;**
- **выявление пробелов в требованиях или излишних требований, не соответствующих заданным рамкам.**

Документирование

Представление и хранение знаний о требованиях определенным способом. Например, в письменные требования, диаграммы пригодные для дальнейшего использования.

Утверждение требований

На этом этапе подтверждается правильность имеющегося набора требований, которые позволят реализовать решение, удовлетворяющее бизнес-целям.

Нельзя создать идеальные требования. Если смотреть с практической точки зрения, то цель разработки требований — накопить общее понимание требований необходимое для разработки очередной порции продукта.

Управление требованиями[19]

Это процесс, включающий идентификацию, выявление, документирование, анализ, отслеживание, установку приоритета требований, достижение соглашения по требованиям и затем управление изменениями и уведомление соответствующих заинтересованных лиц. Управление требованиями — непрерывный процесс на протяжении всего проекта разработки программного обеспечения.

Цель управления требованиями состоит в том, чтобы гарантировать, что организация документирует, проверяет и удовлетворяет потребности и ожидания её клиентов и внутренних или внешних заинтересованных лиц. Управление требованиями начинается с выявления и анализа целей и ограничений клиента. Управление требованиями, далее, включает поддержку требований, интеграцию требований и

организацию работы с требованиями и сопутствующей информацией, поставляющейся вместе с требованиями.

Установленная таким образом отслеживаемость требований используется для того, чтобы уведомлять заинтересованных участников об их выполнении, с точки зрения их соответствия, законченности, охвата и последовательности. Отслеживаемость также поддерживает управление изменениями как часть управления требованиями, так как она способствует пониманию того, как изменения воздействуют на требования или связанные с ними элементы, и облегчает внесение этих изменений.

Управление требованиями включает общение между проектной командой и заинтересованными лицами с целью корректировки требований на протяжении всего проекта. Постоянное общение всех участников проекта важно для того, чтобы ни один класс требований не доминировал над другими. Например, при разработке программного обеспечения для внутреннего использования у бизнеса могут быть столь сильные потребности, что он может проигнорировать требования пользователей, или полагать, что созданные сценарии использования покроют также и пользовательские требования.

1.4 ТЕСТИРОВАНИЕ ТРЕБОВАНИЙ, ТЕХНИКИ РАБОТЫ С ТРЕБОВАНИЯМИ

Качество программного обеспечения во многом зависит от качества сформированных требований, т.к. требования к программному продукту являются базой для разработки и последующего тестирования.

Тестирование требований выполняется на предмет их соответствия критериям качества требований (рисунок 3) [10].

Рисунок 3 - Критерии качества требований

Завершённость (completeness)

Требование является полным и законченным с точки зрения представления в нём всей необходимой информации, ничто не пропущено по соображениям «это и так всем понятно».

Типичные проблемы с завершённостью:

- **Отсутствуют нефункциональные составляющие требования или ссылки на соответствующие нефункциональные требования (пример: «пароли должны храниться в зашифрованном виде», а каков алгоритм шифрования?).**
- **Указана лишь часть некоторого перечисления (пример: «экспорт осуществляется в форматы PDF, PNG и т.д.», а что следует понимать под «и т.д.»?).**
- **Приведённые ссылки неоднозначны (пример: «см. выше» вместо «см. раздел 123.45.b»).**

Атомарность, единичность (atomicity)

Требование является атомарным, если его нельзя разбить на отдельные требования без потери завершённости, и оно описывает одну и только одну ситуацию.

Типичные проблемы с атомарностью:

- **В одном требовании, фактически, содержится несколько независимых (пример: «кнопка “Restart” не должна отображаться при остановленном сервисе, окно “Log” должно вмещать не менее 20-ти записей о последних действиях пользователя»: здесь в одном предложении описаны совершенно разные элементы интерфейса в совершенно разных контекстах).**
- **Требование допускает разночтение в силу грамматических особенностей языка (пример: «если пользователь подтверждает заказ и редактирует заказ или откладывает заказ, должен выдаваться запрос на оплату»: здесь описаны три разных случая, и это требование стоит разбить на три отдельных требования во избежание путаницы). Такое нарушение атомарности часто влечёт за собой возникновение противоречивости.**
- **В одном требовании объединено описание нескольких независимых ситуаций (пример: «когда пользователь входит в систему, должно отображаться приветствие; когда пользователь вошёл в систему, должно отображаться имя пользователя; когда пользователь выходит из системы, должно отображаться прощание»: все эти три ситуации заслуживают того, чтобы быть описанными отдельными и более детальными требованиями) [10].**

Непротиворечивость, последовательность (consistency)

Требование не должно содержать внутренних противоречий и противоречий другим требованиям и документам.

Типичные проблемы с непротиворечивостью:

- **Противоречия внутри одного требования (пример: «после успешного входа в систему пользователя, не имеющего права входить в систему…»: а как пользователь вошёл в систему, если не имел такого права?).**
- **Противоречия между двумя и более требованиями, между таблицей и текстом, рисунком и текстом, требованием и прототипом и т.д. (пример: «712.a Кнопка “Close” всегда должна быть красной» и**

«36452.x Кнопка “Close” всегда должна быть синей»: так всё же красной или синей?).

- **Использование неверной терминологии или использование разных терминов для обозначения одного и того же объекта или явления (пример: «в случае, если разрешение окна составляет менее 800x600…»: разрешение есть у экрана, у окна есть размер) [10].**

Недвусмысленность (unambiguousness, clearness)

Требование описано без использования жаргона, неочевидных аббревиатур и расплывчатых формулировок и допускает только однозначное объективное понимание. Требование атомарно в плане невозможности различной трактовки сочетания отдельных фраз.

Типичные проблемы с недвусмысленностью:

- **Использование терминов или фраз, допускающих субъективное толкование (пример: «приложение должно поддерживать передачу больших объёмов данных»: насколько «больших»?) Вот лишь небольшой перечень слов и выражений, которые можно считать верными признаками двусмысленности: адекватно, быть способным, легко, обеспечивать, как минимум, быть способным, эффективно, своевременно, применимо, если возможно, будет определено позже, по мере необходимости, если это целесообразно, но не ограничиваясь, быть способно, иметь возможность, нормально, минимизировать, максимизировать, оптимизировать, быстро, удобно, просто, часто, обычно, большой, гибкий, устойчивый, по последнему слову техники, улучшенный, результативно.**
- **Использование неочевидных или двусмысленных аббревиатур без расшифровки (пример: «доступ к ФС осуществляется посредством системы прозрачного шифрования» и «ФС предоставляет возможность фиксировать сообщения в их текущем состоянии с хранением истории всех изменений»: ФС здесь обозначает файловую систему или какой-нибудь «Фиксатор Сообщений»?)**
- **Формулировка требований из соображений, что нечто должно быть всем очевидно (пример: «Система конвертирует входной файл из формата PDF в выходной файл формата PNG» и при этом автор считает совершенно очевидным, что имена файлов система получает из командной строки, а многостраничный PDF конвертируется в несколько PNG-файлов, к именам которых добавляется «page-1», «page-2» и т.д.). Эта проблема перекликается с нарушением корректности[10].**

Выполнимость (feasibility)

Требование технологически выполнимо и может быть реализовано в рамках бюджета и сроков разработки проекта.

Типичные проблемы с выполнимостью:

- **Так называемое «озолочение» (gold plating) требования, которые крайне долго и/или дорого реализуются и при этом практически бесполезны для конечных пользователей (пример: «настройка параметров для подключения к базе данных должна поддерживать**

распознавание символов из жестов, полученных с устройств трёхмерного ввода»).

- **Технически нереализуемые на современном уровне развития технологий требования (пример: «анализ договоров должен выполняться с применением искусственного интеллекта, который будет выносить однозначное корректное заключение о степени выгоды от заключения договора»).**
- **В принципе нереализуемые требования (пример: «система поиска должна заранее предусматривать все возможные варианты поисковых запросов и кэшировать их результаты») [10].**

Обязательность, нужность (obligation) и актуальность (up-to-date)

Если требование не является обязательным к реализации, оно должно быть просто исключено из набора требований. Если требование нужное, но «не очень важное», для указания этого факта используется указание приоритета. Также исключены (или переработаны) должны быть требования, утратившие актуальность.

Типичные проблемы с обязательностью и актуальностью:

- **Требование было добавлено «на всякий случай», хотя реальной потребности в нём не было и нет.**
- **Требованию выставлены неверные значения приоритета по критериям важности и/или срочности.**
- **Требование устарело, но не было переработано или удалено[10].**

Прослеживаемость (traceability)

Прослеживаемость бывает вертикальной и горизонтальной. Вертикальная позволяет соотносить между собой требования на различных уровнях требований, горизонтальная позволяет соотносить требование с тест-планом, тест-кейсами, архитектурными решениями и т.д. Для обеспечения прослеживаемости часто используются специальные инструменты по управлению требованиями и/или матрицы прослеживаемости.

Типичные проблемы с прослеживаемостью:

- **Требования не пронумерованы, не структурированы, не имеют оглавления, не имеют работающих перекрёстных ссылок.**
- **При разработке требований не были использованы инструменты и техники управления требованиями.**
- **Набор требований неполный, носит обрывочный характер с явными «пробелами» [10].**

Модифицируемость (modifiability)

Это свойство характеризует простоту внесения изменений в отдельные требования и в набор требований. Можно говорить о наличии модифицируемости в том случае, если при доработке требований искомую информацию легко найти, а её изменение не приводит к нарушению иных описанных в этом перечне свойств.

Типичные проблемы с модифицируемостью:

- **Требования неатомарны (см. «атомарность») и непрослеживаемы, а потому их изменение с высокой вероятностью порождает противоречивость.**
- **Требования изначально противоречивы. В такой ситуации внесение изменений (не связанных с устранением противоречивости) только усугубляет ситуацию, увеличивая противоречивость и снижая прослеживаемость.**
- **Требования представлены в неудобной для обработки форме (например, не использованы инструменты управления требованиями, и в итоге команде приходится работать с десятками огромных текстовых документов)[10].**

Проранжированность по важности, стабильности, срочности (ranked for importance, stability, priority)

Важность характеризует зависимость успеха проекта от успеха реализации требования. Стабильность характеризует вероятность того, что в обозримом будущем в требование не будет внесено никаких изменений. Срочность определяет распределение во времени усилий проектной команды по реализации того или иного требования.

Типичные проблемы с проранжированностью состоят в её отсутствии или неверной реализации и приводят к следующим последствиям.

Проблемы с проранжированностью по важности повышают риск неверного распределения усилий проектной команды, направления усилий на второстепенные задачи и конечного провала проекта из-за неспособности продукта выполнять ключевые задачи с соблюдением ключевых условий.

Проблемы с проранжированностью по стабильности повышают риск выполнения бессмысленной работы по совершенствованию, реализации и тестированию требований, которые в самое ближайшее время могут претерпеть кардинальные изменения (вплоть до полной утраты актуальности).

Проблемы с проранжированностью по срочности повышают риск нарушения желаемой заказчиком последовательности реализации функциональности и ввода этой функциональности в эксплуатацию.

Корректность (correctness) и проверяемость (verifiability)

Фактически эти свойства вытекают из соблюдения всех вышеперечисленных (или можно сказать, что они не выполняются, если нарушено хотя бы одно из вышеперечисленных). В дополнение можно отметить, что проверяемость подразумевает возможность создания объективного тест-кейса (тест-кейсов), однозначно показывающего, что требование реализовано верно и поведение приложения в точности соответствует требованию.

К типичным проблемам с корректностью также можно отнести:

- **опечатки (особенно опасны опечатки в аббревиатурах, превращающие одну осмысленную аббревиатуру в другую также осмысленную, но не имеющую отношения к некоему контексту; такие опечатки крайне сложно заметить);**
- **наличие неаргументированных требований к дизайну и архитектуре;**
- **плохое оформление текста и сопутствующей графической информации,**

- **грамматические, пунктуационные и иные ошибки в тексте;**
- **неверный уровень детализации (например, слишком глубокая детализация требования на уровне бизнес-требований или недостаточная детализация на уровне требований к продукту);**
- **требования к пользователю, а не к приложению (пример: «пользователь должен быть в состоянии отправить сообщение»: мы не можем влиять на состояние пользователя) [10].**

Техники тестирования требований

1. Одной из наиболее активно используемых техник анализа требований является просмотр или рецензирование. Данная техника может быть реализована в форме:

- **беглого просмотра (показ автором своей работы коллеге; самый быстрый, самый дешёвый и наиболее широко используемый вид просмотра);**
- **технического просмотра (выполняется группой специалистов, каждый из которых представляет свою область знаний: просматриваемый продукт не может считаться достаточно качественным, пока хотя бы у одного просматривающего остаются замечания);**
- **формальной инспекцией (структурированный, систематизированный и документируемый подход к анализу документации, для выполнения которого привлекается большое количество специалистов, само выполнение занимает достаточно много времени, и потому этот вариант просмотра используется достаточно редко: как правило, при получении на сопровождение и доработку проекта, созданием которого ранее занималась другая компания) [10].**

2. Следующей техникой тестирования и повышения качества требований является (повторное) использование такой техники выявления требований, как формулировка вопросов. Если хоть что-то в требованиях вызывает непонимание или подозрение – задавайте вопросы[10].

3. Хорошее требование является проверяемым, а значит, должны существовать объективные способы определения того, верно ли реализовано требование. Продумывание чек-листов или даже полноценных тест-кейсов в процессе анализа требований позволяет определить, насколько требование проверяемо. Помимо использования для тестирования требований в дальнейшем такие чек-листы и тест-кейсы могут составить основу тестовой документации[10].

4. Рисунки, схемы. Чтобы увидеть общую картину требований целиком, очень удобно использовать рисунки, схемы, диаграммы, интеллект-карты и т.д. Графическое представление удобно одновременно своей наглядностью и краткостью (например, UML-схема базы данных, занимающая один экран, может быть описана несколькими десятками страниц текста) [10].

5. Исследование поведения и прототипирование. Можно сказать, что прототипирование часто является следствием создания графического представления и анализа поведения системы. С использованием специальных инструментов можно очень быстро сделать наброски пользовательских интерфейсов, оценить применимость тех или иных решений и даже создать не просто «прототип ради прототипа», а заготовку для

дальнейшей разработки, если окажется, что реализованное в прототипе (возможно, с небольшими доработками) устраивает заказчика[10].

2 СТРУКТУРНОЕ И ФУНКЦИОНАЛЬНОЕ ТЕСТИРОВАНИЕ ПО

2.1 ПЛАНИРОВАНИЕ ПРОЦЕССА ТЕСТИРОВАНИЯ. ТЕСТ-ПЛАН. РИСКИ И СТРАТЕГИЯ ТЕСТИРОВАНИЯ

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

Каждая методология или процесс пытаются навязать нам свои форматы оформления планов тестирования. Предлагаю вам, как пример, шаблоны тест планов от RUP (Rational Unified Process) [15] и стандарт IEEE 829 [16]:

- **Test Plan Template RUP [15];**
- **Test Plan Template IEEE 829 [16].**

Присмотревшись внимательнее становится ясно, что оба документа описывают одно и тоже, но в разной форме. В случае, если вам не подходит стандартный шаблон или вы решили придумать свой собственный более подходящий для вас формат документа, то из опыта могу сказать, что хороший тест план должен как минимум отвечать на следующие вопросы:

Что надо тестировать?

- **описание объекта тестирования: системы, приложения, оборудование**

Что будете тестировать?

- **список функций и описание тестируемой системы и её компонент.**

Как будете тестировать?

- **стратегия тестирования, а именно: виды тестирования и их применение по отношению к тестируемому объекту.**

Когда будете тестировать?

- **последовательность проведения работ: подготовка (Test Preparation), тестирование (Testing), анализ результатов (Test Result Analisys) в разрезе запланированных фаз разработки**

Критерии начала тестирования:

- **готовность тестовой платформы (тестового стенда);**
- **законченность разработки требуемого функционала;**
- **наличие всей необходимой документации.**

Критерии окончания тестирования:

- **результаты тестирования удовлетворяют критериям качества продукта;**
- **требования к количеству открытых багов выполнены;**

- **выдержка определенного периода без изменения исходного кода приложения Code Freeze (CF);**
- **выдержка определенного периода без открытия новых багов Zero Bug Bounce (ZBB).**

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

- **Окружение тестируемой системы;**
- **Необходимое для тестирования оборудование и программные средства;**
- **Риски и их разрешение.**

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

- **Мастер Тест План (Master Plan or Master Test Plan);**
- **Тест План (Test Plan, назовем его детальный тест план);**
- **План Приемочных Испытаний (Product Acceptance Plan) - документ, описывающий набор действий, связанных с приемочным тестированием: стратегия, дата проведения, ответственные работники и т.д. (Шаблон плана приемо-сдаточных испытаний от RUP).**

Явное отличие Мастер Тест Плана от просто Тест Плана в том, что мастер тест план является более статичным в силу того, что содержит в себе высокоуровневую (High Level) информацию, которая не подвержена частому изменению в процессе тестирования и пересмотра требований. Сам же детальный тест план, который содержит более конкретную информацию по стратегии, видам тестировании, расписанию выполнения работ, является "живым" документом, который постоянно претерпевает изменения, отражающие реальное положение вещей на проекте.

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

Когда документ пишет один человек, то он получается "однобоким", поэтому советуем проводить периодическое рецензирование со стороны участников проектной группы, а также провести процедуру утверждения документа. Как пример, привожу список участников проектной группы, утверждение которых я считаю необходимым:

- **Ведущий тестировщик;**
- **Тест менеджер (менеджер по качеству);**
- **Руководитель разработки;**
- **Менеджер Проекта.**

Каждый из перечисленных участников проекта, перед утверждением, проведет рецензию и внесет свои комментарии и предложения, которые помогут сделать Ваш тест план более полным и качественным.

2.2 ТЕСТ-ДИЗАЙН. ПРИНЦИПЫ РАЗРАБОТКИ ТЕСТОВ

Тест дизайн – это этап процесса тестирования ПО, на котором проектируются и создаются тестовые случаи (тест кейсы), в соответствии с определёнными ранее критериями качества и целями тестирования. Попросту говоря, задача тест сводится к тому, чтобы, используя различные стратегии и техники тест дизайна, создать набор тестовых случаев, обеспечивающий оптимальную проверку тестируемого приложения.

Цели тест дизайна

- **Обеспечить покрытие функционала приложения тестами:**
- **Тесты должны покрывать весь функционал**
- **Тестов должно быть минимально достаточно**

Тест дизайн задачи

- **Проанализировать требования к продукту**
- **Оценить риски возможные при использовании продукта**
- **Написать достаточное минимальное количество тестов**
- **Разграничить тесты на приемочные, критические, расширенные**

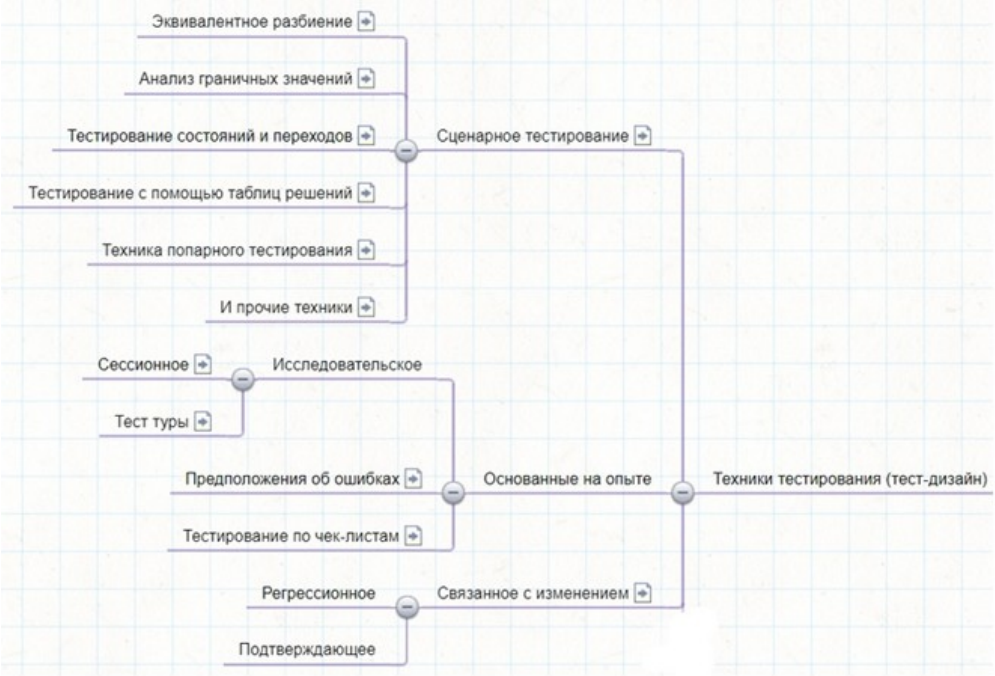

Рисунок 4 - Техники текст-дизайна

Рассмотрим несколько основных методик, однако, будем помнить, что зачастую их используют в комплексе. Одной техники может быть недостаточно, поскольку она не обеспечит максимальный охват тестовых сценариев.

Техника анализа классов эквивалентности - это техника, при которой мы разделяем функционал (часто диапазон возможных вводимых значений) на группы эквивалентных по своему влиянию на систему значений. Такое разделение помогает убедиться в правильном функционировании целой системы — одного класса эквивалентности, проверив только один элемент этой группы. Эта техника заключается в разбиении всего набора тестов на классы эквивалентности с последующим сокращением числа тестов.

Классы эквивалентности выделяются путем выбора каждого входного условия (обычно это предложение или фраза в спецификации) и разбиением его на две или более групп. Для проведения этой операции используют таблицу.

Таблица 1 - Пример таблицы

Входные условия	**Правильные классы эквивалентности**	**Неправильные классы эквивалентности**

Заметим, что различают два типа классов эквивалентности: *правильные* классы эквивалентности, представляющие правильные входные данные программы, и *неправильные* классы эквивалентности, представляющие все другие возможные состояния условий (т. е. ошибочные входные значения).

Алгоритм использования:

1. **Определить классы эквивалентности.**
2. **Это главный шаг техники, т.к. во многом от него зависит эффективность её применения.**
3. **Выбрать одного представителя от каждого класса эквивалентности.**
4. **На этом этапе следует выбрать один тест из эквивалентного набора тестов.**
5. **Выполнение тестов.**
6. **На этом шаге следует выполнить тесты от каждого класса эквивалентности.**
7. **Если есть время, можно протестировать еще несколько представителей от каждого класса эквивалентности. Следует иметь ввиду, при правильном определение классов эквивалентности дополнительные тесты скорее всего будут избыточными и дадут такой же результат.**

Техника анализа граничных значений - это техника проверки поведения продукта на крайних (граничных) значениях входных данных. Граничное тестирование также может включать тесты, проверяющие поведение системы на входных данных, выходящих за допустимый диапазон значений. При этом система должна определённым (заранее оговоренным) способом обрабатывать такие ситуации. Например, с помощью исключительной ситуации или сообщения об ошибке.

Граничные значения очень важны и их обязательно следует применять при написании тестов, т.к. именно в этом месте чаще всего и обнаруживаются ошибки.

На каждой границе диапазона следует проверить по три значения:

- **граничное значение;**
- **значение перед границей;**
- **значение после границы.**

Цель этой техники — найти ошибки, связанные с граничными значениями.

Алгоритм использования техники граничных значений:

1. **Выделить классы эквивалентности;**
2. **Как и в предыдущей технике, этот шаг является очень важным и от того, насколько правильным будет разбиение на классы эквивалентности, зависит эффективность тестов граничных значений.**
3. **Определить граничные значения этих классов;**
4. **Нужно понять, к какому классу будет относиться каждая граница;**
5. **Нужно провести тесты по проверке значения до границы, на границе и сразу после границы.**

Количество тестов для проверки граничных значений будет равен количеству границ, умноженному на 3. Рекомендуется проверять значения вплотную к границе. К примеру, есть диапазон целых чисел, граница находится в числе 100. Таким образом, будем проводить тесты с числом 99 (до границы), 100 (сама граница), 101 (после границы).

Попарное тестирование

Суть этого метода, также известного как pairwise testing, в том, что каждое значение каждого проверяемого параметра должно быть протестировано на взаимодействие с каждым значением всех остальных параметров. После составления такой матрицы мы убираем тесты, которые дублируют друг друга, оставляя максимальное покрытие при минимальном необходимом наборе сценариев.

Попарное тестирование позволяет обнаружить максимум ошибок без избыточных проверок.

Пример

Для Parwise достаточно, чтобы каждое значение всех параметров хотя бы единожды сочеталось с другими значениями остальных параметров. Таким образом, матрицу можно значительно сократить. Пример:

№	Браузер	Операционная система	Язык
1	Opera	Windows	RU
2	Google Chrome	Linux	RU
3	Opera	Linux	EN
4	Google Chrome	Windows	EN

При составлении матрицы принятия решений для двух браузеров, двух ОС и двух языков было бы нужно 8 сценариев. При попарном тестировании достаточно четырех.

Все это можно просчитать и вручную, но не обязательно – гораздо удобнее автоматизировать процесс. Для этого существует программа попарного независимого комбинированного тестирования – Pairwise Independent Combinatorial Testing (PICT).

Таблица состояний и переходов — способ компактного представления модели со сложной логикой; инструмент для упорядочения сложных бизнес требований, которые должны быть реализованы в продукте. Это взаимосвязь между множеством условий и действий.

Таблица переходов представляет собой все возможные комбинации начальных и конечных состояний, включая действительные и недействительные переходы, инициирующие события, защитные условия и результирующие действия. Диаграммы

состояний и переходов обычно, показывают только действительные переходы и исключают недействительные переходы.

Тесты создаются для покрытия типичной последовательности состояний, покрытия каждого возможного состояния, покрытия каждого возможного перехода, проверки специфических последовательностей переходов, или для проверки недействительных переходов.

2.3 РАЗРАБОТКА И ДОКУМЕНТИРОВАНИЕ ТЕСТОВ

Разработка дымового-теста

Дымовой тест (англ. Smoke testing или smoke test, дымовое тестирование) — в тестировании программного обеспечения означает минимальный набор тестов на явные ошибки. Дымовой тест обычно выполняется программистом; не проходившую этот тест программу не имеет смысла отдавать на более глубокое тестирование[17].

Примеры

Ошибки инсталляции: если программный продукт не устанавливается, его тестирование, скорее всего, окажется невозможным.

Ошибки при соединении с базой данных (актуально для архитектуры клиент-сервер).

Ошибки загрузки конфигурации и получения настроек для инициализации при запуске.

Если мы говорим про сайт интернет-магазина, то сценарий будет примерно следующим:

1. **Сайт открывается.**
2. **Можно выбрать случайный товар и добавить его в корзину.**
3. **Можно оформить и оплатить заказ.**

Если мы говорим про мобильное приложение, например, messenger, то:

1. **Приложение устанавливается и запускается.**
2. **Можно авторизоваться.**
3. **Можно написать сообщение случайном контакту.**

Что тестируем?

1) **Проверка сборки: первым и самым важным шагом дымового теста является проверка сборки, номера сборки и доступности среды. Все усилия по тестированию будут потрачены впустую, если сборка неправильная.**
2) **Создание учетной записи: если ваше приложение предполагает создание пользователя, вам следует попытаться создать нового пользователя и проверить, успешно ли система позволяет вам это сделать. Это важный момент, который часто упускается из виду, поскольку тестировщики продолжают использовать старые учетные данные без тестирования для нового пользователя.**

3) **Вход в систему и Выход из системы: если это возможно в вашей SUT (тестируемая система), в рамках дымового теста вы должны попытаться успешно войти в систему со старыми и вновь созданными учетными данными. Также убедитесь, что вы можете успешно выйти из системы без каких-либо ошибок.**

4) **Критически важные для бизнеса функции: это очень важно. Для всех основных или критически важных функций мы должны провести простой тест, чтобы убедиться, что наиболее часто используемые функции не нарушены.**

5) **Сценарии интеграции: это самая важная часть дымового теста. Эффективность этой части зависит от понимания тестером интеграции системы. Например, если тестировщик знает, что какие-то данные текут из системы A в систему B, то он должен сделать это как часть дымового теста (любое значение 1). Это также делается для того, чтобы система не сломалась ни в одной из этих точек интеграции.**

6) **Добавить / изменить / удалить: данные всегда сохраняются в базе данных. Три основных операции в базе данных: добавление записи, редактирование записи и удаление записи. Таким образом, чтобы обеспечить надлежащее соединение с базой данных, в рамках дымового теста необходимо попытаться создать, отредактировать и удалить запись, которая применима в тестируемой системе.**

7) **Общая навигация: последняя часть - общая навигация. То есть нужно пройти через приложение, попытаться коснуться часто используемых функций и страниц, чтобы убедиться, что вся навигация работает должным образом.**

Разработка тест-кейса

Тест-кейс — набор входных значений, предусловий выполнения, ожидаемых результатов и постусловий выполнения, разработанный для определённой цели или тестового условия, таких как выполнения определённого пути программы или же для проверки соответствия определённому требованию.

Определение тест-кейса языком обывателя:

Тест-кейс — это чёткое описание действий, которые необходимо выполнить, для того чтобы проверить работу программы (поля для ввода, кнопки и т.д.). Данное описание содержит: действия, которые надо выполнить до начала проверки — предусловия; действия, которые надо выполнить для проверки — шаги; описание того, что должно произойти, после выполнения действий для проверки — ожидаемый результат.

Правила написания тест-кейсов

Заголовок:

- **должен быть чётким, кратким, понятным и однозначно характеризующим суть тест-кейса;**
- **не может содержать выполняемые шаги и ожидаемый результат.**

Предусловие:

- **может содержать полную информацию о состоянии системы или объекта, необходимом для начала выполнения шагов тест-кейса;**

- **может содержать ссылки на информационные источники, которые необходимо изучить перед прохождением тест-кейса (инструкции, описание систем…);**
- **не может содержать ссылки на тестируемый ресурс, если у информационной системы более одной среды (прод, тест, препрод…), данная информация должна быть вынесена в инструкцию, и ссылка приложена в предусловии;**
- **не может содержать данные для авторизации, данная информация должна быть вынесена в инструкцию, и ссылка приложена в предусловии;**
- **не может содержать выполняемые шаги и ожидаемый результат, если нам нужно, чтобы до выполнения шагов проверки у нас была открыта главная страница, то мы в предусловии указываем «открыта главная страница сайта»;**
- **не может содержать ожидаемый результат.**

Шаги проверки:

- **должны быть чёткими, понятными и последовательными;**
- **следует избегать излишней детализации шагов. Правильно: «ввести в поле число 12».**
- **Неправильно: «нажать на клавиатуре на цифру ‘1’, следующим шагом нажать на клавиатуре на цифру ‘2’»;**
- **должны использоваться безличные глаголы.**
- **Правильно: нажать, ввести, перейти.**
- **Неправильно: нажмите, введите, идите;**
- **не должно быть комментариев и пояснений, если есть необходимость привести мини-инструкцию, то оформляем инструкции в базе-знаний и в предусловии ссылаемся на неё;**
- **не должно быть жёстко прописанных статических данных (логины, пароли, имена файлов) и примеров, для исключения эффекта пестицида.**

Ожидаемый результат:

- **должен быть у каждого шага проверки;**
- **должно быть кратко и понятно описано состояние системы или объекта, наступающее после выполнения соответствующего шага;**
- **не должно быть избыточного описания.**

Общие требования к тест-кейсам:

- **язык описания тест-кейсов должен быть понятен широкому кругу пользователей, а не узкой группе лиц;**
- **тест-кейс должен быть максимально независим от других тест-кейсов и не ссылаться на другие тест-кейсы (лучшая практика, когда зависимостей нет вообще);**

- **тест-кейсы группируются в функциональные блоки по их назначению;**
- **в тест-кейсах, проверяющих работу функционала скриншотов быть не должно, иначе вы будете посвящать сотни часов на изменение всех скриншотов в тысячах тест-кейсах при изменении интерфейса тестируемой программы. Скриншоты могут быть добавлены только в тест-кейсы, проверяющие отображение страниц и форм.**

На самом деле правила простые, однако их не так-то просто соблюдать. Если же придерживаться данных правил, то тест-кейсы будут легко поддерживаемыми, легко читаемыми, не будут вызывать отторжения и могут быть использованы всеми участниками команды в процессе разработки программного обеспечения.

2.4 ОПИСАНИЕ ДЕФЕКТОВ. ЖИЗНЕННЫЙ ЦИКЛ ДЕФЕКТОВ

Программная ошибка – это изъян в разработке программного продукта, который вызывает несоответствие ожидаемых результатов выполнения программы и фактических.

Дефект – это ошибка или неточность, которая может быть (может и не быть) следствием сбоя в работе программы.

Дефект – это поведение программы, затрудняющее, или делающее невозможным достижение целей пользователя или удовлетворение интересов участников.

Дефект – это несоответствие требованиям или функциональным спецификациям.

Где можно найти баги:

1. **Требования**
2. **Архитектурная документация**
3. **Код**
4. **Дизайн**

Кто может обнаружить дефект:

1. **Программист**
2. **Тестировщик**
3. **Технический писатель**
4. **Менеджер проекта**

После обнаружения дефекта составляется отчет об ошибке. Отчет об ошибке – это технический документ, который составляется с целью:

1. **Предоставить информацию о проблеме, ее свойствах и последствиях.**
2. **Классифицировать проблему по важности и скорости устранения.**
3. **Помочь программистам обнаружить и устранить источник проблемы.**

Отчет об ошибке (баг-репорт) – это один из основных результатов работы тестировщика.

Формула хорошего баг-репорта:

1. **Что мы сделали?**
2. **Что получили?**
3. **Что должны были получить?**

Жизненный цикл дефекта:

1. **Обнаружен**
2. **Назначен**
3. **Исправлен**
4. **Проверен**
5. **Закрыт/открыт заново**

Атрибуты отчета о дефекте:

1. **Идентификатор**
2. **Дата**
3. **Краткое описание (что, где и при каких условиях)**
4. **Подробное описание (ожидаемый результат, фактический результат, ссылка на требование)**
5. **Шаги воспроизведения (1. Перейти по ссылке www.omg.org 2. Ввести в поле «логин» значение «admin» 3. Ввести в поле «пароль» значение «abcd» 4. Кликнуть левой кнопкой мыши по кнопке войти. Баг: В левом верхнем углу вместо логотипа пустое место.)**
6. **Воспроизводимость (указывается частота воспроизведения дефекта (всегда или иногда)) (пройти шаги воспроизведения минимум три раза).**
7. **Важность (на сколько серьезна ошибка). Критическая - дефекты вызывают крах системы, либо повреждения данных. Высокая - серьезные ошибки, например, потеря данных пользователя, падение некоторой части функциональности программы, падение браузера. Средняя – ошибки, затрагивающие функции приложения, но их можно обойти. Низкая – ошибки, не мешающие работе с приложением (опечатки).**
8. **Срочность (как быстро необходимо исправить ошибку). Наивысшая присваивается ошибка, наличие которых делает невозможной дальнейшую работу над проектом. Высшая – нужно исправить в самое ближайшее время. Обычная – исправляется в порядке общей очереди. Низкая – если остается время.**
9. **Симптом. Косметический дефект (опечатки, поврежденные картинки, не тот шрифт и т.д.). Повреждения/потеря данных. Проблема в документации. Некорректная операция. Проблема инсталляции. Ошибка локализации. Нереализованная функциональность. Низкая производительность. Крах системы. Неожиданное поведение. Недружественное поведение. Расхождение с требованием, под этот симптом попадает почти любая ошибка, но рекомендуется его писать только тогда, когда другие симптомы не подходят. Предложения по**

улучшению. Дополнительные атрибуты: возможность обойти баг, дополнительная информация и приложения (скриншот, видео).

Преимущества хорошего отчета об ошибках:

Хороший отчет об ошибках помогает сократить количество ошибок, отклоненных, или открытых заново. Ускорить устранение ошибки. Сократить стоимость исправления ошибки. Повысить репутацию тестировщика.

Баг-трекинговые системы

Система отслеживания ошибок (англ. bug tracking system) — прикладная программа, разработанная с целью помочь разработчикам программного обеспечения (программистам, тестировщикам и др.) учитывать и контролировать ошибки и неполадки, найденные в программах, пожелания пользователей, а также следить за процессом устранения этих ошибок и выполнения или невыполнения пожеланий.

Главный компонент такой системы — база данных, содержащая сведения об обнаруженных дефектах. Эти сведения могут включать в себя:

1. **Номер (идентификатор) дефекта;**
2. **Короткое описание дефекта;**
3. **Кто сообщил о дефекте;**
4. **Дата и время, когда был обнаружен дефект;**
5. **Версия продукта, в которой обнаружен дефект;**
6. **Серьёзность (критичность) дефекта и приоритет решения[1];**
7. **Описание шагов для выявления дефекта (воспроизведения неправильного поведения программы);**
8. **Ожидаемый результат и фактический результат;**
9. **Кто ответственен за устранение дефекта;**
10. **Обсуждение возможных решений и их последствий;**
11. **Текущее состояние (статус) дефекта;**
12. **Версия продукта, в которой дефект исправлен.**

Кроме того, развитые системы предоставляют возможность прикреплять файлы, помогающие описать проблему (например, дамп памяти или скриншот).

TRELLO – построена на основе доски, все функции находятся на одном экране. Слишком простая и не предназначена для больших команд. Задачи не имеют номера, что затрудняет контроль.

YouTrack – доска и диаграмма очень удобные, статусы и приоритеты можно выделять для наглядности.

JIRA AGILE – данную систему называют «№1». Она обладает наиболее широкой функциональностью среди других систем. Идеально подходит для крупных проектов с большим штатом тестировщиков, одновременно можно работать под различными операционными системами, создавать и вести схемы безопасности для каждого из проектов.

TARGETPROCESS – система очень дорогая. Доска удобная, размещение задач сделано по принципу trello, приспособлено к большим командам и большому количеству задач. Разработчики ставили цель сделать программу более простой и удобной в использовании, но количество функций урезали.

Процесс баг-трекинга

1. **Создаваясь, сообщение, обязательно имеет ответственного**
2. **Каждому дефекту определяется приоритет важности, возможно добавить комментарий**
3. **Сообщениям устанавливаются статусы «в процессе», где указывается время начала и окончания работы.**

3 ОРГАНИЗАЦИЯ ТЕСТИРОВАНИЯ ПО

3.1 МОДУЛЬНОЕ ТЕСТИРОВАНИЕ

Задачи и цели модульного тестирования

Каждая сложная программная система состоит из отдельных частей - модулей, выполняющих ту или иную функцию в составе системы. Для того, чтобы удостовериться в корректной работе всей системы, необходимо вначале протестировать каждый модуль системы по отдельности. В случае возникновения проблем при тестировании системы в целом это позволяет проще выявить модули, вызвавшие проблему, и устранить соответствующие дефекты в них. Такое тестирование модулей по отдельности получило называние модульного тестирования (unit testing).

Для каждого модуля, подвергаемого тестированию, разрабатывается тестовое окружение, включающее в себя драйвер и заглушки, готовятся тест-требования и тест-планы, описывающие конкретные тестовые примеры [18].

Основная цель модульного тестирования - удостовериться в соответствии требованиям каждого отдельного модуля системы перед тем, как будет произведена его интеграция в состав системы.

При этом в ходе модульного тестирования решаются следующие основные задачи:

- **Поиск и документирование несоответствий требованиям**
- **Поддержка разработки и рефакторинга низкоуровневой архитектуры системы и межмодульного взаимодействия**
- **Поддержка рефакторинга модулей**
- **Поддержка устранения дефектов и отладки**
- **Первая задача - классическая задача тестирования, включающая в себя не только разработку тестового окружения и тестовых примеров, но и выполнение тестов, протоколирование результатов выполнения, составление отчетов о проблемах.**

Вторая задача больше свойственна "легким" методологиям типа XP, где применяется принцип тестирования перед разработкой (Test-driven development), при котором основным источником требований для программного модуля является тест, написанный до реализации самого модуля. Однако, даже при классической схеме тестирования модульные тесты могут выявить проблемы в дизайне системы и нелогичные или запутанные механизмы работы с модулем.

Третья задача связана с поддержкой процесса изменения системы. Достаточно часто в ходе разработки требуется проводить рефакторинг модулей или их групп - оптимизацию или полную переделку программного кода с целью повышения его сопровождаемости, скорости работы или надежности. Модульные тесты при этом являются мощным инструментом для проверки того, что новый вариант программного кода выполняет те же функции, что и старый.

Последняя, четвертая, задача сопряжена с обратной связью, которую получают разработчики от тестировщиков в виде отчетов о проблемах. Подробные отчеты о проблемах, составленные на этапе модульного тестирования, позволяют локализовать

и устранить многие дефекты в программной системе на ранних стадиях ее разработки или разработки ее новой функциональности.

В силу того, что модули, подвергаемые тестированию, обычно невелики по размеру, модульное тестирование считается наиболее простым (хотя и достаточно трудоемким) этапом тестирования системы. Однако, несмотря на внешнюю простоту, с модульным тестированием сопряжены две проблемы.

Первая из них связана с тем, что не существует единых принципов определения того, что в точности является отдельным модулем.

Вторая заключается в различиях в трактовке самого понятия модульного тестирования - понимается ли под ним обособленное тестирование модуля, работа которого поддерживается только тестовым окружением, или речь идет о проверке корректности работы модуля в составе уже разработанной системы. В последнее время термин "модульное тестирование" чаще используется во втором смысле, хотя в этом случае речь скорее идет об интеграционном тестировании.

Понятие модуля и его границ. Тестирование классов

Традиционное определение модуля с точки зрения его тестирования: "модуль - это компонент минимального размера, который может быть независимо протестирован в ходе верификации программной системы". В реальности часто возникают проблемы с тем, что считать модулем.

Существует несколько подходов к данному вопросу:

- **модуль - это часть программного кода, выполняющая одну функцию с точки зрения функциональных требований;**
- **модуль - это программный модуль, т.е. минимальный компилируемый элемент программной системы;**
- **модуль - это задача в списке задач проекта (с точки зрения его менеджера);**
- **модуль - это участок кода, который может уместиться на одном экране или одном листе бумаги;**
- **модуль - это один класс или их множество с единым интерфейсом.**

Обычно за тестируемый модуль принимается либо программный модуль (единица компиляции) в случае, если система разрабатывается на процедурном языке программирования, или класс, если система разрабатывается на объектно-ориентированном языке.

В случае систем, написанных на процедурных языках, процесс тестирования модуля происходит так: для каждого модуля разрабатывается тестовый драйвер, вызывающий функции модуля и собирающий результаты их работы, и набор заглушек - они имитируют поведение функций, которые содержатся в других модулях, не попадающих под тестирование данного модуля. При тестировании объектно-ориентированных систем существует ряд особенностей, прежде всего вызванных инкапсуляцией данных и методов в классах.

В случае объектно-ориентированных систем более мелкое деление классов и использование отдельных методов в качестве тестируемых модулей нецелесообразно, поскольку для тестирования каждого метода потребуется разработка тестового окружения, сравнимого по сложности с уже написанным программным кодом класса. Кроме того, декомпозиция класса нарушает принцип инкапсуляции, согласно которому

объекты каждого класса должны вести себя как единое целое с точки зрения других объектов.

Процесс тестирования классов как модулей иногда называют компонентным тестированием. В ходе такого тестирования проверяется взаимодействие методов внутри класса и правильность доступа методов к внутренним данным класса. Здесь возможно обнаружение не только стандартных дефектов, связанных с выходами за границы диапазона или неверно реализованными требованиями, но и специфических дефектов объектно-ориентированного программного обеспечения:

- **дефектов инкапсуляции, в результате которых, например, сокрытые данные класса оказываются недоступными для соответствующих публичных методов;**
- **дефектов наследования, при наличии которых схема наследования блокирует важные данные или методы от классов-потомков;**
- **дефектов полиморфизма, при которых полиморфное поведение класса оказывается распространенным не на все возможные классы;**
- **дефектов инстанцирования, при которых во вновь создаваемых объектах класса не устанавливаются корректные значения по умолчанию параметров и внутренних данных класса.**

Однако, выбор класса в качестве тестируемого модуля имеет и ряд сопряженных проблем.

Определение степени полноты тестирования класса

В том случае, если в качестве тестируемого модуля выбран класс, не совсем ясно, как определять степень полноты его тестирования. С одной стороны, можно использовать классический критерий полноты покрытия программного кода тестами: если полностью выполнены все структурные элементы всех методов, как публичных, так и скрытых, то тесты можно считать полными.

Однако существует альтернативный подход к тестированию класса, в котором все публичные методы должны предоставлять пользователю данного класса согласованную схему работы - тогда достаточно проверить типичные корректные и некорректные сценарии работы с данным классом. Т.е., например, в классе, объекты которого представляют записи в телефонной книжке, одним из типичных сценариев работы будет "Создать запись \to искать запись и найти ее \to удалить запись \to искать запись вторично и получить сообщение об ошибке".

Различия в этих двух методах напоминают различия между тестированием черного и белого ящиков, но, на самом деле, второй подход отличается от черного ящика тем, что функциональные требования на систему могут быть составлены на уровне более высоком, чем отдельные классы, и установление адекватности тестовых сценариев требованиям остается на откуп тестировщику[18].

Протоколирование состояний объектов и их изменений

Некоторые методы класса предназначены не для выдачи информации пользователю, а для изменения внутренних данных объекта класса. Значение внутренних данных объекта определяет его состояние в каждый отдельный момент времени, а вызов методов, изменяющих данные, изменяет и состояние объекта. При тестировании классов необходимо проверять, что класс адекватно реагирует на внешние вызовы в

любом из состояний. Однако, зачастую из-за инкапсуляции данных невозможно определить внутреннее состояние класса программными способами внутри драйвера.

В этом случае может помочь составление схемы поведения объекта как конечного автомата с определенным набором состояний. Такая схема может входить в низкоуровневую проектную документацию (например, в составе описания архитектуры системы), а может составляться тестировщиком или разработчиком на основе функциональных требований к системе. В последнем случае для определения всех возможных состояний может потребоваться ручной анализ программного кода и определение его соответствия требованиям. Автоматизированное тестирование в этом случае может лишь определить, по всем ли выявленным состояниям осуществлялись переходы и все ли возможные реакции проверялись[18].

Тестирование изменений

Как уже упоминалось выше, модульные тесты - мощный инструмент проверки корректности изменений, внесенных в исходный код при рефакторинге. Однако, в результате рефакторинга только одного класса, как правило, не меняется его внешний интерфейс с другими классами (интерфейсы меняются при рефакторинге сразу нескольких классов). В результате обычных эволюционных изменений системы у класса может меняться внешний интерфейс, причем как по формальным (изменяются имена и состав методов, их параметры), так и по функциональным (при сохранении внешнего интерфейса меняется логика работы методов) признакам. Для проведения модульного тестирования класса после таких изменений потребуется изменение драйвера и, возможно, заглушек. Но только модульного тестирования в данном случае недостаточно, необходимо также проводить и интеграционное тестирование данного класса вместе со всеми классами, которые связаны с ним по данным или по управлению.

Вне зависимости от того, на какие модули, подвергаемые тестированию, разбивается система, рекомендуется изложить принципы выделения тестируемых модулей в плане и стратегии тестирования, а также составить на базе структурной схемы архитектуры системы новую структурную схему, на которой нужно отметить все тестируемые модули. Это позволит спрогнозировать состав и сложность драйверов и заглушек, требуемых для модульного тестирования системы. Такая схема также может использоваться позже на этапе модульного тестирования для выделения укрупненных групп модулей, подвергаемых интеграции[18].

Организация модульного тестирования

Модульное тестирование, с точки зрения тестировщика, - это комплекс работ по выявлению дефектов в тестируемых модулях. В эти работы включается анализ требований, разработка тест-требований и тест-планов, разработка тестового окружения, выполнение тестов, сбор информации об их прохождении.

Однако, с точки зрения руководителя группы тестирования (или с точки зрения руководителя проекта, если в нем не выделена отдельная группа тестирования), модульное тестирование является более широким понятием. Для того, чтобы процесс модульного тестирования мог функционировать совместно с другими процессами разработки, он должен включать в себя несколько фаз: планирование процесса, разработку тестов, выполнение тестов, сбор статистики, управление отчетами о выявленных дефектах[18].

Согласно стандарту IEEE 1008[15] процесс модульного тестирования состоит из трех фаз, в состав которых входит 8 видов деятельности (этапов).

Фаза планирования тестирования

1. **Этап планирования основных подходов к тестированию, ресурсное планирование и календарное планирование.**
2. **Этап определения свойств, подлежащих тестированию.**
3. **Этап уточнения основного плана, сформированного на этапе (1).**

Фаза получения набора тестов

1. **Этап разработки набора тестов.**
2. **Этап реализации уточненного плана.**

Фаза измерений тестируемого модуля

1. **Этап выполнения тестовых процедур.**
2. **Этап определения достаточности тестирования.**
3. **Этап оценки результатов тестирования и тестируемого модуля.**

Во время этапа планирования основных подходов в качестве входных данных используется общий план проекта (модульное тестирование, как часть проектных работ, должно укладываться в общий график) и требования к системе (для оценки трудоемкости работ и любого планирования необходимо проводить анализ сложности системы на основании требований к ней).

Основные задачи, решаемые в ходе этапа планирования, включают в себя:

- **определение общего подхода к тестированию модулей - определяются риски и на их основе - степень полноты и охвата тестирования системы. Определяются источники входных и выходных данных. Определяются технологии проверки результатов тестирования и форматы записи данных о проведенном тестировании. Описывается внешний интерфейс тестируемых модулей и их информационное окружение;**
- **определение требований к полноте тестирования - определяется необходимая степень покрытия программного кода различных участков тестируемого модуля, определяется подходы к классам эквивалентности (требуется ли тестирование за границами диапазона);**
- **определение требований к завершению тестирования - определяются условия, проверка которых позволяет утверждать, что тестирование модуля завершено, и условия, при которых дальнейшее тестирование модуля считается невозможным до его изменения и доработки. Примером таких условий может служить достижение определенного уровня покрытия исходного кода тестами и невозможность компиляции модуля соответственно;**
- **определение требований к ресурсам - для разработки и выполнения тестов, а также для анализа результатов тестирования необходимы ресурсы - как технические (компьютеры и программное обеспечение), так и людские (тестировщики). При решении этой задачи необходимо указывать требования к программному и аппаратному обеспечению, требования к необходимой квалификации людей, а также должно определяться необходимое для проведения количество ресурсов и время их занятости;**

- **определение общего плана-графика работ - на основании общего плана проекта составляется план работ по модульному тестированию. Основной критерий начала работ по тестированию - готовность модулей, т.е. общий план работ по тестированию согласуется по датам начала работ с датами окончания работ общего плана разработки.**

После завершения этапа планирования начинается этап определения свойств системы, подлежащих тестированию.

Основные задачи, которые решаются в ходе деятельности по определению свойств системы, подлежащих тестированию, включают в себя:

- **изучение функциональных требований - определение тестопригодности требований, при необходимости запрашивается уточнение требований;**
- **определение дополнительных требований и связанных процедур - определение требований, которые не попадают под функциональные требования, но могут быть протестированы на уровне модульного тестирования (например, это могут быть требования к производительности системы, входящие в состав системных требований);**
- **определение состояний тестируемого модуля - если тестируемый модуль может быть представлен в виде конечного автомата с определенным набором состояний, то каждое состояние должно быть идентифицировано, а также должны быть выделены все требования, относящиеся к этому состоянию;**
- **определение характеристик входных и выходных данных - для всех данных, которые поступают в модуль, а также выходят из него, должны быть определены форматы, частота поступления, допустимые значения и т.п.;**
- **выбор элементов, подвергаемых тестированию - в случае, когда не может применяться полное тестирование, необходимо выбрать элементы тестируемого модуля, которые будут подвергаться тестированию. Основной источник информации здесь - данные о рисках, проанализированные на уровне структуры исходного кода тестируемого модуля. Для тестирования в первую очередь должны отбираться элементы с максимальной степенью риска.**

И, наконец, в завершение фазы планирования производится уточнение основного плана - уточняется общий подход к тестированию, формулируются специальные и дополнительные требования к ресурсам, составляется детальный план-график работ.

По завершению этих этапов фаза планирования считается оконченной и начинается фаза разработки тестов. При этом процесс разработки тестов подчиняется тем планам и требованиям, которые были созданы на предыдущем этапе. Таким образом, если на первом этапе основную роль выполнял руководитель группы тестирования, то на втором этапе основную роль начинает играть тестировщик, действующий в согласии с указаниями руководителя.

Фаза разработки тестов начинается с собственно разработки набора тестов, который будет использован для тестирования модуля. Основные документы, которые используются на этом этапе: функциональные требования к модулю, архитектура

модуля, список элементов, подвергаемых тестированию, план-график работ, определения тестовых примеров от предыдущей версии модуля (если они существовали) и результаты тестирования прошлой версии (если они существовали).

В ходе этого этапа должны быть решены следующие задачи:

- **разработка архитектуры тестового набора - под тестовым набором здесь понимается не набор конкретных тестовых примеров, а общая структура системы тестов для проверки функциональности тестируемого модуля. Организация тестов в такой системе как правило отражает структуру функциональных требований и зачастую представляет собой иерархию, на каждом уровне которой определяется свой набор тестов;**
- **разработка явных тестовых процедур (тест-требований) - в случае достаточно подробных функциональных требований и четко прописанной концепции разработки тестов явные тестовые процедуры могут и не разрабатываться. Однако, при наличии необходимых ресурсов, разработка тест-требований позволит более четко интерпретировать подвергаемые тестированию функциональные требования - тест-требования снижают риск неоднозначной трактовки функциональных требований;**
- **разработка тестовых примеров - тестовые примеры должны соответствовать требованиям к полноте тестирования и составляться либо на базе тест-требований, либо на основании функциональных требований. Данный вид деятельности наиболее продолжителен во времени;**
- **разработка тестовых примеров, основанных на архитектуре (в случае необходимости) - некоторые тестовые примеры основываются не на функциональных требованиях, а на особенностях архитектуры тестируемого модуля. Для разработки тестовых примеров, основанных на архитектуре, необходимо использовать подход стеклянного ящика. Эти тестовые примеры пишутся либо на основе низкоуровневых требований к системе, либо на основе низкоуровневых тест-требований, если они разрабатывались на одном из предыдущих этапов;**
- **составление спецификации тестовых примеров - результатом деятельности тестировщика в ходе данного этапа составляется документ Test Design Specification (формат которого описан в стандарте IEEE 829[15]).**

На следующем этапе проводится реализация тестов (например, в виде тестового окружения и формализованных описаний тестовых примеров). В ходе этого этапа формируются тестовые наборы данных, которые используются в тестовых примерах, создается тестовое окружение, и также осуществляется интеграция тестового окружения с тестируемым модулем.

После того, как все тесты реализованы, они выполняются на тестовом стенде в ручном или автоматическом режиме. Вне зависимости от вида тестирования в ходе этого этапа решаются две задачи: выполнение тестовых примеров, и сбор, и анализ результатов тестирования.

Сбору подлежит следующая информация:

- **результат выполнения каждого тестового примера (прошел/не прошел);**
- **информация об информационном окружении системы в случае, если тест не прошел;**
- **информация о ресурсах, которые потребовались для выполнения тестового примера.**

По результатам анализа этой информации составляются запросы на изменение проектной документации, программного кода тестируемого модуля или тестового окружения.

Этапы разработки (доработки), реализации и выполнения тестов продолжаются до тех пор, пока не будет достигнут критерий завершения модульного тестирования. Примером такого критерия может служить отсутствие не прошедших тестовых примеров при 75% покрытии строк исходного кода.

После прекращения тестирования выполняются работы по оценке проведенного тестирования, в ходе которых:

- **описываются отличия реального процесса тестирования от запланированного;**
- **отличия поведения тестируемого модуля от описанного в требованиях (с целью дальнейшей коррекции требований);**
- **составляется общий отчет о прохождении тестов, включающий в себя и информацию о покрытии.**

В завершение модульного тестирования необходимо проверить, что все созданные в его ходе артефакты - документы, программный код, файлы отчетов и данных - помещены в базу данных проекта, которая хранит все данные, используемые и создаваемые в процессе разработки программной системы.

3.2 ИНТЕГРАЦИОННОЕ ТЕСТИРОВАНИЕ И СИСТЕМНОЕ ТЕСТИРОВАНИЕ

Результатом тестирования и верификации отдельных модулей, составляющих программную систему, должно быть заключение о том, что эти модули являются внутренне непротиворечивыми и соответствуют требованиям. Однако отдельные модули редко функционируют сами по себе, поэтому следующая задача после тестирования отдельных модулей - тестирование корректности взаимодействия нескольких модулей, объединенных в единое целое.

Такое тестирование называют интеграционным[19]. Его цель – удостовериться в корректности совместной работы компонент программы.

Интеграционное тестирование называют еще тестированием архитектуры системы. С одной стороны, это название обусловлено тем, что интеграционные тесты включают в себя проверки всех возможных видов взаимодействий между программными модулями и элементами, которые определяются в архитектуре системы - таким образом, интеграционные тесты проверяют полноту взаимодействий в тестируемой реализации системы. С другой стороны, результаты выполнения интеграционных тестов - один из основных источников информации для процесса улучшения и уточнения архитектуры

системы, межмодульных и межкомпонентных интерфейсов. Т.е., с этой точки зрения, интеграционные тесты проверяют корректность взаимодействия компонент системы.

Примером проверки корректности взаимодействия могут служить два модуля, один из которых накапливает сообщения протокола о принятых файлах, а второй выводит этот протокол на экран. В функциональных требованиях к системе записано, что сообщения должны выводиться в обратном хронологическом порядке. Однако, модуль хранения сообщений сохраняет их в прямом порядке, а модуль вывода использует стек для вывода в обратном порядке. Модульные тесты, затрагивающие каждый модуль по отдельности, не дадут здесь никакого эффекта - вполне реальна обратная ситуация, при которой сообщения хранятся в обратном порядке, а выводятся с использованием очереди. Обнаружить потенциальную проблему можно только проверив взаимодействие модулей при помощи интеграционных тестов. Ключевым моментом здесь является то, что в обратном хронологическом порядке сообщения выводит система в целом, т.е., проверив модуль вывода и обнаружив, что он выводит сообщения в прямом порядке, мы не сможем гарантировать, что мы обнаружили дефект.

В результате проведения интеграционного тестирования и устранения всех выявленных дефектов получается согласованная и целостная архитектура программной системы, т.е. можно считать, что интеграционное тестирование - это тестирование архитектуры и низкоуровневых функциональных требований.

Интеграционное тестирование, как правило, представляет собой итеративный процесс, при котором проверяется функциональной все более и более увеличивающейся в размерах совокупности модулей.

Структурная классификация методов интеграционного тестирования

Как правило, интеграционное тестирование проводится уже по завершении модульного тестирования для всех интегрируемых модулей. Однако это далеко не всегда так. Существует несколько методов проведения интеграционного тестирования:

- **восходящее тестирование;**
- **монолитное тестирование;**
- **нисходящее тестирование.**

Все эти методики основываются на знаниях об архитектуре системы, которая часто изображается в виде структурных диаграмм или диаграмм вызовов функций. Каждый узел на такой диаграмме представляет собой программный модуль, а стрелки между ними представляют собой зависимость по вызовам между модулями. Основное различие методик интеграционного тестирования заключается в направлении движения по этим диаграммам и в широте охвата за одну итерацию.

Восходящее тестирование. При использовании этого метода подразумевается, что сначала тестируются все программные модули, входящие в состав системы и только затем они объединяются для интеграционного тестирования. При таком подходе значительно упрощается локализация ошибок: если модули протестированы по отдельности, то ошибка при их совместной работе есть проблема их интерфейса. При таком подходе область поиска проблем у тестировщика достаточно узка, и поэтому гораздо выше вероятность правильно идентифицировать дефект.[2]

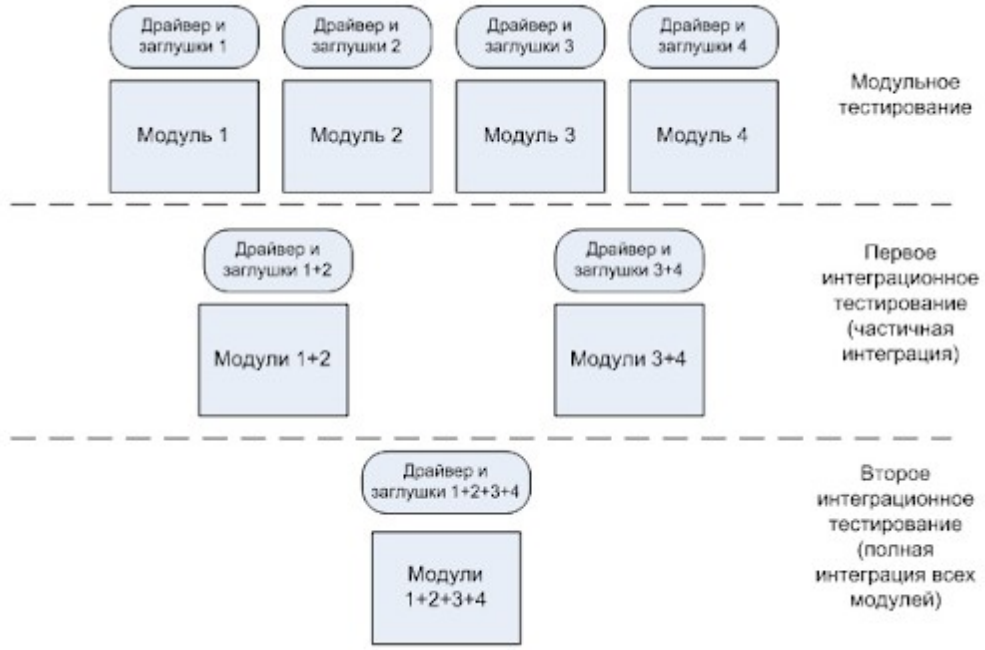

Рисунок 5 - Схема восходящего тестирования

Однако, у восходящего метода тестирования есть существенный недостаток - необходимость в разработке драйвера[2] и заглушек для модульного тестирования перед проведением интеграционного тестирования и необходимость в разработке драйвера и заглушек при интеграционном тестировании части модулей системы.

С одной стороны драйверы и заглушки - мощный инструмент тестирования, с другой - их разработка требует значительных ресурсов, особенно при изменении состава интегрируемых модулей, иначе говоря, может потребоваться один набор драйверов для модульного тестирования каждого модуля, отдельный драйвер и заглушки для тестирования интеграции двух модулей из набора, отдельный - для тестирования интеграции трех модулей и т.п. В первую очередь это связано с тем, что при интеграции модулей отпадает необходимость в некоторых заглушках, а также требуется изменение драйвера, которое поддерживает новые тесты, затрагивающие несколько модулей.

Монолитное тестирование предполагает, что отдельные компоненты системы серьезного тестирования не проходили. Основное преимущество данного метода - отсутствие необходимости в разработке тестового окружения, драйверов и заглушек. После разработки всех модулей выполняется их интеграция, затем система проверяется вся в целом. Этот подход не следует путать с системным тестированием, которому посвящена следующая лекция. Несмотря на то, что при монолитном тестировании проверятся работа всей системы в целом, основная задача этого тестирования - определить проблемы взаимодействия отдельных модулей системы. Задачей же системного тестирования является оценка качественных и количественных характеристик системы с точки зрения их приемлемости для конечного пользователя.

Монолитное тестирование имеет ряд серьезных недостатков.

1. **Очень трудно выявить источник ошибки (идентифицировать ошибочный фрагмент кода). В большинстве модулей следует предполагать наличие ошибки. Проблема сводится к определению того, какая из ошибок во всех вовлечённых модулях привела к полученному результату. При этом возможно наложение эффектов ошибок. Кроме того, ошибка в одном модуле может блокировать тестирование другого.**

2. **Трудно организовать исправление ошибок. В результате тестирования тестировщиком фиксируется найденная проблема. Дефект в системе, вызвавший эту проблему, будет устранять разработчик. Поскольку, как правило, тестируемые модули написаны разными людьми, возникает проблема - кто из них является ответственным за поиск устранение дефекта? При такой "коллективной безответственности" скорость устранения дефектов может резко упасть.**

3. **Процесс тестирования плохо автоматизируется. Преимущество (нет дополнительного программного обеспечения, сопровождающего процесс тестирования) оборачивается недостатком. Каждое внесённое изменение требует повторения всех тестов.**

Нисходящее тестирование предполагает, что процесс интеграционного тестирования движется следом за разработкой. Сначала тестируют только самый верхний управляющий уровень системы, без модулей более низкого уровня. Затем постепенно с более высокоуровневыми модулями интегрируются более низкоуровневые. В результате применения такого метода отпадает необходимость в драйверах (роль драйвера выполняет более высокоуровневый модуль системы), однако сохраняется нужда в заглушках.

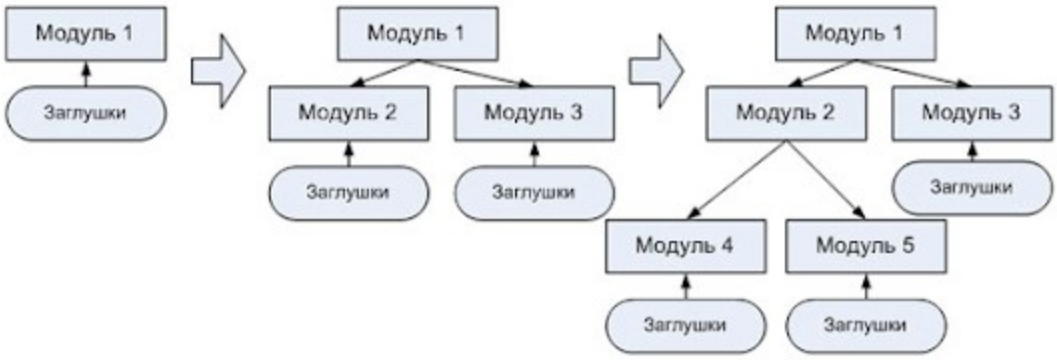

Рисунок 6 - Схема монолитного тестирования

Временная классификация методов интеграционного тестирования

На практике чаще всего в различных частях проекта применяются все рассмотренные в предыдущем разделе методы в совокупности. Каждый модуль тестируют по мере готовности отдельно, а потом включают в уже готовую композицию. Для одних частей тестирование получается нисходящим, для других - восходящим. В связи с этим представляется полезным рассмотреть еще один тип классификации типов интеграционного тестирования - классификацию по времени интеграции.

В рамках этой классификации выделяют:

- **тестирование с поздней интеграцией;**
- **тестирование с постоянной интеграцией;**
- **тестирование с регулярной или послойной интеграцией.**

Тестирование с поздней интеграцией - практически полный аналог монолитного тестирования. Интеграционное тестирование при такой схеме откладывается на как можно более поздние сроки проекта. Этот подход оправдан в том случае, если система является конгломератом слабо связанных между собой модулей, которые

взаимодействуют по какому-либо стандартному интерфейсу, определенному вне проекта (например, в случае, если система состоит из отдельных Web-сервисов).

Тестирование с постоянной интеграцией подразумевает, что, как только разрабатывается новый модуль системы, он сразу же интегрируется со всей остальной системой. При этом тесты для этого модуля проверяют как сугубо его внутреннюю функциональность, так и его взаимодействие с остальными модулями системы. Таким образом, этот подход совмещает в себе модульное тестирование и интеграционное. Разработки заглушек при таком подходе не требуется, но может потребоваться разработка драйверов. В настоящее время именно этот подход называют unit testing, несмотря на то, что в отличие от классического модульного тестирования здесь не проверяется функциональность изолированного модуля. Локализация ошибок межмодульных интерфейсов при таком подходе несколько затруднена, но все же значительно ниже, чем при монолитном тестировании. Большая часть таких ошибок выявляется достаточно рано именно за счет частоты интеграции и за счет того, что за одну итерацию тестирования проверяется сравнительно небольшое число межмодульных интерфейсов.

При тестировании с регулярной или послойной интеграцией интеграционному тестированию подлежат сильно связанные между собой группы модулей (слои), которые затем также интегрируются между собой. Такой вид интеграционного тестирования называют также иерархическим интеграционным тестированием, поскольку укрупнение интегрированных частей системы, как правило, происходит по иерархическому принципу. Однако, в отличие от нисходящего или восходящего тестирования, направление прохода по иерархии в этом подходе не задано.

Задачи и цели системного тестирования

Системное тестирование - один из самых сложных видов тестирования. На этом этапе проводится не только функциональное тестирование, но и оценка характеристик качества системы - ее устойчивости, надежности, безопасности и производительности. На этом этапе выявляются многие проблемы внешних интерфейсов системы, связанные с неверным взаимодействием с другими системами, аппаратным обеспечением, неверным распределением памяти, отсутствием корректного освобождения ресурсов и т.п.

Виды системного тестирования

Принято выделять следующие виды системного тестирования:

- **функциональное тестирование;**
- **тестирование производительности;**
- **нагрузочное или стрессовое тестирование;**
- **тестирование конфигурации;**
- **тестирование безопасности;**
- **тестирование надежности и восстановления после сбоев;**
- **тестирование удобства использования.**

В ходе системного тестирования проводятся далеко не все из перечисленных видов тестирования - конкретный их набор зависит от тестируемой системы.

Исходной информацией для проведения перечисленных видов тестирования являются два класса требований: функциональные и нефункциональные. Функциональные

требования явно описывают, что система должна делать и какие выполнять преобразования входных значений в выходные. Нефункциональные требования определяют свойства системы, напрямую не связанные с ее функциональностью. Примером таких свойств может служить время отклика на запрос пользователя (например, не более 2 секунд), время бесперебойной работы (например, не менее 10000 часов между двумя сбоями), количество ошибок, которые допускает начинающий пользователь за первую неделю работы (не более 100), и т.п.

Рассмотрим каждый вид тестирования подробнее.

Функциональное тестирование. Данный вид тестирования предназначен для доказательства того, что вся система в целом ведет себя в соответствии с ожиданиями пользователя, формализованными в виде системных требований. В ходе данного вида тестирования проверяются все функции системы с точки зрения ее пользователей (как пользователей-людей, так и "пользователей" - других программных систем). Система при функциональном тестировании рассматривается как черный ящик, поэтому в данном случае полезно использовать классы эквивалентности. Критерием полноты тестирования в данном случае будет полнота покрытия тестами системных функциональных требований (или системных тест-требований) и полнота тестирования классов эквивалентности, а именно:

- **все функциональные требования должны быть протестированы;**
- **все классы допустимых входных данных должны корректно обрабатываться системой;**
- **все классы недопустимых входных данных должны быть отброшены системой, при этом не должна нарушаться стабильность ее работы;**
- **в тестовых примерах должны генерироваться все возможные классы выходных данных системы;**
- **во время тестирования система должна побывать во всех своих внутренних состояниях, пройдя при этом по всем возможным переходам между состояниями.**

Результаты системного тестирования протоколируются и анализируются совершенно аналогично тому, как это делается для модульного и интеграционного тестирования. Основная сложность здесь заключается в локализации дефектов в программном коде системы и определении зависимостей одних дефектов от других (эффект "четного числа ошибок").

Тестирование производительности. Данный вид тестирования направлен на определение того, что система обеспечивает должный уровень производительности при обработке пользовательских запросов. Тестирование производительности выполняется при различных уровнях нагрузки на систему, на различных конфигурациях оборудования. Выделяют три основных фактора, влияющие на производительность системы: количество поддерживаемых системой потоков (например, пользовательских сессий), количество свободных системных ресурсов, количество свободных аппаратных ресурсов[19].

Тестирование производительности позволяет выявлять узкие места в системе, которые проявляются в условиях повышенной нагрузки или нехватки системных ресурсов. В этом случае по результатам тестирования проводится доработка системы, изменяются алгоритмы выделения и распределения ресурсов системы.

Все требования, относящиеся к производительности системы, должны быть четко определены и обязательно должны включать в себя числовые оценки параметров производительности. Т.е., например, требование "Система должна иметь приемлемое время отклика на запрос пользователя" является непригодным для тестирования. Напротив, требование "Время отклика на запрос пользователя не должно превышать 2 секунды" может быть протестировано.

То же самое относится и к результатам тестирования производительности. В отчетах по данному виду тестирования сохраняют такие показатели, как загрузка аппаратного и системного программного обеспечения (количество циклов процессора, выделенной памяти, количество свободных системных ресурсов и т.п.). Также важны скоростные характеристики тестируемой системы (количество обработанных в единицу времени запросов, временные интервалы между началом обработки каждого последующего запроса, равномерность времени отклика в разные моменты времени и т.п.).

Для проведения тестирования производительности требуется наличие генератора запросов, который подает на вход системы поток данных, типичных для сеанса работы с ней. Тестовое окружение должно включать в себя кроме программной компоненты еще и аппаратную, причем на таком тестовом стенде должна существовать возможность моделирования различного уровня доступных ресурсов.

Стрессовое тестирование. Стрессовое тестирование имеет много общего с тестированием производительности, однако его основная задача - не определить производительность системы, а оценить производительность и устойчивость системы в случае, когда для своей работы она выделяет максимально доступное количество ресурсов, либо когда она работает в условиях их критической нехватки. Основная цель стрессового тестирования - вывести систему из строя, определить те условия, при которых она не сможет далее нормально функционировать. Для проведения стрессового тестирования используются те же самые инструменты, что и для тестирования производительности. Однако, например, генератор нагрузки при стрессовом тестировании должен генерировать запросы пользователей с максимально возможной скоростью либо генерировать данные запросов таким образом, чтобы они были максимально возможными по объему обработки.

Стрессовое тестирование очень важно при тестировании web-систем и систем с открытым доступом, уровень нагрузки на которые зачастую очень сложно прогнозировать.

Тестирование конфигурации. Большинство программных систем массового назначения предназначено для использования на самом разном оборудовании. Несмотря на то, что в настоящее время особенности реализации периферийных устройств скрываются драйверами операционных систем, которые имеют унифицированный с точки зрения прикладных систем интерфейс, проблемы совместимости (как программной, так и аппаратной) все равно существуют.

В ходе тестирования конфигурации проверяется, что программная система корректно работает на всем поддерживаемом аппаратном обеспечении и совместно с другими программными системами. Необходимо также проверять, что система продолжает стабильно работать при горячей замене любого поддерживаемого устройства на аналогичное. При этом система не должна давать сбоев ни в момент замены устройства, ни после начала работы с новым устройством.

Также необходимо проверять, что система корректно обрабатывает проблемы, возникающие в оборудовании, как штатные (например, сигнал конца бумаги в принтере), так и нештатные (сбой питания)[19].

Тестирование безопасности. Если программная система предназначена для хранения или обработки данных, содержимое которых представляет собой тайну определенного рода (личную, коммерческую, государственную и т.п.), то к свойствам системы, обеспечивающим сохранение этой тайны, будут предъявляться повышенные требования. Эти требования должны быть проверены при тестировании безопасности системы. В ходе этого тестирования проверяется, что информация не теряется, не повреждается, ее невозможно подменить, а также к ней невозможно получить несанкционированный доступ, в том числе при помощи использования уязвимостей в самой программной системе.

Выделяют следующие группы свойств программной системы, подлежащие проверке (некоторые группы свойств укрупнены для сокращения списка):

- **разграничение и контроль доступа - предотвращение доступа к "чужой" информации;**
- **очистка и защита памяти - предотвращение доступа к остаточной информации после удаления объектов из памяти;**
- **маркировка и защита информации, передаваемой во внешний мир - сохранение уровня секретности даже вне системы;**
- **идентификация и аутентификация - предоставление доступа только санкционированным пользователям и отказ в доступе всем остальным;**
- **регистрация (аудит событий) - регистрация в специальном журнале всех событий системы, связанных с безопасностью для последующего анализа;**
- **гарантии проектирования и архитектуры - система должна быть спроектирована таким образом, чтобы гарантировать защищенность информации с определенным уровнем уверенности;**
- **тестирование - все функции по обеспечению безопасности должны быть протестированы во всех режимах;**
- **целостность и восстановление средств защиты - система должна иметь средства контроля корректности всех правил разграничения доступа и системы безопасности в целом, а также средства их восстановления при сбое;**
- **документация разработчика, администратора и пользователя - все средства системы по обеспечению безопасности должны быть описаны в соответствующих руководствах.**

Тестирование надежности и восстановления после сбоев. Для корректной работы системы в любой ситуации необходимо удостовериться в том, что она восстанавливает свою функциональность и продолжает корректно работать после любой проблемы, прервавшей ее работу. При тестировании восстановления после сбоев имитируются сбои оборудования или окружающего программного обеспечения либо сбои программной системы, вызванные внешними факторами. При анализе поведения системы в этом случае необходимо обращать внимание на два фактора - минимизацию потерь данных в результате сбоя и минимизацию времени между сбоем и продолжением нормального функционирования системы[19].

Тестирование удобства использования. Отдельная группа нефункциональных требований - требования к удобству использования пользовательского интерфейса системы. Этот вид тестирования будет рассмотрен в следующей лекции.

В результате выполнения всех рассмотренных выше видов тестирования делается заключение о функциональности и свойствах системы, после чего узкие места системы дорабатываются до реализации необходимой функциональности или до достижения системой необходимых свойств.

3.3 ОТЛАДКА ПО И ЕЁ ВИДЫ

Классификация ошибок

Отладка – это процесс локализации и исправления ошибок, обнаруженных при тестировании программного обеспечения. Локализацией называют процесс определения оператора программы, выполнение которого вызвало нарушение нормального вычислительного процесса. Для исправления ошибки необходимо определить ее причину, т. е. определить оператор или фрагмент, содержащие ошибку. Причины ошибок могут быть как очевидны, так и очень глубоко скрыты[20].

В целом сложность отладки обусловлена следующими причинами:

- **требует от программиста глубоких знаний специфики управления используемыми техническими средствами, операционной системы, среды и языка программирования, реализуемых процессов, природы и специфики различных ошибок, методик отладки и соответствующих программных средств;**
- **психологически дискомфортна, так как необходимо искать собственные ошибки и, как правило, в условиях ограниченного времени;**
- **возможно взаимовлияние ошибок в разных частях программы, например, за счет затирания области памяти одного модуля другим из-за ошибок адресации;**
- **отсутствуют четко сформулированные методики отладки.**

В соответствии с этапом обработки, на котором проявляются ошибки, различают:

- **синтаксические ошибки - ошибки, фиксируемые компилятором (транслятором, интерпретатором) при выполнении синтаксического и частично семантического анализа программы;**
- **логические ошибки;**
- **ошибки компоновки - ошибки, обнаруженные компоновщиком (редактором связей) при объединении модулей программы;**
- **ошибки выполнения - ошибки, обнаруженные операционной системой, аппаратными средствами или пользователем при выполнении программы.**

Синтаксические ошибки. Синтаксические ошибки относят к группе самых простых, так как синтаксис языка, как правило, строго формализован, и ошибки сопровождаются развернутым комментарием с указанием ее местоположения.

Определение причин таких ошибок, как правило, труда не составляет, и даже при нечетком знании правил языка за несколько прогонов удается удалить все ошибки данного типа[20].

Следует иметь в виду, что чем лучше формализованы правила синтаксиса языка, тем больше ошибок из общего количества может обнаружить компилятор и, соответственно, меньше ошибок будет обнаруживаться на следующих этапах. В связи с этим говорят о языках программирования с защищенным синтаксисом и с незащищенным синтаксисом. К первым, безусловно, можно отнести Pascal, имеющий очень простой и четко определенный синтаксис, хорошо проверяемый при компиляции программы, ко вторым - Си со всеми его модификациями. Чего стоит хотя бы возможность выполнения присваивания в условном операторе в Си, например:

if (c = n) x = 0; /* в данном случае не проверятся равенство с и n, а выполняется присваивание с значения n, после чего результат операции сравнивается с нулем, если программист хотел выполнить не присваивание, а сравнение, то эта ошибка будет обнаружена только на этапе выполнения при получении результатов, отличающихся от ожидаемых.

Ошибки компоновки. Ошибки компоновки, как следует из названия, связаны с проблемами, обнаруженными при разрешении внешних ссылок. Например, предусмотрено обращение к подпрограмме другого модуля, а при объединении модулей данная подпрограмма не найдена или не стыкуются списки параметров. В большинстве случаев ошибки такого рода также удается быстро локализовать и устранить.

Ошибки выполнения. К самой непредсказуемой группе относятся ошибки выполнения. Прежде всего они могут иметь разную природу, и соответственно по-разному проявляться. Часть ошибок обнаруживается и документируется операционной системой. Выделяют четыре способа проявления таких ошибок:

- **появление сообщения об ошибке, зафиксированной схемами контроля выполнения машинных команд, например, переполнении разрядной сетки, ситуации «деление на ноль», нарушении адресации и т. п.;**
- **появление сообщения об ошибке, обнаруженной операционной системой, например, нарушении защиты памяти, попытке записи на устройства, защищенные от записи, отсутствии файла с заданным именем и т. п.;**
- **«зависание» компьютера, как простое, когда удается завершить программу без перезагрузки операционной системы, так и «тяжелое», когда для продолжения работы необходима перезагрузка;**
- **несовпадение полученных результатов с ожидаемыми[20].**

Примечание. Отметим, что, если ошибки этапа выполнения обнаруживает пользователь, то в двух первых случаях, получив соответствующее сообщение, пользователь в зависимости от своего характера, степени необходимости и опыта работы за компьютером, либо попробует понять, что произошло, ища свою вину, либо обратится за помощью, либо постарается никогда больше не иметь дела с этим продуктом. При «зависании» компьютера пользователь может даже не сразу понять, что происходит что-то не то, хотя его печальный опыт и заставляет волноваться каждый раз, когда компьютер не выдает быстрой реакции на введенную команду, что также целесообразно иметь в виду. Также опасны могут быть ситуации, при которых пользователь получает неправильные результаты и использует их в своей работе.

Причины ошибок выполнения очень разнообразны, а потому и локализация может оказаться крайне сложной. Все возможные причины ошибок можно разделить на следующие группы:

- **неверное определение исходных данных,**
- **логические ошибки,**
- **накопление погрешностей результатов вычислений.**

Рисунок 7 - Схема ошибок

Методы отладки программного обеспечения

Отладка программы в любом случае предполагает обдумывание и логическое осмысление всей имеющейся информации об ошибке. Большинство ошибок можно обнаружить по косвенным признакам посредством тщательного анализа текстов программ и результатов тестирования без получения дополнительной информации. При этом используют различные методы:

- **ручного тестирования;**
- **индукции;**
- **дедукции;**
- **обратного прослеживания.**

Метод ручного тестирования

Это - самый простой и естественный способ данной группы. При обнаружении ошибки необходимо выполнить тестируемую программу вручную, используя тестовый набор, при работе с которыми была обнаружена ошибка. Метод очень эффективен, но не применим для больших программ, программ со сложными вычислениями и в тех случаях, когда ошибка связана с неверным представлением программиста о выполнении некоторых операций. Данный метод часто используют как составную часть других методов отладки.

Метод индукции

Метод основан на тщательном анализе симптомов ошибки, которые могут проявляться как неверные результаты вычислений или как сообщение об ошибке. Если компьютер просто "зависает", то фрагмент проявления ошибки вычисляют, исходя из последних полученных результатов и действий пользователя. Полученную таким образом информацию организуют и тщательно изучают, просматривая соответствующий фрагмент программы. В результате этих действий выдвигают гипотезы об ошибках, каждую из которых проверяют. Если гипотеза верна, то детализируют информацию об ошибке, иначе - выдвигают другую гипотезу. Последовательность выполнения отладки методом индукции показана на рисунке в виде схемы алгоритма.

Самый ответственный этап - выявление симптомов ошибки. Организуя данные об ошибке, целесообразно записать все, что известно о её проявлениях, причем фиксируют, как ситуации, в которых фрагмент с ошибкой выполняется нормально, так и ситуации, в которых ошибка проявляется. Если в результате изучения данных никаких гипотез не появляется, то необходима дополнительная информация об ошибке. Дополнительную информацию можно получить, например, в результате выполнения схожих тестов. В процессе доказательства пытаются выяснить, все ли проявления ошибки объясняет данная гипотеза, если не все, то либо гипотеза не верна, либо ошибок несколько.

Метод дедукции

По методу дедукции вначале формируют множество причин, которые могли бы вызвать данное проявление ошибки. Затем анализируя причины, исключают те, которые противоречат имеющимся данным. Если все причины исключены, то следует выполнить дополнительное тестирование исследуемого фрагмента. В противном случае наиболее вероятную гипотезу пытаются доказать. Если гипотеза объясняет полученные признаки ошибки, то ошибка найдена, иначе - проверяют следующую причину.

Метод обратного прослеживания

Для небольших программ эффективно применение метода обратного прослеживания. Начинают с точки вывода неправильного результата. Для этой точки строится гипотеза о значениях основных переменных, которые могли бы привести к получению имеющегося результата. Далее, исходя из этой гипотезы, делают предложения о значениях переменных в предыдущей точке. Процесс продолжают, пока не обнаружат причину ошибки.

Методы и средства получения дополнительной информации

Для получения дополнительной информации об ошибке можно выполнить добавочные тесты или использовать специальные методы и средства:

- **отладочный вывод;**
- **интегрированные средства отладки;**
- **независимые отладчики.**

Отладочный вывод. Метод требует включения в программу дополнительного отладочного вывода в узловых точках. Узловыми считают точки алгоритма, в которых основные переменные программы меняют свои значения. Например, отладочный вывод следует предусмотреть до и после завершения цикла изменения некоторого массива значений. (Если отладочный вывод предусмотреть в цикле, то будет выведено слишком много значений, в которых, как правило, сложно разбираться.) При этом

предполагается, что, выполнив анализ выведенных значений, программист уточнит момент, когда были получены неправильные значения, и сможет сделать вывод о причине ошибки.

Данный метод не очень эффективен и в настоящее время практически не используется, так как в сложных случаях в процессе отладки может потребоваться вывод большого количества - «трассы» значений многих переменных, которые выводятся при каждом изменении. Кроме того, внесение в программы дополнительных операторов может привести к изменению проявления ошибки, что нежелательно, хотя и позволяет сделать определенный вывод о ее природе.

Примечание. Ошибки, исчезающие при включении в программу или удалению из нее каких-либо «безобидных» операторов, как правило, связаны с «затиранием» памяти. В результате добавления или удаления операторов область затирания может сместиться в другое место и ошибка либо перестанет проявляться, либо будет проявляться по-другому.

Интегрированные средства отладки. Большинство современных сред программирования (Delphi, Builder C++, Visual Studio и т. д.) включают средства отладки, которые обеспечивают максимально эффективную отладку. Они позволяют:

- **выполнять программу по шагам, причем как с заходом в подпрограммы, так и выполняя их целиком;**
- **предусматривать точки останова;**
- **выполнять программу до оператора, указанного курсором;**
- **отображать содержимое любых переменных при пошаговом выполнении;**
- **отслеживать поток сообщений и т. п.**

Отладка с использованием независимых отладчиков.

При отладке программ иногда используют специальные программы - отладчики, которые позволяют выполнить любой фрагмент программы в пошаговом режиме и проверить содержимое интересующих программиста переменных. Как правило такие отладчики позволяют отлаживать программу только в машинных командах, представленных в 16-ричном коде.

Общая методика отладки программного обеспечения[20]

Суммируя все сказанное выше, можно предложить следующую методику отладки программного обеспечения:

1 этап - изучение проявления ошибки - если выдано какое-либо сообщение или выданы неправильные, или неполные результаты, то необходимо их изучить и попытаться понять, какая ошибка могла так проявиться. При этом используют индуктивные и дедуктивные методы отладки. В результате выдвигают версии о характере ошибки, которые необходимо проверить. Для этого можно применить методы и средства получения дополнительной информации об ошибке. Если ошибка не найдена или система просто «зависла», переходят ко второму этапу.

2 этап - локализация ошибки - определение конкретного фрагмента, при выполнении которого произошло отклонение от предполагаемого вычислительного процесса. Локализация может выполняться:

- **путем отсечения частей программы, причем, если при отсечении некоторой части программы ошибка пропадает, то это может означать как то, что ошибка связана с этой частью, так и то, что внесенное изменение изменило проявление ошибки;**
- **с использованием отладочных средств, позволяющих выполнить интересующих нас фрагмент программы в пошаговом режиме и получить дополнительную информацию о месте проявления и характере ошибки, например, уточнить содержимое указанных переменных.**

При этом если были получены неправильные результаты, то в пошаговом режиме проверяют ключевые точки процесса формирования данного результата. Как подчеркивалось выше, ошибка не обязательно допущена в том месте, где она проявилась. Если в конкретном случае это так, то переходят к следующему этапу.

3 этап - определение причины ошибки - изучение результатов второго этапа и формирование версий возможных причин ошибки. Эти версии необходимо проверить, возможно, используя отладочные средства для просмотра последовательности операторов или значений переменных.

4 этап - исправление ошибки - внесение соответствующих изменений во все операторы, совместное выполнение которых привело к ошибке.

5 этап - повторное тестирование - повторение всех тестов с начала, так как при исправлении обнаруженных ошибок часто вносят в программу новые.

Следует иметь в виду, что процесс отладки можно существенно упростить, если следовать основным рекомендациям структурного подхода к программированию:

- **программу наращивать «сверху-вниз», от интерфейса к обрабатывающим подпрограммам, тестируя ее по ходу добавления подпрограмм;**
- **выводить пользователю вводимые им данные для контроля и проверять их на допустимость сразу после ввода;**
- **предусматривать вывод основных данных во всех узловых точках алгоритма (ветвлениях, вызовах подпрограмм).**

Кроме того, следует более тщательно проверять фрагменты программного обеспечения, где уже были обнаружены ошибки, так как вероятность ошибок в этих местах по статистике выше. Это вызвано следующими причинами. Во-первых, ошибки чаще допускают в сложных местах или в тех случаях, если спецификации на реализуемые операции недостаточно проработаны. Во-вторых, ошибки могут быть результатом того, что программист устал, отвлекся или плохо себя чувствует. В-третьих, как уже упоминалось выше, ошибки часто появляются в результате исправления уже найденных ошибок.

3.4 ТЕСТИРОВАНИЕ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА

Часть программной системы, обеспечивающая работу интерфейса с пользователем - один из наиболее нетривиальных объектов для верификации. Нетривиальность заключается в двояком восприятии термина "пользовательский интерфейс".

С одной стороны, пользовательский интерфейс - часть программной системы. Соответственно, на пользовательский интерфейс пишутся функциональные и низкоуровневые требования, по которым затем составляются тест-требования и тест-планы. При этом, как правило, требования определяют реакцию системы на каждый ввод пользователя (при помощи клавиатуры, мыши или иного устройства ввода) и вид информационных сообщений системы, выводимых на экран, печатающее устройство или иное устройство вывода. При верификации таких требований речь идет о проверке функциональной полноты пользовательского интерфейса - насколько реализованные функции соответствуют требованиям, корректно ли выводится информация на экран.

С другой стороны, пользовательский интерфейс - "лицо" системы, и от его продуманности зависит эффективность работы пользователя с системой. Факторы, влияющие на эффективность работы, слабо поддаются формализации в виде конкретных требований к отдельным элементам, однако должны быть учтены в виде общих рекомендаций и принципов построения пользовательского интерфейса программной системы. Проверка интерфейса на эффективность человеко-машинного взаимодействия получила название проверки удобства использования (usability verification; в русскоязычной литературе в качестве перевода термина usability часто используют слово "практичность").

Функциональное тестирование пользовательских интерфейсов[21]

Функциональное тестирование пользовательского интерфейса состоит из пяти фаз:

- **анализ требований к пользовательскому интерфейсу;**
- **разработка тест-требований и тест-планов для проверки пользовательского интерфейса;**
- **выполнение тестовых примеров и сбор информации о выполнении тестов;**
- **определение полноты покрытия пользовательского интерфейса требованиями;**
- **составление отчетов о проблемах в случае несовпадения поведения системы и требований либо в случае отсутствия требований на отдельные интерфейсные элементы.**

Все эти фазы точно такие же, как и в случае тестирования любого другого компонента программной системы. Отличия заключаются в трактовке некоторых терминов в применении к пользовательскому интерфейсу и в особенностях автоматизированного сбора информации на каждой фазе.

Так, тест-планы для проверки пользовательского интерфейса, как правило, представляют собой сценарии, описывающие действия пользователя при работе с системой. Сценарии могут быть записаны либо на естественном языке, либо на формальном языке какой-либо системы автоматизации пользовательского интерфейса. Выполнение тестов при этом производится либо оператором в ручном режиме, либо системой, которая эмулирует поведение оператора.

При сборе информации о выполнении тестовых примеров обычно применяются технологии анализа выводимых на экран форм и их элементов (в случае графического интерфейса) или выводимого на экран текста (в случае текстового), а не проверка значений тех или иных переменных, устанавливаемых программной системой.

Под полнотой покрытия пользовательского интерфейса понимается то, что в результате выполнения всех тестовых примеров каждый интерфейсный элемент был использован хотя бы один раз во всех доступных режимах.

Отчеты о проблемах в пользовательском интерфейсе могут включать в себя как описания несоответствий требований и реального поведения системы, так и описания проблем в требованиях к пользовательскому интерфейсу. Основной источник проблем в этих требованиях - их тестонепригодность, вызванная расплывчатостью формулировок и неконкретностью.

Проверка требований к пользовательскому интерфейсу[21]

Требования к пользовательскому интерфейсу могут быть разбиты на две группы:

- **требования к внешнему виду пользовательского интерфейса и формам взаимодействия с пользователем;**
- **требования по доступу к внутренней функциональности системы при помощи пользовательского интерфейса.**

Другими словами, первая группа требований описывает взаимодействие подсистемы интерфейса с пользователем, а вторая - с внутренней логикой системы.

К первой группе можно отнести следующие типы требований:

Требования к размещению элементов управления на экранных формах

Данные требования могут определять общие принципы размещения элементов пользовательского интерфейса или требования к размещению конкретных элементов.

Например, общие требования по размещению элементов на графической экранной форме могут выглядеть следующим образом:

Каждое окно приложения должно быть разбито на три части: строка меню, рабочая область и статусная строка. Строка меню должна быть горизонтальной и прижатой к верхней части окна, статусная строка должна быть горизонтальной и прижатой к нижней части окна, рабочая область должна находиться между строкой меню и статусной строкой и занимать всю оставшуюся площадь окна.

При тестировании данного требования достаточно определить, что в каждом окне системы действительно присутствуют три части, которые расположены и прижаты согласно требованиям даже при изменении размеров окна, его сворачивании/разворачивании, перемещении по экрану, при перекрытии его другими окнами.

Примером требований по размещению конкретного элемента может служить следующее:

Кнопка "Начать передачу" должна находиться непосредственно под строкой меню в левой части рабочей области окна.

При тестировании такого требования также необходимо определить, сохраняется ли расположение элемента при изменении размера окна, а также при использовании элемента (в данном случае - при нажатии).

Требования к содержанию и оформлению выводимых сообщений

Требования к содержанию и оформлению выводимых сообщений определяют текст сообщений, выводимых системой, его шрифтовое и цветовое оформление. Также часто в таких требованиях определяется, в каких случаях выводится то или иное сообщение.

Так, например, для тестирования требования

Сообщение "Невозможно открыть файл" должно выводиться в статусную строку прижатым к левому краю, красным цветом, полужирным шрифтом в случае недоступности открываемого файла по чтению.

необходимо проверить, что при возникновении указанной ситуации сообщение действительно выводится согласно требованиям.

Однако в случае тестирования требования вида

Сообщения об ошибках должны выводиться в статусную строку прижатыми к левому краю красным цветом полужирным шрифтом.

необходимо проверять форматы всех возможных сообщений об ошибках программы во всех возможных ошибочных ситуациях. Таким образом, очевидно, что при тестировании пользовательского интерфейса не всегда можно однозначно определить количество тестовых примеров, которые понадобятся для тестирования требования. Эта проблема вызвана тем, что требования к пользовательскому интерфейсу зачастую кажутся слишком очевидными для их точной формулировки. Эта неконкретность требований и вызывает большое количество тестов для каждого требования.

Требования к форматам ввода

Данная группа требований определяет, в каком виде информация поступает от пользователя в систему. При этом кроме собственно требований, определяющих корректный формат, к этой группе относятся требования, определяющие реакцию системы на некорректный ввод. Для проверки таких требований необходимо проверять как корректный ввод, так и некорректный. Желательно при этом разбивать различные варианты ввода на классы эквивалентности (как минимум на два - корректные и некорректные).

Ко второй группе относятся следующие типы требований:

Требования к реакции системы на ввод пользователя

Данный тип требований определяет связь внутренней логики системы и интерфейсных элементов. Например,

При нажатии кнопки "Сброс" значение таймера синхронизации передачи должно сбрасываться в 0.

Для проверки такого требования в тестовом примере должно быть сымитировано нажатие на кнопку "Сброс", после чего должна проводиться проверка значения таймера. Однако некоторые требования определяют в качестве реакции системы не то, как меняется ее внутреннее состояние, а реакцию пользовательского интерфейса.

Например, в требовании

При нажатии кнопки "Отложенный сброс" должно выводиться окно "Ввод значения времени для отложенного сброса".

в качестве реакции на использование одного интерфейсного элемента определяется появление другого интерфейсного элемента. Такие требования проверяются при помощи имитации ввода пользователя и анализа появляющихся интерфейсных элементов.

Требования к времени отклика на команды пользователя

В качестве отдельного типа требований можно выделить требования к времени отклика системы на различные пользовательские операции. Это связано с тем, что подсознательно пользователь воспринимает операции продолжительностью более 1 секунды как длительные. Если в этот момент система не сообщает пользователю о том, что она выполняет какую-либо операцию, пользователь начнет считать, что система зависла или работает в неверном режиме. В связи с этим либо каждое предельное время отклика должно быть указано в требованиях и пользовательской документации, либо во время длительных операций должны выводиться информационные сообщения (например, индикатор прогресса). Значения предельного времени и равномерность увеличения значений индикатора прогресса должны проверяться соответствующими тестами.

Тестопригодность требований к пользовательскому интерфейсу

Некоторые требования к пользовательскому интерфейсу могут оказаться тестонепригодными, либо их тестирование будет значительно затруднено. К таким требованиям в первую очередь относятся требования, описывающие субъективные характеристики интерфейса, которые не могут быть точно определены или измерены при выполнении тестовых примеров. При анализе требований к пользовательскому интерфейсу необходимо четко представлять, какой элемент интерфейса и каким образом будет проверяться, какая его характеристика будет измеряться в ходе тестирования.

Примером тестонепригодного требования может служить классическое требование

Пользовательский интерфейс должен быть интуитивно понятным.

Без определения четких критериев интуитивной понятности проверка такого требования невозможна. При этом необходимо понимать, что критерий в данном случае может быть двух видов: детерминированным или вероятностным. Примером детерминированного критерия может быть дополнение к требованию вида

Под интуитивной понятностью интерфейса понимается доступность любой функции системы при помощи не более чем 5 щелчков мыши по интерфейсным элементам.

Требование с таким уточнением поддается как ручному, так и автоматизированному тестированию, более того, результат такого тестирования не будет зависеть от субъективного мнения тестировщика (понятия об интуитивной понятности у всех разные).

Примером вероятностного критерия может служить следующее дополнение:

Под интуитивной понятностью интерфейса понимается, что пользователь обращается к руководству пользователя не чаще, чем раз в пять минут на этапе обучения и не чаще, чем раз в 2 часа на этапе активного использования системы. Значения должны быть получены на репрезентативной выборке пользователей не менее 1000 человек.

Проверка требования с таким дополнением не является задачей классической верификации и относится уже скорее к проверке удобства применения пользовательского интерфейса. Однако здесь также вводится четкий критерий, при использовании которого результаты тестирования могут быть воспроизведены.

Полнота покрытия пользовательского интерфейса[21]

При определении понятия покрытия пользовательского интерфейса можно ввести следующие его уровни:

- **функциональное покрытие - покрытие требований к пользовательскому интерфейсу;**
- **структурное покрытие - для обеспечения полного структурного покрытия каждый интерфейсный элемент должен быть использован в тестовых примерах хотя бы один раз;**
- **структурное покрытие с учетом состояния элементов интерфейса - для обеспечения этого уровня покрытия необходимо не только использовать каждый элемент интерфейса, но и привести его во все возможные состояния (*например, для чек-боксов - отмечен/не отмечен, для полей ввода - пустое/заполненное не целиком/заполненное полностью и т.п.*)**
- **структурное покрытие с учетом состояния элементов интерфейса и внутреннего состояния системы - поведение некоторых интерфейсных элементов может изменяться в зависимости от внутреннего состояния системы. Каждое такое различимое поведение интерфейсного элемента должно быть проверено. *Например, система может иметь два режима работы - нормальный и для начинающего пользователя, в котором нажатие каждого элемента сопровождается появлением всплывающей подсказки. В этом случае нужно проверить оба режима и при этом проверить, что подсказки появляются только в режиме для начинающих.***

При определении степени покрытия необходимо учитывать, что реакция на некоторые интерфейсные элементы определяется не программной системой, а на уровне операционной системы или среды выполнения. Так, например, реакция на использование многих интерфейсных элементов стандартного диалогового окна открытия файла определяется операционной системой и может не тестироваться.

Если уровень покрытия интерфейсных элементов тестами недостаточен, это является сигналом либо к уточнению требований к пользовательскому интерфейсу, либо к снижению степени подробности тестирования.

Методы проведения тестирования пользовательского интерфейса, повторяемость тестирования пользовательского интерфейса

Функциональное тестирование пользовательского интерфейса может проводиться различными методами - как вручную при непосредственном участии оператора, так и при помощи различного инструментария, автоматизирующего выполнение тестовых примеров.

Ручное тестирование

Ручное тестирование пользовательского интерфейса проводится тестировщиком-оператором, который руководствуется в своей работе описанием тестовых примеров в виде набора сценариев. Каждый сценарий включает в себя перечисление последовательности действий, которые должен выполнить оператор, и описание важных для анализа результатов тестирования ответных реакций системы, отражаемых в пользовательском интерфейсе. Типичная форма записи сценария для проведения ручного тестирования - таблица, в которой в одной колонке описаны действия (шаги сценария), в другой - ожидаемая реакция системы, а третья предназначена для записи того, совпала ли ожидаемая реакция системы с реальной и перечисления несовпадений.

Ручное тестирование пользовательского интерфейса удобно тем, что контроль корректности интерфейса проводится человеком, т.е. основным "потребителем" данной части программной системы. К тому же при чисто косметических изменениях в интерфейсах системы, не отраженных в требованиях (например, при перемещении кнопок управления на 10 пикселей влево), анализ успешности прохождения теста будет выполняться не по формальным признакам, а согласно человеческому восприятию.

При этом ручное тестирование имеет и существенный недостаток - для его проведения требуются значительные человеческие и временные ресурсы. Особенно сильно этот недостаток проявляется при проведении регрессионного тестирования и вообще любого повторного тестирования - на каждой итерации повторного тестирования пользовательского интерфейса требуется участие тестировщика-оператора. В связи с этим в последнее десятилетие получили распространение средства автоматизации тестирования пользовательского интерфейса, снижающие нагрузку на тестировщика-оператора.

ЗАКЛЮЧЕНИЕ

Сегодня тестирование является неотъемлемой частью процесса производства программных продуктов. Качественное тестирование помогает своевременно выявлять и исправлять ошибки, тем самым уменьшая риски и затраты на разработку программного обеспечения. При автоматизации тестирования скорость и качество тестирования может повыситься, что приведет к еще большему снижению издержек и повышению качества.

В результате данного анализа был изучены терминология, основы тестирования, виды и методы тестирования ПО, функциональное и структурное тестирование, организация тестирования ПО.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. **Гагарина, Л. Г. Технология разработки программного обеспечения : учебное пособие / Л.Г. Гагарина, Е.В. Кокорева, Б.Д. Сидорова-Виснадул ; под ред. Л.Г. Гагариной. — Москва : ФОРУМ : ИНФРА-М, 2022. — 400 с. — (Высшее образование: Бакалавриат). - ISBN 978-5-8199-0707-8. - Текст : электронный. - URL: https://znanium.com/catalog/product/1699927 – Режим доступа: по подписке.**
2. **Дубовой, Н. Д. Основы метрологии, стандартизации и сертификации : учебное пособие / Н. Д. Дубовой, Е. М. Портнов. - Москва : ФОРУМ : ИНФРА-М, 2019. - 256 с. : ил. - (Профессиональное образование). - ISBN 978-5-8199-0338-4. - Текст : электронный. - URL: https://znanium.com/catalog/product/991962 – Режим доступа: по подписке.**
3. **Исаченко, О. В. Программное обеспечение компьютерных сетей : учебное пособие / О.В. Исаченко. — 2-е изд., испр. и доп. — Москва : ИНФРА-М, 2022. — 158 с. — (Среднее профессиональное образование). - ISBN 978-5-16-015447-3. - Текст : электронный. - URL: https://znanium.com/catalog/product/1860121 – Режим доступа: по подписке.**
4. **Мартишин, С. А. Основы теории надежности информационных систем : учебное пособие / С. А. Мартишин, В. Л. Симонов, М. В. Храпченко. — Москва : ФОРУМ : ИНФРА-М, 2020. — 255 с. — (Высшее образование: Бакалавриат). - ISBN 978-5-8199-0757-3. - Текст : электронный. - URL: https://znanium.com/catalog/product/1062374 – Режим доступа: по подписке.**
5. **Морозова, Ю. В. Тестирование программного обеспечения : учебное пособие / Ю. В. Морозова. - Томск : Эль-Контент, 2019. - 120 с. - ISBN 978-5-4332-0279-5. - Текст : электронный. - URL: https://znanium.com/catalog/product/1845910 – Режим доступа: по подписке.**
6. **Попова, Ю. Б. Тестирование и отладка программного обеспечения: пособие / Ю. Б. Попова. – Минск : БНТУ, 2020. – 66 с.**
7. **Федорова, Г. Н. Разработка, внедрение и адаптация программного обеспечения отраслевой направленности : учебное пособие / Г.Н. Федорова. — Москва : КУРС : ИНФРА-М, 2022. — 336 с. — (Среднее профессиональное образование). - ISBN 978-5-906818-41-6. - Текст : электронный. - URL: https://znanium.com/catalog/product/1858587 – Режим доступа: по подписке.**
8. **Царев, Р. Ю. Основы распределенной обработки информации: Учебное пособие / Царёв Р.Ю., Прокопенко А.В., Никифоров А.Ю. - Краснояр.:СФУ, 2015. - 180 с.: ISBN 978-5-7638-3386-7. - Текст : электронный. - URL: https://znanium.com/catalog/product/967646 – Режим доступа: по подписке.**
9. **Черников, Б. В. Управление качеством программного обеспечения : учебник / Б.В. Черников. — Москва : ФОРУМ : ИНФРА-М, 2022. — 240 с. — (Среднее профессиональное образование). - ISBN 978-5-8199-0902-7. - Текст : электронный. - URL: https://znanium.com/catalog/product/1850732 – Режим доступа: по подписке.**

10. **Тестирование, оценка програмного обеспечения. Учебно-методическое пособие по дисциплине «Тестирование, оценка программного обеспечения» для студ. всех форм обуч. спец. 1-58-01-01 Инженерно-психологическое обеспечение информационных технологий и направл. спец. 1-40 05 01-09 Информационные системы и технологии (в обеспечении промышленной безопасности), 1-40 05 01-10 Информационные системы и технологии (в бизнес-менеджменте) / М. М. Меженная, Т. В. Гордейчук, М. М. Борисик, О. С. Медведев, И.Ф. Киринович. – Минск: БГУИР, 2016. – 64 с. : ил.**

11. **Портал специалистов по тестированию и обеспечению качества ПО, библиотека статей по тестированию ПО [Электронный ресурс]. URL: http://softwaretesting.ru/library. Яз. русский.**