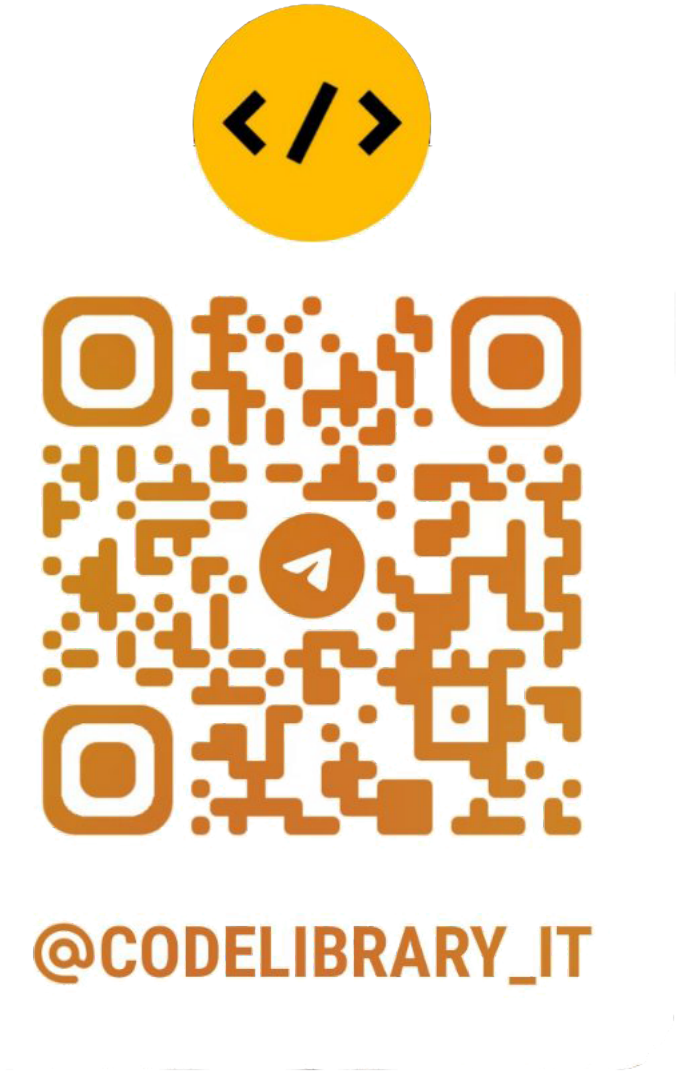
ЗАХАРОВ ВИКТОР

# ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ. ОСНОВЫ

# Виктор Захаров
# Тестирование программного обеспечения. Основы



Захаров Виктор

# Тестирование программного обеспечения. Основы

***Настольная книга специалиста по тестированию***

Первое издание

Бердск, 2024

# Аннотация

Погрузитесь в увлекательный мир тестирования программного обеспечения вместе с книгой, которая является настоящим концентратом чистейших знаний для новичков и профессионалов! Автор делится секретами мастерства, подробно рассказывая о более 15 видах тестирования и более 20 методах проектирования тестов. И это только вершина айсберга знаний, изложенных в книге. Вы будете поражены глубиной информации и открытием знаний, о которых даже не догадывались. Книга насыщена ценнейшими советами, основанными на практическом опыте. Многочисленные примеры помогут быстрее освоить представленный в книге материал. Вооружившись знаниями из этой книги, вы будете уверенно разбираться в нюансах тестирования программного обеспечения и с лёгкостью применять знания на практике! Книга может по праву считаться настольной книгой специалиста по тестированию.

# Предисловие

На момент написания данной книги я работаю в одной из крупнейших компаний на должности руководителя управления, отвечающего за качество программного обеспечения. За плечами у меня 17 лет опыта в области информационных технологий, из них 13 лет я руковожу подразделениями тестирования.

В процессе работы регулярно возникала необходимость подбора новых сотрудников. Проводя собеседования, я видел низкую квалификацию кандидатов и большие провалы в знаниях и навыках тестирования. На должность специалистов приходят плохо подготовленные кадры!

Многие проходят курсы по тестированию, которые в наше время не создаёт только ленивый. По заверениям многих учебных центров их обучающие курсы ведут сертифицированные специалисты (сертификат ISTQB[1]). Однако огромному их количеству рано учить, им надо учиться. Это я понял, когда знакомился с программами, материалами и лекциями преподаваемых курсов. Я был разочарован качеством!

Тогда я принял решение создать учебную программу, где все имеющиеся у меня знания будут чётко структурированы, где не окажется противоречий, где есть большой практический блок, способный помочь в обучении собственных сотрудников. После длительной кропотливой работы был разработан курс для начинающих специалистов по тестированию, содержащий фундаментальные основы по тестированию.

Создавался он около года: писались материалы, проводилась сверка со стандартами, разрабатывалось программное обеспечение для практической работы и так далее. В подготовке мне помогала супруга. Через год курс был готов. Новые сотрудники, приходящие в подразделение, начали обучаться, повышая квалификацию. Результат не заставил себя долго ждать.

Курс представлял собой концентрат чистейших знаний по тестированию. В ходе разработки было подготовлено и переработано множество материалов, и в процессе обучения сотрудников он постоянно совершенствовался. Мне пришла в голову мысль, что все

имеющиеся у меня на руках и в голове знания и опыт необходимо передать всем желающим. Настал момент написания книги.

Сейчас вы читаете ту самую книгу, которая поможет вам постичь фундаментальные основы тестирования программного обеспечения. Написана она простым и доступным языком. Материал будет понятен даже неискушённому читателю. Книга интересна и полезна как новичкам, так и опытным специалистам – каждый найдёт для себя необходимые знания и советы!

Располагайтесь удобнее. Мы с вами начинаем постигать таинственный мир тестирования программного обеспечения. Вас ждут новые открытия и знания. В добрый путь.

# История тестирования

История тестирования компьютерных программ началась в 1950-х годах, когда впервые появились компьютеры. В то время программисты[2] самостоятельно проверяли свои программы на работоспособность, чтобы убедиться, что они работают правильно. В эти годы появилось одно из определений: тестирование – это процесс проверки программы с целью демонстрации её правильной работы.

Первые серьёзные программы создавались для научных исследований и для нужд министерств обороны. Требовалась чёткая и бесперебойная работа, а также отсутствие ошибок. В связи с этим процесс решили формализовать и стандартизировать. Проверка работоспособности программ проводилась формализовано с фиксированием всех изучаемых данных и полученных результатов.

В 1960-е годы люди стремились охватить программы полностью, то есть проверять все возможные передаваемые программе данные и все варианты выполнения программ. К примеру, программа может складывать большие числа, и вместо проверки сложения нескольких чисел, пробовали все возможные варианты без исключения: 1 + 1, 1 + 2, 1 + 3… 2 + 10, 2 + 11… 1259 + 15, 1259 + 16… и так далее.

Миллионы проверок! Это оказалось нереально, поскольку существует много данных, которые необходимо вводить или передавать в программу, много вариантов обработки передаваемых данных, и на их проверку уйдут годы. Также в документах, где описано, как создавать программу, присутствовало большое количество ошибок, которые трудно было найти. Поэтому метод полной проверки отклонили и признали неработоспособным. Появился один из принципов тестирования – исчерпывающее тестирование невозможно.

До 1970-х годов проверка программы означала демонстрацию её правильной работы. Но это занимало много времени и не давало полной информации о качестве программы. Подход оказался неэффективным. В 1970-х годах произошли изменения: вместо демонстрации правильной работы программы использовали поиск существующих в ней ошибок. Удачной проверкой считалась та, с

помощью которой обнаруживали ранее неизвестную ошибку. В эти годы появилось очередное определение тестирования – это процесс проверки программы с целью нахождения ошибок.

В 1980-е годы начинается формирование концепции тестирования, дошедшей до наших дней. Проверка программ включила в себя такое понятие как «предупреждение ошибок».

*Предупреждение ошибок – это информирование о возможной ошибке до того, как она станет серьёзной проблемой.*

Это помогает исправить ошибку прежде, чем программа попадёт к пользователям и станет критической. Одним из эффективных способов предотвращения ошибок является проектирование тестов. То есть перед тем, как проверять программу, необходимо продумать, какие проверки мы будем делать, далее зафиксировать их списком и только после этого проверять работоспособность программы по составленному списку проверок.

В эти же годы стало понятно: необходимо сформировать принципы и подходы тестирования. Это позволит решать конкретные задачи и формализует процесс, чтобы он стал управляемым, дающим возможность контролировать качество программ на протяжении всех этапов их создания. В дальнейшем мы познакомимся с каждым этапом.

Тестирование, существовавшее до начала 1980-х годов, проверяло только собранные и работающие программы. Это означает, что проверялась программа, которую можно было запустить и работать с ней. Однако с течением времени специалисты по тестированию начали принимать участие на всех этапах её создания, что позволяло выявлять проблемы заранее и уменьшать сроки и стоимость разработки. Тестирование стало больше чем просто поиск ошибок или демонстрация правильной работы программы.

В 1990-е годы понятие «тестирование» означало не только проверку программы, но и планирование процесса проверки, создание, поддержку и выполнение тестов, а также окружений, в которых работала и проверялась программа. Тогда тестирование стало важной составляющей для поддержания и улучшения качества программ. Это дало развитие инструментам, используемым для поддержки процессов тестирования: многофункциональные системы и инструменты для

автоматизации тестирования; инструменты формирования отчётов; системы написания и хранения тестов и проведения тестирования; системы для проверки работы программ под высокой нагрузкой.

В настоящее время тестирование является важным инструментом для гарантии качества программ и уверенности в их правильной работе. Оно продолжает развиваться и не стоит на месте. Инновации в данной области позволяют постоянно улучшать качество и надёжность программ. Сейчас специалисты по тестированию играют важную роль в процессе разработки программ.

# Кто он – специалист по тестированию

Мы пользуемся множеством различных программ каждый день, даже не задумываясь об этом. Например, покупая что-то в магазине, используются кассовые аппараты; снимая наличные деньги, мы используем банкоматы; посещая различные сайты, мы используем компьютеры, все это работает благодаря специальным программам.

Кто-то разрабатывает их, а кто-то проверяет работоспособность – тестирует. Специалистов, тестирующих программы, называют «специалистами по тестированию» или «тестировщиками». Есть профессиональный стандарт, где указано чёткое наименование профессии – «специалист по тестированию», поэтому в данной книге будем использовать термин оттуда.

Те, кто никогда не сталкивались с тестированием, иногда заблуждаются, думая, будто специалисты по тестированию – это люди, которые в процессе работы бессмысленно нажимают различные кнопки в проверяемой программе. Это не так. Тогда кто же он – этот специалист по тестированию?

Этот человек проверяет программы и в процессе проверки проводит их глубокий анализ и исследование. Он имеет аналитический склад ума, постоянно развивает навыки логически мыслить и анализировать большой объём информации, прежде чем решать поставленные перед ним задачи. Благодаря гибкости ума он моделирует различные ситуации, в которых программа может работать. И если она работает не так, как должна, а её ожидаемое поведение обязательно закреплено в специальной документации, специалист по тестированию должен сообщить программистам об ошибке. Специалисту по тестированию в этот момент понадобится важный для его профессии навык – умение чётко формулировать мысли и грамотно доносить информацию до других. Зачем нужен этот навык? Если программист не поймёт, что именно работает неправильно, он не сможет исправить ошибку или потратит на поиск непозволительно много времени.

Однако на этом работа специалиста по тестированию не заканчивается. Он следит за тем, чтобы программу или оборудование, в которое встроена программа, было удобно использовать. Например,

если кто-то разместит монитор банкомата на уровне колен человека, пользоваться им будет затруднительно. Тестировщик должен сообщать о подобных недочётах тем, кто отвечает за проектирование удобства использования, чтобы исправить проблему до того, как такой банкомат перешёл в массовое производство. Аналогичная ситуация и с программами. Если с интерфейсом существуют значительные проблемы в удобстве её использования, специалист по тестированию должен сообщать об этом.

Он хорошо знает, как работает программа, которую проверяет. Порой даже лучше программистов, которые её создают. Если программа очень большая, команда программистов делится на группы. Каждая знает только ту часть, за разработку которой отвечает, а тестировщикам приходится изучать и знать функционирование всей программы целиком, чтобы проверять её работу комплексно. По опыту могу сказать: программисты ценят квалифицированных специалистов по тестированию, и когда у новых программистов возникают вопросы связанные с работой программы, они идут к ним за помощью. Встречались ситуации, когда новые программисты или аналитики[3], изучая работу новой для них программы, обучались у опытных тестировщиков.

Надо помнить, что на плечи специалистов по тестированию ложится ответственность за качество работы программ. И от качества их работы зависит очень многое: от размера прибыли компании до жизни людей. Практически каждая компания, связанная с разработкой программ, имеет в своём штате тестировщиков.

# Зачем нужны специалисты по тестированию

Никто не идеален, все мы допускаем ошибки. Они могут быть как незначительными, так и очень серьёзными, некоторые имеют разрушительные последствия. Поэтому, когда мы создаём какой-либо продукт, в том числе компьютерные программы, необходима проверка, чтобы его использование было безопасным и эффективным. В этот момент и нужны специалисты по тестированию.

У многих возникает вполне закономерный вопрос: «Зачем привлекать специалистов по тестированию для проверки программ, если это могут делать программисты, разрабатывавшие её?». В некоторых компаниях, где нет специалистов по тестированию, так и происходит. Программисты сами и разрабатывают, и тестируют. Однако не всё так просто.

Во-первых, если программист сам занимается тестированием своей программы, у него будет меньше времени, чтобы фокусироваться на прямых обязанностях: разработке программы и устранении ошибок. Таким образом, время, которое уходит на разработку и тестирование в целом, значительно больше, чем если тестирование выполняется специалистами по тестированию. Конечно же программист проверяет программу перед тем, как передать её специалистам по тестированию, однако делает это поверхностно, чтобы удостовериться, что логика, которую он реализовал, работает. Если он этого не сделает, и программа, переданная на тестирование, не будет функционировать, специалисты по тестированию вернут её на доработку.

Во-вторых, программист не всегда может предусмотреть все возможные способы использования программы, поскольку его мышление отличается от мышления специалиста по тестированию. Последний задумывается обо всех возможных способах использования программы, в том числе, учитывая и возможные варианты её сломать, так как пользователь программы, если у него что-то не получается, может начать щелкать все по очереди, надеясь, что что-то заработает. Программист обычно думает, как правильно использовать программу, и может не предусматривать случаи, когда программа может

сломаться. В результате, такой подход способен привести к пропуску критических ошибок, которые могут обнаружить конечные пользователи.

В-третьих, специалист по тестированию, используя различные техники, методы и виды тестирования проводит более тщательную проверку, ведь это его профессия, он совершенствует навыки годами.

Отсюда и вывод: специалисты по тестированию нужны везде, где есть программисты. Программисты разрабатывают, а специалисты по тестированию – тестируют.

# Что мы знаем о программах

В предыдущих главах мы упоминали программы, которые разрабатываются и тестируются. В жизни вы слышали термин «программное обеспечение». Кто работает в сфере информационных технологий, могли ещё слышать термин «информационная система». Многие считают, что программа, программное обеспечение и информационная система – это одно и то же. Давайте попробуем открыть завесу тайны и понять, что означают данные термины.

Начнём с «программы». Сначала приведём общепринятое понятие.

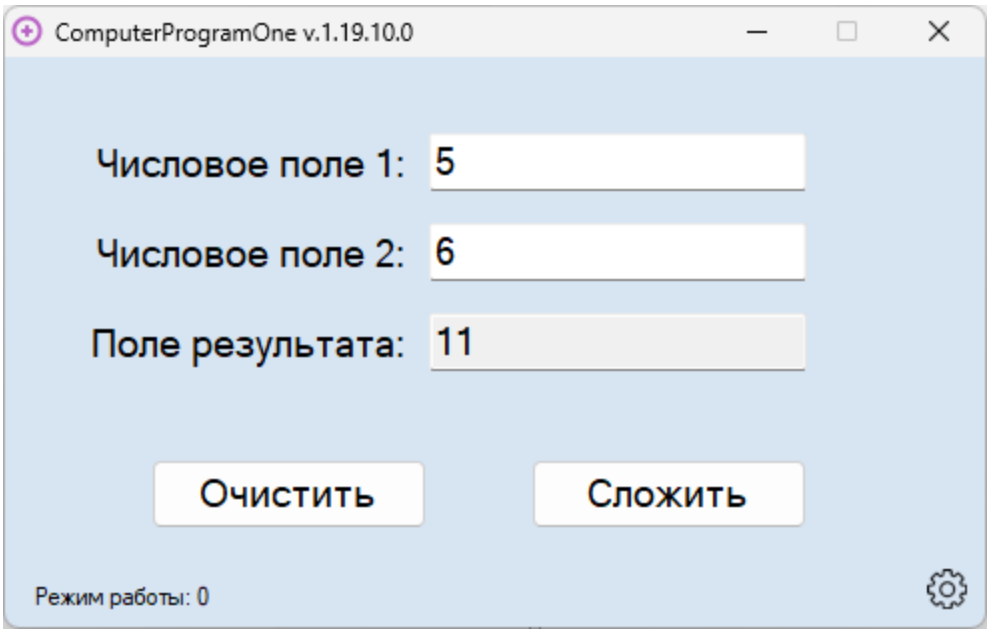
*Программа – это набор инструкций, написанных на определённом языке программирования, которые компьютер может исполнять, для выполнения определённой задачи.*

Простыми словами: программа представляет собой рецепт для компьютера. Когда мы готовим еду, нужен рецепт, чтобы знать, что и как делать. То же самое и с компьютером – ему необходима программа, чтобы выполнить определённую задачу. Рецепт содержит инструкции, как приготовить блюдо. Аналогично и программа содержит программные инструкции в коде. Но в отличие от рецепта, компьютерная программа исполняет инструкции автоматически или при определённом условии. Это значит, что, когда мы запускаем программу на компьютере, она самостоятельно выполняет определённые инструкции, прописанные в ней, или выполняет прописанные инструкции при определённом действии пользователя.

Рассмотрим пример простой программы, где есть две инструкции:

1) При нажатии на кнопку «Сложить» выполнить операцию сложения чисел и вывести результат сложения.

2) При нажатии на кнопку «Очистить» очистить все имеющиеся у программы поля.




Данная программа для сложения чисел будет сопровождать нас на протяжении всей книги, поэтому можете скачать её для ознакомления на сайте автора[4].

Переходим к термину «программное обеспечение».

*Программное обеспечение (ПО) – совокупность программ, используемых для управления компьютером.*

Исходя из определения можно сказать, что программное обеспечение – это набор программ, установленных на компьютере и предоставляющих возможности для выполнения различных задач. Данный набор включает все программы на компьютере или устройстве, необходимые для их работы.

Теперь мы понимаем, что программа и программное обеспечение – это два разных понятия. Программа – часть программного обеспечения, а программное обеспечение – совокупностью всех программ, установленных на компьютере.

Вы часто будете сталкиваться с ситуациями, когда под словами «программное обеспечение» люди подразумевают программу. Так повелось, и на это надо реагировать спокойно.

Теперь нам предстоит понять, что такое «информационная система».

*Информационная система – это комплекс программ и устройств, которые работают вместе для сбора, обработки, хранения, предоставления и передачи информации с целью решения определённых задач.*

Информационные системы могут состоять из программ, компьютеров, сети передачи информации, баз данных[5], устройств ввода-вывода[6] и так далее. Цель информационных систем – облегчение, повышение эффективности и производительности процессов. Они используются для решения широкого спектра задач.

Теперь рассмотрим все три понятия в связке на простом примере. Мы покупаем компьютер. На нём ничего не установлено, и это является оборудованием. Далее устанавливаем операционную систему, которая включает набор различных программ: калькулятор, редактор текста и т. д. Теперь у нас есть компьютер с программным обеспечением.

Мы разрабатываем программу, в которой можно заполнять налоговые отчёты. Она установлена на первом компьютере. На втором устанавливаем базу данных, в которой наша программа хранит все данные включая созданные отчёты. На третий устанавливаем программу, которая получает из базы данных информацию по отчётам и автоматически отправляет их в налоговые органы. Всё описанное является информационной системой, включающей три компьютера, без которых не будут функционировать программы; программы для создания отчётов, их отправки, база данных; программное обеспечение с операционными системами со всеми перечисленными программами; сеть, которой связаны компьютеры и по которой они обмениваются информацией.

Есть ли элемент меньше программы? Есть – программный компонент.

*Программный компонент – это автономный наименьший элемент программы, который создан для выполнения конкретных функций или задач.*

Можно сказать, что программные компоненты – кирпичики, из которых строятся программы. Как и в случае с настоящими кирпичиками, программисты могут использовать различные готовые компоненты в своих программах, чтобы не приходилось писать новый программный код с нуля.

Рассмотрим на примере. Представим, что нам необходимо создать программу для рисования. Вместо того, чтобы писать код для каждой функции программы (например, для создания линий, окружностей, прямоугольников), можно использовать готовые компоненты, созданные не нами, которые уже выполняют эти функции. Это бывает как стандартный набор графических компонентов, предоставляемый операционной системой, так и специализированные компоненты, которые мы можем загрузить из интернета.

Другой пример: создание онлайн-магазина. Вместо того, чтобы писать код для каждой функции, такой как добавление товаров в корзину, оформление заказа, обработка платежей и отправка уведомлений покупателям, мы можем использовать готовые компоненты, в которых уже реализованы данные функции.

В процессе чтения книги вы будете сталкиваться со всеми рассмотренными понятиями: программа, программное обеспечение, информационная система, компонент. Это сделано для того, чтобы вы привыкали к данным понятиям. Они будут использоваться, только если это уместно в определённом контексте.

# Клиент-серверная архитектура программ

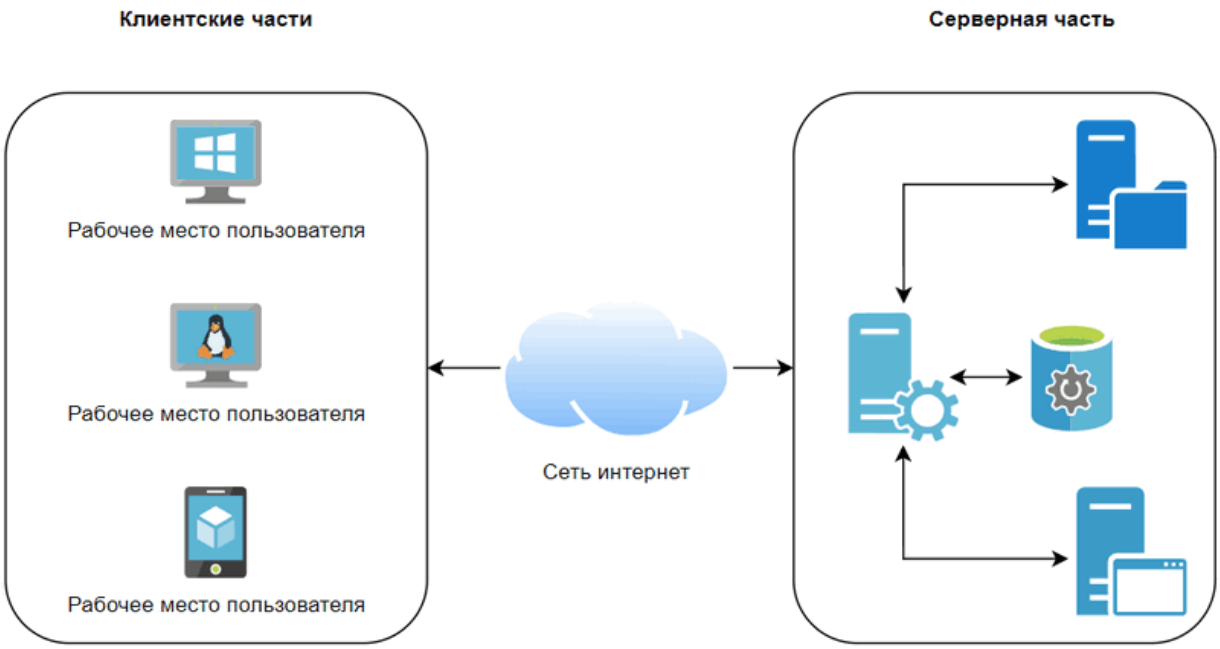
В разделе, описывающем виды тестирования, мы столкнёмся с понятиями клиент-серверной архитектуры. В связи с этим разберём, что это такое. Также данная информация будет вам полезна для общего понимания работы ряда информационных систем.

*Клиент-серверная архитектура – это принцип построения информационных систем и программ, которые работают через интернет или локальную сеть.*

Это как разделение обязанностей между компьютерами. В данной системе есть два типа компьютеров: «клиент» – клиентская часть, «сервер» – серверная часть. Они являются двумя составными частями информационных систем и программ, работающих по сети. Давайте посмотрим на них на примере интернет-магазина.

Клиентская часть – то, что работает у пользователя на компьютере или телефоне. В нашем случае – сайт или программа интернет-магазина, его интерфейс, где отображается каталог товаров, корзина, личный кабинет.

Серверная часть – это программа или набор программ, установленных на удалённом компьютере или компьютерах организации. Там хранятся все данные о товарах, заказах, пользователях. Туда приходят ваши запросы из клиентской части при нажатии на кнопки, а обратно с сервера приходят данные в клиентскую часть для отображения.




Клиентская часть информационной системы – это то, что использует пользователь напрямую, а серверная работает на стороне организации, обеспечивая все основные функции. Таким образом, первая ориентирована на взаимодействие с пользователем, в то время как вторая заботится о более сложных задачах и обеспечивает функциональность, которую видит пользователь в клиентской части. Вместе они образуют работающую информационную систему.

Программа для сложения чисел, текстовый редактор не имеют клиент-серверной архитектуры и их можно назвать «автономными программами». Им не требуется для работы серверная часть. Интернет-магазины, мобильные программы различных магазинов, сетевые игры – информационные системы с клиент-серверной архитектурой. Для их работы нужна серверная часть.

# Жизненный цикл программы

Как любая сущность в мире, программы появляются на свет, «живут» определённое время и «умирают», и на протяжении этого периода проходят различные стадии своего существования.

Идея. Появление программы начинается с неё. Есть заинтересованное лицо или группа лиц, которые в определённый период времени понимают: нужна программа, которая будет помогать им решать определённые задачи. В момент появления идеи и принятия решения о её создании начинается «жизнь» любой программы.

Анализ. На данном этапе определяются нужды и ожидания пользователей будущей программы и собираются их пожелания. По мере того, как выясняются потребности, они преобразуются в требования, которым должна удовлетворять новая программа. Они чётко, структурировано и формализовано фиксируются в документе. О требованиях поговорим в следующих главах.

Проектирование. На этапе анализа определяется, что должна делать будущая программа; определяется, как она будет выполнять свои задачи и какие функции необходимо заложить. Именно на данном этапе определяется структура программы, базы данных, как программа будет взаимодействовать с другими программами, как будет выглядеть интерфейс и прочие моменты.

Разработка. На этом этапе происходит написание кода, создание файлов данных и разработка баз данных.

Тестирование. В этот период проводятся исследования и испытания программы. Данный этап тесно связан с разработкой, так как в процессе разработки идёт постоянное тестирование.

Внедрение. Происходит ввод программы в эксплуатацию и передача её пользователям.

Сопровождение. Программа поддерживается и обновляется, чтобы оставаться актуальной и исправно работающей.

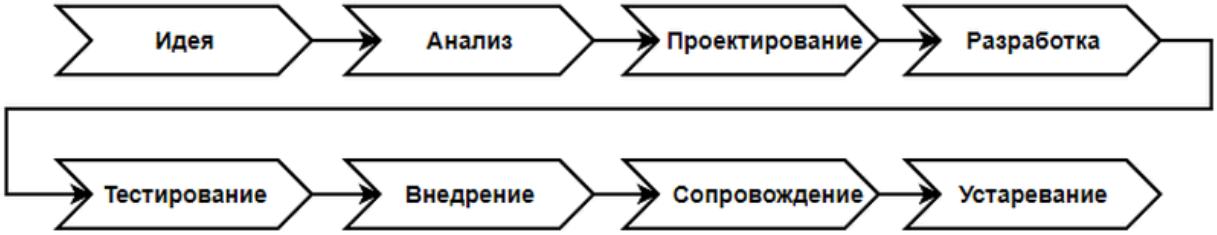
Устаревание. На этом этапе программа выводится из эксплуатации из-за устаревания или замены на более новую версию. Прекращается сопровождение и разработка, завершается «жизнь» программы.

Рассмотрим все описанные этапы на примере строительства. Изначально у человека появляется идея постройки дома. Через определённое время он начинает проводить анализ и задаваться вопросами: «сколько комнат должно быть в доме», «нужна ванна или душевая кабинка», «какой фундамент выбрать», «сколько должно быть этажей», а также он спрашивает о потребностях свою семью. Это и есть период анализа, сбора требований и пожеланий. Организованный человек все пожелания фиксирует на бумаге, и у него появятся зафиксированные требования к дому. После анализа человек идёт к архитекторам, чтобы они разработали ему индивидуальный проект под его требования. Архитекторы начинают проектирование дома, коммуникаций, придомовой территории. Дальше человек получает на руки всю проектную документацию и направляется к строителям, начинается разработка дома – строительство. Затем он принимает дом и проводит тестирование: крутит краны, топает по полу, изучая прочность, проверяет систему отопления, кондиционирования. Если всё удовлетворяет, он въезжает, и этап переезда – это внедрение. Далее уже идёт эксплуатация – период проживания. Если строители дали на дом гарантию или на платной основе готовы постоянно поддерживать его в хорошем состоянии, в случае поломки чего-либо, человек обращается в организацию, строившую дом, и они исправляют проблему, а возможно что-то совершенствуют – это сопровождение. Через много лет дом станет непригодным для проживания – тогда его выведут из эксплуатации и снесут; на этом этапе завершается жизнь дома – этап устаревания.

Теперь возьмём пример операционной системы. В далёкие времена у кого-то появилась **идея** создания операционной системы Windows. Вдохновлённый ею человек начал проводить **анализ** существующих программ на рынке и собирать техническую информацию. Накопив необходимые данные стало понятно – нужно создавать. После началось **проектирование** операционной системы – продумывалось, из каких программ она будет состоять, как программы будут между собой взаимодействовать, как будет выглядеть интерфейс операционной системы. Пройдя этап проектирования, началась **разработка** – написание кода операционной системы. Создав первую версию, программисты с коллегами начали **тестирование** работы операционной системы и её компонентов. Убедившись, что

операционная система работает, провели **внедрение** – передали пользователям и научили их работать с операционной системой. Люди пользовались операционной системой, обнаруживали ошибки и сообщали создателям. Те, в свою очередь их исправляли и обновляли операционную систему. Это этап **сопровождения**. Создатели выпустили новую версию, а затем ещё одну. Первую версию вывели из эксплуатации, так как произошло **устаревание,** её больше не поддерживали. Так и закончилась жизнь первой версии операционной системы. Это применимо ко всем существующим программам.

Описанные стадии жизни программы и есть её жизненный цикл, состоящий из этапов:




При этом, пока не произошло вывода из эксплуатации (устаревания), процесс цикличен, т. е. все этапы кроме «Устаревания» многократно повторяются.

*Жизненный цикл программы – период времени, который начинается с момента принятия решения о необходимости создания программы и заканчивается в момент её полного изъятия из эксплуатации.*

# Требования к программе

Мы с вами получили представление о жизненном цикле программ. В процессе его рассмотрения мы затронули такое понятие как «требования». Рассмотрим, что они из себя представляют, для чего нужны и откуда берутся.

Чтобы разработать программу, необходимо понять, что пользователь хочет получить от неё, какие нужды хочет закрыть. На основании полученной информации аналитики начинают подробнее разбирать потребности пользователей и фиксировать, как должна работать программа, что должна делать, с какими программами должна взаимодействовать, а также прописывают многие другие аспекты на основании постоянного общения с пользователями и выявления их потребностей и желаний. Все данные фиксируются в виде требований к программе. Из озвученного выведем определение, что такое «требование к программе».

*Требование к программе – это структурированное описание определённых свойств программы (поведения, внешнего вида, качества и т. д.), которые должны отвечать потребностям пользователя. Требования могут представляться в виде документа или набора документов.*

К примеру, в ходе общения с пользователями аналитик выясняет, что кнопка в программе должна быть зелёная – это требование. Программа должна запускаться за 5 секунд – это требование. Итого мы уже имеем два требования.

Они фиксируются в документах, которые имеют своё название – «спецификация».

*Спецификация – документ, исчерпывающе, однозначно и доступно описывающий требования, дизайн, поведение и иные характеристики программы, которую требуется разработать.*

Спецификацию ещё могут называть «спецификация требований». Ниже приведена часть спецификации с таблицей требований к программе:

| Номер | Название | Требование |
|---|---|---|
| ФТ-1.1 | Появление формы | Форма появляется при запуске программы.<br>Форма появляется по центру экрана. |
| ФТ-1.2 | Масштабирование формы | Размер формы не изменяется (фиксированный). |
| ФТ-1.3 | Перемещение формы | Форма перемещается по экрану, если навести курсор мыши на заголовок формы, зажать левую кнопку мыши и произвести перемещение формы. |
| ФТ-1.4 | Поле ввода первого числа | В поле можно вводить только положительные целые числа.<br>Максимальное количество вводимых и отображаемых символов 4 (четыре).<br>В поле можно вводить информацию с клавиатуры.<br>В поле можно вставлять данные из буфера обмена.<br>Из поля можно копировать данные в буфер обмена. |
| ФТ-1.5 | Поле ввода второго числа | В поле можно вводить только положительные целые числа.<br>Максимальное количество вводимых и отображаемых символов 4 (четыре).<br>В поле можно вводить информацию с клавиатуры.<br>В поле можно вставлять данные из буфера обмена.<br>Из поля можно копировать данные в буфер обмена. |

Спецификации, которые составляют аналитики, от компании к компании отличаются, нет чёткого шаблона, которого придерживаются все аналитики мира.

Она может содержать функциональные и нефункциональные требования – это два основных вида требований к программному обеспечению. В чём их отличия?

Функциональные требования – описывают функциональность, предоставляемую программным обеспечением. Они определяют, что должна делать программа. Сюда входят: возможность ввода и редактирования данных в поле программы; возможность очистки полей для ввода данных; возможность сложения чисел и т. д. На рисунке выше как раз отражена спецификация с функциональными требованиями.

Нефункциональные требования – описывают свойства программы, но не её поведение. Они определяют, как должна работать программа. Сюда относятся: удобство использования (к примеру, расположение кнопки или её название); производительность; безопасность и т. д. Ниже приведена спецификация с нефункциональными требованиями:

| Номер | Название | Требование |
|---|---|---|
| НТ-3.1 | Название файла | Файл логирования имеет имя «ComputerProgramOne.log». |
| НТ-3.2 | Путь к файлу | Файл логирования располагается рядом с исполняемым файлом программы. |
| НТ-3.3 | Формат файла | Файл имеет текстовый формат и хранит данные в соответствующем формате |

Основное отличие заключается в том, что функциональные описывают функционал, нефункциональные – свойства и качества системы.

В ходе своей профессиональной деятельности специалисты по тестированию часто сталкиваются с ситуацией, когда в организации отсутствуют чётко задокументированные требования к программе или они являются минимальными и не полностью информативными. В таких случаях специалисты по тестированию вынуждены активно взаимодействовать с аналитиками и программистами, чтобы получить необходимую информацию. Иногда им даже приходится самостоятельно проводить исследование программ, чтобы полноценно провести тестирование. Это требует от специалиста по тестированию гибкости и способности самостоятельно разбираться в функциональности программ.

# Дефект, ошибка и отказ

Собрав информацию о потребностях пользователей и проработав требования, все данные передаются программистам. Они, изучив их, начинают разрабатывать программу, реализовывая то, что зафиксировано в требованиях. В процессе написания кода программисты могут допустить ошибку и прописать неверные данные в коде. К примеру, разрабатывая программу сложения чисел программист в строке, где должно происходить сложение, вместо знака «плюс» прописал знак «минус». Так была допущена ошибка:

```
private void МетодСложенияЧисел()
{
    int _первоеЧисло = Int32.Parse(полеВводаПервогоЧисла.Text);
    int _второеЧисло = Int32.Parse(полеВводаВторогоЧисла.Text);
    int _суммаЧисел = _первоеЧисло - _второеЧисло;
    полеОтображенияРезультата.Text = _суммаЧисел.ToString();
}
```

*Ошибка – действие человека, которое приводит к неправильному результату.*

После написания кода программы её специальными инструментами «собирают», чтобы она стала полноценной работающей программой, а не набором файлов со строками кода. Собрали, внедрили и начали процесс эксплуатации. В момент использования программы пользователи столкнутся с ситуацией, когда программа работает не так, как ожидалось, когда реальный результат не равен ожидаемому.

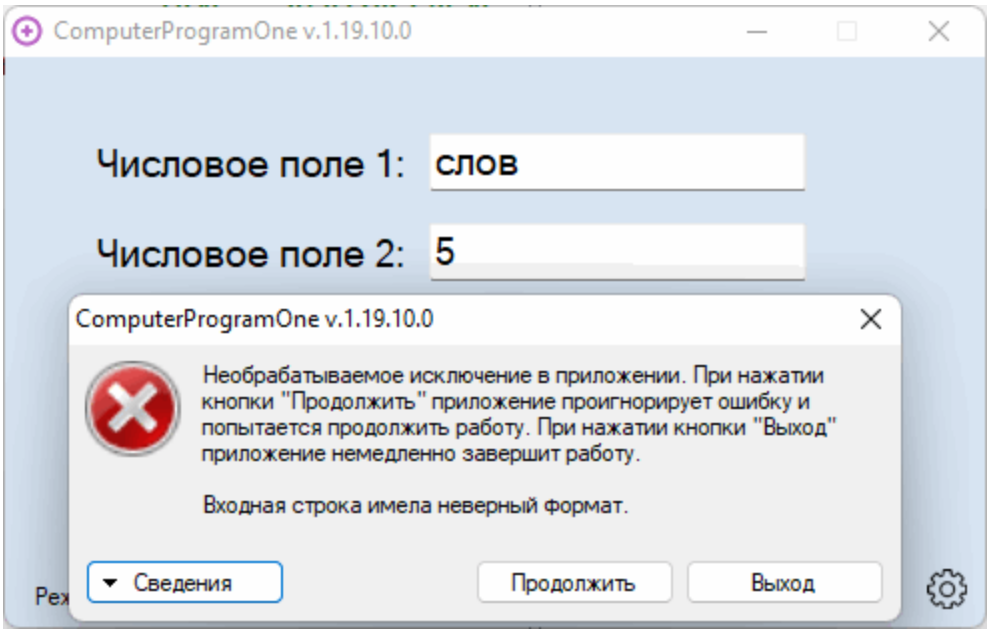
Если рассматривать приведённый пример с программой сложения чисел, пользователь ожидает, что, введя два числа и нажав на кнопку сложения, получит результат сложения. Однако его ждёт разочарование, так как программа будет вычитать и выводить результат вычитания, а это означает, что она не сможет выполнять требуемую от неё функцию. Почему? Потому что на этапе разработки

допустили ошибку, и пользователь столкнулся с последствиями ранее допущенной ошибки – с дефектом в программе.

*Дефект – изъян в программе, который может привести к невозможности выполнить требуемую функцию.*

Чтобы дефекты не попадали к пользователям, они должны выявляться специалистами по тестированию до передачи программы в эксплуатацию.

В ходе работы программы дефекты могут привести к отказу. Пример: программист, создавая программу, не добавил в код проверку, запрещающую программе складывать текст с числами. В этом случае появится дефект, который приведёт к отказу программы, и она попытается аварийно завершить свою работу:




*Отказ – нарушение работоспособности программы, при котором она перестаёт выполнять целиком или частично свои функции.*

В итоге получаем следующую последовательность: допущенная программистом в коде ошибка приводит к появлению в программе дефекта, и при работе он может повлечь отказ программы. В свою очередь отказ может привести к большим финансовым потерям компании или к катастрофе, если речь идёт об атомных реакторах или самолётах. Всё зависит от информационной системы, в которой произошёл отказ.

Дефекты могут появляться не только из-за допущенных в коде программ ошибок, но и из-за ошибок в настройках. К примеру, у нас есть требования, что в интернет-магазине заказы могу оплачивать все категории покупателей. Настраивая интернет-магазин, специалист по невнимательности указал, что оплачивать заказы могут только пользователи, которые не вошли в систему (анонимные пользователи). В этом случае не смогут оплачивать заказы те, кто вошли в систему (авторизованные пользователи). В итоге допущенная в настройках ошибка привела к появлению дефекта (невозможно выполнить требуемую функцию) и отказу системы (программа перестаёт выполнять свои функции) при оплате заказов авторизованными пользователями.

# Критичность и приоритет

Мы определили следующее: чтобы дефекты не попадали к пользователям, они должны выявляться специалистами по тестированию до передачи программы в эксплуатацию. Если на ранних этапах дефекты не обнаружатся и попадут с программой к пользователям, то они будут оказывать влияние на работу самих пользователей. К примеру, бухгалтер на неработающем программном калькуляторе не может складывать числа, а это влияет на его труд. Он работает медленно, так как ему надо просчитать множество значений столбиком на листочке. Если бухгалтер вовремя не заполнит и не отправит отчёт в контролирующий орган, тот выставит штраф организации за несвоевременное предоставление отчётов, и организация понесёт финансовые потери.

Ещё пример. Разрабатывая калькулятор, допустили ошибку и вместо знака «плюс» на кнопке отображается буква «П». Бухгалтер, немного изучив программу, понял: по этой кнопке происходит сложение чисел, вовремя заполнил отчёты и сдал их в контролирующий органы. Во время работы он чувствовал неудобство, поскольку приходилось постоянно помнить, что означает буква «П» на кнопке калькулятора, но это не оказало критичного влияния на его работу и на организацию в целом.

Мы видим, что дефекты оказывают разное влияние на программу и пользователей. Одни дефекты незначительны и не доставляют особых проблем, а другие полностью блокируют работу. Всё зависит от степени критичности дефекта – её ещё называют «серьёзностью дефекта».

*Критичность – важность воздействия конкретного дефекта на функционирование и возможность использования программы.*

У дефектов есть пять уровней критичности:
– блокирующий;
– критический;
– значительный;

– незначительный;
– тривиальный.

*Блокирующий – дефект полностью блокирует выполнение функционала программы, и нет никакого способа его обойти.*

Рассмотрим блокирующий дефект на примере программы для сложения чисел. В ней есть кнопка «Сложить», однако она заблокирована, и введя числа для сложения мы не можем их никак сложить. У нас нет возможности обойти этот дефект и каким-либо образом запустить функцию сложения в программе.

*Критический – дефект блокирует выполнение функционала программы, но есть альтернативный путь для его обхода.*

Рассмотрим критический дефект на примере. В программе есть кнопка «Сложить», однако она заблокирована, и введя числа для сложения мы не можем их сложить с помощью данной кнопки. Но у нас есть возможность обойти данный дефект: необходимо после ввода чисел установить курсор мыши на любое поле для ввода числа и нажать кнопку «Enter» на клавиатуре, и программа произведёт сложение.

*Значительный – дефект, указывающий на некорректную работу функционала программы. Проявляется не тем, что функция не работает, а тем, что она работает неправильно.*

Рассмотрим значительный дефект на том же примере программы для сложения чисел. Кнопка «Сложить» работает: мы вводим числа в поля для ввода и нажимаем на неё. Программа складывает числа и выводит результат сложения, однако при этом программа перестаёт отвечать на действия пользователя на определённое время (зависает). Через минуту программа снова начинает отвечать на действия пользователя. И так происходит после каждого сложения.

*Незначительный – дефект, не относящийся к функциональности программы, очевидная проблема пользовательского интерфейса.*

Пример незначительного дефекта. У программы есть кнопка, при нажатии на которую в программе очищаются все поля. В требованиях прописано, что кнопка должна именоваться «Очистить». При разработке программы указали некорректную надпись «Пусто» у кнопки. Она продолжает выполнять свою функцию, но название неверное – не соответствует требованиям.

Незначительная критичность указывается у тех дефектов, которые относятся к удобству использования программы или к интерфейсу программы.

*Тривиальный – дефект, не затрагивающий функциональность программы, а также оказывающий минимальное влияние на общее качество работы программы.*

Обычно тривиальные дефекты – это грамматические ошибки в интерфейсе программы или в сопроводительной документации к программе, а также дефекты сторонних библиотек или сервисов, не относящихся к самой программе. Часто они трудно отличимы от дефектов незначительной критичности.

Пример тривиального дефекта. Есть библиотека[7] стороннего программиста, в которой прописано: если хочешь получить картинку с природой, то запроси картинку, передав библиотеке слово «природа», картинку с небом – «небо», и так далее. Разрабатывая программу сложения чисел мы решили, что будет замечательно, если интерфейс (фон) программы сделать не одноцветным, а использовать какое-либо изображение, к примеру – изображение неба. Мы это реализовали. Теперь, когда мы запускаем программу, она подключается к библиотеке стороннего программиста и запрашивает у неё картинку неба, передавая библиотеке слово «небо». Однако библиотека возвращает картинку с природой, и мы видим оформление программы с фоном природы. Это явная ошибка в сторонней библиотеке, а не нашей программы. Но так как наша программа использует в своей работе эту стороннюю библиотеку, мы заводим дефект для нашего программиста, чтобы он исправил дефект, связанный с внешним оформлением нашей программы.

Для корректного определения критичности дефектов специалисты по тестированию должны в деталях знать функциональность тестируемой программы и особенности её работы. Это один из признаков, который характеризует хороших тестировщиков.

Кроме критичности у дефектов есть приоритет. В процессе разработки программы специалисты по тестированию находят множество дефектов и каждому присваивают критичность. Через определённое время набирается несколько десятков дефектов, которые необходимо устранять с помощью правки программного кода программистом. Он готов устранять дефекты, однако не может одновременно исправлять все. В этот момент ответственные специалисты собираются для определения порядка исправления дефектов. То есть занимаются приоритизацией[8]. Она помогает определить, какие дефекты нужно устранить в первую очередь. Когда специалисты приоритизируют дефекты, они присваивают им определённый приоритет.

*Приоритет – степень важности, присваиваемая объекту, которая указывает на очерёдность устранения дефекта или очерёдность выполнения задачи.*

Обратите внимание: приоритет есть не только у дефектов, но и у задач[9] (заявок на изменение). Приоритеты у дефектов сообщают нам, в какой очерёдности необходимо устранять дефекты. Приоритеты у задач сообщают, в какой очерёдности необходимо выполнять задачи. Ещё важный момент: в отличие от критичности, приоритет есть и у дефектов, и у задач. Критичность же есть только у дефектов.

У приоритета существует три уровня:

– высокий;

– средний;

– низкий.

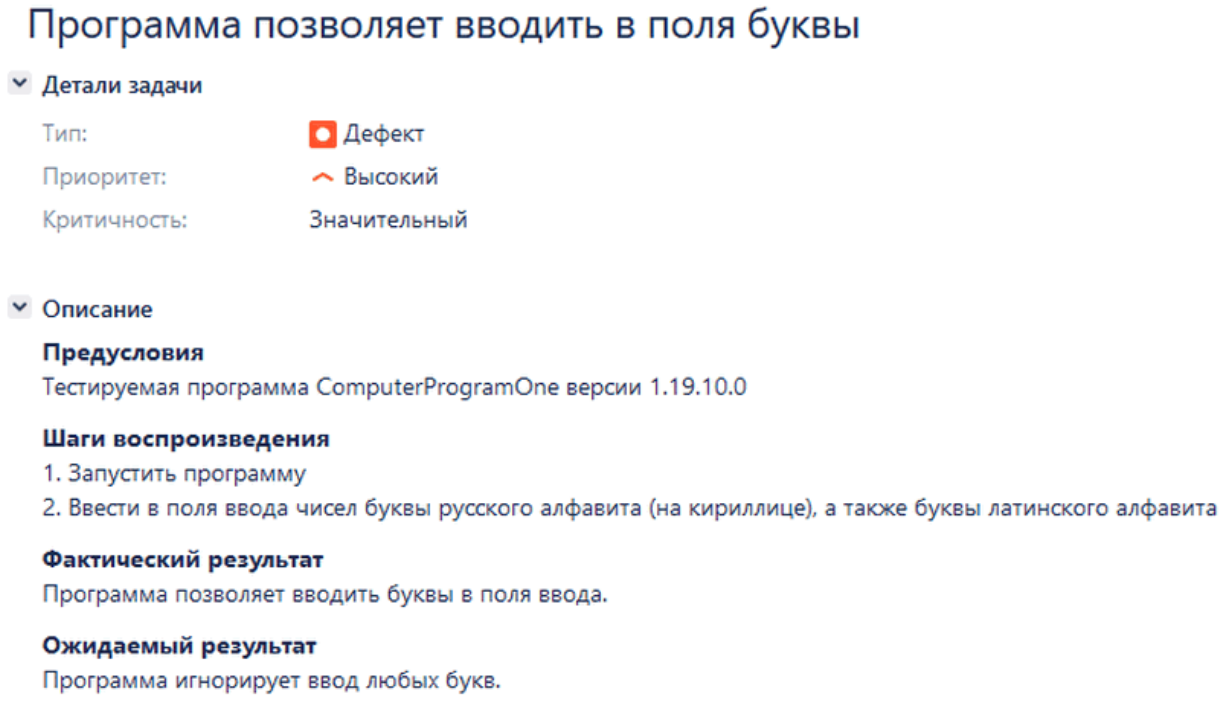
В ряде компаний могут вводить дополнительные уровни.

*Высокий – требуется устранить или выполнить в первую очередь.*

*Средний – требуется устранить или выполнить во вторую очередь, когда нет дефектов и задач с высоким приоритетом.*

*Низкий – требуется устранить или выполнить в последнюю очередь, когда все дефекты и задачи с более высокими приоритетами уже выполнены.*

Указание критичности и приоритета является важной частью процесса разработки и тестирования, так как данные атрибуты однозначно классифицируют дефекты по степени их влияния на систему и очерёдность их исправления:




Не во всех организациях используют одновременно два атрибута, чаще всего только приоритет, который логически объединяет оба атрибута, однако это некорректно.

Зачем нужны оба атрибута? Допустим, есть программа, и в ней нашли два дефекта.

Дефект № 1 – не работает функциональность формирования годового отчёта для бухгалтера, которым он должен будет воспользоваться в начале следующего года. Дефект по критичности «блокирующий».

Дефект № 2 – на главной странице сайта есть логотип, в котором имеется опечатка, буквы «к» и «ё» заменили на «т» и «е», и название

компании «Клён» читается как «Тлен». Дефект по критичности «тривиальный».

Дефект № 2 может сильно подорвать доверие пользователей к компании и продукции, которую они предлагают, что в свою очередь может повлиять на продажи организации и прибыль. Это зависит от компании и её места на рынке. Дефект № 1 блокирующий, но функционалом не будут пользоваться ещё более полугода, так как сейчас, допустим, лето. В связи с вышесказанным, руководители установят дефекту № 2 высокий приоритет на устранение, а дефекту № 1 средний. Первым будет устраняться дефект № 2.

Приоритеты постоянно пересматриваются, так как они зависят от многих факторов. К примеру, если наступит начало года и отчётный период, а дефект № 1 не позволяет бухгалтерам создавать и отправлять отчёты, то приоритеты у дефектов будут изменены. Дефект № 1 получит высокий приоритет на устранение, а дефект № 2 – средний.

# Что такое тестирование

Мы познакомились со специалистами по тестированию и с деятельностью, которой они занимаются. Также изучили, что из себя представляют программы и информационные системы. Настал момент рассмотреть подробнее, что же такое тестирование.

В современном мире программы являются неотъемлемой частью нашей жизни. Они применяются во многих сферах: программы для бизнеса; персональные приложения и игры; программы и микропрограммы в технике (например, телевизоры, стиральные машины). В своей повседневности вы постоянно сталкиваетесь с программами определённого рода.

Мы уже знаем, что программы, работающие некорректно, могут привести к проблемам – потеря времени, денег, деловой репутации, а также могут стать причиной травм и даже смерти. Скорее всего кто-то из вас имел опыт использования программ, которые работали не так, как ожидалось.

Специалисты по тестированию минимизируют риски, выявляя ошибки в программах. Однако это не единственная цель их работы и тестирования в целом. В чём тогда заключается цель тестирования? Она заложена в самом его определении. Рассмотрим ряд существующих неполных и неточных определений тестирования:

– Тестирование ПО – это процесс поиска ошибок в программе. Здесь подразумевается, что мы целенаправленно стараемся найти в программе максимальное количество ошибок.

– Тестирование ПО – это процесс проверки, который гарантирует, что программа работает безошибочно. Здесь имеется в виду, что мы не ищем ошибки, а проводим ряд проверок, чтобы убедиться, что требуемые нам операции отработали без ошибок.

– Тестирование ПО – это процесс, который направлен на подтверждение соответствия программы требованиям, поставленным заказчиком[10]. Здесь подразумевается следующее: мы проверяем, что в программе реализовано то, что попросил заказчик.

Ни одно из этих определений не является полностью верным или точным, так как тестирование – это более сложный и многогранный

процесс, который включает в себя множество активностей, методов и подходов. На основании всего вышесказанного выведем определение.

*Тестирование ПО – это процесс исследования, испытания программы, с целью продемонстрировать заказчикам и/или заинтересованным лицам, что программа соответствует установленным требованиям, а также это процесс проверки соответствия между реальным поведением программы и ожидаемым поведением, и выявление ситуаций, в которых поведение программы является неправильным или нежелательным.*

Данное определение даёт полное понимание, что из себя представляет тестирование программного обеспечения. И ничего сверх того, что сказано в определении, не добавить.

# Цели тестирования

Мы уже знаем и понимает, что такое тестирование, кто его проводит и что тестируется. Но с какой целью всё это осуществляется? Чтобы ответить на этот вопрос, необходимо рассмотреть цели тестирования программного обеспечения.

Цель – это то, ради чего осуществляется какой-либо процесс. Правильно определённая цель позволяет эффективно организовать работу и направить усилия всех участников на достижение желаемого результата. Перечислим цели тестирования, которые применимы к любой разработке.

Цель № 1. Оценка состояния требований, пользовательских историй, проектной документации или написанного кода для выявления расхождений с первоначально запланированными результатами и формирования предложений по их улучшению. Специалисты проводят детальный анализ всех имеющихся артефактов проектирования и разработки программного обеспечения – требований, пользовательских историй, технического задания, дизайн-документации, написанного кода. Цель данного анализа – проверить их корректность и соответствие первоначальному замыслу и плану. Специалисты ищут расхождения между тем, что было описано или запланировано на этапе проектирования, и тем, как это выглядит на данный момент на этапе разработки. Выявленные расхождения анализируются и на их основе разрабатываются конкретные предложения по улучшению.

Цель № 2. Проверка выполнения всех требований, т. е. проверка, что разработанное программное обеспечение полностью отвечает всем изначально заявленным пользовательским нуждам и ожиданиям. При разработке программного обеспечения сначала определяются все необходимые функции и возможности, которыми должна обладать программа. Это фиксируется в виде требований к программе. Затем приступают к непосредственной разработке самого программного кода. Программисты стараются реализовать все заявленные в требованиях функции. Однако чтобы убедиться, что разработанная программа действительно соответствует изначальным требованиям,

проводится его тестирование. В ходе тестирования проверяется, был ли воплощён в жизнь каждый пункт требований. То есть проверяется, реализован ли в программном коде весь функционал, который был описан на этапе определения требований.

Цель № 3. Проверка, что объект тестирования завершён и работает, как ожидают пользователи и заинтересованные лица. В этом случае подтверждаем: программа соответствует ожиданиям и потребностям пользователей, которые будут взаимодействовать с этой программой. Специалисты по тестированию проверяют, работает ли программа так, как этого ожидают пользователи при выполнении тех или иных действий. Проверяется, насколько удобным и понятным является интерфейс программы для пользователей, насколько быстро и эффективно пользователи смогут выполнять свои задачи с помощью этой программы. Также проверяется, учтены ли потребности и ожидания других заинтересованных сторон, таких как администраторы, менеджеры, техподдержка и другие сотрудники организации, в которой будет использоваться данная программа. В результате проверки подтверждается, что разработанная программа полностью соответствует целям её создания и будет удовлетворять нужды всех целевых групп пользователей.

Цель № 4. Создание уверенности в уровне качества объекта тестирования. Проверяется программа, чтобы убедиться, что она соответствует предъявляемым к ней критериям и параметрам качества. Есть, к примеру, зафиксированные параметры качества программы: она должна запускаться и отображать свой интерфейс через 5 секунд после запуска; документ в программе должен сохраняться за 2 секунды; в программе не должно быть критических дефектов и т. д. Специалисты проверяют, что программа соответствует установленным параметрам качества.

Цель № 5. Предотвращение дефектов, т. е. максимальное сокращение количества дефектов путём их предотвращения на ранних этапах создания программного обеспечения. Предотвращение дефектов на этапе сбора и анализа требований. На этом этапе тестировщик проверяет требования на корректность, непротиворечивость и полноту описания. При обнаружении несоответствий или неясностей в требованиях они уточняются ещё на этапе сбора, до начала разработки, что позволяет избежать ошибок на

последующих этапах. Предотвращение дефектов на этапе проектирования. На этом этапе специалист по тестированию проверяет архитектурные решения, дизайн и проектирование программы на соответствие требованиям. Если обнаруживаются расхождения, они устраняются специалистами по проектированию на этапе проектирования, до начала реализации. Предотвращение дефектов на этапе разработки. Здесь специалист по тестированию проверяет программу на соответствие требованиям ещё до встраивания программы в общую экосистему. Программисты, в свою очередь, проверяют исходный программный код на корректность и соответствие их стандартам. Это позволяет выявлять и устранять ошибки на самых ранних этапах жизненного цикла программы.

Цель № 6. Обнаружение отказов и дефектов в ПО. В этом случае мы выявляем уже занесённые в результате разработки ПО дефекты. Это могут быть: дефекты в логике работы программы; ошибки в алгоритмах; дефекты в интерфейсе и взаимодействии с пользователем; дефекты в работе с внешними системами и базами данных; недочёты в документации и описании функционала. Проводится систематический поиск и выявление таких дефектов. Это позволяет устранить их до выхода в продуктивное окружение, чтобы обеспечить его корректную и безаварийную работу. В идеале, все дефекты должны быть найдены и исправлены на этапе тестирования, прежде чем программа будет поставлена пользователю.

Цель № 7. Предоставление заинтересованным лицам достаточной информации, позволяющей им принять обоснованные решения в отношении уровня качества объекта тестирования. В ходе тестирования специалисты по тестированию проверяют соответствие программы требованиям, выявляют дефекты. На основании полученных результатов формируют отчёты, в которых указывают количество и критичность найденных дефектов, степень реализации функциональных и нефункциональных требований и другие показатели качества. Эта информация позволяет заинтересованным сторонам принять обоснованное решение о готовности программы: нужна ли доработка для устранения дефектов, возможен ли запуск в эксплуатацию или требуется ещё один цикл разработки и тестирования.

Цель № 8. Соблюдение договорных, правовых или нормативных требований, или стандартов и/или проверка соответствия объекта тестирования таким требованиям и стандартам. Специалисты по тестированию проверяют, что разрабатываемая программа соответствует государственным или международным стандартам, законодательству и т. д. Пример: программа работает с платёжными системами и должна соответствовать их стандартам и требованиям. При тестировании специалисты должны убедиться в этом, в противном случае данная программа не будет допущена к работе с платёжными системами.

Цели тестирования могут отличаться, в зависимости от этапа жизненного цикла программы, на котором проводится тестирование, а также в зависимости от назначения и типа тестируемого компонента или программы.

# Верификация и валидация

Проводя тестирование, мы в этот момент проводим верификацию и валидацию программы. Кого-то смутят уже сами эти слова и явно не появится желания погружаться в определения непонятных терминов. Однако, придётся в них разобраться, так как специалисты по тестированию сталкиваются с ними постоянно.

*Верификация – подтверждение того, что заданные требования полностью реализованы в программе.*

Подтвердить, что заданные требования полностью реализованы в программе – означает необходимость убедиться, что программисты сделали то, что заказчик зафиксировал в требованиях. Рассмотрим на примере. У заказчика есть требование к программе по сложению чисел. Смотрите таблицу:

| Номер | Название | Требование |
|---|---|---|
| ФТ-1.4 | Поле ввода первого числа | В поле можно вводить только положительные целые числа.<br>Максимальное количество вводимых и отображаемых символов 4 (четыре).<br>В поле можно вводить информацию с клавиатуры.<br>В поле можно вставлять данные из буфера обмена.<br>Из поля можно копировать данные в буфер обмена. |

Верифицируя программу при проведении тестирования, мы должны проверить, что изложенные требования реализованы и всё сделано так, как требовалось. Мы проверяем, что можно вводить только положительные целые числа, что максимальное количество вводимых и отображаемых символов равно четырём, что в поле можно вводить информацию с клавиатуры, вставлять данные из буфера обмена и из поля можно копировать данные в буфер обмена. Если программа одну из проверок не пройдёт, к примеру, из поля нельзя будет копировать данные, то программа не она верификацию, так как не все требования

реализованы. Если же все требования соблюдены, программа пройдёт верификацию. С этим разобрались.

Теперь рассмотрим валидацию.

*Валидация – подтверждение того, что функции программы при её использовании соответствуют требованиям и ожиданиям заказчика и программа способна выполнять задачи, которые от неё ожидают.*

Валидируя программу, мы должны проверить, что реализованная программистами функциональность соответствует требованиям и ожиданиям заказчика. Рассмотрим на примере. У заказчика есть требование к программе по сложению чисел. Смотрите таблицу:

| Номер | Название | Требование |
|---|---|---|
| ФТ-1.6 | Поле отображения результатов | В поле отображаются результаты сложения чисел полей «ввода первого числа» и «ввода второго числа».<br>Поле заблокировано от ввода информации с клавиатуры и от вставки из буфера обмена.<br>Из поля можно копировать данные в буфер обмена. |

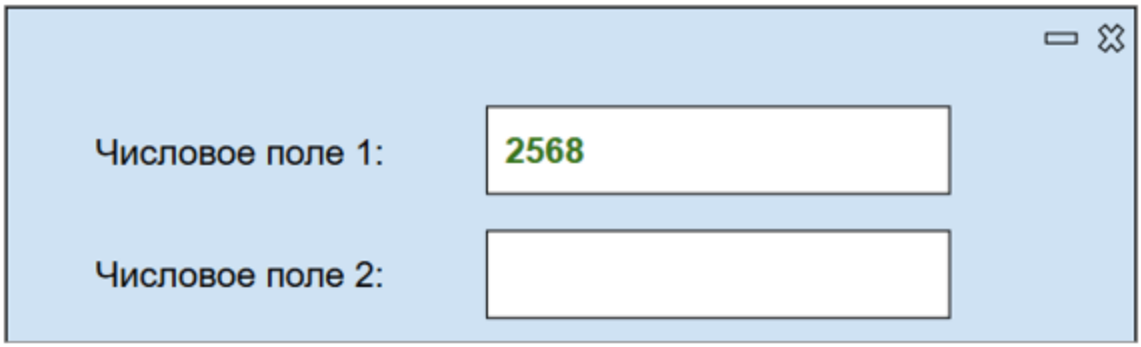
Проводя верификацию, мы убедились, что в поле отображения результатов после сложения чисел отображается результат. Однако через 10 секунд он исчезает. В требованиях не сказано, сколько времени отображать результат сложения, поэтому верификация пройдена. Однако заказчики ожидают, что он не будет исчезать. И логически мы это понимаем. Т. е. программа соответствует требованиям, но не соответствует ожиданиям заказчика, а это означает, что она не прошла валидацию. Если все требования соблюдены, и программа работает, как ожидает заказчик, она пройдёт валидацию.

# Позитивное и негативное тестирование

В программе есть заявленный функционал, который описан в требованиях, и, проверяя данный функционал, мы проводим позитивное тестирование, т. е. убеждаемся, что программа работает так, как описано в требованиях, при использовании допустимых и корректных данных. Нам также необходимо проверить, как программа будет работать, если использовать некорректные данные и непредусмотренные ситуации – это уже негативное тестирование.

*Позитивное тестирование – тестирование, которое определяет, что программа работает так, как ожидалось, и использует для проверок только корректные данные.*

Пример позитивного теста. В программе есть поле ввода, которое может принимать только цифры. При тестировании мы проверяем, что в поле можно вводить цифры. Ввод других данных не проверяется:

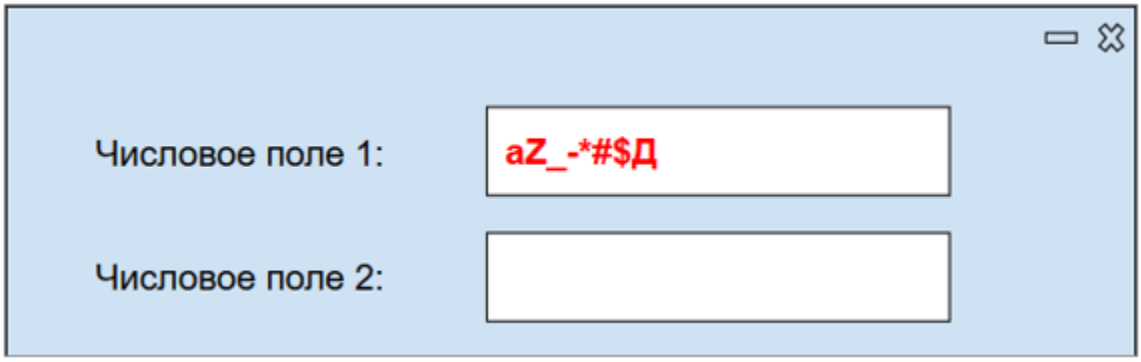



Если в процессе позитивного тестирования мы не сможем ввести цифры в поле ввода, тест не пройден. Если сможем – тест пройден.

*Негативное тестирование – тестирование, которое гарантирует, что программа может корректно обрабатывать неверный ввод*

*данных или неожиданное поведение пользователя.*

Пример негативного теста. В программе есть поле ввода, способное принимать только цифры. При тестировании мы проверяем, что в поле нельзя ввести буквы, спецсимволы, знаки препинания. Программа при этом должна сообщать нам о невозможности ввода некорректных данных, а также не должна аварийно завершать работу:




Если в процессе негативного тестирования программа позволит ввести вместо цифр буквы и попытаться их сложить, а ещё хуже – при этом аварийно завершит свою работу значит тест не пройден. Если программа не позволит вводить ничего кроме цифр или даст вводить буквы, но не позволит запустить функцию сложения, тест пройден.

# Ложноположительные и ложноотрицательные результаты тестирования

В процессе проведения тестирования специалисты по тестированию могут принять верное поведение программы за дефект, а неверное поведение – за норму. В этом случае говорят, что получены ложноотрицательный или ложноположительный результаты тестирования. Иногда их называют «ложнонегативными» и «ложнопозитивными» результатами тестирования.

*Ложноположительные результаты тестирования – во время проверки функционала программы принимаем неправильное поведение программы за её корректное поведение.*

Ложноположительные результаты тестирования возникают в следующей ситуации. Представим, что мы проверяем работу некой функции программы. Ожидается, что при определённых входных данных функция должна возвращать конкретный результат. Однако в тесте мы подаём на вход другие данные, при которых поведение функции не определено требованиями. И если на эти непредвиденные данные функция вернула какой-то результат, мы по ошибке принимаем такое поведение за правильное. Хотя на самом деле функция работает неправильно, выдавая случайный результат вместо сообщения об ошибке. То есть мы ошибочно интерпретируем неправильную работу программы как корректную. Из-за этого подобные дефекты остаются незамеченными во время тестирования.

*Ложноотрицательные результаты тестирования – во время проверки функционала программы, принимаем правильное поведение*

*программы за её некорректное поведение.*

Ложноотрицательные результаты тестирования возникают в следующей ситуации. Допустим, мы проверяем работу функции на некотором входном значении. Спецификация гласит, что на этих данных функция должна вернуть определённый результат. Однако в тесте функция вернула другое значение, отличное от ожидаемого. На первый взгляд это выглядит как дефект в работе функции. На самом деле возвращаемое значение также является корректным согласно дополнительным неформальным требованиям, о которых специалист по тестированию не знал. То есть функция работает правильно, но из-за неполноты требований тестировщик ошибочно интерпретирует это как дефект. Таким образом из-за ложноотрицательного результата корректно работающий функционал маркируется как дефектный. Это приводит к необоснованной потере времени на поиск несуществующих дефектов.

# Ручное и автоматизированное тестирование

Тестирование может проводиться как вручную, так и автоматизировано с помощью специализированных инструментов, которые позволяют выполнять тесты не вручную, а автоматизировано. Рассмотрим эти понятия.

*Ручное тестирование – процесс тестирования программного обеспечения вручную без использования программных средств, которые выполняют проверки функционала программы с помощью автоматизированных сценариев тестирования.*

Пример: запускаем программу и начинаем её тестировать, вручную нажимать на кнопки, вводить данные и смотреть на получаемые результаты. Если вы сейчас самостоятельно начнёте проверять корректность работы любой программы, вы как раз будете проводить ручное тестирование.

В его определении мы упомянули автоматизированный сценарий тестирования, который также именуют автоматизированным тестом.

*Автоматизированный сценарий тестирования – это набор действий, описанный на определённом языке программирования, которые выполняются автоматически с использованием специальных инструментов или программного обеспечения для проверки определённой функции тестируемой программы.*

Их разрабатывают специалисты по автоматизированному тестированию. Такие сценарии тестирования позволяют проверить работоспособность и качество программы, автоматизируя процесс. Автоматизированные сценарии тестирования могут включать в себя различные действия, такие как ввод данных, выполнение определённых операций, проверку результатов и сравнение их с ожидаемыми значениями. Они помогают ускорить и упростить процесс тестирования, а также повысить его надёжность и точность, и

являются основой автоматизированного тестирования.

*Автоматизированное тестирование – процесс тестирования программного обеспечения с использованием программных средств для выполнения автоматизированных сценариев тестирования и проверки результатов выполнения, с целью сокращения времени тестирования и упрощения процесса.*

Чтобы заниматься таким тестированием необходимо обладать специализированными навыками и знать языки программирования.

Какое тестирование можно автоматизировать?

– функциональное тестирование сайтов;

– функциональное тестирование настольных приложений;

– функциональное тестирование мобильных приложений;

– тестирование API информационных систем.

Есть и другие сферы применения автоматизированного тестирования. Некоторые ключевые характеристики автоматизированного тестирования:

– Можно запускать тесты регулярно и независимо от занятости человека. Это обеспечивает непрерывное тестирование.

– Ускоряет и упрощает процесс тестирования.

– Повышает надёжность тестирования благодаря исключению из процесса человеческого фактора.

К примеру, необходимо ежедневно проверять, что после выпуска очередной новой версии программы в ней не ломаются основные функции, которые до этого работали без ошибок. Для этого требуется каждый день выполнять сотни тестов – однообразная и не воодушевляющая специалистов деятельность. В таком случае при наличии навыков и опыта специалисты разрабатывают сотни автоматизированных сценариев тестирования, которые в дальнейшем будут автоматически запускаться ежедневно, проверять программу, и заинтересованным людям будут отправляться на почту отчёты с результатами тестирования. Специалисты в это время могут посвятить себя более интеллектуальному тестированию.

# Тестовое окружение и не только

В процессе своей трудовой деятельности специалисты по тестированию работают с окружениями. Тестовое окружение и продуктивное окружение – это два основных типа окружений при разработке программного обеспечения. Под окружением в данном случае понимается функционально связанный между собой набор компьютеров, программ, информационных систем.

*Продуктивное окружение – это окружение, в котором запускается готовое программное обеспечение для конечных пользователей.*

Когда вы посещаете интернет и заходите на сайт поисковой системы, чтобы осуществить поиск какой-либо информации, вы попадаете на сайт, работающий в промышленном окружении, куда поместили его создатели, чтобы конечные пользователи им пользовались.

Основные характеристики продуктивного окружения:

– в нём работает окончательная версия программы после всех тестов;

– в нём не разрабатывается и не тестируется программа;

– оно не используется для исправления дефектов;

– обеспечивает максимальную производительность и доступность;

– содержит реальные данные и учётные записи пользователей;

– настроена на безопасность и конфиденциальность данных;

– требует высокой доступности и производительности по сравнению с тестовым окружением.

*Тестовое окружение – окружение, предназначенное для тестирования и отладки программ.*

Перед тем как поместить сайт в промышленное окружение создатели помещают его в тестовое окружение, которое не доступно для конечных пользователей. В тестовом окружении проводят

тестирование поискового сайта, чтобы убедиться, что сайт работает и его можно в дальнейшем поместить в промышленное окружение.

Основные характеристики тестового окружения:

– в нём разрабатывается и тестируется программа до выпуска в продуктивное окружение;

– используется для исправления дефектов и проверки функциональности до переноса в промышленное окружение;

– имитирует реальные условия работы, но может иметь меньшие мощности по сравнению с промышленным окружением;

– содержит не реальные данные, а тестовые.

Таким образом, тестовое окружение – для разработки и тестирования, продуктивное окружение – для реальной эксплуатации готовой программы. Основная работа специалистов по тестированию проводится на тестовом окружении.

В различных компаниях могут использоваться и другие окружения. Примеры названий: dev (дев), test (тест), qa (кью-эй), preprod (препрод), stage (стейдж), preview (превью) и т. д. То, какие окружения используются, зависит от специфики работы организации и разрабатываемых программ, поэтому мы не будем их рассматривать. Но вы должны знать, что кроме продуктивного и тестового окружения в ряде организаций существуют и другие окружения.

# Тестовая документация и артефакты

В процессе работы специалисты по тестированию создают различные артефакты[11] и документы. Кстати, дефект – это один из артефактов работы тестировщика. Познакомимся поближе с документами и артефактами, которые могут создавать и с которыми могут работать специалисты по тестированию.

# План тестирования

Тестирование, как и любой другой процесс, должно планироваться. Планирование тестирования – это одна из активностей, о которой мы поговорим в других главах книги. Планирование является непрерывной деятельностью, которая выполняется в течение всего жизненного цикла ПО, и в процессе планирования создаётся план тестирования (тест-план).

*План тестирования – документ, описывающий стратегию и тактику тестирования программного обеспечения.*

Это документ, который помогает организовать и планировать процесс тестирования на этапе разработки программного обеспечения, что позволяет провести тестирование эффективно и качественно.

В плане тестирования фиксируется следующая информация:

– Цели и задачи тестирования. Здесь определяется, что именно нужно проверить в программном обеспечении.

– Объекты тестирования[12]. Какие модули, функциональные возможности, интерфейсы и т. д. будут тестироваться.

– Уровни, типы и виды тестов. Например, функциональное тестирование, нефункциональное тестирование, тестирование производительности, и т. д.

– Приоритеты тестов. В какой последовательности будут выполняться тесты.

– Ответственные за тестирование. Кто конкретно будет разрабатывать, выполнять и отслеживать результаты тестов.

– График тестирования. План со сроками, в котором указано, когда и в какие сроки должно быть проведено и завершено тестирование.

– Тестовое окружение. Какое тестовое окружение будет использоваться в тестах.

– Риски. Что может увеличить сроки тестирования или блокировать тестирование и как эти риски нивелировать.

– Критерии успешности тестирования. Что считать успешным тестированием и какие условия должны выполняться.

В зависимости от организации в план тестирования могут включать и другую информацию, которую посчитают важной. С примером плана тестирования вы можете ознакомиться на сайте автора[13].

Разработку планов тестирования проводят опытные специалисты по тестированию или руководители и менеджеры по тестированию.

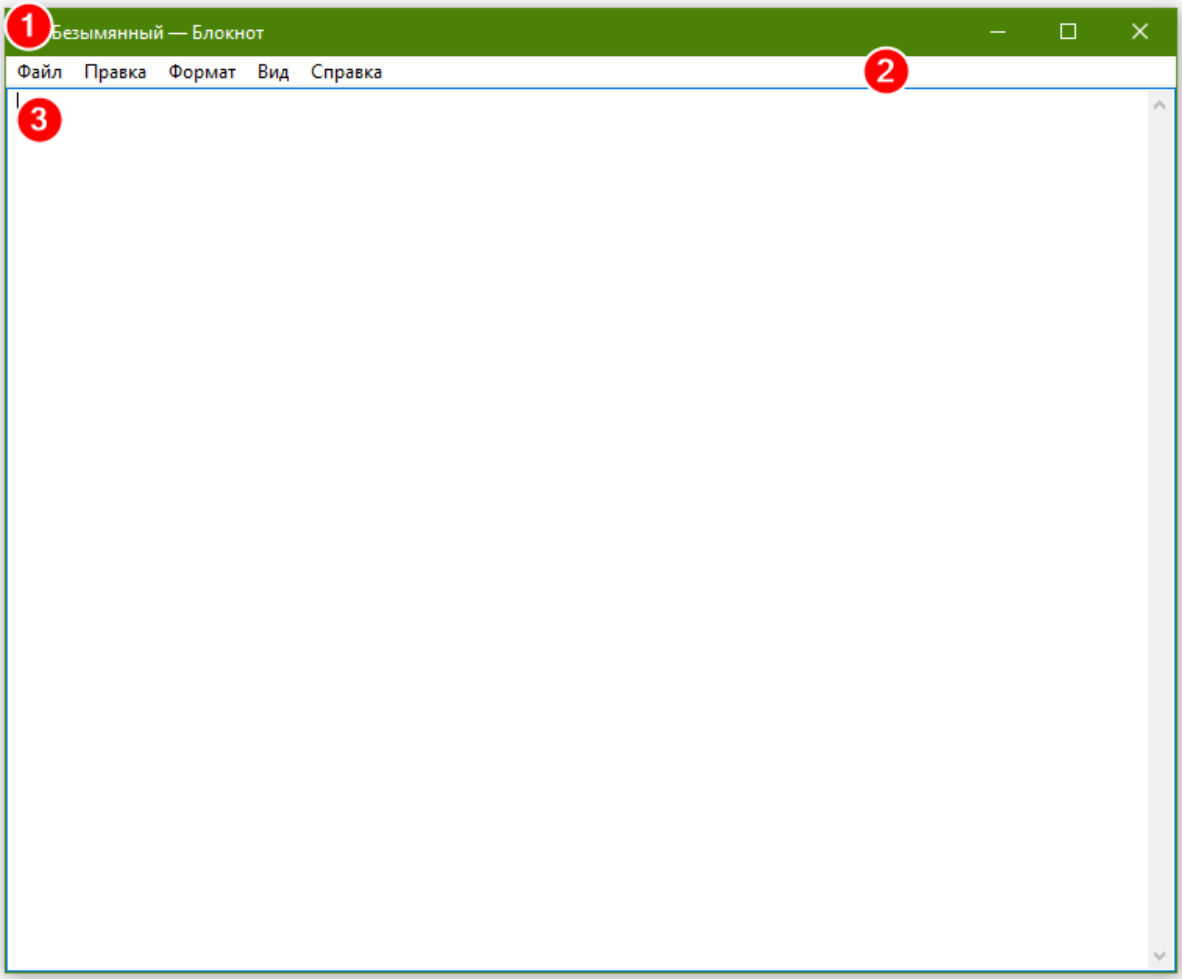
# Функциональная карта

Функциональную карту ещё называют «интеллектуальной картой», «диаграммой связей», «ассоциативной картой» и т. д. Мы её называем «функциональной», так как на ней отражены функциональные области программы, которые необходимо тестировать.

*Функциональная карта программы – схема, которая визуально отображает основные функциональные возможности программы и позволяет быстро понять её назначение и принцип работы.*

Цель функциональной карты – дать общее представление о структуре и логике работы программы, выделить её основные возможности. Она используется на этапе планирования и проектирования тестов. В функциональной карте отображают основные модули или разделы программы, вспомогательные функции, обеспечивающие работу основных модулей.

Как функциональные карты применяются специалистами по тестированию в работе? Создавая функциональную карту программы, специалист разбивает программу на логические функциональные блоки и описывает её ветвлениями. Рассмотрим на простом примере, взяв за основу программу «Блокнот», которая имеется в операционной системе Windows:




У программы «Блокнот» есть «Заголовок» (1), «Строка меню» (2), «Форма ввода данных» (3). Названные блоки (элементы) в свою очередь делятся на дополнительные элементы.

Заголовок (1) имеет:

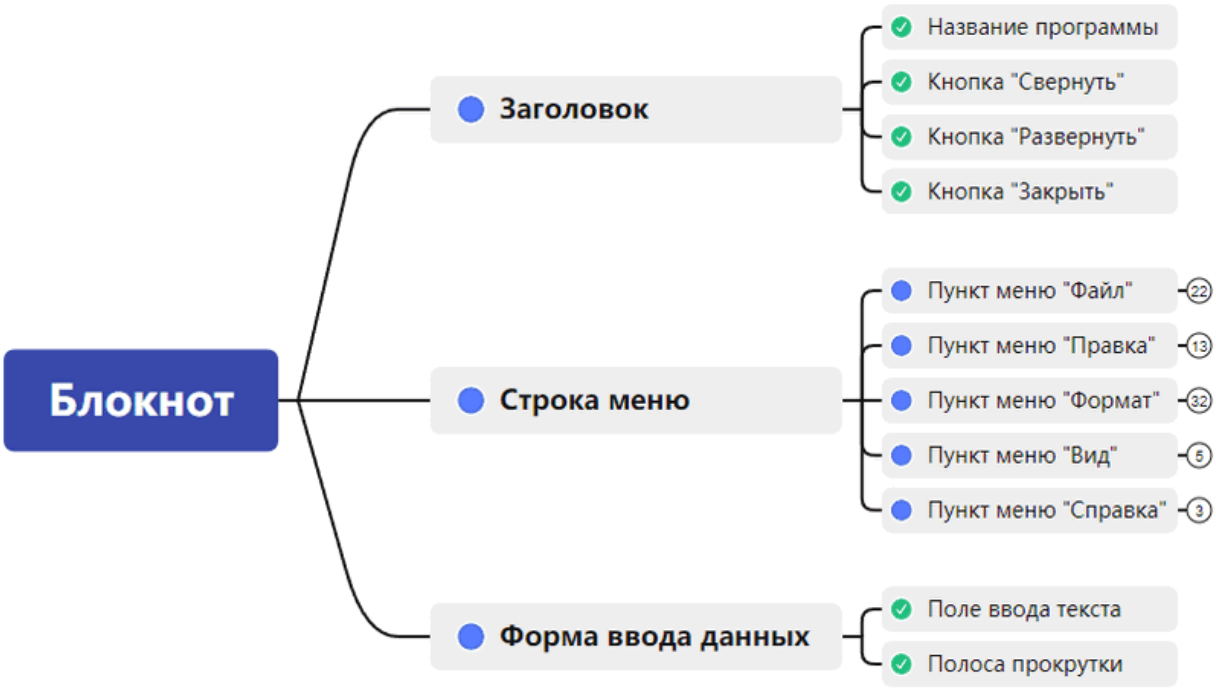
– название программы в заголовке;
– кнопка «Свернуть»;
– кнопка «Развернуть»;
– кнопка «Закрыть».

Строка меню (2) имеет:

– пункт меню «Файл»;
– пункт меню «Правка»;
– пункт меню «Формат»;
– пункт меню «Вид»;
– пункт меню «Справка».

Форма ввода данных (3) имеет:

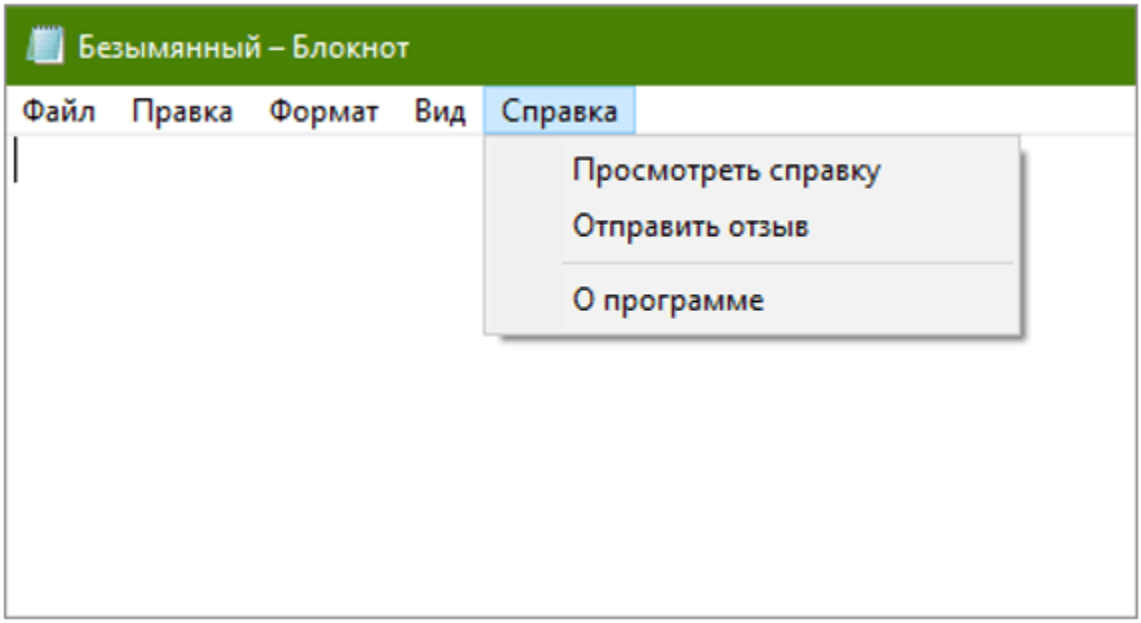
– поле ввода текста;
– полоса прокрутки.
Всё перечисленное можно отобразить на функциональной карте:



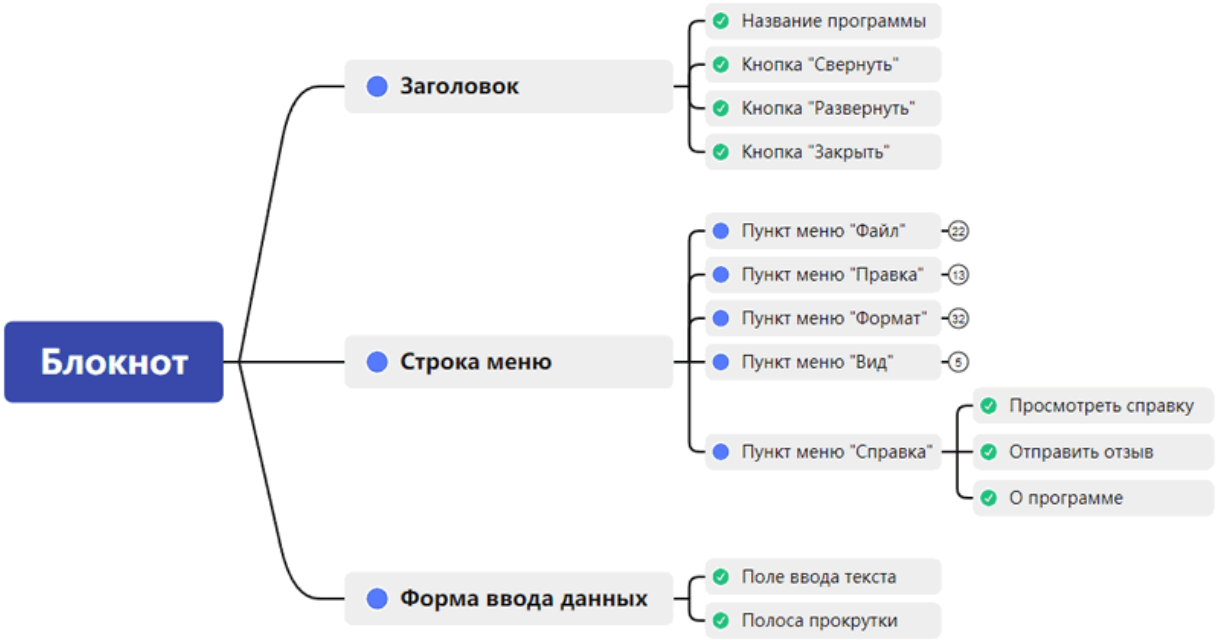

На функциональной карте видим, как ветвится функционал программы. Обратите внимание, что для удобства восприятия различные пункты помечаем двумя видами значков, которые могут быть другими:

– синий круг – это значит ветвление будет дальше продолжаться;
– зелёный круг с галочкой – это является конечной проверкой.

Визуально видим, как ветвится функционал программы и остаётся только продолжить следование по пунктам меню или функционалу программы. Для большей наглядности продолжим разбор пункта меню «Справка»:




Пункт меню «Справка» имеет пункты:
– просмотреть справку.
– отправить отзыв.
– о программе.
Функциональная карта получит следующее продолжение:




Таким образом следуя по всей программе, специалист описывает её функциональной картой. Если он где-то оставил значок синего круга и

начал описывать другие ветки, то точно не забудет вернуться, чтобы продолжить описывать функциональную ветку программы, пока не доберётся до конечных пунктов проверки.

После того как тестировщик полностью создаст функциональную карту, он будет видеть все проверки, которые необходимо провести в программе.

# Тест-кейс

*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.*

Предполагаю, что, прочитав это официальное описание, вы загрустили, ничего не поняв. Попробую сформулировать определение, используя понятные для всех формулировки.

*Тест-кейс – это чёткое описание действий, которые необходимо выполнить, 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 проверки; описание того, что должно произойти, после выполнения действий для проверки – ожидаемый результат; действия, которые необходимо выполнить в самом конце, чтобы привести систему в первоначальное состояние, сбросив все внесённые нами изменения – постусловия.*

Второе определение понятнее, однако всё равно необходимо напрячься, чтобы понять написанное, поэтому следующее определение будет ещё проще и дано на языке простого обывателя.

*Тест-кейс – это описание того, что надо сделать, чтобы проверить определённый функционал программы и что должно произойти после того, как мы выполним описанные действия.*

Почему здесь приведено так много определений? Для того, чтобы вы чётко осознали, что такое тест-кейс, так как в своей работе специалист по тестированию очень много времени уделяет работе с тест-кейсами: написание, правка, проверка программ по тест-кейсам и т. д.

Тест-кейсы также называют «контрольными примерами» или «сценариями тестирования». В определениях терминов в данной книге вы с этим названием будете сталкиваться.

Рассмотрим основные атрибуты, из которых состоит тест-кейс и которые используются в большинстве организаций:

1) Номер тест-кейса – уникальный идентификатор тест-кейса. Если у вас тысячи тест-кейсов, то при общении с коллегами вам будет проще сообщить номер тест-кейса, ссылаясь на него, а не пытаться словами рассказать, где и как найти определённый тест-кейс.

2) Заголовок – краткое, понятное и ёмкое описание сути проверки.

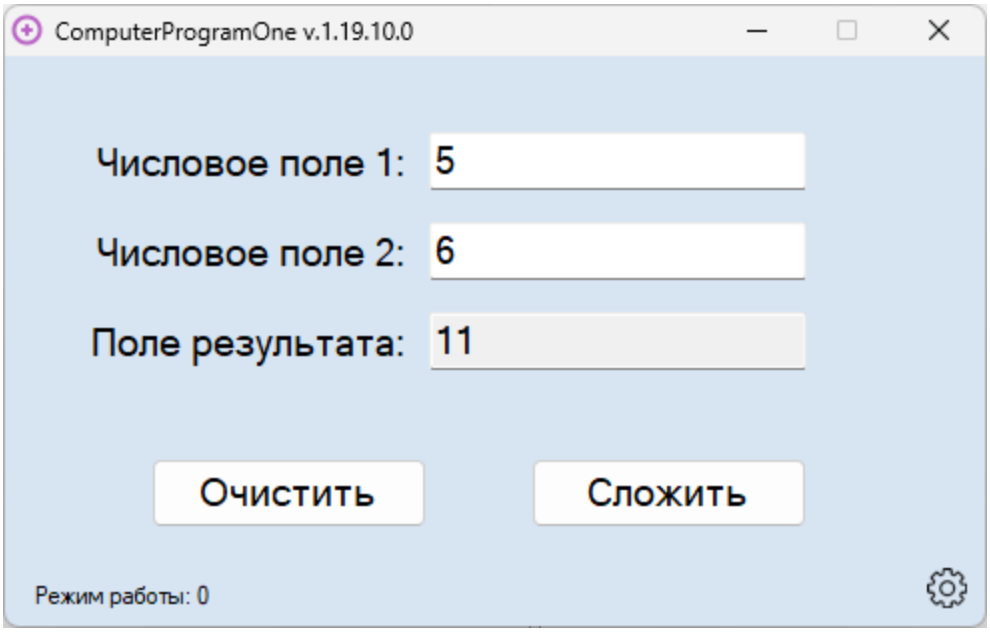
3) Предусловия – описание действий, которые необходимо предварительно выполнить или учесть перед началом проведения проверок.

4) Шаги проверки – описание последовательности действий, которые надо выполнить для проверки.

5) Ожидаемый результат – проверка, которая устанавливает, что мы ожидаем получить после выполнения определённых действий в соответствующем шаге.

В зависимости от специфики организации могут присутствовать дополнительные атрибуты для заполнения: постусловия, приоритет, функциональный блок, программа, ссылка на требование, номер требования и т. д.

Рассмотрим тест-кейс на примере программы для сложения чисел:




Для тестирования программы необходимо подготовить список проверок и написать тест-кейсы. Одной из проверок будет сложение однозначных положительных целых чисел. Специалисту по

тестированию надо на эту проверку написать тест-кейс. Предположим, что это будет двенадцатый тест-кейс, который он начал писать на данную программу:

| | | |
|---|---|---|
| | **Номер тест-кейса:** | 12 |
| | **Название:** | Сложение однозначных положительных целых чисел |
| | **Предусловия:** | Программа «Computer Program One» запущена |
| | | |
| | **Шаги** | **Ожидаемый результат** |
| 1 | Ввести с клавиатуры в поле «Числовое поле 1» любое однозначное положительное целое число. | В поле «Числовое поле 1» отображается введённое число. |
| 2 | Ввести с клавиатуры в поле «Числовое поле 2» любое однозначное положительное целое число. | В поле «Числовое поле 2» отображается введённое число. |
| 3 | Нажать на кнопку «Сложить». | В поле «Поле результата» отображается сумма чисел «Числовое поле 1» + «Числовое поле 2». |

Видим готовый тест-кейс. Используя его, уже можно провести одну проверку программы на сложение положительных целых чисел. Суть понятна. Теперь рассмотрим правила написания тест-кейсов. Их не нужно заучивать. При необходимости можете открыть данную книгу и повторить правила перед тем, как начать писать тест-кейсы.

Правило № 1. Заголовок:

– должен быть чётким, кратким, понятным и однозначно характеризующим суть тест-кейса;

– не может содержать выполняемые шаги и ожидаемый результат.

Правило № 2. Предусловие:

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

– может содержать ссылки на информационные источники, которые нужно изучить перед прохождением тест-кейса (инструкции, описание систем…);

– не может содержать ссылки на тестируемый ресурс, если у информационной системы более одного окружения (продуктивное окружение, тестовое окружение…), данная информация должна быть вынесена в инструкцию, и ссылка приложена в предусловии;

– не может содержать данные для авторизации, эта информация должна быть вынесена в инструкцию, и ссылка приложена в предусловии;

– не может содержать выполняемые шаги и ожидаемый результат, если нам надо, чтобы до выполнения шагов проверки у нас была открыта главная страница, то в предусловии указываем «открыта главная страница сайта»;

– не может содержать ожидаемый результат.

Правило № 3. 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»;

– не должно быть комментариев и пояснений. Если есть необходимость привести мини-инструкцию, то оформляем инструкции в базе-знаний и ссылаемся на неё в предусловии;

– не должно быть жёстко прописанных статических данных (логины, пароли, имена файлов) и примеров для исключения эффекта пестицида (о нём поговорим в другой главе).

Правило № 4. 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.

Правило № 5. Общие требования к тест-кейсам:

– язык описания тест-кейсов должен быть понятен широкому кругу пользователей, а не узкой группе лиц;

– тест-кейс должен быть независим от других тест-кейсов и не должен ссылаться на другие тест-кейсы;

– тест-кейсы группируются в функциональные блоки по их назначению;

– в тест-кейсах, проверяющих работу функционала, картинок (снимков экранов) быть не должно, иначе вы посвятите сотни часов на изменение всех картинок в тысячах тест-кейсах при изменении интерфейса тестируемой программы. Картинки можно добавить только в тест-кейсы, проверяющие визуальное отображение страниц и форм.

Правила простые, однако первое время их непросто соблюдать, так как хочется всё сделать быстро, несмотря ни на что. Если придерживаться данных правил, тест-кейсы станут легко поддерживаемыми, легко читаемыми и могут быть использованы всеми участниками команды в процессе разработки программного обеспечения.

# Чек-лист

Чек-лист отличается от тест-кейсов степенью детализации. В чек-листе вы не встретите подробных шагов, которые есть в тест-кейсах, поэтому при использовании чек-листа в тестировании необходимо много информации держать в голове в момент проведения тестирования и хорошо знать логику работы программы. Отсюда следует, что проводить тестирование программы по чек-листам могут опытные специалисты по тестированию, хорошо знающие тестируемую программу.

Чек-листы используются в тестировании в нескольких случаях:

1) Вместо тест-кейсов, если у тестировщика не хватает времени создавать и обновлять тест-кейсы.

2) Предварительно создаются чек-листы, на основании которых пишутся тест-кейсы. В этом случае быстро составляется список проверок и потом пишутся тест-кейсами.

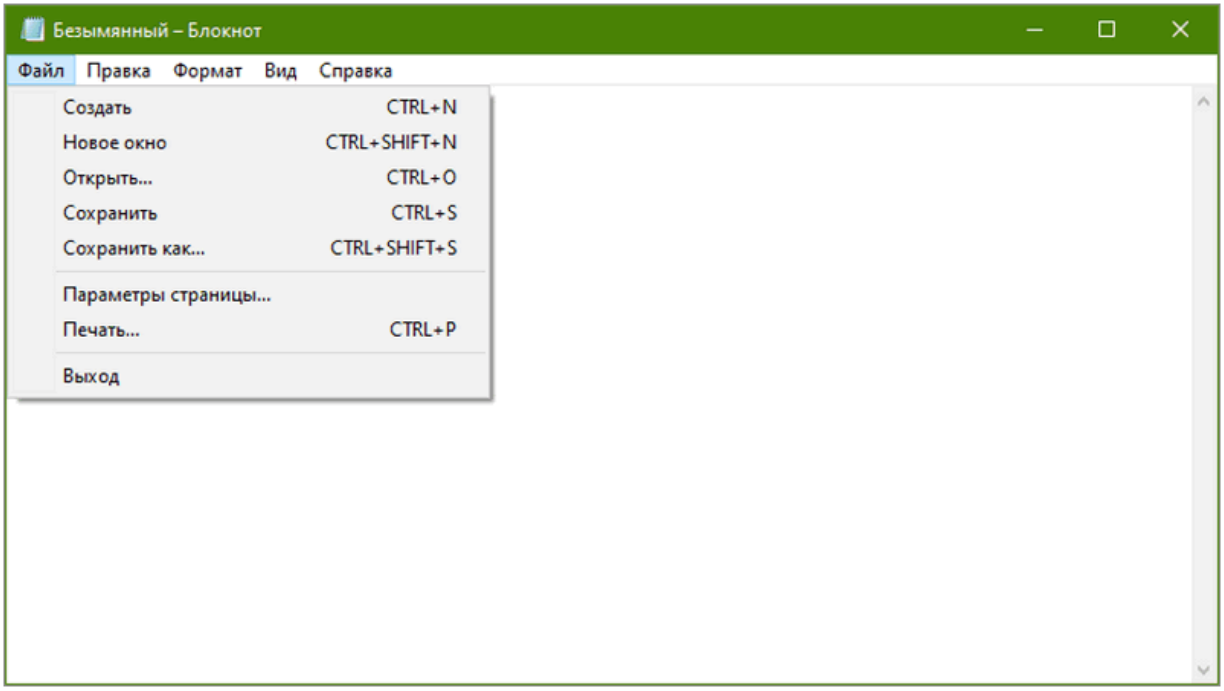
Чек-листы с проверками могут создаваться на основании требований, знаний о программе, функциональной карты программы.

Рассмотрим пример составления чек-листа для тестирования программы «Блокнот» на основании наших знаний о программе:


Начнём составление с меню программы. У нас есть пять пунктов меню, поэтому в чек-листе появятся первые пять проверок (тестов) программы:

– раскрытие пункта меню «Файл»;
– раскрытие пункта меню «Правка»;
– раскрытие пункта меню «Формат»;
– раскрытие пункта меню «Вид»;
– раскрытие пункта меню «Справка».

Вот и готовы пункты чек-листа для тестирования функционала раскрытия пунктов меню. Теперь создадим чек-лист из списка проверок работы пунктов меню, которые вложены в меню «Файл»:




В процессе составления списков проверок буду использовать различные техники наименования проверок, чтобы вы могли видеть варианты, которые сможете применять в будущем:

– проверка работы пункта меню «Файл – Создать»;

– проверить пункт меню «Файл – Новое окно»;

– проверка работы функционала пункта меню «Файл – Открыть»;

– работа функционала пункта меню «Файл – Сохранить»;

– функционал пункта меню «Файл – Сохранить как»;

– работа пункта меню «Файл – Параметры страницы»;

– работа пункта меню «Файл – Печать»;

– работа пункта меню «Файл – Выход».

Так как пункты меню в программе могут отрабатывать по нажатию горячих клавиш, то создадим проверки работы функционала этих же пунктов, только с помощью горячих клавиш:

– работа функционала «Файл – Создать» одновременным нажатием клавиш «CTRL + N»;

– проверка работы функционала «Файл – Новое окно» одновременным нажатием клавиш «CTRL + SHIFT + N»;

– проверка работы функционала «Файл – Открыть» одновременным нажатием клавиш «CTRL + O»;

– проверка работы функционала «Файл – Сохранить» одновременным нажатием клавиш «CTRL + S»;

– проверка работы функционала «Файл – Сохранить как» одновременным нажатием клавиш «CTRL + SHIFT + S»;

– проверка работы функционала «Файл – Печать» одновременным нажатием клавиш «CTRL + P».

Соберём все написанные проверки и получим чек-лист для проверки определённого функционала программы «Блокнот»:

– раскрытие пункта меню «Файл»;

– раскрытие пункта меню «Правка»;

– раскрытие пункта меню «Формат»;

– раскрытие пункта меню «Вид»;

– раскрытие пункта меню «Справка»;

– проверка работы пункта меню «Файл – Создать»;

– проверить пункт меню «Файл – Новое окно»;

– проверка работы функционала пункта меню «Файл – Открыть»;

– работа функционала пункта меню «Файл – Сохранить»;

– функционал пункта меню «Файл – Сохранить как»;

– работа пункта меню «Файл – Параметры страницы»;

– работа пункта меню «Файл – Печать»;

– работа пункта меню «Файл – Выход»;

– работа функционала «Файл – Создать» одновременным нажатием клавиш «CTRL + N»;

– проверка работы функционала «Файл – Новое окно» одновременным нажатием клавиш «CTRL + SHIFT + N»;

– проверка работы функционала «Файл – Открыть» одновременным нажатием клавиш «CTRL + O»;

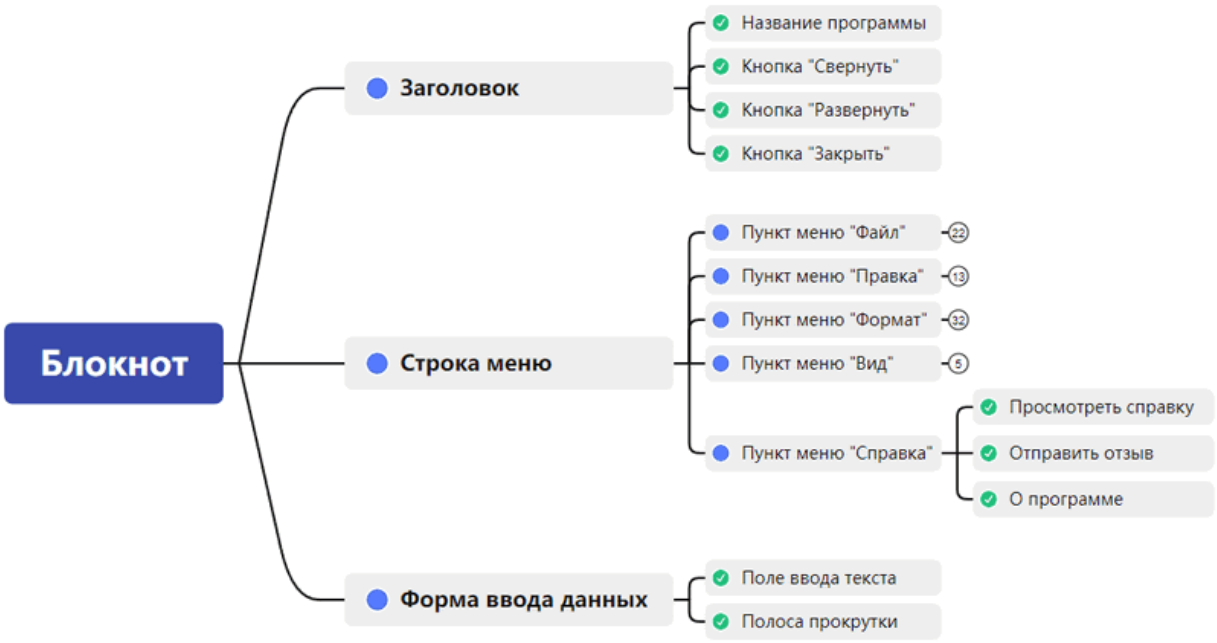
– проверка работы функционала «Файл – Сохранить» одновременным нажатием клавиш «CTRL + S»;

– проверка работы функционала «Файл – Сохранить как» одновременным нажатием клавиш «CTRL + SHIFT + S»;

– проверка работы функционала «Файл – Печать» одновременным нажатием клавиш «CTRL + P».

Это не полный список проверок программы «Блокнот», а его небольшая часть. Размер чек-листа, содержащего проверки, зависит от сложности программы и её знания тестировщиком.

Стоит помнить: если функционал программы большой и нет полных требований, то при создании чек-листов специалист скорее всего будет пропускать часть функционала, который необходимо тестировать, так как в момент, когда список увеличивается до сотни или тысячи пунктов, он не может запомнить, какие проверки описал в чек-листе, а какие забыл. В этом случае рекомендую разрабатывать чек-лист на основании функциональной карты программы. После того как специалист полностью подготовит функциональную карту программы, ему останется только перенести все конечные проверки с функциональной карты в чек-лист. Рассмотрим на примере:




На приведённой функциональной карте мы видим конечные проверки. Нам остаётся перенести их в чек-лист:

– проверить корректное отображение названия заголовка программы;

– работа кнопки «Свернуть программу» в заголовке программы;

– работа кнопки «Развернуть программу» в заголовке программы;

– работа кнопки «Закрыть программу» в заголовке программы;

– работа пункта меню «Справка» – «Просмотреть справку»;

– работа пункта меню «Справка» – «Отправить отзыв»;

– работа пункта меню «Справка» – «О программе»;

– ввод текста в поле ввода текста (кириллица, латиница, цифры, спецсимволы);

– работа полосы прокрутки.

В функциональной карте девять конечных проверок и в чек-листе аналогичное количество проверок. Составляя чек-лист на основании функциональной карты, специалист по тестированию максимально покроет программу проверками, ничего не упуская. Главное предварительно качественно составить функциональную карту.

Посмотрите на чек-лист. На что похожи названия проверок? На заголовки тест-кейсов. Имея чек-лист проверок, специалист по ним может писать тест-кейсы. Из примера выше можно подготовить девять тест-кейсов, в которых будет подробно указано, как проводить каждую проверку.

Кстати, составляя чек-лист проверок по функциональной карте мы применили один из методов проектирования тестов. Когда изучите все методы проектирования тестов, вспомните о данной главе и самостоятельно определите, какой метод проектирования тестов был применён.

# Отчёт о дефекте

*Отчёт о дефекте (defect report, bug report) – документ, содержащий отчёт о недостатке в компоненте или программе, который может привести компонент или программу к невозможности выполнить требуемую функцию.*

Если специалист по тестированию находит в программе дефект, он должен о нём сообщить всем заинтересованным лицам, чтобы дефект был исправлен. Для этого он составляет отчёт о дефекте и отправляет его аналитикам или программистам, в зависимости от договорённостей в организации.

Рассмотрим принципы и правила оформления отчётов о дефектах.

Правило № 1. Тема или заголовок. Заголовок должен быть кратким и в то же время описывать суть проблемы. В заголовок не надо помещать описание всего дефекта – оно будет непосредственно в описании.

Правило № 2. Описание дефекта. Должно содержать следующие данные:

– Предусловие. Описание действий, которые необходимо предварительно выполнить или учесть. Это блок, где поясняется вводная информация.

– Шаги воспроизведения. Описание последовательности действий, которые необходимо выполнить для воспроизведения дефекта.

– Фактический результат. Результат, который получил автор отчёта, проделав последовательность действий, описанных в блоке «Шаги воспроизведения».

– Ожидаемый результат. Что мы ожидаем получить согласно требованиям после выполнения определённых действий в условиях корректной работы программы.

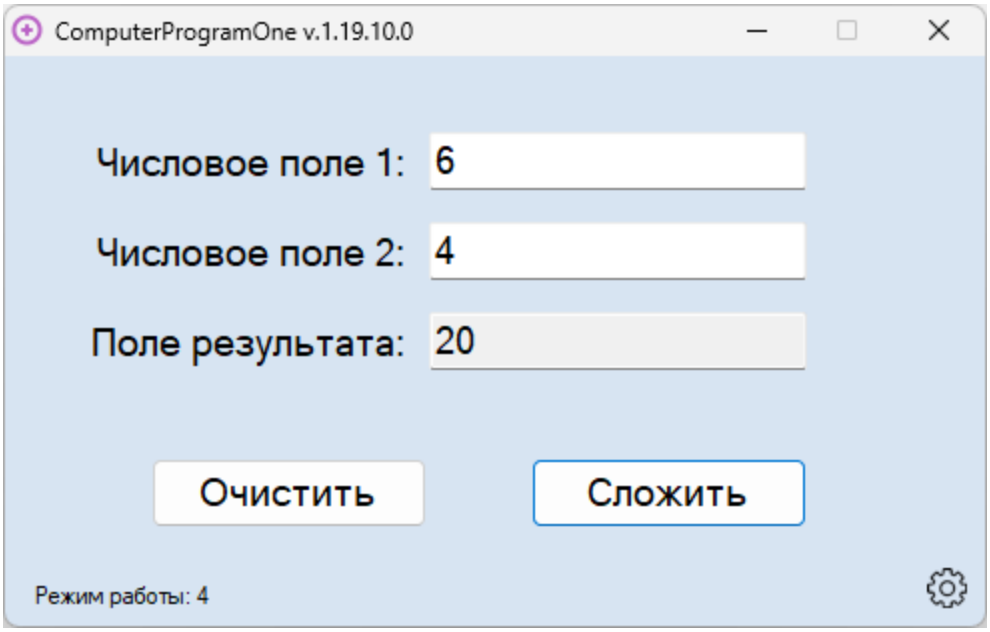
– Дополнительная информация. Необязательный блок, в нём добавляется информация, которая считается важной и которая поможет оперативно исправить дефект.

Правило № 3. Вложение. Прикладываем к описанию снимки экранов и видео (если это возможно), служебные файлы и логи, которые можно использовать для воспроизведения и локализации

дефекта. Видео полезно прикладывать, когда последовательность действий для воспроизведения дефекта очень длинная, – целесообразней снять видео и приложить его, а не описывать всю последовательность на пару листов А4 (в этом случае фиксируются крупные шаги в описании). На снимках экранов должно быть выделено место, на которое нужно обратить внимание. Если это снимки экранов к шагам воспроизведения, надо выделить место, куда следует нажать, куда что-то ввести и т. п., а также снимок экрана должен вставляться по тексту в шаге воспроизведения, к которому он относится, а не прикладываться в самом конце описания.

В отчёте могут заполняться дополнительные атрибуты: приоритет, важность, заявитель, исполнитель, версия программы, окружение и прочее. Это делается на усмотрение заявителя, если не являются обязательными для заполнения, а также согласно установленным в организации правилам.

Рассмотрим простой пример составления отчёта о дефекте на примере программы для сложения чисел, которая при их сложении выводит некорректный результат:




Отчёт о дефекте:

| **Заголовок:** |
| --- |
| Программа «Computer Program One» некорректно складывает числа |
| **Описание:** |
| Предварительные условия<br>Версия тестируемой программы 1.19.10.0<br>Шаги воспроизведения<br>1. Запустить программу<br>2. Ввести в «Числовое поле 1» число 6.<br>3. Ввести в «Числовое поле 2» число 4.<br>4. Нажать на кнопку «Сложить»<br>Ожидаемый результат<br>В поле «Поле результата» отображается число 10 – сумма чисел полей «Числовое поле 1» и «Числовое поле 2».<br>Фактический результат<br>В поле «Поле результата» отображается число 20. Число больше ожидаемого результата сложения на 10.<br>Дополнительная информация<br>После сложения любых чисел, в поле «Поле результата» всегда отображается результат, который больше ожидаемого результата сложения на 10. |

Придерживаясь приведённых правил составления отчётов о дефектах, специалист по тестированию облегчает работу себе, программистам и остальным членам команды. Если в отчёте будет указан минимум информации, программист не сможет локализовать проблему, руководитель разработки отклонит дефект, что повлечёт за собой дополнительную потерю времени на повторное заведение/ открытие дефекта и общение с программистами. Для того чтобы программист мог приступить к исправлению дефекта, он должен иметь всю необходимую информацию по дефекту, поэтому автору отчёта надо подробно описывать все детали.

# Отчёт о тестировании

*Отчёт о тестировании – это документ, в котором суммируются все результаты проведённых испытаний программного обеспечения.*

Основная цель отчёта о тестировании – проинформировать всех заинтересованных лиц о ходе тестирования, выявленных дефектах и степени готовности программного обеспечения. Он составляется по окончании всех этапов тестирования и является обязательным атрибутом данного процесса. В нём детально описываются все аспекты проведённых испытаний: этапы, сроки, задачи, участники, полученные результаты. Отчёт, подготовленный во время тестирования, называется «отчётом о ходе тестирования». Отчёт, подготовленный по итогам, называется «итоговым отчётом о тестировании», который содержит статистику по всем этапам, сводную информацию об ошибках и доработках.

Отчёт о тестировании имеет важное значение для всех участников команды, работающей над разработкой программы. Для заказчика он демонстрирует полноту проверок и качество выполненной работы. Программистам отчёт помогает определить основные направления доработки. При его составлении внимание уделяется детальности и структурированности представления информации. Это обеспечивает его понятность даже для лиц, не участвовавших непосредственно в тестировании. Отчёт должен быть максимально объективным и сбалансированным. В нём не допускается субъективная оценка, все выводы должны быть обоснованы конкретными фактами и данными.

Содержимое любого типа отчёта о тестировании зависит от тестируемой программы, требований организации или команды, жизненного цикла разработки программного обеспечения. С примером отчёта о тестировании вы можете ознакомиться на сайте автора[14].

# Базис тестирования

Периодически вы будете сталкиваться с понятием «базис тестирования», поэтому раскроем его.

*Определение согласно ISTQB:*

*Базис тестирования – документ, на основании которого определяются требования к компоненту или системе. Документация, на которой базируются тест-кейсы.*

*Определение согласно ГОСТ Р 56920-2016/ISO/IEC/IEEE 29119-1:2013:*

*Базис тестирования – свод знаний, используемых в качестве базы проекта тестирования и контрольных примеров. Может иметь форму документов, таких как спецификация требований, спецификация проекта или спецификация модуля, но может также представлять собой недокументированное понимание требуемого поведения.*

Оба определения говорят об одном и том же, однако второе точнее, и на мой взгляд корректнее, и понятнее первого. Особенно важна формулировка «*…может также представлять собой недокументированное понимание требуемого поведения.*», т. е. здесь говорится, что базис тестирования не только документ, но и знания в голове специалиста, если нет никакой документации.

Из всего сказанного делаем вывод: базис тестирования – это документированные или недокументированные знания о программе, которые используются для дальнейшего формирования тест-кейсов и проведения тестирования.

Рассмотрим примеры, что может выступать в качестве базиса тестирования:

– спецификация требований;

– пользовательская история[15];

– руководство пользователя (инструкция);

– программный код;

– любая документация, в которой зафиксированы знания о программе;

– знание специалистов о программе.

Изучив базис тестирования, специалист по тестированию может разрабатывать тест-кейсы, чек-листы и тестировать программу.

# Принципы тестирования

Принцип тестирования ПО – это основополагающие положения, которыми руководствуются при проведении тестирования программного обеспечения. За последние десятилетия выработаны и предложены принципы тестирования, которые являются общими для тестирования в целом. Давайте рассмотрим их.

# Тестирование демонстрирует наличие дефектов, а не их отсутствие

Данный принцип подразумевает, что тестирование программного обеспечения может показать, что дефекты присутствуют в программе, но не может доказать, что их нет. Тестирование снижает вероятность наличия дефектов, находящихся в программе, однако, даже если дефекты не были обнаружены, тестирование не доказывает их отсутствие и корректность работы программы.

В каждой программе есть дефекты: какие-то явные, какие-то могут проявляться редко и в определённых условиях, поэтому они часто не обнаруживаются или обнаруживаются при определённых условиях.

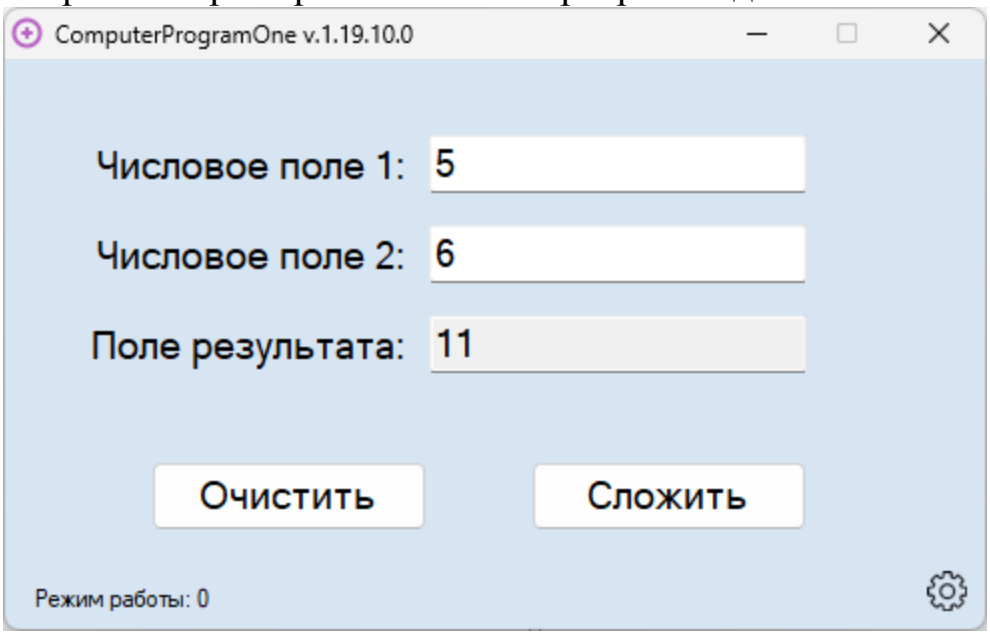
Пример. У нас есть программа, которая сохраняет документ за 2 секунды после нажатия на кнопку «Сохранить». Однако она работает с такой скоростью, если в ней одновременно документ сохраняют не более 1 000 человек. Если в программе одновременно 1 000 человек нажмут «Сохранить», у каждого из них программа будет сохранять документ 15 минут. Как видим, в программе есть дефект, который воспроизводится в определённых условиях, и ряд организаций никогда с ним не столкнётся. Зато те, у кого в штате десятки тысяч сотрудников, могут с этим дефектом столкнуться рано или поздно. Он есть, но может длительное время не обнаруживаться, так как о нём и об условиях его воспроизведения не подозревают. Это подтверждает, что тестирование не сможет продемонстрировать отсутствие дефектов в программе.

# Исчерпывающее тестирование недостижимо

*Исчерпывающее тестирование – методика тестирования, в которой набор тестов включает в себя все комбинации входных данных и предусловий.*

Исчерпывающее тестирование – это, по сути, полное всеобъемлющее тестирование с использованием всех возможных комбинаций ввода и предусловий. И рассматриваемый принцип говорит, что данное тестирование недостижимо, если только это не простейшая программа, имеющая одну кнопку и функцию. Поэтому в своей практике специалист по тестированию должен вместо попыток проведения исчерпывающего тестирования акцентировать усилия на максимальной пользе, которую может принести тестирование в свете имеющихся в определённый момент времени ограничений. Для этого ему надо планировать и проектировать тесты с использованием определённых практик и методов.

Рассмотрим на примере. У нас есть программа для сложения чисел:




Допустим, каждое поле для ввода чисел может содержать числа в двадцать знаков, и максимальное число 99 999 999 999 999 999 999. Чтобы провести исчерпывающее тестирование и полностью проверить

функционал сложения в этом случае необходимо протестировать сложение, перебрав в каждом поле все числа начиная с однозначных и заканчивая двадцатизначными, сложив все числа со всеми. Эта задача для тестировщика может затянуться на долгое время. С более сложными программами процесс займёт годы. Поэтому в таких случаях используют специальные техники и методы для уменьшения количества проводимых тестов. С ними мы будем знакомиться в других главах книги. К примеру, можно провести только эти проверки, исключив все остальные:

- 1 + 99 999 999 999 999 999 999;
- 99 999 999 999 999 999 999 + 1;
- 99 999 999 999 999 999 999 + 99 999 999 999 999 999 999.

Конечно же тут приведены не все проверки, так как их будет больше, если проектировать тесты с использованием различных методов, но суть, надеюсь, вам понятна. Это касательно методов.

Приоритеты и риски. Допустим, мы применили методы проектирования тестов, и у нас вместо сотен тысяч проверок имеется три тысячи проверок и один день на тестирование. В этом случае мы определяем критически важные для бизнеса или организации функции программы и выставляем проверкам этого функционала высокий приоритет. Потом определяем менее критические функции программы и выставляем проверкам этого функционала средний приоритет и т. д. В итоге у нас получится перечень проверок по приоритетам – когда на тестирование придёт программа, и срок на него будет минимальным, мы начнём проверки в порядке приоритета. В этом случае при нехватке времени у нас останется непроверенным не критический функционал.

Рассмотрим на примере программы сложения. У нас есть проверки:

– проверка корректности надписей на кнопках;

– проверка корректности надписей названия полей;

– сложение чисел;

– очистка полей от введённых данных.

И так далее. На тестирование имеется четыре секунды, и не секундой больше. Мы берём программу, за четыре секунды успеваем сделать только первые две проверки максимум. После этого программу передадут пользователям и окажется, что она не может складывать числа, так как до проверки сложения мы не дошли, а это критичный

для пользователей функционал. Поэтому предварительно необходимо приоритизировать проверки и приоритизировать их в порядке критичности:

– сложение чисел (высокая критичность);

– очистка полей от введённых данных (средняя критичность);

– проверка корректности надписей на кнопках (не критично);

– проверка корректности надписей названия полей (не критично).

И так далее. Теперь, когда нам передадут программу на тестирование, мы за четыре секунды успеем проверить критичный для пользователей функционал. Если у нас в запасе одна минута, мы успеем проверить все четыре указанные проверки.

# Раннее тестирование сохраняет время и деньги

Эффективное тестирование на более ранних этапах разработки программного продукта играет ключевую роль в обеспечении его качества и успешной реализации. Когда тестирование внедряется на ранних стадиях жизненного цикла разработки, это предоставляет ценные преимущества и минимизирует риски. Раннее тестирование способствует ускорению цикла разработки, поскольку выявление и устранение дефектов на ранних стадиях снижает вероятность возникновения серьёзных проблем и ошибок в более поздних фазах разработки.

На ранних стадиях исправление является более доступным и менее затратным процессом. Чтобы организации, которая занимается разработкой программного обеспечения, сократить финансовые расходы, необходимо проводить тестирование на ранних этапах жизненного цикла программного обеспечения. Тогда дефекты будут выявляться до того, как попадут к пользователям.

В октябре 2008 года компания IBM провела исследование[16] и, согласно ему, стоимость исправления дефекта варьируется в зависимости от того, на каком этапе жизненного цикла программы дефект обнаруживается. Смотрите таблицу:

| Дизайн и архитектура | Разработка | Интеграционное тестирование | Пользовательское бета-тестирование | После выхода в промышленную эксплуатацию |
|---|---|---|---|---|
| 1X* | 5X | 10X | 15X | 30X |

* X – единица стоимости, которая может выражаться в человеко-часах, денежных единицах и т. д.

Теперь попробуем данные из таблицы наложить на деньги. Допустим, на устранение дефекта у нас затрачивается один рабочий час. Один рабочий час сотрудников стоит 1 000 денежных единиц (X, единица стоимости). В этом случае мы будем иметь определённые затраты при исправлении дефекта на определённом этапе жизненного

цикла программы. Смотрите таблицу:

| Дизайн и архитектура | Разработка | Интеграционное тестирование | Пользовательское бета-тестирование | После выхода в промышленную эксплуатацию |
|---|---|---|---|---|
| 1 * 1000 = 1 000 | 5 * 1 000 = 5 000 | 10 * 1 000 = 10 000 | 15 * 1 000 = 15 000 | 30 * 1 000 = 30 000 |

Как видите, картина для организации не радужная. Чем позже дефект исправляется, тем больше финансов организация затрачивает. В связи с этим, если в организации нет специалистов по тестированию, ей стоит задуматься о развитии центра тестирования и не экономить на этом, так как финансовые потери на исправлении дефектов на поздних стадиях выше затрат на тестировщиков.

Что влияет на стоимость исправления? Рассмотрим несколько случаев и по итогам сделаем выводы.

Случай первый. Аналитик написал требования. Они готовы к передаче в разработку. Специалист по тестированию проверил требования и нашёл в них ошибку, сообщил аналитику, и тот сразу же исправил эту ошибку. Программист разработал программу по требованиям без ошибки. Специалист по тестированию протестировал программу. Программисты её собрали и передали пользователям. Дефект в программе не появился, так как ошибка в требованиях была исправлена на ранних стадиях.

Случай второй. Аналитик написал требования. Они готовы к передаче в разработку. В них есть ошибка. Требования переданы в разработку. Программист написал код. Специалист по тестированию провёл тестирование, нашёл дефект в программе и сообщил программисту и аналитику. Аналитик проанализировал дефект и подтвердил, что это действительно дефект и так программа не должна работать. Программист исправил дефект в программном коде. Тестировщик перепроверил исправление. Программисты программу собрали и передали пользователям.

На всех этапах, где участвуют люди, организация несла затраты на оплату труда и т. д. Если посчитать, сколько раз специалисты были задействованы, мы видим, что во втором случае их больше. Если

дефект обнаружится на ещё более поздних этапах, когда попадёт к пользователям, могут появиться и потери организации в связи с некорректной работой программы из-за дефекта. Затраты могут вырасти кратно. К примеру, из-за дефекта в программе организация не заключит своевременно договор на миллионы денежных единиц, и это всё будут потери организации.

Из вышеизложенного понимаем: чем позже найден дефект, тем большие затраты понесёт организация. Чем позже дефект обнаружен, тем большее количество людей, систем, инфраструктуры окажется задействовано, а также появляются факторы неисполнения организацией обязательств и недополученная прибыль.

# Кластеризация дефектов

Кластер – совокупность объектов, связанных между собой или объединяемых по наличию у них сходных признаков. В нашем случае объекты – это дефекты, объединённые между собой сходными признаками – функциональными блоками программы.

*Кластеризация дефектов – это выявление функциональных блоков или компонентов программы, которые содержат наибольшее количество дефектов.*

Обычно небольшое количество функциональных блоков программы содержит основную часть всех дефектов, найденных во время тестирования перед выпуском программы, или отвечает за большинство сбоев в работе программы в процессе её эксплуатации. Выявление таких «проблемных» функциональных блоков программы или предсказание, в каких блоках может быть больше дефектов, является важным этапом анализа рисков. Это позволяет сосредоточить усилия специалистов по тестированию именно на проверке критически важных блоков программы. Таким образом, кластеризация дефектов помогает эффективно распределить ресурсы на тестирование, сфокусировав их на тех функциональных блоках программного обеспечения, которые содержат наибольшее количество дефектов. В итоге это повышает качество программного продукта и оптимизирует трудозатраты.

Представим программу, у которой есть два функциональных блока. Первый отвечает за отправку обязательных отчётов в государственные органы. Второй – за создание (заполнение) этих отчётов, которые отправляет первый блок. В процессе эксплуатации программы нам необходимо собирать статистику о возникающих в программе дефектах, а также строить прогнозы, где могут чаще всего появляться дефекты, которые будут критичны для пользователей. На основании прогнозируемых данных и полученной информации мы покрываем

проблемный блок большим количеством проверок, а менее проблемные блоки – меньшим количеством.

К примеру, сейчас мы теоретически можем прогнозировать, что ошибок будет больше в блоке заполнения отчётов, так как в данном процессе используется много полей, справочников и различная логика расчётов различных данных. На основании этого мы создаём максимально возможный набор тестов на проверку функционала данного блока, а для блока отправки отчётов – минимальный набор самых необходимых тестов: проверка отправки отчётов, проверка получения ответов. Ситуация в определённый момент может измениться, поэтому постоянно необходимо анализировать данные.

# Парадокс пестицида

*Парадокс пестицида – это явление, когда одни и те же тесты перестают быть эффективными для обнаружения дефектов в программе.*

Суть парадокса заключается в следующем: если выполнять одни и те же тесты, которые имеют не изменяемые входные данные, снова и снова для проверки программы, со временем при появлении дефектов эти тесты перестанут находить новые дефекты. Подобное происходит потому, что тесты проверяют только те аспекты, на которые были нацелены при их написании. Если ошибки в этих аспектах уже найдены и исправлены, тесты больше не смогут обнаружить новые проблемы.

Это можно сравнить с сельским хозяйством. Если использовать одни и те же пестициды для борьбы с вредителями, со временем они перестанут быть эффективны, так как вредители адаптируются и вырабатывают устойчивость к этим пестицидам.

Для обнаружения новых дефектов в программе нужно менять существующие тесты, изменяя и/или добавляя входные тестовые данные и сценарии, а также разрабатывать новые тесты, проверяющие другие аспекты программы.

Разберём на примере. У нас есть точка **А** на одной стороне леса и точка **Б** на другой стороне леса. Наша задача найти максимальное количество грибов, следуя из точки **А** в точку **Б**. Первый день мы проложили маршрут из одной точки в другую, и дальше изо дня в день мы ходим по нему, пытаясь найти грибы. Через пару дней грибы на нашем пути закончатся, и в последующие дни мы не будем достигать поставленной цели. Мы ничего не найдём, но грибы в лесу будут. Что в этом случае можно сделать? Ежедневно немного изменять наш маршрут, изучая окрестности, но продолжая следовать из точки **А** в точку **Б**. В этом случае, мы будем регулярно находить грибы, если они есть в лесу.

Как это используется в тестировании? Если в программе необходимо проверять сложение, не надо изо дня в день проверять 2 + 2. Ежедневно корректируйте данные: сегодня – 2 + 2, завтра – 15 + 3 и

так далее. Эта методика применима и для более глобальных программ и программ, работающих с другими типами данных.

# Тестирование зависит от контекста

Под контекстом понимается назначение программы, её функциональные возможности, целевая аудитория и область применения. Каждая программа уникальна и имеет свои особенности, поэтому тестирование необходимо подстраивать под конкретный контекст.

Например, если программа отвечает за доступ в корпоративную сеть и хранение конфиденциальных данных, при тестировании надо уделить особое внимание проверке защиты от взлома и уязвимостей. Нужно провести тесты на проникновение, стресс-тесты и другие тесты, проверяющие безопасность.

Для мобильного приложения онлайн-магазина акцент будет делаться на удобство использования, корректную работу функций покупки и оплаты. Здесь важно протестировать все возможные пользовательские сценарии взаимодействия с приложением.

В связи с этим специалист по тестированию перед тестированием программы обязан изучить её и «погрузиться» в её контекст, чтобы понять, на какие аспекты следует обратить основное внимание, какие виды тестирования применить, какие методы проектирования тестов применить при разработке тестов, чтобы обеспечить нужное качество готовой программы. Это позволит максимально эффективно использовать время и ресурсы.

# Заблуждение об отсутствии ошибок

*Заблуждение об отсутствии ошибок – уверенность в том, что в программе полностью отсутствуют ошибки.*

Заблуждение происходит, когда руководители или заказчики ожидают и уверенны, что тестировщики найдут все дефекты в программе.

В этом случае им необходимо помнить о следующих принципах тестирования:

Принцип № 1. Исчерпывающее тестирование недостижимо. Если специалисты по тестированию попытаются провести исчерпывающее тестирование, то на тестирование некоторых программ у них уйдут годы. За это время программы станут неактуальны и могут морально устареть. Никто не будет выделять годы на одну итерацию[17] тестирования программы. Из этого следует, что все дефекты не будут найдены.

Принцип № 2. Тестирование демонстрирует наличие дефектов, а не их отсутствие. Как мы ранее говорили, в программе невозможно найти все дефекты, даже если протестировать все возможные сценарии, ведь некоторые дефекты всё равно могут ускользнуть, так как их проявление зависит от редких и случайных факторов.

В связи со сказанным должно быть понимание, что тестировщики не смогут гарантированно найти все дефекты. Они могут обнаружить наиболее критичные дефекты и снизить вероятность появления дефектов в программе, но полностью исключить их не смогут. Однако это не значит, что не надо пытаться выявлять максимально возможное количество дефектов, способных привести к различным потерям. И самое главное – не надо оправдывать некачественную работу специалистов по тестированию, ссылаясь на данный принцип тестирования.

# Активности процесса тестирования

На данный момент не существует универсального процесса тестирования программ, который подошёл бы для любой программы и которого все придерживаются. Однако есть общий набор тестовых действий и активностей, без которых тестирование не сможет достичь поставленных целей.

Давайте предварительно определим, что такое процесс тестирования.

*Процесс тестирования – это последовательность взаимосвязанных активностей и действий, направленных на достижение целей тестирования.*

Процесс тестирования состоит из следующих групп активностей:

– планирование;
– мониторинг и контроль;
– анализ;
– проектирование тестов;
– реализация тестов;
– выполнение тестов;
– завершение тестирования.

Каждая из указанных групп может состоять из своих отдельных активностей. Многие из перечисленных групп активностей выстроены последовательно, однако они часто выполняются не только последовательно, но и многократно повторяются или выполняются одновременно. К примеру, тестирование (выполнение тестов) текущей версии программы может проводиться одновременно с проектированием тестов новой версии программы и т. д.

# Планирование тестирования

*Планирование тестирования – это комплекс мероприятий и действий, направленных на подготовку и организацию процесса тестирования программного обеспечения.*

Тестирование начинается с планирования. Оно состоит из активностей, которые определяют цели тестирования и подход, с помощью которого мы достигнем целей тестирования. Планирование зависит от используемых видов тестирования, жизненных циклов разработки программы, объёма тестирования, целей, рисков, ограничений, тестируемости программы и доступности ресурсов.

На данном этапе формируется или корректируется план тестирования. При составлении план можно оформить как в виде общего плана тестирования, так и в виде различных уровневых/ целевых планов: план системного тестирования; план приёмочного тестирования; планы для отдельных видов тестирования. Планы тестирования могут быть скорректированы в ходе проведения работ, так как в процессе разработки программы становится доступно больше информации. Также он может быть скорректирован по итогам работ в рамках активности по мониторингу и контролю тестирования. Рассмотрим задачи планирования тестирования.

Анализ требований. Изучаются функциональные и нефункциональные требования к программе, технические задания.

Определение областей тестирования. На основании полученных данных в ходе анализа требований выделяют основные разделы, модули, функции программы, которые необходимо протестировать.

Разработка стратегии и подходов тестирования. Выбирают, какие виды тестирования использовать: функциональное тестирование, тестирование производительности и т. д.

Расчёт объёмов работ. Подсчитывают, сколько времени и ресурсов потребуется на выполнение всех тестов.

Определение сроков. Зная объёмы работ и сроки окончания работ, планируется график тестирования.

Анализ рисков. Выявляются области, которые могут повлиять на успешный результат работ.

Корректировка. План корректируют по мере получения новой информации о программе.

# Мониторинг и контроль тестирования

В процессе тестирования мы должны следить, чтобы процесс шёл, согласно разработанному плану. Для этого существуют такие активности, как мониторинг и контроль.

*Мониторинг тестирования – это процесс сбора данных о ходе тестирования и проверки статуса тестирования.*

Мониторинг тестирования предполагает сбор информации и непрерывное сравнение текущего состояния хода работ с тем, что прописан в плане тестирования, при этом используются показатели (метрики) тестирования, определённые и зафиксированные в плане тестирования.

Примеры показателей, которые могут собираться для оценки:

– процент выполненных работ по подготовке тест-кейсов;

– процент разработанных тест-кейсов;

– процент выполненных работ по подготовке тестовой инфраструктуры;

– метрики выполнения тестов: количество пройденных/не пройденных тестов; количество успешно/не успешно пройденных тестов;

– информация о дефектах: количество обнаруженных и исправленных дефектов; количество дефектов по критичности;

– покрытие требований тест-кейсами при написании тест-кейсов по требованиям;

– информация о выполнении задач, распределении и использовании ресурсов, трудозатратах.

Мониторинг провели и поняли, что процесс тестирования у нас начинает расходиться с ранее разработанным планом. Что делать? В этом случае на помощь приходит контроль тестирования.

*Контроль тестирования – это процесс, в рамках которого проводится разработка и применение комплекса корректирующих мероприятий для возвращения тестирования к ранее*

*запланированным показателям.*

Контроль тестирования, в рамках которого мы принимаем меры, необходим для достижения целей, озвученных в плане тестирования.

Примеры мероприятий, применяемых при контроле тестирования:

– повторная приоритизация тестов при появлении риска срывов сроков тестирования;

– увеличение команды тестирования при появлении риска срывов сроков тестирования;

– изменение графика тестирования из-за доступности или недоступности тестовых стендов или человеческих ресурсов.

Рассмотрим на примере, как могут применяться мониторинг и контроль тестирования. В плане по тестированию обозначено, что оно должно начаться в феврале и закончиться в ноябре. Первого числа каждого месяца в процессе тестирования проводился мониторинг тестирования. Никаких отклонений до сентября не было выявлено. Первого сентября в ходе мониторинга определили, что дата окончания тестирования может быть перенесена на конец декабря. Это недопустимо. В этот момент запускается активность по контролю тестирования. В процессе определили, что в первых числах августа в отпуск ушёл профессиональный программист, а на его место временно взяли неопытного. В ходе своей работы он допускал большое количество ошибок в коде, как итог – все его разработки проходили тестирование по три-четыре раза, что отнимало время, и другие разработки не тестировались. В связи с этим накопилась очередь разработок на тестирование, и за ноябрь их в текущем режиме не успеть протестировать. Были выработаны мероприятия: заменить неопытного программиста на профессионала; увеличить количество специалистов по тестированию. Тестирование завершили в срок. Пользователи получили в ожидаемый срок новую версию программы вовремя.

В процессе тестирования мониторинг и контроль важны, чтобы придерживаться планов и стратегий тестирования и вовремя нивелировать риски.

# Анализ тестирования

*Анализ тестирования – процесс анализа базиса тестирования для определения целей тестирования и тестируемых функций.*

Прежде чем тестировать, надо понять, как и что конкретно тестировать. Другими словами, на этапе анализа тестирования нужно ответить на вопрос «Что тестировать?».

Анализ тестирования состоит из следующих активностей:

Активность № 1. Анализ базиса тестирования. На данном этапе мы анализируем всю имеющуюся у нас документацию и информацию о программе, которую будем тестировать.

Активность № 2. Оценка базиса тестирования для выявления дефектов различных типов, таких как:

– Неоднозначность реализации. В базисе тестирования выявляем неточности, которые не позволяют точно определить, как будет работать реализованный функционал в программе. К примеру, в документе написано программа должна отправлять письма, но не сказано как. По нажатию на кнопку в программе? По нажатию на кнопку на клавиатуре? Автоматически?

– Пропуски в логике реализации. Смотрим, что реализация описана чётко и логично, нет пропусков в её описании, чтобы программисту потом не нужно было самому додумывать, как функционал программы должен работать.

– Несоответствие. Выявляем моменты, когда зафиксированные данные в базисе тестирования не соответствуют фактическому положению дел. К примеру, в программу можно войти по логину и паролю или по номеру телефона, а в базисе тестирования написано «реализовать функцию, которая доступна только тем, кто авторизовался по Email». По Email никто никогда не авторизовывается в программе, и как итог функция никому не будет доступной из-за неверной информации, зафиксированной в требованиях.

– Неточность описания. В базисе тестирования может быть неточное описание. К примеру, описан алгоритм работы с какой-либо функцией, однако указано, что для вызова уведомлений надо нажать на значок звёздочки. На самом деле должно было написано «на значок

колокольчика». При нажатии на значок звёздочки пользователь попадает в избранное.

– Противоречивость. В базисе тестирования может говориться в двух местах об одном и том же, но по-разному. К примеру, в одном месте написано сделать кнопку зелёной, а в другом – сделать кнопку синей. Это противоречие.

– Избыточные утверждения. Пример избыточности утверждения: «Я вчера вечером ужинал. Ужин состоял из супа, горячего блюда и десерта. Я съел суп, потом горячее блюдо, а затем десерт. Еда была вкусной». В этом утверждении излишне упоминается, что ужин состоял из супа, горячего блюда и десерта, а также подробно описывается последовательность еды – суп, потом горячее, потом десерт. Эта информация не несёт дополнительной смысловой нагрузки и является избыточной. Достаточно было бы сказать: «Я ел вчера вечером вкусный суп и десерт» или «Вчера на ужин я съел суп, второе блюдо и десерт».

Активность № 3. Определение функций программы. Это позволит составить полную картину того, что необходимо протестировать, а также позволит провести эффективное и полное тестирование.

Активность № 4. Определение функций, которые нужно проверить при тестировании, и какие из них проверять в первую очередь. Для этого мы анализируем, для чего предназначена программа, какие функции она должна выполнять. Также учитываем другие важные детали. Потом расставляем функции в порядке приоритета, чтобы сначала проверить самые важные.

Найденные в базисе тестирования дефекты передаются аналитикам или ответственным лицам. Остальная информация, полученная в процессе анализа, будет использоваться специалистами по тестированию при проектировании тестов.

Рассмотрим анализ тестирования на примере программы для сложения чисел.

1) Определяем базис тестирования – это требования к программе, которые нам передали аналитики:

| Номер | Название | Требование |
|---|---|---|
| ФТ-1.1 | Появление формы | Форма появляется при запуске программы.<br>Форма появляется по центру экрана. |
| ФТ-1.2 | Масштабирование формы | Размер формы не изменяется (фиксированный). |
| ФТ-1.3 | Перемещение формы | Форма перемещается по экрану, если навести курсор мыши на заголовок формы, зажать левую кнопку мыши и произвести перемещение формы. |
| ФТ-1.4 | Поле ввода первого числа | В поле можно вводить только положительные целые числа.<br>Максимальное количество вводимых и отображаемых символов 4 (четыре).<br>В поле можно вводить информацию с клавиатуры.<br>В поле можно вставлять данные из буфера обмена.<br>Из поля можно копировать данные в буфер обмена. |
| ФТ-1.5 | Поле ввода второго числа | В поле можно вводить только положительные целые числа.<br>Максимальное количество вводимых и отображаемых символов 4 (четыре).<br>В поле можно вводить информацию с клавиатуры.<br>В поле можно вставлять данные из буфера обмена.<br>Из поля можно копировать данные в буфер обмена. |

2) Читаем и анализируем данный базис тестирования – требования.

3) Оцениваем базис тестирования – выявляем различные дефекты в описании, о которых говорилось выше. Дефекты передаём ответственным за написание требований людям.

4) Определяем, какие функции есть у программы на основании той информации, которую получили из базиса тестирования. Пример:

– функционал масштабирование формы;

– функционал очистки полей;

– функционал сложения чисел.

И так далее. Анализ проведён, можно приступать к следующей активности.

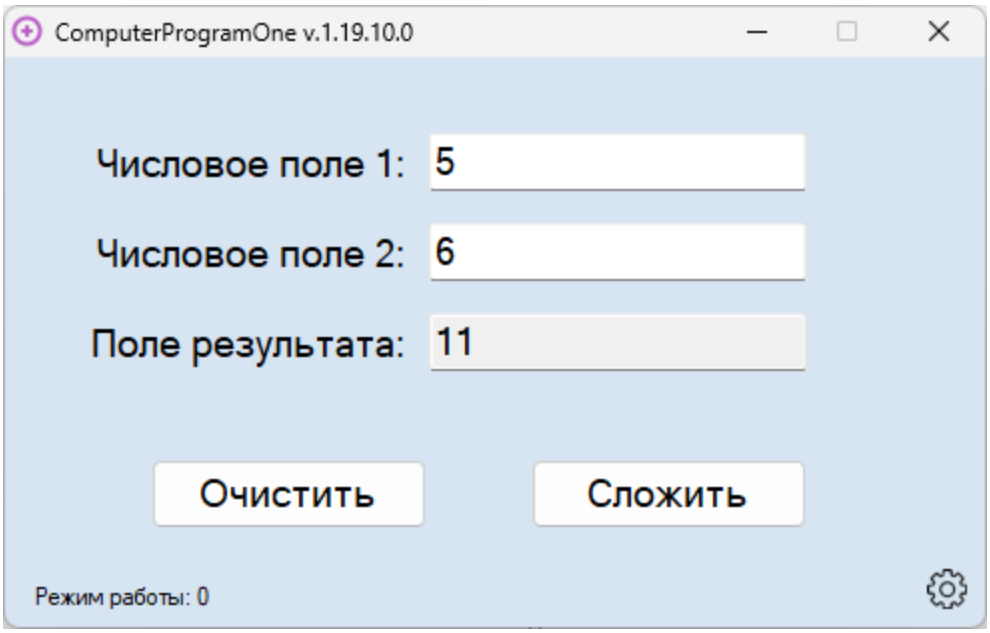
# Проектирование тестов

Во время проектирования тестов создаются тесты в виде тест-кейсов, чек-листов и т. д., которые будут использоваться при тестировании. Анализ тестирования отвечает на вопрос: «Что тестировать?», а проектирование тестов отвечает на вопрос: «Как тестировать?».

*Проектирование тестов – это процесс создания тестовых артефактов, которые будут использоваться для проверки (тестирования) программы.*

На этапе проектирования определяют, с какими параметрами и данными нужно проверять имеющиеся в программе функции. Результатом проектирования тестов будет набор инструкций, собранных в тест-кейсы, которые потом выполнят для проверки программы.

Рассмотрим процесс на примере программы для сложения чисел и спроектируем для неё тесты:




В рамках активности по анализу тестирования мы определили, какие функции есть у программы. Теперь на основании полученной ранее информации мы проектируем тесты и пишем тест-кейсы. Здесь мы фиксируем только заголовки тест-кейсов, а тест-кейсы полностью можете составить самостоятельно по примеру, рассмотренному в предыдущих главах.

Тест-кейсы для проверки функционала масштабирования формы:

– изменение размера формы (форма не должна изменяться).

Тест-кейсы для проверки функционала очистки полей:

– очистка полей ввода только при заполненном первом поле ввода;

– очистка полей ввода только при заполненном втором поле ввода;

– очистка полей при заполненном первом и втором поле ввода и отображении результата сложения.

Тест-кейсы для проверки функционала сложения чисел:

– сложение однозначных чисел;

– сложение двузначных чисел;

– сложение трёхзначных чисел;

– сложение четырёхзначных чисел.

Подробные тест-кейсы специалистом по тестированию подготовлены, и он готов к следующему этапу.

# Реализация тестов

Проектирование тестов отвечает на вопрос «Как тестировать?», в то время как реализация тестов отвечает на вопрос «У нас есть все для выполнения тестов?».

*Реализация тестов – процесс расстановки приоритетов тестов, группировка тестов в наборы, создание тестовых данных, подготовка тестового окружения и написание автоматизированных сценариев тестирования.*

Реализация тестов состоит из следующих активностей:

– формирование тестовых наборов из тест-кейсов и организация наборов таким образом, чтобы тесты выполнялись эффективно;

– расстановка приоритетов тестирования;

– создание автоматизированных сценариев тестирования;

– построение тестового окружения и проверка правильности настройки окружения;

– подготовка тестовых данных и загрузка их в тестовое окружение.

Разберём каждую активность, используя в качестве примера программу для сложения чисел.

Формирование тестовых наборов из тест-кейсов и организация наборов тестов таким образом, чтобы они выполнялись эффективно. Ранее мы создали тест-кейсы:

– сложение трёхзначных чисел;

– изменение размера формы (форма не должна изменяться);

– очистка полей ввода только при заполненном первом поле ввода;

– сложение однозначных чисел;

– очистка полей ввода только при заполненном втором поле ввода;

– сложение двузначных чисел;

– очистка полей при заполненном первом и втором поле ввода и отображении результата сложения;

– сложение четырёхзначных чисел.

Мы видим, что все тест-кейсы смешаны. Что можно сделать, чтобы скорость и эффективность тестирования была выше? Объединить тест-кейсы в наборы. Каким принципом руководствоваться? Набор должен

содержать тест-кейсы из одного функционального блока или области программы. В итоге у нас появится три набора тест-кейсов.

Набор «масштабирования»:

– изменение размера формы (форма не должна изменяться).

Набор «очистки полей»:

– очистка полей ввода только при заполненном первом поле ввода;

– очистка полей ввода только при заполненном втором поле ввода;

– очистка полей при заполненном первом и втором поле ввода и отображении результата сложения.

Набор «сложения чисел»:

– сложение однозначных чисел;

– сложение двузначных чисел;

– сложение трёхзначных чисел;

– сложение четырёхзначных чисел.

Расстановка приоритетов тестирования. Есть три набора тест-кейсов, созданных нами. Необходимо разместить их в порядке приоритета выполнения. Это делается с целью оптимизации процесса. Если выполнение тестирования по каждому набору будет длиться по одной минуте, то на все три набора понадобится три минуты. При поступлении программы в тестирование и наличии всего одной минуты на тестирование нужно выполнить тесты из набора с самым высоким приоритетом. Если не приоритизируем наборы, начнём проводить тестирование по первому попавшемуся в списке, а это неверное поведение. Что для пользователя важнее, чтобы программа складывала числа или чтобы форма программы не масштабировалась (не изменяла размер)? Ответ очевиден: пользователю важно, чтобы программа складывала числа, а масштабированием можно пренебречь. В процессе расстановки приоритетов мы ведём общение с аналитиками, заказчиками, заинтересованными лицами, а также опираемся на свой опыт, чтобы определить важность функционала программы и разместить его списком по убыванию в порядке важности. Это и будет приоритизированный список. В нашем случае разместим наборы в следующем порядке:

Набор № 1. Набор «сложения чисел».

Набор № 2. Набор «очистки полей».

Набор № 3. Набор «масштабирования».

Создание автоматизированных сценариев тестирования. В процессе работы может быть определено, что часть тестов можно проводить с помощью инструментов автоматизированного тестирования, т. е. разработать автоматизированные сценарии тестирования, чтобы они проверяли программу без прямого участия тестировщика. В каком случае это может понадобиться? Если времени на ручную проверку по всем тест-кейсам недостаточно и не успевают всё тестировать вручную, или если есть множество простых и монотонных проверок, или в другом подходящем случае. Тогда определяют, какие тест-кейсы можно автоматизировать, а какие проверять вручную. В нашем случае необходимо учитывать, что на тестирование выделяется всего одна минута, а нам требуется три минуты, чтобы провести все проверки по трём наборам. В данной ситуации на первые два набора пишем автоматизированные сценарии тестирования, а третий проверяем вручную. В итоге один набор будем проверять вручную и в это же время автоматизированные сценарии проверят остальные два набора:

– набор «сложения чисел» будет проверяться автоматизированными сценариями за 5 секунд;

– набор «очистки полей» будет проверяться автоматизированными сценариями за 5 секунд;

– набор «масштабирования» будет проверяться вручную за 1 минуту.

Построение тестового окружения и проверка правильности настройки окружения. В этом случае нам нужно подготовить компьютер, на котором будет запускаться и проверяться программа для сложения чисел. Если бы тестировалась какая-либо крупная информационная система, связанная с другими информационными системами, нам бы пришлось подготовить и настроить десятки или сотни компьютеров (серверов[18]).

Подготовка тестовых данных и загрузка их в тестовое окружение. Наша программа для сложения чисел может работать в нескольких режимах – 8 режимов. От выбора режима зависит, как программа будет реагировать на ввод различных данных. Для смены режима работы программы необходимо заменить ей файл настроек «ComputerProgramOne.exe.config». На этапе подготовки тестовых данных нужно подготовить восемь различных файлов настроек, чтобы в процессе тестирования мы не занимались подготовкой этих данных,

а полностью посвятили себя тестированию программы и использовали уже подготовленные файлы настроек.

# Выполнение тестов

*Выполнение тестов – процесс запуска тестов на исследуемом компоненте или программе, приводящий к определённым результатам.*

В процессе выполнения тестов мы тестируем программу, воспроизводя реальные действия пользователей или внешних систем, с которыми взаимодействует тестируемая программа. Тестирование проводим по тест-кейсам, в которых описаны воспроизводимые действия и ожидаемые результаты или по чек-листам.

Возможно тестирование не по тест-кейсам и чек-листам, а имея только знания о системе и спецификацию требований. Данный вариант применяется в организациях с незрелыми процессами тестирования, когда по разным причинам не проводится анализ, проектирование и реализация тестов, а тестирование провести требуется. Такого следует избегать и немедленно приступать к выстраиванию в организации процессов тестирования.

Выполнение тестов состоит из следующих активностей:

– выполнение тестов вручную или с помощью специализированных инструментов;

– сравнение фактических и ожидаемых результатов;

– фиксирование результатов выполнения тестов (например, пройден – не пройден);

– анализ полученных отклонений для установления их причин;

– составление отчётов о дефектах при наличии дефектов.

Проводим тестирование программы для сложения чисел. Берём все тест-кейсы, которые у нас собраны в тестовые наборы, и начинаем проверять программу по тест-кейсам, т. е. выполнять действия, написанные в тест-кейсе, и сравнивать то, что получили в программе, с тем, что должны получить по описанию в тест-кейсе. Рассмотрим на примере ранее составленного тест-кейса:

| | | |
|---|---|---|
| | **Номер тест-кейса:** | 12 |
| | **Название:** | Сложение однозначных положительных целых чисел |
| | **Предусловия:** | Программа «Computer Program One» запущена |
| | | |
| | **Шаги** | **Ожидаемый результат** |
| 1 | Ввести с клавиатуры в поле «Числовое поле 1» любое однозначное положительное целое число | В поле «Числовое поле 1» отображается введённое число |
| 2 | Ввести с клавиатуры в поле «Числовое поле 2» любое однозначное положительное целое число | В поле «Числовое поле 2» отображается введённое число |
| 3 | Нажать на кнопку «Сложить» | В поле «Поле результата» отображается сумма чисел «Числовое поле 1» + «Числовое поле 2» |
| | | |
| | **Результат проверки:** | **Пройден** |

По тест-кейсу выполним предусловие, т. е. запустим программу. Далее выполним то, что написано в шагах:

1) Введём в поле «Числовое поле 1» число 7, так как можно вводить любое однозначное положительное число. Проверим, что введённое число отображается.

2) Введём в поле «Числовое поле 1» число 9. Проверим, что введённое число отображается.

3) Нажмём на кнопку «Сложить». Проверим, что в «Поле результата» отображается число 16, которое является результатом сложения введённых чисел.

4) Если все ожидаемые результаты совпали, то в «Результат проверки» фиксируем, что тест «Пройден». Если хоть один из ожидаемых результатов не совпал с результатом, полученным в программе, к примеру, сумма равна не 16, а 20, то в «Результате проверки» пришлось бы указать, что тест «Не пройден».

Таким образом проверяем программу по всем имеющимся тест-кейсам, фиксируем по каждому результат проверки, проводим все активности этапа выполнения тестов.

# Завершение тестирования

*Завершение тестирования – активность, во время которой собираются данные обо всех завершённых процессах и прошедших активностях тестирования с целью объединения опыта, фактов, чисел, данных, а также происходит оценка процесса тестирования, включающая в себя подготовку отчёта о тестировании.*

На этапе завершения тестирования мы убеждаемся: всё, что требовалось протестировать, протестировано, собираем все артефакты тестирования, размещаем их в специально предназначенных для этого системах, готовим отчёт о тестировании и анализируем всю проделанную работу, чтобы сделать выводы и, при необходимости, улучшить процесс тестирования.

Завершение тестирования состоит из следующих активностей:

– проверка, что старые отчёты о дефектах закрыты, а новые зафиксированы и отправлены на исправления;

– проверка, что запросы на изменение, в рамках которых проводилось тестирование, закрыты;

– создание сводного отчёта по тестированию для передачи заинтересованным лицам;

– завершение и архивирование тестовых артефактов, тестовых данных, инфраструктуры тестирования;

– передача тестовых артефактов команде сопровождения, другим командам или другим заинтересованным лицам, которые могут извлечь выгоду из их использования;

– анализ полученного опыта и использование собранной информации для дальнейшего улучшения процесса тестирования.

На этом процесс тестирования не завершается, поскольку все активности запускаются повторно, и так до тех пор, пока программа не будет выведена из эксплуатации.

# Динамическое и статическое тестирование

В процессе тестирования специалист по тестированию использует два известных метода тестирования – динамическое и статическое тестирование. В данной главе рассмотрим с вами оба метода тестирования.

*Динамическое тестирование – тестирование, проводимое во время выполнения кода компонента или программы.*

Динамический метод тестирования программы предполагает выполнение кода программы для оценки её поведения и функциональности в реальном времени. Он включает в себя запуск программы с определёнными входными данными и анализ её выходных результатов. Проверка осуществляется с помощью ручного или автоматизированного выполнения заранее подготовленных тест-кейсов.

Что означает фраза «предполагает выполнение кода программы»? Это означает, что если при проведении тестирования выполняется код программы, в этот момент применяется динамический метод тестирования. Когда может выполняться код программы? В процессе работы программы. Т. е. если программа запущена и работает, в этот момент на компьютере выполняется код самой программы.

Данный метод можно назвать самым популярным, поскольку он используется всеми тестировщиками, которые тестируют программное обеспечение. Большинство видов тестирования, рассматриваемых в последующих главах, относятся к динамическому методу тестирования.

Ранее мы тестировали программу для сложения чисел. В момент, когда мы её запустили, и программа отобразила свой интерфейс, начал выполняться код программы на компьютере, и все последующие тесты по сложению чисел и очистке полей проводились с применением

динамического метода тестирования, хотя в тот момент вы этого не подозревали и об этом не задумывались.

*Статический метод тестирования – тестирование, направленное на проверку артефактов разработки программного обеспечения, таких как требования, дизайн или программный код, проводимое без исполнения этих артефактов.*

Статический метод тестирования программного обеспечения не включает в себя выполнение кода программы. Вместо этого он фокусируется на анализе программного кода, документации и других артефактах без их выполнения.

Статический метод тестирования может быть как ручным (рецензирование), так и автоматизированным (статический анализ). Оба указанных вида оценивают работу тестируемого продукта без фактического исполнения кода или работы тестируемой программы и будут нами рассмотрены в последующих главах.

Данный метод тестирования менее популярен по разным причинам: у компаний не хватает времени и ресурсов; его считают менее важным; в компании не развита практика тестирования.

Статический метод тестирования по умолчанию используется при исследовании следующих артефактов:

– спецификации требований;

– пользовательские истории;

– архитектурные спецификации и схемы;

– программный код. Статический анализ кода и обзор кода (code review[19]);

– тестовая документация, включая тест-планы, тест-кейсы, автоматизированные сценарии тестирования;

– руководства пользователя и прочая документация.

Могут исследоваться и другие неперечисленные здесь артефакты.

В разделе, который посвящён активностям процесса тестирования, описана активность «анализа тестирования». В её рамках проводится анализ и оценка базиса тестирования для выявления дефектов. И в этот момент используется статический метод тестирования, так как в

процессе анализа требований, а также другой документации не происходит выполнения кода программы, ведь специалисты работают с документацией.

Если применять статический метод к программе по сложению чисел, в этом случае мы тестируем не саму программу, а документ, содержащий требования к программе. В нём ищем ошибки и несоответствия. Также программисты могут друг за другом перепроверять написанный код программы. В обоих случаях используется статический метод тестирования.

Проводя тестирование, невозможно выбрать какой-то метод тестирования в конкретной ситуации. Они используются по умолчанию. Если код программы выполняется в процессе тестирования, по умолчанию применяется динамический метод. Если код программы не выполняется, по умолчанию будет использоваться статический метод.

# Уровни тестирования

*Уровень тестирования – это группа активностей, реализующая процесс тестирования программы, которая находится на конкретном этапе разработки.*

Для большего понимания попробуем упростить определение. Уровень тестирования – это процесс тестирования программы на определённом этапе разработки программы. Что подразумевается под этапами? Рассмотрим на примере и перечислим некоторые этапы разработки программы.

Этап № 1. Программист начал разрабатывать программу и разработал один компонент программы – часть программы, которая не имеет визуальной составляющей, а представляет собой программный код, выполняющий определённую функцию программы.

Этап № 2. Программист разработал несколько функциональных компонентов программы, которые могут взаимодействовать между собой.

Этап № 3. Программист разработал программу полностью, и она передана специалистам по тестированию на тестирование.

Этап № 4. Специалист по тестированию протестировал программу, программист исправил все найденные дефекты, и программу передали заказчикам, чтобы они её приняли.

На всех перечисленных этапах программа тестируется, и тестирование программы на определённом этапе реализации называют «уровнем тестирования».

Рассмотрим четыре уровня тестирования:

1) Компонентное тестирование.
2) Интеграционное тестирование.
3) Системное тестирование.
4) Приёмочное тестирование.

# Компонентное тестирование

*Компонентное тестирование – уровень тестирования отдельных компонентов программы: модули программы; объекты; классы; функции.*

Компонентное тестирование также известно как «модульное тестирование». Вы часто будете слышать оба названия.

При нём тестирование проводится, вызывая программный код программы, который необходимо проверить. Компонентное тестирование фокусируется на компонентах, которые могут быть проверены отдельно.

На данном уровне мы проверяем отдельные части (компоненты) программы. Для этого запускаем программный код только этих частей, а не всей программы сразу. Так можно детально проверить работоспособность каждой отдельной части.

Рассмотрим несколько примеров:

Пример № 1. Проверка модуля ввода данных. Запускается код этого модуля и проверяется, правильно ли он считывает введённые пользователем значения.

Пример № 2. Тестирование базы данных. Проверяется, корректно ли работает программный код для добавления, удаления и обновления записей в базе данных.

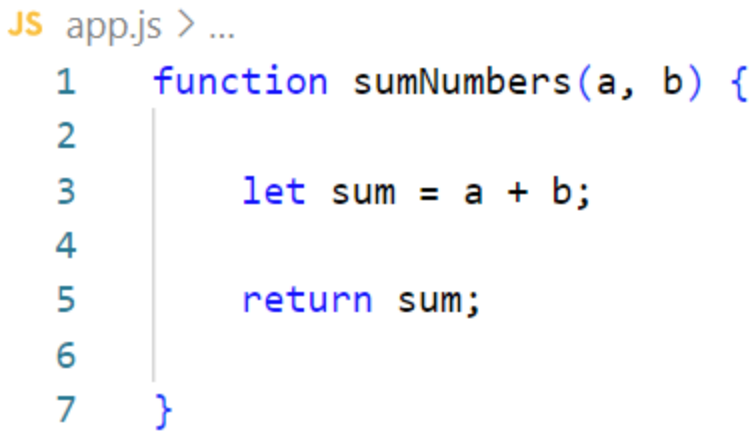
Пример № 3. Тестирование модуля расчётов. Запускается код этого модуля с разными тестовыми данными и проводится наблюдение, правильно ли он выполняет необходимые операции и возвращает ли ожидаемые результаты.

Пример № 4. Проверка интерфейса. Тестируется отдельный код управления интерфейсом – правильно ли он отображает элементы, реагирует ли на действия пользователя.

Компонентное тестирование требует доступа к исходному коду и, как правило, выполняется программистом, написавшим этот код. Программисты могут чередовать процессы создания компонентов и поиска/исправления ошибок. Иногда они пишут юнит-тесты[20] сразу после завершения работы над компонентом.

При компонентном тестировании проверяются как функциональные, так и нефункциональные характеристики отдельных компонентов программы. К функциональным тестам относится проверка логики и правильности вычислений, а в нефункциональные тесты входят, например, тесты поиска утечек памяти.

Теперь рассмотрим данный уровень тестирования на примере программы для сложения чисел. Программист написал программный код, который описывает метод сложения чисел:




Это один из компонентов (модулей) программы. Теперь программисту необходимо проверить, что данный метод будет отрабатывать корректно. Для этого он пишет дополнительный программный код, который проверит работу метода сложения чисел, вызывая его. Это и есть юнит-тест. В дальнейшем будет создан метод очистки полей, и для его проверки напишется ещё один юнит-тест, который уже проверит работу метода очистки полей. В будущем, внося какие-либо изменения в код программы, программист будет запускать все ранее написанные юнит-тесты, чтобы они проверяли, что при изменении какого-либо участка кода в программе другие участки кода работают корректно.

# Интеграционное тестирование

*Интеграционное тестирование – уровень тестирования, проводимого с целью проверки интеграционных интерфейсов и взаимодействия между интегрированными компонентами или программами.*

Интеграционное тестирование направлено на проверку взаимодействия между различными частями программы. При таком тестировании мы проверяем, как взаимодействуют между собой отдельные компоненты или модули одной программы. Также интеграционное тестирование направлено на проверку взаимодействия между различными программами, которые обмениваются данными через специальные интерфейсы. Цель интеграционного тестирования состоит в том, чтобы проверить работу взаимосвязей и взаимодействия между различными частями программы или между различными программами.

Если речь идёт о тестировании взаимодействия между компонентами внутри одной программы, это называется «компонентным интеграционным тестированием». Оно фокусируется на проверке взаимодействий и интерфейсов между интегрированными компонентами. Такое тестирование проводится после тестирования отдельных компонентов.

Если тестируется взаимодействие между различными программами или информационными системами, это называется «системным интеграционным тестированием». Оно фокусируется на проверке взаимодействий и интерфейсов между разными программами. Также в его задачи может входить тестирование взаимодействия с внешними программами, например программами сторонних организаций.

Как и при компонентном тестировании, так и при интеграционном тестировании автоматизированные тесты позволяют нам быть уверенными в том, что изменения не повредили существующие интеграционные интерфейсы, компоненты или программы.

Целью компонентных и системных интеграционных тестов является проверка взаимодействия интегрируемых компонентов/программ, а не

отдельных элементов. Так, при интеграции модуля **А** и модуля **В** одной программы тесты должны фокусироваться на проверке их взаимодействия, а не функциональности каждого модуля по отдельности – последнее проверяется на этапе просто компонентного тестирования. Аналогично, интегрируя программу **Б** с программой **Д**, целью тестирования будет проверка их взаимодействия, а не отдельных возможностей каждой программы. Функционал отдельных программ проверяется на этапе системного тестирования, о чем поговорим далее.

Тестирование интеграции на уровне компонентов выполняется программистами, в то время как тестирование интеграции на уровне программ выполняется специалистами по тестированию.

В крупных организациях информационные системы имеют сложную внутреннюю интеграцию и интеграцию со множеством других программ организации. Допустим у интернет-магазина есть несколько внутренних методов (модулей):

– оплата заказа;

– проверка остатков на складе.

При попытке пользователя оплатить заказ на сайте модуль оплаты обращается к модулю проверки остатков на складе и «спрашивает» у него: «Есть товар на складе?», модуль проверки остатков проверяет это и сообщает модулю оплаты заказа о наличии или отсутствии товара. Пользователь этого не видит, так как всё взаимодействие модулей происходит на уровне программного кода. В процессе разработки программист как раз проверяет взаимодействие данных модулей в рамках интеграционного компонентного тестирования.

После оплаты заказа интернет-магазин отправляет данные о заказе другой программе, которая отвечает за доставку и которой пользуются специалисты по доставке. В свою очередь программа доставки, после доставки заказа покупателю, сообщает интернет-магазину, что заказ доставлен. Как мы видим, здесь есть интеграция между интернет-магазином и программой доставки. Когда программист дорабатывает одну из них, специалист по тестированию проверяет, что интеграция между программами работает. Для этого ему не нужно видеть код программ, достаточно оформить заказ в интернет-магазине и проверить, что информация о заказе появилась в программе доставки. После этого отметить в программе доставки заказ как доставленный и

проверить, что информация о доставке заказа передана в интернет-магазин. Он провёл системное интеграционное тестирование.

# Системное тестирование

*Системное тестирование – уровень тестирования программы в целом с целью проверки того, что она соответствует установленным требованиям.*

На каком уровне тестирования больше всего работают тестировщики? На уровне системного тестирования. Тут специалисты тестируют готовые к работе программы и проводят основные виды тестирования.

Системное тестирование направлено на проверку работоспособности программы в целом, оценивая её функциональные и нефункциональные характеристики. При этом уделяется внимание не только отдельным функциям, но и сквозным бизнес-процессам[21], а также поведению программы при выполнении различных задач. Системное тестирование должно быть сфокусировано на общем функциональном и нефункциональном поведении программы с точки зрения конечных пользователей. При системном тестировании следует использовать наиболее подходящие методы проектирования тестов, чтобы проверить конкретные аспекты тестируемой программы, которые ей присущи.

При проведении системного тестирования у тестировщиков могут возникать трудности. Во-первых, неполные или ошибочные требования в спецификациях приводят к непониманию специалистами желаемого поведения программы. Отсутствие чётких требований к программе значительно осложняет работу специалистов по тестированию. Не имея полного представления о правильном и ожидаемом поведении системы, они не могут правильно интерпретировать результаты тестирования. Это приводит к риску принятия неверных решений о корректности работы программы. Для исключения подобной ситуации крайне важно на этапе проектирования тестов уточнить все неясные моменты в требованиях. Это позволит выработать единое понимание ожидаемой функциональности у всех заинтересованных сторон – программистов, тестировщиков, заказчиков и конечных пользователей. Во-вторых, из-

за этого при тестировании получают как ложноположительные, так и ложноотрицательные результаты. Это отнимает время и снижает эффективность выявления реальных дефектов.

Примеры системного тестирования:

Пример № 1. При тестировании браузера системное тестирование включает в себя проверку его основных функций, таких как открытие веб-сайтов, вкладок, закладок, загрузка файлов и безопасность (например, блокировка вредоносных сайтов).

Пример № 2. При тестировании офисных программ (например, Microsoft Word или Excel) системное тестирование оценивает способность создавать, редактировать и сохранять документы или таблицы, а также взаимодействие с другими программами, такими как электронная почта.

Пример № 3. При тестировании мессенджера для обмена сообщениями системное тестирование может включать проверку отправки и получения сообщений, создания групповых чатов, отправки медиафайлов и защиту конфиденциальности сообщений.

Пример № 4. При тестировании игр системное тестирование оценивает работу игрового движка, графику, управление, сохранение игры и групповой режим игры.

# Приёмочное тестирование

Программисты разработали программу, специалисты по тестированию провели тестирование программы, и она готова к передаче заказчику. Как и любой товар программа должна быть проверена заказчиком, чтобы убедиться, что она работает так, как ожидалось, и выполняет требуемые от неё функции. Если программа не удовлетворяет требованиям заказчика и не соответствует заявленным требованиям, она возвращается на доработку. Проверка программы заказчиком проводится в рамках приёмочного тестирования.

Взаимодействие между заказчиками и разработчиками программ может быть организовано двумя способами. Первый подразумевает, что обе стороны находятся в одной организации. Например, бизнес-подразделение инициирует создание корпоративной информационной системы, а ИТ-подразделение берёт на себя её разработку. Второй способ, когда заказчик и разработчик представлены разными организациями, работающими по договору. Организация-заказчик поручает сторонней организации выполнить проект по созданию и внедрению необходимой программы на возмездной договорной основе. Таким образом, модель взаимоотношений между заказчиком и разработчиком программы может быть как внутренней, так и внешней в зависимости от потребностей и возможностей организации-заказчика.

*Приёмочное тестирование – это проверка соответствия готовой программы требованиям и ожиданиям заказчика перед вводом её в эксплуатацию.*

Цели приёмочного тестирования:

– продемонстрировать заказчикам и заинтересованным лицам качество программы в целом;

– проверить, что программа завершена и будет работать как ожидалось;

– проверить, соответствует ли функциональное и нефункциональное поведение программы установленным требованиям.

По результатам приёмочных испытаний заказчик принимает решение о готовности программы к промышленной эксплуатации и передаче конечным пользователям. Таким образом, приёмочное тестирование является «воротами» между разработкой и внедрением программы в эксплуатацию. При проведении приёмочного тестирования возможно выявление отдельных дефектов программы, но их поиск не является целью данного процесса. Главная задача приёмочного тестирования – подтвердить соответствие разработанного решения изначальным требованиям и ожиданиям заказчика. Приёмочное тестирование, как и системное тестирование, обычно фокусируется на поведении и возможностях программы в целом.

Оно имеет несколько форм, о которых сейчас и поговорим. Формы приёмочного тестирования:

– пользовательское приёмочное тестирование;
– эксплуатационное приёмочное тестирование;
– контрактное приёмочное тестирование;
– нормативное приёмочное тестирование;
– бета-тестирование.

# Пользовательское приёмочное тестирование

*Пользовательское приёмочное тестирование – это процесс непосредственной оценки пользователями готовности программы к промышленной эксплуатации.*

Пользовательское приёмочное тестирование фокусируется на оценке удобства и практичности использования информационной системы целевыми пользователями. Оно проводится либо в реальном рабочем окружении, либо в максимально приближенных к работе условиях. Главная цель – убедиться, что конечные пользователи могут эффективно использовать программу для решения своих ежедневных задач, а также подтвердить соответствие функциональности системы изначальным требованиям и ожиданиям заказчика. Приёмочные испытания позволяют выявить все имеющиеся сложности, ограничения и риски практического использования программы в рабочих процессах. Это даёт возможность своевременно устранить их перед запуском в эксплуатацию и свести к минимуму последующие накладные расходы организации.

Основные характеристики пользовательского приёмочного тестирования:

– проводится ключевыми конечными пользователями или представителями заказчика;

– фокус на проверке соответствия функционала программы требованиям;

– проверяются типовые и критичные к бизнесу сценарии работы с программой;

– оценивается не только функционал, но и удобство/простота работы пользователей с программой;

– выявляются дефекты с точки зрения реальных пользователей;

– результаты учитываются для окончательного решения о запуске программы в промышленную эксплуатацию.

Таким образом, пользовательское тестирование является одним из важных этапов приёмки программы в части соответствия бизнес-ожиданиям. Оно проводится конечными пользователями со стороны

заказчика, т. е. теми пользователями, которые в дальнейшем будут работать с программой.

## Эксплуатационное приёмочное тестирование

*Эксплуатационное приёмочное тестирование – это финальная стадия приёмки программы перед выводом в промышленную эксплуатацию, с целью удостовериться, что программа полностью готова к передаче в опытную эксплуатацию[22] и последующей промышленной эксплуатации.*

Главная задача эксплуатационного приёмочного тестирования – убедиться, что технические специалисты и администраторы программ смогут обеспечить её стабильное функционирование для пользователей. При этом программа должна сохранять работоспособность в заявленных пределах даже при возникновении сбоев, пиковых нагрузок или других нештатных ситуаций.

Основные характеристики эксплуатационного приёмочного тестирования:

– проводится в окружении, максимально приближенном к реальным условиям и нагрузкам;

– тестируется работа в комплексе всех компонентов программы;

– проверяется не только функционал, но и показатели производительности, отказоустойчивости, восстанавливаемости;

– проверяются требования по информационной безопасности;

– специалисты максимально эмулируют поведение реальных пользователей.

Эксплуатационное приёмочное тестирование проводится администраторами, которые будут обслуживать программу, а также техническими специалистами со стороны заказчика.

## Контрактное приёмочное тестирование

*Контрактное приёмочное тестирование – это независимая экспертиза готовности программы к вводу в эксплуатацию, которая проводится третьей стороной – экспертной тестовой организацией.*

В этом случае организация-заказчик обращается к организации-тестировщику, чтобы она провела приёмочное тестирование программы, разработанной организацией-разработчиком.

Основные характеристики контрактного приёмочного тестирования:

– проводится на основании отдельного договора между заказчиком и тестовой организацией;

– направлено на независимую проверку соответствия программы требованиям договора между заказчиком и разработчиком;

– базируется на тест-кейсах, разработанных специалистами по тестированию организации на основе анализа требований и технической документации;

– результаты тестирования оформляются в виде отчёта с перечнем дефектов, замечаний и рекомендаций по исправлению недостатков;

– заключение специалистов по тестированию учитывается при окончательной приёмке заказчиком готовой программы и принятии решения о вводе её в эксплуатацию.

## Нормативное приёмочное тестирование

*Нормативное тестирование – проверка соответствия разработанной программы обязательным государственным или отраслевым стандартам, техническим регламентам.*

При проведении данного вида тестирования проверяется, что программа соответствует обязательным стандартам, нормам и правилам, которые предъявляются к подобным программам. К примеру, есть программы криптографической защиты информации, которые шифруют данные. Если они используются в государственных органах, к ним государственными органами и стандартами предъявляются специальные требования, которые обязательно должны быть соблюдены при разработке этих программ. На этапе нормативного приёмочного тестирования проверяют соблюдение в программе этих стандартов.

Рассмотрим на примере программы для сложения чисел. Допустим, она используется в государственных органах, а согласно государственным стандартам программы для сложения чисел должны хранить свои настройки в специальном файле настроек и в

определённом формате. В этом случае после разработки программы мы её передаём экспертам, чтобы они её проверили и дали заключение о соответствии программы стандартам. Проводя нормативное приёмочное тестирование, они проверят, что программа соответствует стандартам и действительно хранит свои настройки в специальном файле настроек и в том формате, который описан в стандарте.

Основные характеристики нормативного приёмочного тестирования:

– проводится на основе государственных стандартов, отраслевых регламентов, технических условий и других нормативных документов;

– ориентировано на проверку надёжности, безопасности, соответствия унифицированным требованиям по разработке программ;

– может проводиться как сторонними экспертами, так и представителями регулирующих органов;

– позволяет получить официальное разрешение на запуск программы в эксплуатацию или осуществить обязательную сертификацию программы;

– подтверждает соответствие программы необходимым нормативно-правовым актам.

# Бета-тестирование

*Бета-тестирование – это предварительные испытания программы группой потенциальных пользователей на их реальных рабочих задачах с целью сбора отзывов и пожеланий перед окончательной сдачей в эксплуатацию.*

Особенности бета-тестирования:

– проводится на этапе, когда разработка системы почти завершена;

– участвует ограниченная группа предполагаемых пользователей (заказчиков);

– пользователи тестируют программу на своих реальных задачах и данных;

– оценивается соответствие функционала программы требованиям и удобство использования;

– выявляются дефекты и недочёты перед выходом финальной версии;

– результаты тестирования учитывают при финальной приёмке и принятии решения о запуске в промышленную эксплуатацию.

Таким образом бета-тестирование позволяет снизить риски внедрения системы и обеспечить лучшее соответствие ожиданиям конечных пользователей.

Рассмотрим на примере программы для сложения чисел. Допустим, разработчики разработали программу и реализовали функционал сложения таким образом, что программа складывает числа по нажатию на клавиатуре на клавишу «Enter». Кнопку «Сложить» в программе не создавали, так как посчитали её лишней. После этого у них появились сомнения в этом, и они хотят проверить, как пользователи отнесутся к их решению. В данном случае они передали программу пользователям и попросили поработать с ней и предоставить обратную связь. После определённого времени пользования программой пользователи сообщили, что программой крайне неудобно пользоваться. Если не будет кнопки «Сложить», они скорее всего выберут программу от другой организации-разработчика. Прошло бета-тестирование. Разработчики учли эти пожелания и риски отказа от их программы и реализовали в ней кнопку «Сложить».

Есть сторонники того, что бета-тестирование не является формой приёмочного тестирования. Попробую привести аргументы в пользу того, что бета-тестирование – форма приёмочного тестирования.

Аргументы:

1) Проводится на финальных стадиях тестирования перед установкой в продуктивное окружение.

2) Участвуют конечные пользователи (заказчики).

3) Направлено на выявление соответствия программы требованиям и ожиданиям пользователей.

4) Ориентировано на рабочие задачи и реальные данные.

5) Оценивает готовность системы к промышленной эксплуатации.

6) Результаты учитываются при принятии решения о запуске системы.

Исходя из этого, бета-тестирование решает те же задачи, что и классическое приёмочное тестирование конечными пользователями. Поэтому его можно считать одной из разновидностей приёмочного тестирования ПО.

# Типы тестирования

*Тип тестирования – это группировка процессов тестирования по определённому признаку для проверки заданных свойств программы или её компонентов согласно конкретной цели.*

Тип тестирования определяет общий фокус набора тестов исходя из конкретной цели – проверить функционал, совместимость, производительность и т. д. В рамках проверки программы может применяться комплекс разных типов тестирования.

Что подразумевается под группировкой процессов тестирования? Например, функциональное тестирование включает в себя действия по проверке соответствия функций программы требованиям. Нефункциональное тестирование включает в себя действия по проверке соответствия свойств программы, которые не относятся к её функциям. Представьте, что мы тестируем программу, чтобы убедиться, что она работает правильно. Для этого её необходимо всесторонне проверить. В этом случае мы проводим, к примеру, следующие проверки:

Проверка № 1. Проверка функций. Запускаем все меню, кнопки, опции и смотрим, работают ли они так, как задумано.

Проверка № 2. Проверка скорости работы или удобства использования. Измеряем, как быстро запускается программа, и оцениваем, как просто пользоваться программой, все ли понятно для обычного пользователя.

Как видим, для полной проверки программы мы её тестируем с разных сторон, обращая внимание на разные параметры. Они сами по себе сгруппированы и представляют собой определённые типы или группы.

# Функциональный тип тестирования

*Функциональный тип тестирования – тестирование функциональности компонента или программы, основанное на анализе спецификации компонента или программы.*

Основная цель функционального типа тестирования – подтвердить наличие в программе всей необходимой логики и функциональности, которые в неё закладывались.

Функциональные требования, на основании которых мы проводим тестирование, могут быть задокументированы в виде спецификаций, пользовательских сценариев, случаев использования и других описаниях желаемого поведения программы. А могут изначально и не фиксироваться в явном виде. В любом случае функциональные тесты отвечают на вопрос – «что именно должна делать программа с точки зрения логики, процессов и решаемых задач?». То есть проверяют соответствие функционала программы изначальным целям её создания. Функциональные проверки выполняются на всех уровнях тестирования.

Разработка функциональных тестов и их выполнение может потребовать специальных знаний конкретной предметной области, для автоматизации которой создаётся программа. Например, для тестирования программы по геологическому моделированию в нефтегазовой отрасли нужна экспертиза из этой сферы. А для тестирования компьютерных игр или развлекательных приложений требуется понимание принципов игрового дизайна и механики. Без глубоких знаний предметной области сложно корректно спроектировать функциональные тесты для полной проверки функционала программы.

Приведём пример проверок программы для сложения чисел, которые относятся к функциональному типу тестирования:

– проверка сложения чисел;
– проверка очистки полей;
– проверка ввода данных в поля ввода.

# Нефункциональный тип тестирования

*Нефункциональный тип тестирования – тестирование свойств компонента или программы, не относящихся к функциональности, таких как надёжность, эффективность, переносимость, удобство использования и т. д.*

Нефункциональный тип тестирования отвечает на вопрос – «насколько хорошо работает программа?».

Нефункциональный тип тестирования не фокусируется напрямую на проверке бизнес-функций программы. Вместо этого позволяет оценить более глобальные, интегральные характеристики[23] качества программ. К ним относятся: надёжность (устойчивость к ошибкам и сбоям), производительность (быстродействие при различной нагрузке), удобство использования, масштабируемость, безопасность данных и другие характеристики, влияющие на общее восприятие программы.

Нефункциональные требования во многом формируют положительный или отрицательный опыт конечных пользователей от эксплуатации программы. Поэтому при разработке программ важно уделять пристальное внимание тестированию именно этих интегральных качественных характеристик программ. От них напрямую зависит успешность и репутация продукта на рынке.

Для качественного нефункционального тестирования может потребоваться глубокая специальная экспертиза конкретных технологий и предметных областей. Например, для оценки защищённости программы важно знать типичные уязвимости используемых языков программирования. Для тестирования удобства использования критично представлять потребности и возможности целевых групп пользователей программы. Без понимания архитектурных рисков, специфики отрасли и особенностей аудитории сложно корректно спроектировать нефункциональные тесты и оценить достаточность мер по обеспечению производительности, безопасности и других интегральных характеристик программы.

Приведём пример проверок программы для сложения чисел, которые относятся к нефункциональному типу тестирования:

Проверка № 1. Проверка удобства использования программы в работе. Если не будет кнопки «Сложить» и сложение возможно только по клавише «Enter» на клавиатуре, то программой неудобно пользоваться.

Проверка № 2. Производительность – скорость работы. Если программа после запуска отображает свой интерфейс через минуту и выводит результат сложения через две минуты, то производительность у неё низкая.

# Виды тестирования

Видов тестирования программ множество. В общем смысле, виды тестирования представляют собой различные «срезы» процесса проверки качества программ. Каждый такой «срез» позволяет оценить определённый аспект или свойство программы. То есть это различные «точки зрения» на процесс проверки. Комбинируя виды тестирования, можно всесторонне оценить качество сложной программы.

*Вид тестирования – это классификация процесса тестирования программного обеспечения по некоторому конкретному признаку, аспекту или цели.*

Рассмотрим основные виды тестирования программного обеспечения.

# Дымовое тестирование

Когда программу передают специалистам по тестированию, прежде чем начать проводить всевозможные тесты, им необходимо убедиться, работает ли программа и выполняет ли свои базовые функции. Для этого проводится дымовое тестирование.

*Дымовое тестирование – вид тестирования, направленный на предварительную проверку базового, критичного функционала программы перед более глубоким тестированием. Проводится с целью убедиться, что базовые функции программы в целом работают.*

Данный вид тестирования в первую очередь направлен на проверку готовности разработанной программы к проведению более расширенного тестирования и определения общего состояния качества программы. То есть в процессе дымового тестирования проверяются только ключевые функции. Это и есть его задача. Программу, не прошедшую дымовое тестирование, не имеет смысла отдавать на более глубокое тестирование, так как у неё не работает основной функционал.

Представим, что вышла очередная версия программы для мобильного банка. Проводя дымовое тестирование, проверяем:

– запускается ли программа;

– можем ли зайти под своим логином и паролем;

– отображается ли баланс счёта после входа;

– можем ли перевести деньги на другую карту банка.

Допустим, при проведении дымового тестирования специалист по тестированию не смог войти в программу. Раз так, он не может проверить остальные функции программы: перевод денег, получение уведомлений. Любая дальнейшая попытка выполнить другие тесты будет пустой тратой времени и усилий. В этом случае разумно вернуть на доработку программу, которая не прошла дымовое тестирование.

Заглянем немного в историю. Название «дымовое тестирование» пришло из мира электроники и аппаратного обеспечения. Когда производят новую печатную плату или железное устройство, одним из первых тестов является проверка, что при включении оно вообще

заработает, не «задымится» и не сгорит. То есть после сборки проверяют, есть ли признаки работоспособности устройства – светится индикатор, крутятся вентиляторы, происходит загрузка системы. Этот подход, позволяющий сначала убедиться в принципиальной работоспособности, а уже потом проводить глубокие и длительные тесты, перекочевал в сферу тестирования программного обеспечения.

# Тестирование сборки

Чтобы получить готовую программу из исходного набора файлов, содержащих программный код, написанный программистом, необходимо программу собрать из исходных файлов. В итоге получается законченная программа, которая уже может быть запущена на выполнение.

*Сборка программы из исходных файлов – это процесс компиляции исходного программного кода в выполняемые файлы программы.*

После того как собрали воедино исходные файлы с использованием специализированных инструментов, мы получили программу, которую можно запускать и работать с ней. В этом случае мы имеем конкретную сборку программы.

*Сборка программы – подготовленный для использования информационный продукт – программа. Чаще всего сборкой является исполняемый файл (двоичный файл), содержащий исполняемый код программы или библиотеки.*

Т. е. сборка программы – это программа, собранная из исходных файлов, с которой можно работать. Имея сборку программы, мы можем её тестировать, проводя тестирование сборки.

*Тестирование сборки – вид тестирования, направленный на проверку работоспособности основных функций и стабильности новой сборки программы.*

Данный вид тестирования выполняется для каждой новой сборки перед передачей её в тестирование. Обычно данный вид тестирования автоматизируется. При обнаружении дефектов сборка отклоняется и возвращается на доработку.

Внимательно присмотревшись, видим, что тестирование сборки и дымовое тестирование нацелены на одно и то же. Почему тогда они

называются по-разному, в то время как направлены на проверку одного и того же – работоспособности основных функций?

Приведём два основных отличия:

1) Тестирование сборки проводится до передачи в тестирование. Это тестирование проводят с использованием автоматизированных тестов, чтобы принять решение, передавать сборку на тестирование или вернуть её на доработку.

2) Дымовое тестирование проводится после передачи в тестирование, чтобы принять решение о проведении дальнейших видов тестирования или возврате программы на доработку.

Рассмотрим на примере программы для сложения чисел.

Тестирование сборки:

1) Программа автоматически собирается из исходных файлов в сборку.

2) Инструмент, собравший программу, запускает инструмент, отвечающий за автоматизированные тесты, которые начинают проверки:

– проверка запуска программы;

– проверка функции сложения;

– проверка функции очистки полей.

3) Если тесты прошли успешно, сборка передаётся в тестирование.

4) Если тесты выявили дефекты, сборка возвращается на доработку.

Дымовое тестирование:

1) Специалист получает программу после тестирования сборки и начинает проверки:

– проверка запуска программы;

– проверка функции сложения;

– проверка функции очистки полей кнопкой «Очистить»;

– проверка функции очистки полей с помощью клавиатуры;

– смена режима работы.

2) Если тесты прошли успешно, начинают проведение других видов тестирования.

3) Если тесты выявили дефекты, программа возвращается на доработку.

# Функциональное тестирование

Данный вид тестирования одноимённый имеющемуся типу тестирования, поэтому стоит помнить, что есть функциональный тип и вид тестирования.

*Функциональное тестирование – вид тестирования программного обеспечения, направленный на проверку реализованных функциональных требований и способности программы решать задачи пользователей.*

Основная цель функционального тестирования – подтвердить, что разрабатываемая программа реализует весь необходимый набор функций, соответствующий изначальным требованиям и ожиданиям заказчика. Иными словами, функциональные тесты призваны дать уверенность, что программа в полной мере обладает всей бизнес-логикой и выполняет все процессы, критичные для целевых пользователей программы. Это позволяет подтвердить соответствие реализованного решения задачам предметной области.

Рассмотрим функциональное тестирование на примере. Допустим, мы тестируем программу для сложения чисел:

Проверка № 1. Проверяем сложение однозначных чисел – вводим числа в первое поле и во второе поле, нажимаем на кнопку «Сложить». Должен отработать функционал сложения чисел: сложить числа и вывести результат в поле вывода результата.

Проверка № 2. Проверяем сложение двузначных чисел – вводим числа в первое поле и во второе поле, нажимаем на кнопку «Сложить». Должен отработать функционал сложения чисел: сложить числа и вывести результат в поле вывода результата.

Проверка № 3. Проверяем очистку полей – нажимаем на кнопку «Очистить». Поля ввода и поле отображения результатов должны быть очищены.

Проверка № 4. Проверяем сворачивание программы в панель задач – нажимаем на кнопку сворачивания программы. Программа должна свернуться в панель задач операционной системы.

То есть мы запускаем основные функции программы по реальным сценариям и смотрим, работают ли они так, как нужно. Это и есть функциональное тестирование.

# Регрессионное тестирование

В процессе всего периода эксплуатации программа периодически изменяется, так как в ней исправляются найденные дефекты или добавляется новая функциональность. И при изменении программы нам необходимо проверять, что программист, изменяя одну область программы, не повлиял на работоспособность другой области программы. С этой целью проводится регрессионное тестирование.

*Регрессионное тестирование – вид тестирования, направленный на проверку ранее протестированной области программы, который проводится после изменения кода программы с целью убедиться в том, что процесс изменения не внёс или не активизировал ошибки в областях программы, которые не подвергались изменениям.*

Главная задача регрессионного тестирования – гарантировать, что ранее работающие функции программы продолжают корректно работать после добавления нового функционала в программу или изменения смежного функционала и отсутствие деградации общего качества программы.

Регрессионное тестирование является собирательным наименованием для всех видов тестов, ориентированных на поиск новых дефектов в уже проверенных функциях программы после очередных изменений в программном коде. Проводя данные тесты, специалисты выявляют ситуации, когда внесение изменений в одних модулях программы неожиданно приводит к появлению дефектов и поломок в других, неизменённых модулях. Такие дефекты называют «регрессионными».

*Регрессионный дефект – дефект, который возникает в ранее протестированной части программы после внесения изменений или обновлений в код.*

В практике разработки программ нередко происходит повторное возникновение одних и тех же дефектов, даже если они уже были исправлены. Причин тому много. Чтобы избежать подобных

регрессий[24], рекомендуется при исправлении дефектов создавать автоматизированные тесты и в дальнейшем регулярно запускать их после очередных изменений кода, чтобы убедиться, что старые ошибки не появились вновь. Хотя регрессионное тестирование можно проводить вручную, обычно для этого используются специальные инструменты. Они позволяют выполнить все накопленные регрессионные тесты за один запуск. Нередко настраивают автозапуск таких тестов при каждой новой сборке программы или с заданной периодичностью. Это помогает своевременно обнаруживать возможные регрессии.

Рассмотрим пример. Имеется интернет-магазин, в котором при оформлении заказа можно оплатить товар только банковской картой. При очередной доработке добавили новый способ оплаты – предоплаченным сертификатом. Проводя регрессионное тестирование, специалист должен убедиться, что после добавления нового способа оплаты ранее существующие способы оплаты продолжают работать, т. е. оплата банковской картой продолжает работать. Он убеждается, что изменения не повлияли на смежный функционал системы.

# Санитарное тестирование

Данный вид тестирование вызывает много споров в кругу специалистов, профессионально занимающихся тестированием. Давайте его рассмотрим.

*Санитарное тестирование – вид тестирования, направленный на проверку отдельных функций или компонентов программы с целью проверки их корректной работы после очередных небольших изменений кода или функциональности.*

Санитарные тесты проводят, чтобы подтвердить, что конкретный модуль или функция программы по-прежнему работают согласно изначальным техническим требованиям и не были сломаны при незначительном изменении кода. Главная задача – убедиться, что локальные изменения не привели к поломке существующего функционала.

Вчитавшись внимательно в текст, описывающий санитарное тестирование, возникает ощущение, будто речь идёт о регрессионном тестировании «*…убедиться, что локальные изменения не привели к поломке существующего функционала…*». Далее начинает формироваться вопрос «В чём тогда разница?». Разница в деталях, которые мы с вами сейчас и перечислим.

В процессе разработки перед выпуском новой версии программы она постоянно передаётся от программистов тестировщикам и обратно. Это происходит в связи с тем, что программист готовит и собирает программу и передаёт её на тестирование. Специалисты по тестированию проверяют программу, обнаруживают дефекты и отправляют на исправление программистам. И так продолжается до тех пор, пока программа не станет стабильной – будет без дефектов, которые не позволяют передавать программу пользователям.

Первые версии сборок программы обычно нестабильны – имеют много дефектов и недоработок. Поэтому сначала на них проводится дымовое тестирование основных критичных функций. Если базовый функционал работает приемлемо, сборка дополнительно проходит остальные виды тестирования (регрессионное и другие). После

прохождения нескольких итераций дымовых и регрессионных тестов сборка становится относительно стабильной. Если есть какие-либо исправления дефектов, добавленные в стабильную сборку, выполняется санитарное тестирование для повторного тестирования исправления дефектов. Если всё в порядке, выполняется повторное регрессионное тестирование.

Рассмотрим на примере программы для сложения чисел. В очередной сборке программы, после многократного проведения различных видов тестирования, исправили последний дефект – поле отображения результатов сложения не было заблокировано от введения в него данных, которые можно было вносить туда вручную. После исправления дефекта программу передали на тестирование. Специалист по тестированию знает: все остальные функции программы ранее были проверены и работали корректно, программисты их не трогали. Поэтому он протестирует исправления только данной ошибки и проверит конкретно, что поле отображения результатов сложения теперь недоступно для редактирования, и в него нельзя вводить какие-либо данные. В этот момент он провёл санитарное тестирование.

Второй пример. В очередной сборке программы, после многократного проведения различных видов тестирования, был исправлен последний дефект – при изменении режима работы программы данные о новом режиме работы не отображались в интерфейсе программы. Сам функционал изменения режима работы работал. После исправления дефекта программу передали на тестирование. Специалист протестирует исправления только данной ошибки и проверит конкретно, что при изменении режима работы данные о новом режиме работы отображаются в интерфейсе программы. Он проведёт санитарное тестирование. В этот момент он не проводит другие тесты. Если начнёт проверять, не сломан ли был в момент исправления дефекта другой функционал, тогда он переключится на регрессионное тестирование.

Завершающий пример из жизни. Вера по просьбе Игоря одолжила ему фонарик. Игорь вернул Вере фонарик через неделю. Вера обнаружила, что у фонарика разбито стекло – обнаружила дефект. Она попросила Игоря исправить дефект. Он стекло заменил и вновь передал фонарик Вере. Она, получив фонарик, в первую очередь

проверила, что стекло целое и из того же материала, что было ранее. В этот момент она провела санитарное тестирование. И только после этого начала проверять, включается ли фонарик, светит ли так же, как и раньше. В этот момент она переключилась на регрессионное тестирование, убеждаясь, что ранее работающие функции работают, как и должны работать.

# Тестирование установки

Есть программы, которые можно сразу запустить и работать с ними без всякой предварительной установки. Есть программы, которые необходимо установить на компьютер перед тем, как начать с ними работать. Кроме установки есть процесс удаления, а также процесс обновления программы. Все они тестируются. При этом применяется тестирование установки.

*Тестирование установки – вид тестирования программного обеспечения, направленный на проверку стабильности и корректности процедур установки, обновления и удаления программы на различных конфигурациях оборудования и программного окружения пользователей.*

Цель тестирования установки – гарантировать правильную работу процедур установки, удаления и обновления программ в разнообразных условиях эксплуатации у конечных пользователей.

Есть информационные системы промышленного масштаба, которые используются на разнообразных аппаратных платформах[25]. В этом случае процесс установки трудоёмкий и сложный. В случае работы с подобными типами систем, установка представляет из себя процесс поочерёдных операций, которые проводит группа специалистов. Данные операции описываются в плане установки и обновления. С помощью этого документа можно не только узнать алгоритм установки и обновления определённой программы, но и способы её возврата в исходное состояние в случае, если в процессе обновления возникли проблемы.

Тестирование установки – важный вид тестирования. Если им будут пренебрегать организации, у которых имеются информационные системы промышленного масштаба (банки, промышленные предприятия и т. д.), это может привести к серьёзным последствиям. Также организациям-разработчикам программ не следует пренебрегать этим видом тестирования, так как они могут понести репутационные и прочие потери.

Пример проверок при тестировании установки:

– проверка последовательности появления диалоговых окон во время установки программы;

– проверка работы всех кнопок инсталлятора;

– проверка установки на всех поддерживаемых платформах;

– проверка установки с правами, которые требует инсталляция;

– проверка установки с нуля (при отсутствии любых связанных файлов и предыдущих версий);

– проверка при установке подсчёта количества свободного места на диске;

– проверка установки программы по сети;

– проверка восстановления процесса установки после его внезапного прерывания;

– проверка распознавания наличия в системе программ, необходимых для корректной работы устанавливаемой программы;

– проверка наличия созданных ярлыков и корректность их расположения.

По аналогии составляются проверки для тестирования процесса удаления и обновления программы. Попробуйте сделать это самостоятельно, выбрав в качестве примера программу, которую недавно устанавливали.

# Тестирование удобства использования

*Тестирование удобства использования – вид тестирования, направленный на проверку того, насколько легко пользователь может разобраться с программой, насколько быстро он может освоить её и насколько привлекательной программа является при использовании в конкретных условиях эксплуатации.*

Главная цель – определить, насколько понятна и удобна программа в условиях реального применения, и выявить проблемные места с точки зрения удобства использования.

В ходе тестирования удобства использования анализируются:

– интуитивность и лёгкость изучения интерфейса программы;

– эффективность работы в программе при выполнении типовых задач;

– количество и степень серьёзности ошибок пользователей, которые они допускают во время работы;

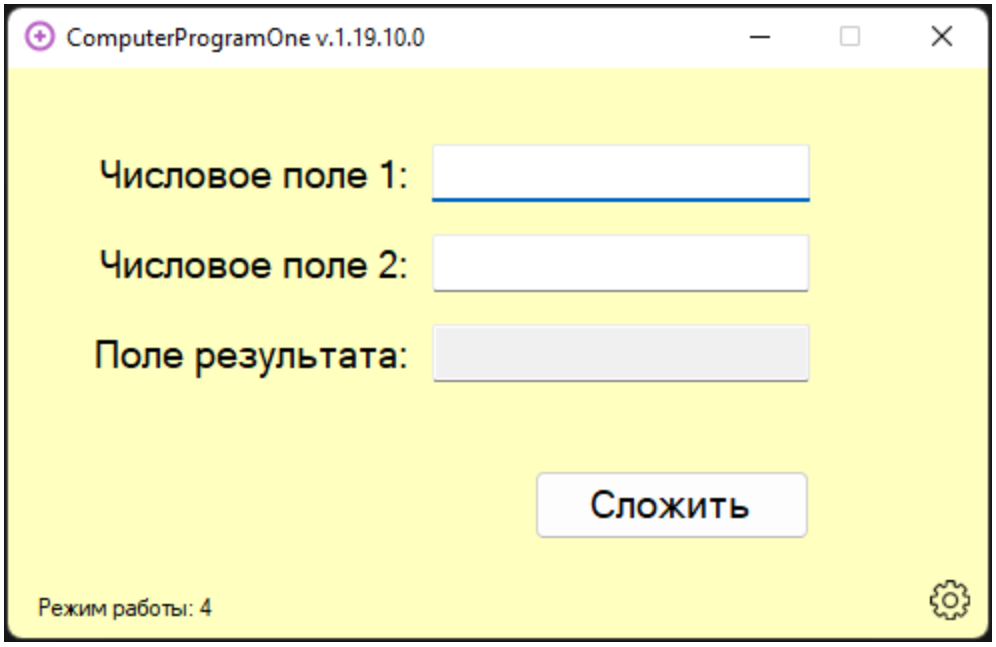
– общее впечатление от использования: удовольствие или раздражение.

Иногда мы сталкиваемся с программами, у которых многие возможности и методы использования неочевидны. После взаимодействия с такой программой у пользователей редко появляется желание воспользоваться ей снова. В таких ситуациях пользователи, скорее всего, начинают искать удобные аналоги. В чём типичные проблемы? Формулировки, термины и пиктограммы в интерфейсе непонятные. Выполнить простые действия не получается или занимает слишком много шагов и времени. Документация отсутствует или напоминает инструкцию по сборке мебели на межгалактическом наречии. В итоге у пользователя складывается негативный опыт. Вместо помощи в решении задач он получает лишь раздражение от хождения по замкнутому кругу по не интуитивным меню. К сожалению, такие ситуации не редкость.

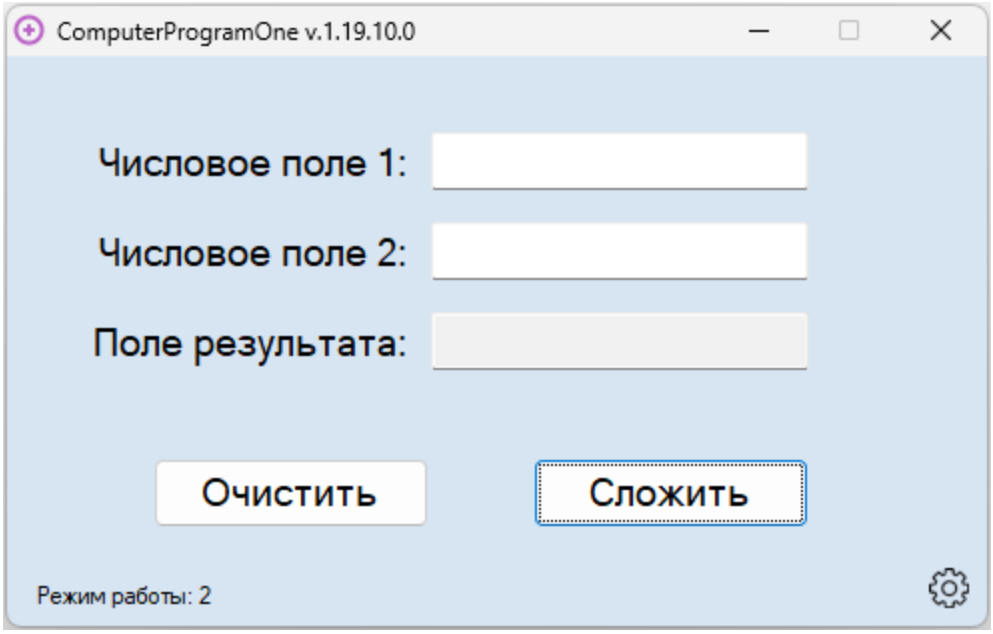
Для того чтобы разрабатываемая программа завоевала популярность, ей недостаточно просто выполнять определённые функции – она должна быть удобной в использовании. Понятные и удобные в использовании программы позволяют сокращать затраты на

обучение пользователей и персонала, что, в свою очередь, делает программу более конкурентоспособной. Исходя из этого, можно сделать вывод, что тестирование удобства использования должно быть неотъемлемой частью проверки программ, предназначенных для массового использования.

Рассмотрим пример. Есть у нас две программы для сложения чисел. Каждую из них разрабатывает отдельная организация. У одной программы поля можно очищать только вручную по отдельности при помощи кнопки «Backspace» на клавиатуре:




У второй программы также можно очищать поля вручную по отдельности, а можно очистить все поля разом одной кнопкой «Очистить»:




Какая программа удобнее в использовании? Какую предпочтут пользователи? Ответ очевиден – программу, у которой есть кнопка «Очистить».

Ещё один пример. Есть два сайта. На первом сайте информация для чтения предоставляется мелким шрифтом текста, и при прочтении требуется напрягать зрение в попытке вчитаться. На втором сайте информация для чтения представляется в удобном для чтения формате – большой размер шрифта текста. Оба сайта имеют одинаковую информацию и функциональность. Выбор пользователей очевиден. Большинство предпочтёт второй сайт для регулярного посещения. Первый будут посещать те, кто не знает о втором, и пока они не узнают о нём.

В обоих примерах организации, создавшие неудобные программы, будут терять аудиторию, а соответственно и деньги. Поэтому тестирование удобства использования – важный вид тестирования, позволяющий специалистам выявлять проблемы в удобстве использования программ, которые необходимо устранять.

# Тестирование взаимодействия

Есть программы, у которых внутренние компоненты взаимодействуют между собой с помощью специальных интерфейсов. Также есть программы, которые взаимодействуют с другими с помощью тех же специальных интерфейсов. В процессе планирования тестирования необходимо не забыть запланировать тестирование указанных взаимодействий. Для тестирования взаимодействий компонентов и программ применяется предназначенный для этого вид тестирования – тестирование взаимодействия.

*Тестирование взаимодействия – вид тестирования, направленный на проверку взаимодействия различных компонентов или модулей программы или оценку взаимодействия различных программ между собой.*

Цель тестирования взаимодействия – удостовериться, что различные элементы программы или сами программы корректно взаимодействуют друг с другом и выполняют свои функции в рамках совместной работы.

В наши дни тестирование взаимодействия приобретает особую актуальность в связи с ростом популярности веб-приложений, мобильных сервисов и сетевых программ. Эти цифровые продукты, как правило, интегрированы с множеством других облачных программ и сервисов. Например, социальные сети взаимодействуют с системами аналитики и рекламными платформами. Интернет-магазины интегрируются с сервисами доставки и банками. В таких условиях важно тщательно протестировать не только отдельную программу, но и правильность её взаимодействия с другими программами.

Как я уже сказал, программы или компоненты программ осуществляют взаимодействие между собой через специальные интерфейсы. У этих интерфейсов есть общепринятое название – API.

*API (Application Programming Interface, интерфейс программирования приложений) – это программный интерфейс, который представляет собой набор готовых классов, функций, методов и протоколов взаимодействия, которые предоставляет одна программа другим программам для взаимодействия друг с другом.*

API позволяет разным программам или компонентам одной программы обмениваться данными и взаимодействовать друг с другом. Через программный интерфейс они обмениваются данными, управляют и управляются.

Для лучшего понимания рассмотрим API на примере ресторанного меню. В ресторане есть меню, в котором представлен список блюд, а также информация о том, как их заказать. API действует подобно этому меню, предоставляя программам (или различным частям программ) специальные «блюда» (функции), которые они могут использовать. Представьте, что мы пришли в ресторан и заказали салат. Наш заказ (запрос) отправляется на кухню, где салат готовится и затем возвращается нам (ответ). В этом контексте ресторанное меню – это API, которое определяет, какие блюда (функции) доступны для заказа и что мы получим в результате (ответ). Точно так же, когда программа использует API, она отправляет запрос на выполнение определённой функции, а затем получает результат в ответ.

Примеры тестирования взаимодействия:

Пример № 1. Есть веб-приложение, включающее клиентскую и серверную части. При тестировании интеграции проверяется, что данные правильно передаются от клиента к серверу и обратно, а также что обе части взаимодействуют без ошибок.

Пример № 2. Есть программа, устанавливаемая на мобильное устройство, использующая API для получения данных с сервера. Тестирование API в данном случае включает проверку того, что запросы к серверу отправляются и принимаются правильно, что полученные данные обрабатываются корректно, а также что в программу возвращаются те данные, которые программа запрашивала у сервера.

Пример № 3. В «организации 1» имеется программа, в которой ведут счета на оплату. В «организации 2» есть программа, в которой

организация ведёт всю свою деятельность. Программа первой организации передаёт в программу второй организации счета. В процессе тестирования взаимодействия необходимо убедиться, что данные корректно отправляются и принимаются в обеих программах.

Во всех перечисленных примерах тестируется API с использованием специализированного инструментария.

# Тестирование переносимости

*Тестирование переносимости (портируемости) – вид тестирования, направленный на проверку возможности переноса программного обеспечения на другие аппаратные или программные платформы без потери функциональности.*

Данный вид тестирования направлен на проверку того, насколько легко программа может быть адаптирована или перенесена из одной среды в другую, сохраняя работоспособность всех функций. Это важно, поскольку пользователи используют разные операционные системы, устройства и окружения.

Рассмотрим на примере компьютерной игры для персонального компьютера. Изначально определённая компьютерная игра разрабатывалась под Windows. Однако программисты решили адаптировать её и под операционную систему Linux, к примеру CentOS. Они это сделали, и специалисту по тестированию требуется проверить:

– корректную установку на Windows и Linux;

– запуск и работу основных функций на Windows и Linux;

– отсутствие проблем с графикой на Windows и Linux;

– сохранение и загрузку игрового прогресса на Windows и Linux;

– работу с периферией (геймпады, мыши и т. д.) на Windows и Linux.

В ходе данного тестирования выявляются проблемы совместимости и портируемости на разных платформах. Это делается для того, чтобы пользователи могли комфортно играть на любой из поддерживаемых операционных систем.

Примеры тестирования на переносимость:

Пример № 1. Тестирование на разных операционных системах. Программа должна быть протестирована на Windows, Mac OS и Linux, чтобы убедиться, что она работает стабильно в различных средах.

Пример № 2. Тестирование на разных устройствах. Если программа предназначена для использования на мобильных устройствах, её следует протестировать на разных моделях устройств и разрешениях экранов.

Пример № 3. Тестирование с разными версиями языков программирования. Если программа написана на языке программирования, который обновляется с течением времени, необходимо проверить её совместимость с разными версиями этого языка.

Пример № 4. Тестирование с разными библиотеками и зависимостями. Программу следует протестировать с различными версиями библиотек и внешних компонентов, чтобы избежать проблем совместимости.

Пример № 5. Тестирование на различных браузерах. Если программа является веб-приложением, то необходимо убедиться, что она корректно работает в различных веб-браузерах, таких как Chrome, Firefox, Safari и т. д.

# Тестирование производительности

*Тестирование производительности – вид тестирования, направленный на проверку работы программы в условиях различных нагрузок с целью определения её производительности.*

Что мы подразумеваем под производительностью?

*Производительность – это мера того, насколько эффективно и быстро программа выполняет свои задачи. Данная характеристика оценивается в различных аспектах, включая время отклика, скорость обработки данных, эффективность использования ресурсов (таких как память и процессор).*

Ключевые аспекты тестирования производительности:

– оценка реакции программы при постепенном или максимальном увеличении нагрузки, чтобы выявить её предельные возможности;

– определение, сколько данных программа может обработать за единицу времени;

– измерение времени, необходимого программе для обработки запросов и возвращения результата;

– оценка способности программы эффективно масштабироваться при увеличении числа пользователей или объёма данных.

Рассмотрим пример. Есть сайт. В час сайт посещает сто пользователей. В планах организации стоит привлечение в ближайшее время новых пользователей. По подсчётам специалистов сайт начнут посещать по двести пользователей в час. Задача специалистов понять, сможет ли сайт продолжать работать в том же режиме и с такой же скоростью, как и сейчас, когда количество пользователей в час увеличится. Для этого с помощью специальных программ они будут эмулировать посещение сайта двумястами пользователями в час, измерять время загрузки страниц и скорость выполнения определённых операций на сайте, которые производят пользователи (авторизация, покупка товаров, написание отзывов). После проведения тестов и анализа полученных данных специалисты должны будут принять решение, как увеличить производительность сайта, если она

недостаточная. Если, к примеру, главная страница сайта начнёт открываться в три раза дольше, специалисты будут изучать причину возникновения долгой загрузки страницы и устранять её.

Таким образом выявляются «узкие» места при росте нагрузки и определяются пороги, после которых производительность программы резко падает. Это позволяет заранее оптимизировать работу программы при росте нагрузки на неё.

# Стрессовое тестирование

*Стрессовое тестирование (стресс-тестирование) – вид тестирования, направленный на проверку работы программы на предельных и запредельных нагрузках, а также в условиях ограниченности системных ресурсов.*

В ходе данного вида тестирования имитируются пиковые рабочие нагрузки, превышающие расчётные. Например, одновременный запуск максимального количества разрешённых пользователей на сайт. Также может проверяться работоспособность при нагрузках за границей проектных значений или при искусственном ограничении ресурсов, таких как памяти, дискового пространства, полосы пропускания канала связи с сервером. Стресс-тесты позволяют оценить устойчивость работы программы в условиях экстремальных нагрузок, а также проверить, насколько быстро программа сможет вернуться в штатный режим работы после снятия аномально высокой нагрузки. Всё это нужно, чтобы убедиться, что программа останется отказоустойчивой даже в экстремальных условиях пиковой активности.

Например, есть сайт, и специалисты знают, что он выдерживает посещение максимум шестисот пользователей в час (десять пользователей в минуту). При превышении данного числа, сайт начинает работать медленно, и некоторые пользователи вместо страниц с информацией видят страницы с ошибками. Задача специалистов – превысить это количество, подавать повышенную нагрузку определённое время, потом снова снизить нагрузку до шестисот пользователей в час и убедиться, что сайт восстановился и начал нормально работать при допустимой для него нагрузке. Если этого не произошло, специалисты ищут причину и устраняют её, изменяя программный код, или переконфигурируют оборудование, на котором развёрнут сайт.

# Объёмное тестирование

*Объёмное тестирование – вид тестирования программы, направленный на оценку способности программы эффективно обрабатывать большие объёмы данных.*

Цель объёмного тестирования – убедиться, что программа продолжает работать стабильно и с прежней производительностью при увеличении объёма информации, с которой она работает. Это позволяет заранее подготовить архитектуру и мощность серверов, на которых развёрнута информационная система к росту объёма данных, с которым программы будут работать в перспективе следующих лет.

Ключевые аспекты объёмного тестирования:

– проверка программы на её способность обрабатывать большие объёмы данных, такие как записи в базе данных, файлы или транзакции[26];

– оценка времени отклика и общей производительности программы при работе с различными объёмами данных;

– оценка использования ресурсов, таких как память и процессор, при обработке больших объёмов данных.

Рассмотрим пример сайта, который уже упоминали. Допустим, сейчас у сайта размер базы данных сто мегабайт и посещает его шестьсот пользователей в час, при этом главная страница сайта открывается за одну секунду. Одна секунда на открытие главной страницы – это максимум, на что согласны в организации. Там знают, что через год размер базы данных может вырасти до ста гигабайт, и они решили определить сколько времени при этом будет открываться главная страница сайта. В этом случае специалисты проводят объёмное тестирование. Специалисты искусственно увеличивают объём базы данных до ста гигабайт. Далее при тестировании специальными инструментами эмулируют посещение сайта шестьюстами пользователей в час. Проведя ряд тестов, они определят, как влияет объём базы данных в сто гигабайт на скорость работы сайта. Если показатели не будут их устраивать и главная страница

будет открываться более одной секунды, то они проведут ряд мероприятий по улучшению работы сайта.

# Тестирование стабильности

*Тестирование стабильности – вид тестирования, направленный на проверку бесперебойной работы программы в течение длительного времени с ожидаемым уровнем нагрузки.*

Тестирование стабильности, также известное как «тестирование надёжности». Задача тестирования стабильности – проверка, что программа работает стабильно при длительной (многочасовой) подаче требуемой нагрузки на программу. Обычно подают нагрузку в районе 70 – 80% от максимального уровня, который может выдержать программа. При данном тестировании ведётся наблюдение за потреблением программой серверных ресурсов: загрузка памяти; загрузка процессора; загрузка сети и т. д. В ходе наблюдения отслеживается, чтобы скорость обработки данных и/или время отклика программы в начале теста и с течением времени не увеличивались. Если это не так, программистами проводятся мероприятия по улучшению стабильности программы. В противном случае в ходе её промышленной эксплуатации вероятны сбои в работе и различного рода потери.

Снова обратим внимание на сайт. Как мы уже знаем, он выдерживает шестьсот пользователей в час. Все мы знаем, что посещаемость в течение дня варьируется: то шестьсот пользователей в час, то четыреста, то двести и т. д. Нам надо ответить на вопросы: «Что будет, если наш сайт будут посещать ежечасно в районе 80% пользователей от максимального количества пользователей в шестьсот человек? Как в этом случае поведёт себя сайт?». И тогда запускается тест и с помощью специализированных инструментов эмулируется на протяжении многих часов (дней) посещение сайта ежечасно по четыреста восемьдесят человек в час (это 80% от максимума). Если сайт в начале, в середине и в конце теста работает, как ожидалось, то сайт стабилен. Если со временем он начнёт испытывать трудности в работе, а возможно и вообще перестанет работать, то с сайтом проблемы, которые необходимо в дальнейшем выявить и устранить.

Такие виды тестирования как тестирование производительности, стрессовое тестирование, объёмное тестирование, тестирование

стабильности проводятся с использованием специализированных инструментов, которые предназначены для проведения данных видов тестирования. Также они обычно проводятся специалистами по нагрузочному тестированию, которые обладают необходимыми специализированными навыками и знаниями.

# Тестирование безопасности

*Тестирование безопасности – вид тестирования, направленный на проверку устойчивости программы к различным атакам на уязвимость и взломам с целью определить наличие уязвимостей, которые могут привести к нарушениям конфиденциальности или целостности данных.*

Цель такого тестирования – выявить уязвимости в программе и обеспечить её защиту от потенциальных атак.

Ключевые аспекты тестирования безопасности:

– поиск и анализ возможных точек входа для потенциальных атак, таких как слабые пароли, ошибки в программном коде и прочее;

– оценка процессов проверки подлинности пользователей и управления их доступом к ресурсам программы;

– проверка, как программа обрабатывает и защищает конфиденциальные данные, используя шифрование;

– анализ устойчивости сетевых элементов программы к атакам, таким как перехват данных, отказ в обслуживании и другие;

– проверка программы на наличие вредоносного кода, троянских программ и других вредных элементов.

Тестирование безопасности не только выявляет уязвимости, но и позволяет организациям предотвратить потенциальные угрозы для конфиденциальности, целостности и доступности данных. Компьютерные системы часто становятся мишенью незаконных взломов и атак. Осуществлять взлом и атаки могут хакеры, стремящиеся проникнуть в систему из любопытства, сотрудники из-за мести работодателю, мошенники с целью незаконного обогащения. Тестирование безопасности направлено на проверку реакции встроенных в систему защитных механизмов на потенциальные вторжения. В процессе тестирования безопасности специалисты моделирует действия потенциального злоумышленника.

Специалист, проверяющий программу, может выполнять следующие шаги:

– попытка получить сохранённые пароли пользователей с использованием специализированных инструментов;

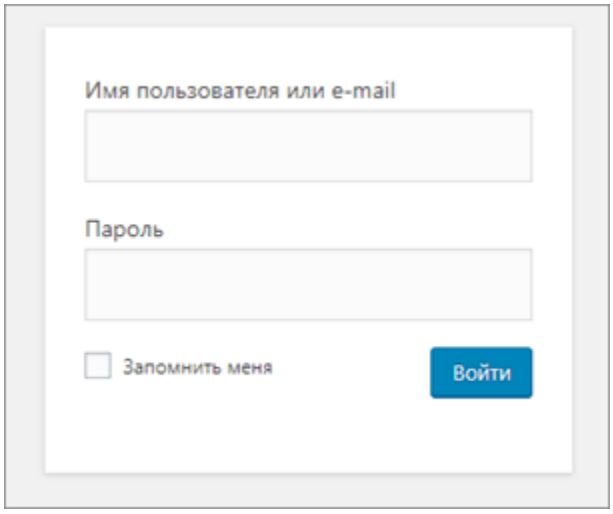
– атака на программу с применением специальных инструментов, которые анализируют уровень защиты программы;

– создание ситуации, при которой программа теряет возможность функционировать, и происходит сбой в обслуживании пользователей;

– умышленное внесение ошибок с надеждой на проникновение в систему в результате нештатного функционирования программы.

Мы должны помнить: если злоумышленник располагает неограниченным временем и ресурсами, через определённое время он способен обнаружить уязвимости в системе защиты. Поэтому в процессе проектирования программ архитекторы сталкиваются с задачей создания такой программы, где стоимость вторжения значительно превышает ценность получаемой в результате информации. К примеру, если злоумышленник потратит финансов на взлом информационной системы организации в десятки раз больше, чем получит от продажи украденной информации, он вряд ли захочет этим заниматься.

Рассмотрим простой пример тестирования безопасности. Есть программа, в которой работают финансисты организации и в которой находится вся финансовая информация организации. У программы существует форма для входа:

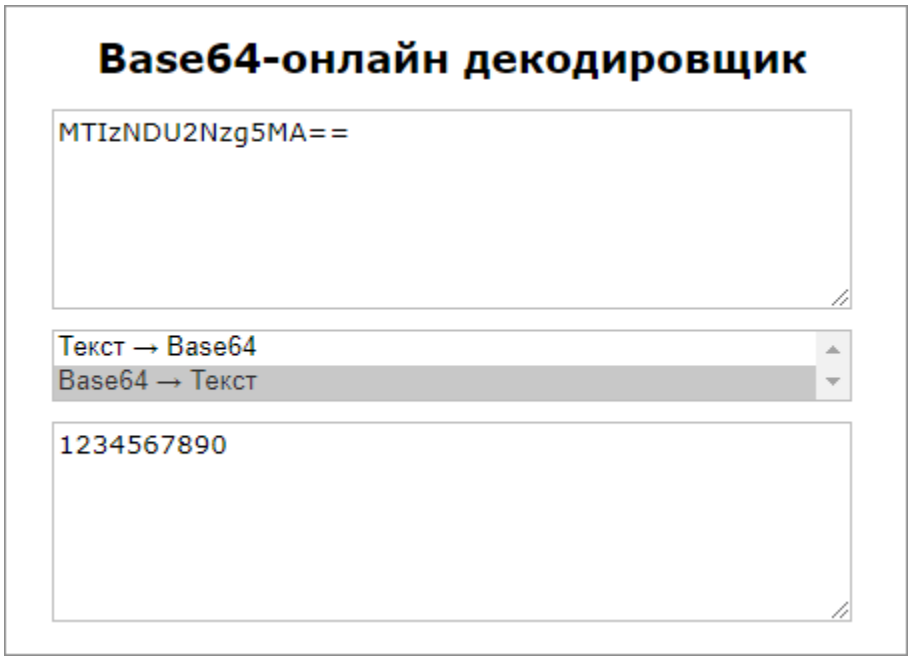



Если при входе в программу пользователь поставит галочку «Запомнить меня», то программа сохраняет пароль в файл настроек на диске. Представим, что в требованиях к программе написано: «программа должна хранить пароль в защищённом виде». Каждый

может это понимать по-своему. Программист скрыл пароль, маскируя его при помощи стандарта кодирования двоичных данных Base64. Этим он замаскировал данные, но не зашифровал. В этом случае пароль «1234567890» в файле настроек будет храниться в таком виде:

MTIzNDU2Nzg5MA==

По требованиям пароль не защищён, а скрыт/замаскирован. Алгоритм общеизвестный, и мы можем легко узнать пароль:




Однако подобная маскировка не защитит хранимый пароль от злоумышленника, который может без проблем прочесть его и в дальнейшем им воспользоваться в своих корыстных целях, нанеся финансовый урон организации.

Специалисты обнаружили данную проблему в ходе тестирования. В этом случае программа не пройдёт тестирование безопасности, и специалисты будут рекомендовать программистам использовать более надёжные алгоритмы шифрования пароля. Возможно, программистам придётся придумать свой алгоритм шифрования пароля.

Тестирование безопасности является важным видом тестирования, которым нельзя пренебрегать.

# Тестирование на отказ и восстановление

*Тестирование на отказ и восстановление – вид тестирования, направленный на определение того, насколько хорошо программное обеспечение или информационная система способны справляться со сбоями, ошибками и другими неполадками.*

Основная цель тестирования на отказ и восстановление – проверить надёжность механизмов сохранности и целостности данных в программе при возникновении различных аварийных ситуаций. В ходе данного вида тестирования искусственно имитируются сбои отдельных компонентов путём отключения питания серверов, обрыва сетевых подключений, остановки критичных сервисов и баз данных. Это требуется, чтобы удостовериться, что в случае реальных инцидентов у пользователей системы не произойдёт непоправимой потери важной информации или нарушения целостности данных. Также это необходимо, чтобы проверить оперативность восстановления штатного режима работы всех сервисов и компонентов программы после устранения условно возникшей неисправности.

Примеры ситуаций отказа:

Пример № 1. Отказ жёсткого диска, выход из строя памяти, сбой в работе процессора;

Пример № 2. Непредвиденное завершение работы программы, ошибки в коде, необработанные исключения;

Пример № 3. Потеря связи, недоступность серверов, задержки в передаче данных.

Примеры тестов на отказ и восстановление:

Пример № 1. Создание ситуации отказа и проверка, насколько программа способна восстановиться автоматически. Например, автоматическое восстановление работы программы после сбоя в сетевом соединении.

Пример № 2. Проверка, как быстро и безопасно программа переключается на резервные ресурсы в случае выхода из строя основных ресурсов.

Пример № 3. Отключение базы данных и проверка переключения информационной системы на резервную базу данных.

Пример № 4. Отключение отдельных компонентов или сервисов и проверка, как программа справляется с их отсутствием и восстанавливает ли свою функциональность.

Рассмотрим пример. Есть сайт, который должен работать 24 часа в сутки, 7 дней в неделю. Если сайт разместить на одном сервере, то, когда сервер выйдет из строя, сайт станет недоступным. Для этого его размещают, к примеру, на двух серверах: один основной, а второй резервный. Это можно назвать кластером серверов. Если первый сервер выйдет из строя, то специальный контроллер мгновенно переключит всё на второй сервер, и сайт продолжит свою работу (сработает отказоустойчивость), а когда первый сервер исправят и включат, контроллер снова переключит всё на первый сервер (восстановится первоначальное состояние).

# Рецензирование

*Рецензирование – вид тестирования, в ходе которого эксперты тестируют программный код, документацию, дизайн или другие артефакты с целью выявления ошибок, улучшения качества и обеспечения соответствия стандартам и требованиям.*

Рецензирование проводит человек-эксперт. Рецензирование может быть применено к любому артефакту, который участники анализа понимают и могут прочесть.

Примеры рецензируемых артефактов:

Пример № 1. Программный код. Программисты могут проверять код, написанный не ими, на наличие ошибок, соответствие стандартам кодирования, эффективность и читаемость.

Пример № 2. Документация. Технические писатели могут проверять технические документы на ясность, полноту и соответствие стандартам документирования.

Пример № 3. Дизайн интерфейса. Дизайнеры могут проверять макеты и элементы пользовательского интерфейса на соответствие дизайн-стандартам и удобство использования.

Пример № 4. Спецификации требований. Специалисты по тестированию могут проверять спецификации требований на полноту, непротиворечивость, отсутствие логических ошибок и т. д.

Во всех перечисленных случаях эксперты изучают и вычитывают различные артефакты, выявляя в них несоответствия или проверяя их на соответствие определённым стандартам и правилам.

Преимущества рецензирования:

1) Обнаружение ошибок. Рецензирование помогает выявить ошибки и недочёты на ранних этапах разработки, что способствует повышению качества продукта.

2) Обмен опытом. Рецензирование предоставляет возможность обмена опытом между членами команды, улучшая общий профессиональный уровень.

3) Соблюдение стандартов. Рецензирование позволяет следить за соблюдением стандартов разработки, документирования или дизайна.

Простой пример рецензирования: специалист по тестированию вычитывает спецификацию требований на разработку программы для сложения чисел и выявляет в ней все несоответствия. Он проводит рецензирование.

# Статический анализ

*Статический анализ – вид тестирования, направленный на анализ программного кода, документации или других артефактов без их фактического выполнения с использованием специализированных инструментов.*

Цель статического анализа – выявление потенциальных ошибок, несоответствий стандартам, а также поиск других проблем на ранних этапах разработки.

Статический анализ можно эффективно применить к любым артефактам разработки программы, имеющим формальную структуру: код, модели, документация, спецификация требований. Главное условие – наличие инструментов анализа для проверки качества конкретного артефакта. Например, для исходного кода существует множество анализаторов, выявляющих различные ошибки. Для проверки требований могут использоваться инструменты анализа текстов для проверки орфографии, лексики, удобочитаемости. Комплексное применение статического анализа на всех этапах разработки позволяет выявлять дефекты ещё до запуска программы и значительно снизить риски появления дефектов в промышленном окружении.

Примеры статического анализа:

Пример № 1. Анализ программного кода с целью обнаружения синтаксических ошибок, потенциальных проблем безопасности и несоответствий стандартам кодирования. Инструменты для статического анализа кода проводят статический анализ кода, выявляя структурные и стилевые проблемы. К примеру, ReSharper, SonarQube для кода, написанного на языке программирования C#, ESLint для кода, написанного на JavaScript и т. д.

Пример № 2. Анализ документации с целью проверки документации на наличие недочётов, несоответствий стандартам. К примеру, инструменты анализа текста, такие как Microsoft Word могут использоваться для статического анализа текста документации и выявления грамматических ошибок.

Пример № 3. Анализ модели данных[27] с целью проверки соответствия модели данных заданным требованиям и выявления возможных проблем в структуре данных. Инструменты для статического анализа модели данных могут помочь в обнаружении несоответствий в структуре баз данных.

Преимущества статического анализа:

1) Позволяет выявить и исправить ошибки до того, как программный код будет собран в программу или программа будет внедрена в промышленную эксплуатацию.

2) Помогает удостовериться, что программный код или документация соответствуют установленным стандартам кодирования и документирования.

3) Позволяет быстро проверить большие объёмы программного кода или документации, выявляя потенциальные проблемы.

Рассмотрим пару простых примеров:

Пример № 1. Специалист по тестированию открывает спецификацию требований в одном из текстовых редакторов, который может проводить проверку орфографии и расстановку знаков препинания. С его помощью проводится проверка документа.

Пример № 2. Разработчик написал программный код и, чтобы проверить, есть ли в сотнях написанных строк ошибка, загружает написанный код в специальную программу для статистического анализа кода. Программа, проверив код, выдаст список ошибок, замечаний и рекомендаций. При этом проверяемый код не собирается в программу, ошибки будут выявлены до её сборки и не попадут в программу.

В заключение по видам тестирования хотелось бы сказать: нет чёткой последовательности проведения видов тестирования. Т. е. нет такого, что сначала проводим «вид тестирования X», потом – «вид тестирования S» и так далее. Всё индивидуально и зависит от правил и договорённостей в конкретно взятой организации и от квалификации специалистов по тестированию.

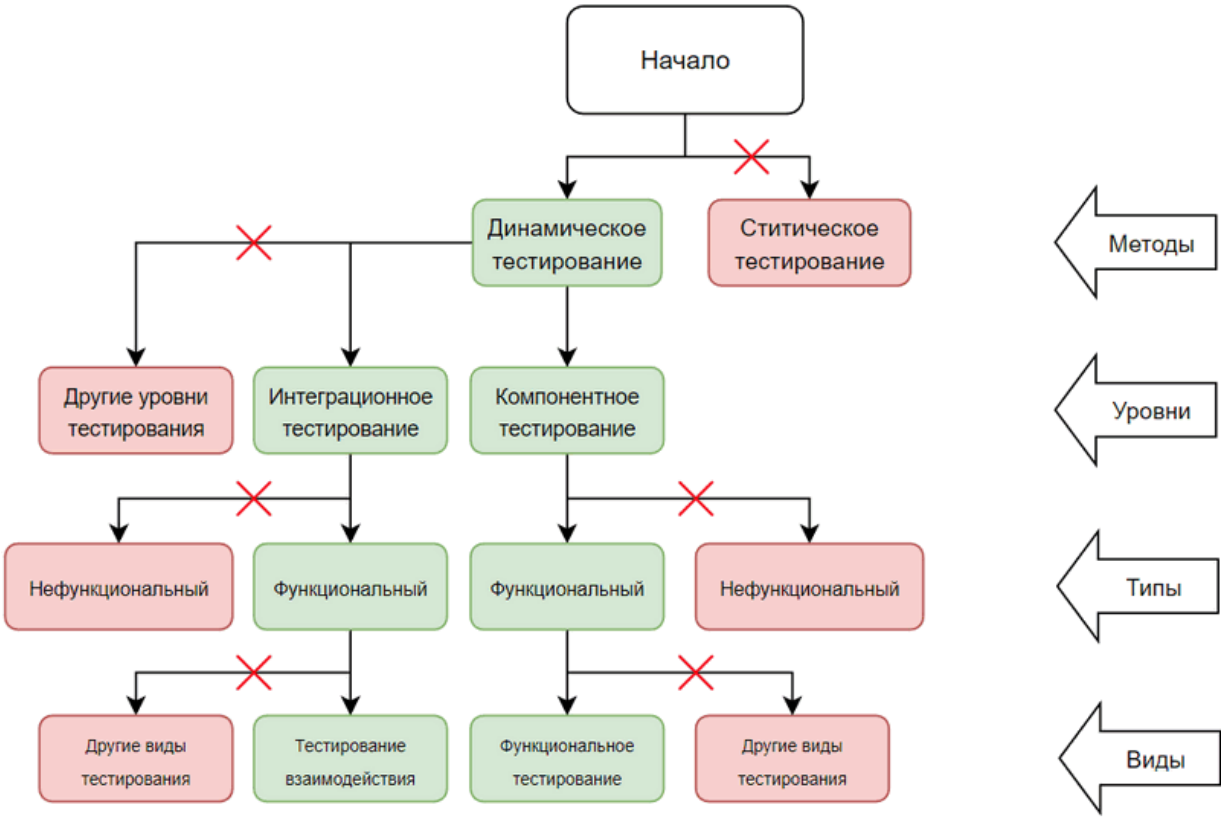
# Взаимосвязь методов, уровней, типов и видов тестирования

После изучения методов, уровней, типов и видов тестирования возникает закономерный вопрос: «Как они между собой связаны?». Чтобы ответить, мы рассмотрим взаимосвязи на примерах.

**Пример № 1**

Есть программа для сложения чисел. Программисты разрабатывают её, и в это же время пишут юнит-тесты, которые тестируют программу.

Юнит-тесты проверяют программу, запуская код самой программы, а это означает, что используется динамический метод тестирования. Юнит-тесты тесты пишутся на уровне (этапе) компонентного тестирования. Юнит-тесты для того, чтобы подтвердить наличие в программе всей необходимой логики и функциональности, которые в неё закладывались, а это означает, что проводится функциональный тип тестирования. Если юнит-тесты проверяют взаимодействие компонентов внутри программы, а также её определённые функции, то проводятся следующие виды тестирования: функциональное тестирование; тестирование взаимодействия. Тестирование взаимодействия? Но ведь это относится к интеграционному тестированию. Совершенно, верно. Это означает, что у нас задействовано тестирование на двух уровнях. Отразим сказанное схематично:




Можно применять и другие виды тестирования, однако в примере мы рассмотрели конкретные случаи.

**Пример № 2**

Есть сайт, на котором заказывают билеты на концерты. Сайт разработан и передан в тестирование специалистам по тестированию. При передаче руководители попросили проверить весь функционал сайта, как сайт взаимодействует с другими программами, которые есть в организации, а также как он будет работать, если в ближайшее время количество посетителей сайта увеличится на 1 000 человек в день.

Рассмотрим три вида тестирования, которые сразу прослеживаются: функциональное тестирование, тестирование взаимодействия, тестирование производительности. Другие виды брать не будем для простоты восприятия материала.

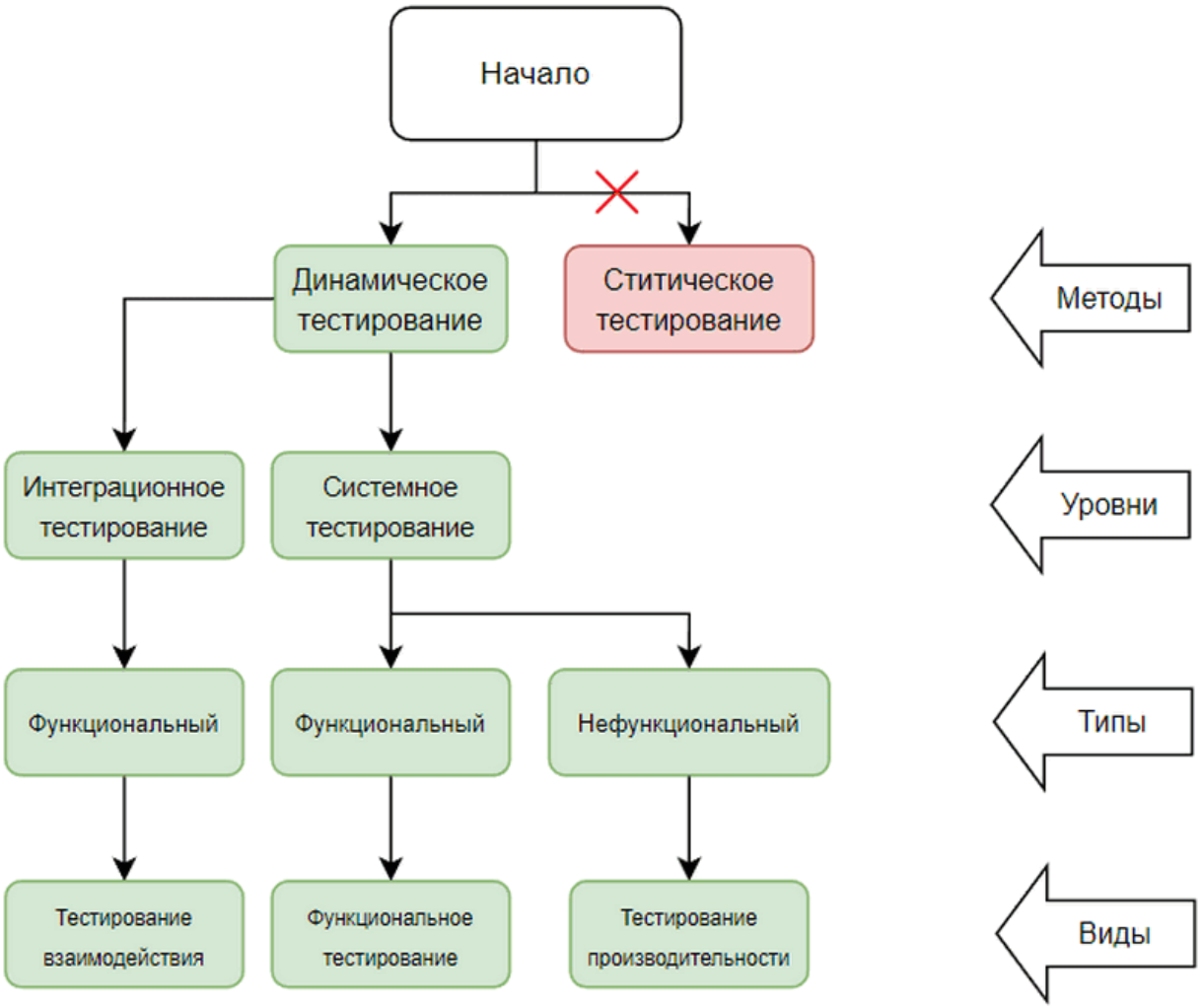
Функциональный вид тестирования, так как проверяем весь функционал сайта. Подтвердить наличие в программе всей необходимой логики и функциональности, которые в неё закладывались, – то есть проводится функциональный тип тестирования. Программу необходимо проверить в целом с целью

удостовериться, что она соответствует установленным требованиям – это уровень системного тестирования. В процессе тестирования будет выполняться код программы – это динамический метод тестирования.

Тестирование взаимодействия, так как проверяем взаимодействие различных программ между собой. Проверить необходимую функциональность программы, значит провести функциональный тип тестирования. Проверить взаимодействие между интегрированными программами –интеграционный уровень тестирования. В процессе тестирования будет выполняться код программы – динамический метод тестирования.

Тестирование производительности, так как требуется проверка работы программы в условиях различных нагрузок (+1 000 пользователей). Производительность – это мера того, насколько эффективно и быстро программа выполняет свои задачи. Мы проверяем свойства программы, не относящиеся к функциональности – скорость работы, а это означает, что проводится нефункциональный тип тестирования. Программа полностью собрана и проверяется в целом, чтобы проверить её соответствие установленным требованиям – уровень системного тестирования. В процессе будет выполняться код программы – это динамический метод тестирования.

Отразим сказанное схематично:




В приведённых примерах показана связь между методами, уровнями, типами и видами тестирования. Полностью все связи рассматривать не будем, так как объёма информации хватит ещё на одну книгу. Вы можете самостоятельно попрактиковаться и придумать ряд дополнительных примеров отразив схематически взаимосвязи. Сначала вам будет трудно, так как вы недостаточно хорошо ориентируетесь в методах, уровнях, типах и видах тестирования. Однако со временем будет получаться лучше.

# Проектирование тестов

*Проектирование тестов – это этап, на котором цели тестирования преобразуются в конкретные тест-кейсы, а также в набор тест-кейсов с целью дальнейшей проверки программы.*

При проектировании тестов необходимо тщательно анализировать требования к программе, выделять ключевые сценарии использования программы, а также выявлять её функциональные возможности. Результатом проектирования тестов будут полностью готовые предписания и шаги для многократного выполнения тестов специалистами по тестированию как вручную, так и автоматизировано.

Простым языком: проектирование тестов – это создание тест-кейсов на основании имеющегося базиса тестирования и имеющейся дополнительной информации по тестируемой программе. Берётся базис тестирования, изучается, после этого определяется, какие функции есть у программы, и на их проверку создаются тест-кейсы.

Тесты проектируются с использованием специальных методов.

*Метод проектирования тестов – это методика, которая используется для создания тест-кейсов.*

Методы проектирования тестов также именуют «техниками тест-дизайна». Методы проектирования тестов используются с целью определения тестовых условий, тест-кейсов и тестовых данных. Методы проектирования позволяют нам правильно спроектировать тест-кейсы. Если проектировать тест-кейсы, не опираясь на какие-либо методы, многое можно не учесть или проделать много лишней работы.

Переходя дорогу, вы руководствуетесь определёнными правилами:

– горит красный свет – дорогу переходить нельзя;

– горит зелёный свет – дорогу переходить можно.

Методы проектирования тестов являются группами правил, которыми руководствуются при разработке тестов. Выбор метода зависит от множества различных факторов, и в разных ситуациях при проектировании тестов могут применяться конкретные методы, а иногда, при возможности, они комбинируются. Обычно тестировщики используют комбинации различных методов в процессе создания тест-кейсов для достижения наилучших результатов.

Методы проектирования тестов группируются по категориям. Рассмотрим с вами три категории методов проектирования тестов:

– методы чёрного ящика;

– методы белого ящика;

– методы, основанные на опыте.

Список методов проектирования тестов, которые будем рассматривать, составлен на основании международного стандарта ISO/IEC/IEEE 29119-4 и ISTQB Foundation Level Syllabus[28].

# Методы чёрного ящика

Методы проектирования тестов из категории чёрного ящика основываются на знании функциональных особенностей и возможностей программы, полученных в ходе изучения базиса тестирования. При проектировании тестов методами чёрного ящика не учитываются знания внутренней структуры объекта тестирования – исходного кода программы.

## Метод эквивалентного разделения

*Эквивалентное разделение – метод проектирования тестов, который разделяет входные данные на группы (классы эквивалентности) таким образом, чтобы можно было выбрать представителя из каждого класса для тестирования.*

Англоязычное именование метода «Equivalence partitioning». Идея метода заключается в том, что если одно значение в группе (классе) ведёт себя определённым образом, то все остальные значения в эту группу должны помещаться по аналогичному признаку и должны вести себя аналогичным образом.

*Класс эквивалентности – часть данных из всего набора данных, обработка которых производится одинаково и приводит к одному и тому же результату, для которой поведение компонента или системы, основываясь на спецификации, считается одинаковым.*

Класс эквивалентности объединяет данные, считающиеся «эквивалентными» по какому-то критерию.

Предположим, в программе есть функция, которая принимает в себя возраст пользователя и возвращает «Доступ разрешён», если пользователь совершеннолетний, и «Доступ запрещён» в противном случае. Допустим, человек может прожить до 150 лет, и минимум ему может быть 0 лет и 1 секунда от роду… В этом случае необходимо при проведении тестирования программы провести 151 тест, вводя

поочерёдно от 0 до 150 и проверяя результат. Это мы проведём позитивное тестирование. Также необходимо провести негативное тестирование, вводя отрицательные значения и значения больше 150. Сколько в этом случае у нас будет тестов? Огромное количество, так как числовой ряд бесконечен. В этом случае нужно сократить количество тестов и сократить трудозатраты. Для этого необходимо разделить числа на группы (классы эквивалентности). У нас появляется четыре класса эквивалентности:

– несовершеннолетние – все числа больше 0 и меньше или равны 18;

– совершеннолетние – все числа больше или равны 18 и меньше или равны 150;

– отрицательные – все числа меньше нуля (отрицательные);

– долгожитель – все числа больше 150 лет.

Все вводимые в программу числа из класса эквивалентности «несовершеннолетние» будут обрабатываться одинаково и приводить к результату «Доступ запрещён». Все вводимые в программу числа из класса эквивалентности «совершеннолетние» будут обрабатываться одинаково и приводить к результату «Доступ разрешён». Все числа из классов «отрицательные» будут обрабатываться одинаково и приводить к результату «Неверный ввод данных». Все числа из классов «долгожитель» будут обрабатываться одинаково и приводить к результату «Долгожитель, позвоните в поддержку».

Если числа из каждого класса обрабатываются одинаково, нам достаточно из каждого класса взять по одному числу для проведения теста. В этом случае у нас будет четыре теста для проверки функционала контроля возраста программы:

Тест № 1. Входные данные: -5. Ожидаемый результат: «Неверный ввод данных». Класс эквивалентности «отрицательные» (возраст меньше 0).

Тест № 2. Входные данные: 14. Ожидаемый результат: «Доступ запрещён». Класс эквивалентности «несовершеннолетние» (возраст больше 0 и меньше 18).

Тест № 3. Входные данные: 46. Ожидаемый результат: «Доступ разрешён». Класс эквивалентности «совершеннолетние» (возраст больше 18 и меньше 150).

Тест № 4. Входные данные: 178. Ожидаемый результат: «Долгожитель, позвоните в поддержку». Класс эквивалентности

«долгожитель» (возраст больше 150).

Разделив числа по группам (классам), в дальнейшем достаточно взять по одному числу из каждого класса, так как предполагается, что на остальные числа из каждой группы (класса) программа будет реагировать так же. После применения метода эквивалентного разделения в процессе проектирования тестов мы сократили количество проверок до четырёх. Если бы этого не было сделано, то количество проверок было бы бесконечным.

Рассмотрим ещё один пример. Предположим, в организации есть веб-приложение для планирования отпусков. В нём заложена логика:

– если отпуск летом и зимой, а также пересекается с отпуском коллеги, то вывести сообщение «Отказано в отпуске»;

– если отпуск летом и зимой, и не пересекается с отпуском коллеги, то вывести сообщение «Отпуск согласован» и запланировать отпуск сотрудника;

– если отпуск весной и осенью, то ограничений нет, вывести сообщение «Отпуск согласован» и запланировать отпуск сотрудника.

Какие тесты нам необходимо подготовить? Не используя эквивалентное разделение, мы постараемся проверить все варианты и создадим следующие тесты:

– отпуск летом пересекается с отпуском коллеги;
– отпуск осенью пересекается с отпуском коллеги;
– отпуск зимой пересекается с отпуском коллеги;
– отпуск весной пересекается с отпуском коллеги;
– отпуск летом не пересекается с отпуском коллеги;
– отпуск осенью не пересекается с отпуском коллеги;
– отпуск зимой не пересекается с отпуском коллеги;
– отпуск весной не пересекается с отпуском коллеги.

Восемь вариантов только на проверку функционала планирования отпусков, а ведь в веб-приложении ещё много другого функционала, и там по десятку проверок. Так мы его не протестируем и за год. Требуется сократить количество проверок. Что у нас будет веб-приложением обрабатываться одинаково?

Эти две проверки, по сути, одинаковы, и их отнесём к группе № 1:

– отпуск летом пересекается с отпуском коллеги;
– отпуск зимой пересекается с отпуском коллеги.

Аналогично отнесём к группе № 2 следующие проверки:

– отпуск летом не пересекается с отпуском коллеги;
– отпуск зимой не пересекается с отпуском коллеги.

Глянем на эти проверки:

– отпуск весной пересекается с отпуском коллеги;
– отпуск осенью пересекается с отпуском коллеги;
– отпуск весной не пересекается с отпуском коллеги;
– отпуск осенью не пересекается с отпуском коллеги.

Когда выбирается отпуск весной и осенью, там нет никаких ограничений у веб-приложения, а значит, все эти проверки эквивалентные и не пересечение/пересечение отпуска с коллегами никак не влияет на них. Поэтому относим их все к группе № 3. Мы знаем, что программа обрабатывает одинаково данные из одной группы, если они правильно разнесены по группам. Зная это, можно из каждой группы выбрать по одной проверке, предполагая, что остальные будут обрабатываться программой аналогично. В итоге появляется следующий список проверок:

Проверка № 1. Отпуск летом, и отпуск пересекается с отпуском коллеги. Результат: отказ в планировании отпуска.

Проверка № 2. Отпуск зимой, и отпуск не пересекается с отпуском коллеги. Результат: отпуск запланирован.

Проверка № 3. Отпуск весной (не важно пересекается с отпуском коллеги или нет). Результат: отпуск запланирован.

Таким образом необходимо будет проанализировать всё веб-приложение по всему функционалу и составить итоговый список проверок с учётом эквивалентного разбиения. Кстати, обратите внимание: в итоге у нас получился чек-лист проверок, который можно использовать для тестирования программы или написания тест-кейсов.

# Метод анализа граничных значений

*Анализ граничных значений – метод проектирования тестов, который фокусируется на проверке значений на границах диапазона входных данных.*

Англоязычное именование метода «Boundary value analysis». Цель данного метода заключается в выявлении дефектов, которые могут возникнуть в окрестности граничных значений, так как именно здесь часто находятся ошибки программного обеспечения.

Рассмотрим на примере. В программе есть поле для ввода чисел. В данное поле можем вводить только положительные целые числа от 1 до 99. Выявляем границы – это 1 и 99. У каждой границы есть минимальное граничное значение и максимальное граничное значение, которые прилегают к самим границам. Их также необходимо проверить. Исходя из сказанного определяем, что на каждую из границ у нас должно приходиться по три проверки, для нижней границы 0, 1, 2, для верхней границы – 98, 99, 100. Всего шесть проверок (тестов):

– значения на границе: 1 и 99;

– минимальные граничные значения – 0 и 98;

– максимальные граничные значения – 2 и 100.


Проверки чисел за пределами интервала 1–99 – это отрицательные проверки (0, 100), так как по условиям, озвученным ранее, они недопустимы к вводу, но мы их проверяем. Проверки чисел в пределах интервала 1–99, включая сами границы – положительные проверки (1, 2, 98, 99), так как они допустимы к вводу.

Окончательный перечень проверок (тестов):

Проверка № 1. Ввод числа 0 в поле ввода. Результат: ввод числа программой игнорируется.

Проверка № 2. Ввод числа 1 в поле ввода. Результат: ввод числа успешный.

Проверка № 3. Ввод числа 2 в поле ввода. Результат: ввод числа успешный.

Проверка № 4. Ввод числа 98 в поле ввода. Результат: ввод числа успешный.

Проверка № 5. Ввод числа 99 в поле ввода. Результат: ввод числа успешный.

Проверка № 6. Ввод числа 100 в поле ввода. Результат: ввод числа программой игнорируется.

Может возникнуть вопрос: «Почему минимальными и максимальными граничными значениями не могут быть числа 0.99, 1.1, 98.9, 99.9?». Они могли бы быть ими, но только если бы в условии не было чётко сказано: «В данное поле можем вводить только положительные целые числа…». Если бы было сказано, что в поле можно вводить числа с десятичными дробями, то числа, указанные в вопросе, могли быть минимальными и максимальными граничными значениями.

Рассмотрим второй пример. Имеется сайт, на котором можно бронировать отели. Отель можно забронировать с 1 января 2099 года по 31 декабря 2099 года. Сразу определяем границы:

– 1 января 2099 года;

– 31 декабря 2099 года.

Определяем минимальные граничные значения:

– 31 декабря 2098 года (минимальное для границы 1 января 2099 года);

– 30 декабря 2099 года (минимальное для границы 31 декабря 2099 года).

Определяем максимальные граничные значения:

– 2 января 2099 года (максимальное для границы 1 января 2099 года);

– 1 января 2100 года (максимальное для границы 31 декабря 2099 года).


Общий список проверок, следующий:

Проверка № 1. Бронирование отеля. Дата начала проживания 1 января 2099 года. Дата окончания проживания любая в пределах допустимых границ (между 1 января 2099 и 31 декабря 2099). Результат: отель забронирован.

Проверка № 2. Бронирование отеля. Дата окончания проживания 31 декабря 2099 года. Дата начала проживания любая в пределах допустимых границ. Результат: отель забронирован.

Проверка № 3. Бронирование отеля. Дата окончания проживания 30 декабря 2099 года. Дата начала проживания любая в пределах допустимых границ. Результат: отель забронирован.

Проверка № 4. Бронирование отеля. Дата начала проживания 2 января 2099 года. Дата окончания проживания любая в пределах допустимых границ. Результат: отель забронирован.

Проверка № 5. Бронирование отеля. Дата начала проживания 31 декабря 2098 года. Дата окончания проживания любая в пределах допустимых границ. Результат: бронирование недоступно.

Проверка № 6. Бронирование отеля. Дата окончания проживания 1 января 2100 года. Дата начала проживания любая в пределах допустимых границ. Результат: бронирование недоступно.

Составляя список проверок, учитываем, что дата начала проживания не может быть больше даты окончания проживания. Можно составить дополнительные проверки, указывая дату начала проживания равной дате окончания проживания. Попробуйте эти проверки составить самостоятельно.

Изучите список проверок и осмыслите его, чтобы понять суть. Для большего понимания смотрите на даты, отображённые на временном отрезке.

# Метод попарного тестирования

*Попарное тестирование – метод проектирования тестов, при котором тестируемые значения проверяемых параметров хотя бы раз сочетаются с тестируемыми значениями остальных проверяемых параметров.*

Англоязычное именование метода «Pairwise testing». Попарное тестирование представляет собой метод, с помощью которого создаются тесты для проверки всех возможных комбинаций пар параметров. Этот метод основан на идее, что многие дефекты программного обеспечения проявляются именно при взаимодействии между различными параметрами.

Перейдём к рассмотрению на примерах. Есть сайт для заказа продуктов. При заказе продуктов нам надо указать определённые данные:

| Параметр | Значения |
|---|---|
| Регион доставки | Город 1<br>Город 2<br>Город 3 |
| Способ оплаты | Кредитная карта<br>Наличные при доставке<br>Накопленными баллами |
| Скидка | Нет<br>5%<br>10% |

Сколько в нашем случае будет тестов, если связать всё со всем? Региона доставки – 3 шт., способов оплаты – 3 шт., скидок – 3 шт. Перемножаем всё: 3 * 3 * 3 = 27.

Двадцать семь проверок, если все значения параметров проверить в связке со всеми остальными значениями других параметров. Составляем полную таблицу тестов. Все значения региона доставки должны пересечься со всеми значениями способами оплаты и со всеми

значениями скидок. Аналогично все значения способы оплаты должны пересечься со всеми значениями региона доставки и скидок. То же самое и со скидками:

| Регион доставки | Способ оплаты | Скидка |
|---|---|---|
| Город 1 | кредитная карта | нет |
| Город 1 | кредитная карта | 5% |
| Город 1 | кредитная карта | 10% |
| Город 1 | наличные при доставке | нет |
| Город 1 | наличные при доставке | 5% |
| Город 1 | наличные при доставке | 10% |
| Город 1 | накопленными баллами | нет |
| Город 1 | накопленными баллами | 5% |
| Город 1 | накопленными баллами | 10% |
| Город 2 | кредитная карта | нет |
| Город 2 | кредитная карта | 5% |
| Город 2 | кредитная карта | 10% |
| Город 2 | наличные при доставке | нет |
| Город 2 | наличные при доставке | 5% |
| Город 2 | наличные при доставке | 10% |
| Город 2 | накопленными баллами | нет |
| Город 2 | накопленными баллами | 5% |
| Город 2 | накопленными баллами | 10% |
| Город 3 | кредитная карта | нет |
| Город 3 | кредитная карта | 5% |
| Город 3 | кредитная карта | 10% |
| Город 3 | наличные при доставке | нет |
| Город 3 | наличные при доставке | 5% |
| Город 3 | наличные при доставке | 10% |
| Город 3 | накопленными баллами | нет |
| Город 3 | накопленными баллами | 5% |
| Город 3 | накопленными баллами | 10% |

Много проверок, не правда ли? Теперь надо сократить количество тестов. Нам необходимо выбрать из имеющегося списка тесты таким образом, чтобы в тестах хотя бы раз проверялось значение каждого параметра, и чтобы оно пересеклось хотя бы раз со значениями других параметров, и постараться повторяться минимальное количество раз.

К примеру, есть регион доставки «Город 1». Нам нужно создать тест, чтобы при проведении тестирования с этим регионом доставки было задействовано любое одно значение из способа оплаты и любое одно значение из скидки. Вот этот тест:

| Регион доставки | Способ оплаты | Скидка |
|---|---|---|
| Город 1 | кредитная карта | нет |

Другие тесты с регионом доставки «Город 1», со способом оплаты «кредитная карта» и со скидкой «нет» создавать не нужно, так как они уже задействованы в одном из тестов. В следующем тесте необходимо задействовать другие значения параметров. Второй тест:

| Регион доставки | Способ оплаты | Скидка |
|---|---|---|
| Город 2 | наличные при доставке | 5% |

Во втором тесте нет значений, которые использовались в первом тесте и используются все параметры – регион доставки, способ оплаты, скидка. Осталось по аналогии создать следующие тесты. В итоге получится финальная таблица не повторяющихся тестов, в которых задействованы все значения имеющихся параметров:

| Регион доставки | Способ оплаты | Скидка |
|---|---|---|
| Город 1 | кредитная карта | нет |
| Город 2 | наличные при доставке | 5% |
| Город 3 | накопленными баллами | 10% |

Список проверок уменьшен с двадцати семи до трёх. Имеющиеся тесты покрыли все имеющиеся значения всех параметров. Только что мы применили метод попарного тестирования. Давайте переведём строки таблицы в осмысленные проверки:

Проверка № 1. Оформить доставку продуктов, выбрав регион доставки «Город 1», способ оплаты «кредитная карта», скидка «нет» (отсутствует скидка). Результат: доставка продуктов оформлена.

Проверка № 2. Оформить доставку продуктов, выбрав регион доставки «Город 2», способ оплаты «наличные при доставке», скидка «5%». Результат: доставка продуктов оформлена.

Проверка № 3. Оформить доставку продуктов, выбрав регион доставки «Город 3», способ оплаты «накопленными баллами», скидка «10%». Результат: доставка продуктов оформлена.

Метод попарного тестирования полезен в случаях, когда количество комбинаций параметров велико, и тестирование всех возможных случаев становится непрактичным. Во многих программах попадается огромное количество параметров и их значений. В этом случае сгруппировать их по методу попарного тестирования вручную трудозатратно или не представляется возможным. В этом случае используют специализированные инструменты, для автоматического составления таблиц, аналогичных нашей.

# Метод проектирования по таблице решений

Метод попарного тестирования помогает проверить все значения параметров из большого множества, но с минимально возможным количеством тестов. Что делать, когда все параметры важны, их все необходимо проверить и ничего нельзя упустить? В данном случае может использоваться метод проектирования тестов по таблице решений.

*Проектирование по таблице решений – метод проектирования тестов, который используется для проектирования тестов с целью тщательной проверки различных комбинаций входных данных в зависимости от имеющихся условий и действий.*

Англоязычное именование метода «Decision table testing». Данный метод особенно полезен, когда существует множество возможных комбинаций условий и необходимо убедиться, что каждая комбинация протестирована. Когда создают таблицу решений, специалист по тестированию продумывает, какие условия и какие действия должны происходить в программе. Потом эти пары условий и действий записываются в строки таблицы. Условия пишутся сверху, а действия – снизу. Каждый нумерованный столбец в таблице представляет собой особое правило, где определены условия и действия, связанные с этим правилом. Часто значения условий записывают в виде «да» или «нет».

Рассмотрим метод на относительно простом примере. Имеется сайт продажи билетов на концерты. У него есть определённые правила (условия) продажи билетов. Как видим в таблице, у нас есть условия (левая часть) и действие (правая часть):

| Условие | Действие |
|---|---|
| Возраст покупателя меньше или равен 18 | Скидка на билет 18% |
| Возраст покупателя больше 18 и меньше или равен 40 | Скидка на билет 40% |
| Возраст покупателя больше 40 | Скидка на билет 70% |
| Возраст покупателя равен 100 | К скидке добавляется 10% |

Нам надо спроектировать тесты, учитывая все условия. Если их много, то мы не сможем удержать в голове и проработать все комбинации. Для этого будем проектировать тесты с помощью таблицы решений:

| номера столбцов тестов >> | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| **Условие** | | | | |
| № 1. Возраст покупателя <= 18 лет? | Д | Н | Н | Н |
| № 2. Возраст покупателя > 18 и <= 40 лет? | Н | Д | Н | Н |
| № 3. Возраст покупателя > 40 лет? | Н | Н | Д | Д |
| № 4. Возраст покупателя = 100 лет? | Н | Н | Н | Д |
| | | | | |
| **Действие** | | | | |
| № 1. Скидка на билет 18%. | Д | Н | Н | Н |
| № 2. Скидка на билет 40%. | Н | Д | Н | Н |
| № 3. Скидка на билет 70%. | Н | Н | Д | Д |
| № 4. К скидке добавляется 10%. | Н | Н | Н | Д |

Условия перечислены в левой верхней части таблицы, а действия – в левой нижней. Каждый нумерованный столбец справа содержит бизнес-правило. Каждое правило утверждает, вкратце: «В этой конкретной комбинации условий (отображённой в верхней части правила) необходимо выполнить конкретную комбинацию действий (отображённую в нижней части правила)».

Д – это «Да», Н – это «Нет». Ещё может быть прочерк: значит всё равно, выполняется дополнительно это условие или нет, совместно с другим условием. В таблице, по сути, каждый нумерованный столбец – один тест-кейс. Разберём подробнее каждый столбец. Далее речь пойдёт о нумерованных столбцах в правой части таблицы решений.

Условия:

Столбец № 1 – столбец для условия № 1. Условие № 1 должно выполняться (Д). Остальные условия могут с ним выполняться? Нет (Н). Т. е. если человеку меньше или равно 18 лет, то ему не может быть

больше 18 или 40, поэтому в столбце 1 «Д» проставляется только у первого условия. Остальные столбцы к первому условию не относятся, так как они для других условий.

Столбец № 2 – столбец для условия № 2. Условие № 2 должно выполняться (Д). Остальные условия могут с этим условием выполняться? Нет (Н).

Столбец № 3 – столбец для условия № 3. Условие № 3 должно выполняться (Д). Остальные условия могут с ним выполняться? Нет (Н).

Столбец № 4 – столбец для условия № 4 и № 3. Условие № 4 должно выполняться (Д). Остальные условия могут с этим условием выполняться? Да, только условие № 3 (Д). Здесь поясним. Если пользователю 100 лет, то в столбце 4 напротив данного условия ставим «Д». Если пользователю 100 лет, то ему больше 40 лет? Да, поэтому в столбце 4 напротив условия № 3 ставим «Д».

Действия:

Столбец № 1. Действие № 1 должно выполняться для условия № 1? Да (Д). Остальные действия должны выполняться с этим действием? Нет (Н).

2. Столбец № 2. Действие № 2 должно выполняться для условия № 2? Да (Д). Остальные действия должны выполняться с этим действием? Нет (Н).

3. Столбец № 3. Действие № 3 должно выполняться для условия № 3? Да (Д). Остальные действия должны выполняться с этим действием? Нет (Н).

4. Столбец № 4. Действие № 4 должно выполняться для условия № 4? Да (Д). Остальные действия должны выполняться с этим действием? Да, только действие № 3 (Д). Здесь поясним. Если пользователю 100 лет, то в столбце 4 напротив данного условия ставим «Д» – ему положена дополнительная скидка в 10%. Если пользователю 100 лет, ему больше 40 лет? Да, ему положена скидка 70%.

Смотрим на подготовленную нами таблицу решений и фиксируем перечень проверок, внимательно читая условия и действия, которые должны выполняться при выполнении условия. Ориентируемся на «Д»:

Проверка № 1. Купить билет пользователю, которому меньше 18 лет. Результат: билет куплен, дана скидка 18%.

Проверка № 2. Купить билет пользователю, которому больше 18 и меньше или равно 40 лет. Результат: билет куплен, дана скидка 40%.

Проверка № 3. Купить билет пользователем, которому больше 40 лет. Результат: билет куплен, дана скидка 70%.

Проверка № 4. Купить билет пользователем, которому ровно 100 лет. Результат: билет куплен, дана скидка 80% (70% – больше 40 лет и дополнительно 10% – ему 100 лет).

Составив таблицу решений, специалист по тестированию, проверяя программу, знает, что проверять и какие правила должны отрабатывать. В итоге при проверке он ничего не упускает. Если условий и/или действий будет больше, чем у нас, таблица будет иметь большее количество нумерованных столбцов, а соответственно и проверок.

Преимущества метода проектирования по таблице решений:

– гарантируется покрытие всех возможных комбинаций условий;

– таблицы решений легко читать и понимать;

– пропущенные комбинации легко заметить, так как они просто отсутствуют в таблице.

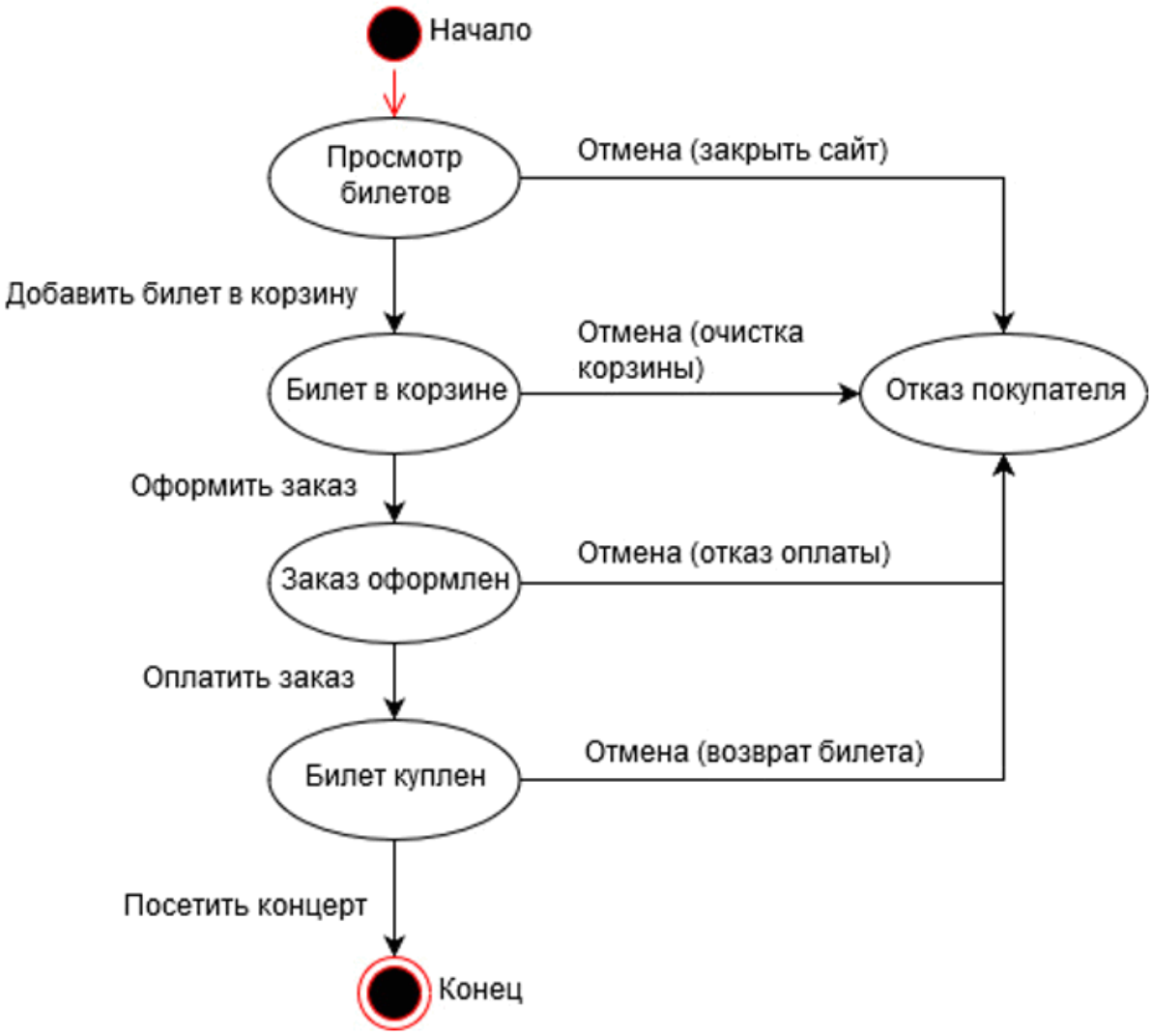
Данный метод полезен при тестировании сложных информационных систем, где существует множество различных вариантов взаимодействия между условиями и действиями.

# Метод проектирования по таблице переходов состояний

*Проектирование по таблице переходов состояний – метод проектирования тестов для тестирования программ, которые могут находиться в различных состояниях и переходить между ними в ответ на определённые события.*

Англоязычное именование метода «State transition testing». Таблица переходов – это как схема, где записаны все возможные переходы между разными состояниями чего-либо, например, игрового персонажа или светильника. В ней есть разные состояния, например, включено или выключено, и события, которые могут их изменить, например, нажатие на кнопку. Мы создаём тесты, чтобы удостовериться, что программа правильно работает в разных ситуациях.

Рассмотрим пример. Имеется сайт покупки билетов на концерты. Рассмотрим процесс покупки билетов. У сайта есть состояния, в которых он пребывает (круги и овалы на диаграмме):




Из одного состояния мы можем привести систему в другое состояние. Из одного состояния система может перейти в другое состояние различными путями. Обратите внимание на состояние «Отказ покупателя», куда система может попасть при выполнении одного из четырёх событий. У системы есть начальное и конечное состояние.

Проанализировав базис тестирования и изучив функционал программы, в нашем случае проанализировав диаграмму, мы можем составить таблицу переходов, которая поможет нам подготовить тесты на её основании:

| | Начало | Просмотр билетов | Билет в корзине | Заказ оформлен | Билет куплен | Отказ покупателя | Конец |
|---|---|---|---|---|---|---|---|
| Начало | - | + | - | - | - | - | - |
| Просмотр билетов | - | - | + | - | - | + | - |
| Билет в корзине | - | - | - | + | - | + | - |
| Заказ оформлен | - | - | - | - | + | + | - |
| Билет куплен | - | - | - | - | - | + | + |
| Отказ покупателя | - | - | - | - | - | - | - |
| Конец | - | - | - | - | - | - | - |

Как видим, в таблице зафиксированы все возможные состояния, включая начало и конец процесса оформления билетов. Таблица читается слева направо.

Первая строка таблицы. Рассматриваем состояние «Начало». Посмотрите на диаграмму. В какое состояние система может попасть из состояния «Начало»? Только в состояние «Просмотр билетов» – пользователь зашёл на сайт и начал просмотр билетов. В таблице на пересечениях «Начало» – «Просмотр билетов» ставим плюс, на остальных пересечениях минусы, так как в другие состояния напрямую система попасть не может.

Вторая строка таблицы. Рассматриваем состояние «Просмотр билетов». Посмотрите на диаграмму. В какое состояние система может попасть из состояния «Просмотр билетов»? В состояние «Билет в корзине», когда пользователь добавит билет в корзину для покупки, и в состояние «Отказ покупателя», когда пользователь уйдёт с сайта. В таблице на пересечениях «Просмотр билетов» – «Билет в корзине» и «Просмотр билетов» – «Отказ покупателя» ставим плюсы, на остальных пересечениях минусы, так как в другие состояния напрямую система попасть не может.

По аналогии проставляем «+» и «-» по остальным строкам, изучая работу процесса покупки билетов на сайте по диаграмме. В итоге у нас

в таблице появляется девять плюсов – девять проверок. Составьте самостоятельно чек-лист проверок на основании имеющейся таблицы.

Преимущества техники проектирования по таблице переходов:

– гарантируется тестирование всех возможных переходов между состояниями;

– таблица чётко представляет все переходы, что облегчает понимание при проектировании тестов.

Этот метод полезен при тестировании программ, в которых важно проверить переходы между различными состояниями.

## Метод проектирования по сценариям использования

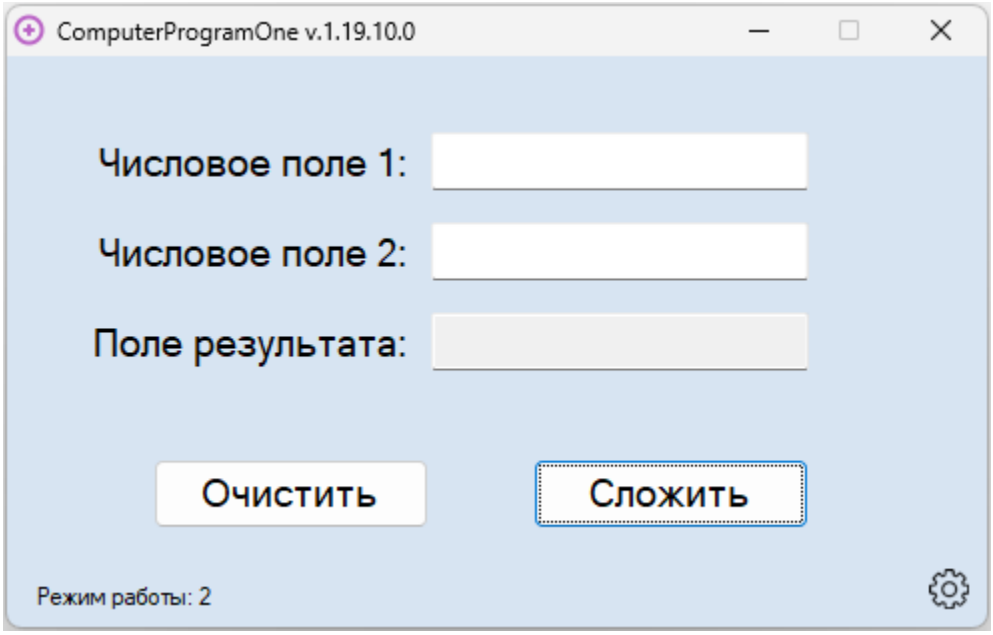
*Проектирование тестов по сценариям использования – метод проектирования тестов, подразумевающий создание тестов на основании сценариев использования системы.*

Англоязычное именование метода «Scenario testing». При проектировании тестов по сценариям использования[29] каждый тест покрывает отдельный сценарий использования программы от начала до конца. Такой подход гарантирует проверку всей цепочки функций согласно реальному применению программы. Сценарий использования выглядит следующим образом:

| | |
|---|---|
| **Цель:** | Проверка наличия полей на форме |
| **Действующее лицо** | Пользователь |
| **Предусловия:** | Пользователь запустил приложение и находится на основной форме |
| **Основной сценарий:** | |
| Шаг | Действие |
| 1. | Пользователь на главной форме программы изучает наличие элементов |
| 2. | Программа отображает поле ввода первого числа |
| 3. | Программа отображает поле ввода второго числа |
| 4. | Программа отображает поле отображения результатов |
| **Результат:** | Пользователь увидел все обязательные поля программы |

Как видите, ничего особенного в сценарии использования нет. Подобные сценарии использования являются основой для проектирования тестов методом, который мы рассматриваем в данный момент.

Пример. Имеется программа для сложения чисел:




Сценарий использования, который приведён в качестве примера, написан для данной программы. Он описывает проверку наличия полей на форме программы. Специалист по тестированию изучает сценарий использования и пишет по нему тест-кейс:

| | | |
|---|---|---|
| | **Номер тест-кейса:** | 4 |
| | **Название:** | Проверка наличия полей ввода и результата программы |
| | **Предусловия:** | Программа «Computer Program One» запущена |
| | | |
| | **Шаги** | **Ожидаемый результат** |
| 1 | Проверить наличие поля «Числовое поле 1» | Поле «Числовое поле 1» отображается на форме программы |
| 2 | Проверить наличие поля «Числовое поле 2» | Поле «Числовое поле 1» отображается на форме программы |
| 3 | Проверить наличие поля «Поле результата» | Поле «Поле результата» отображается на форме программы |

Аналогичным образом пишутся тест-кейсы по всем остальным имеющимся сценариям использования.

## Метод синтаксического тестирования

*Синтаксическое тестирование – метод проектирования тестов, направленный на проверку корректности обработки программой данных в соответствии с форматами, заданными в спецификации.*

Англоязычное именование метода «Syntax testing». Цель тестов, спроектированных с помощью данного метода – убедиться, что программа корректно работает как с корректными, так и с некорректными по формату данными. Суть подхода заключается в подаче на вход программы как корректных данных правильной структуры, так и недопустимых значений с нарушением структуры и последующем анализе реакции программы. Это позволяет протестировать устойчивость программы к потенциально возможному некорректному синтаксису данных на этапах ввода и обработки данных, а также возможности обработки корректного синтаксиса данных.

Примеры проверяемых данных:

– проверка корректности обработки даты и времени программой для бронирования билетов (корректный и некорректный формат дат и времени);

– ввод имени пользователя в поле поиска социальной сети с разными символами;

– загрузка XML файла в программу, которая анализирует файлы XML с корректной и некорректной внутренней структурой XML файла;

– проверка загрузки в программу CSV файла, содержащего данные с соблюдением и нарушением внутренней структуры (разделителей);

– загрузка корректных и некорректных форматов изображений в графический редактор;

– проверка отображения браузером HTML документа с соблюдением и нарушением стандартов;

– загрузка в программу документов поддерживаемых и неподдерживаемых форматов (docx, pdf и т. д.);

– выгрузка отчёта из программы в форматах XML, CSV, XLS и проверка корректности структуры выгруженных данных.

Ожидаемые результаты:

– обработка корректных данных без ошибок;

– выдача сообщений об ошибках при некорректных данных;

– отсутствие зависаний программы и отказ в обработке на некорректных данных.

Рассмотрим простой пример проектирования тестов с целью проверки программы для сложения чисел. Программа предназначена для *сложения положительных целых чисел, минимальное число для ввода 1, максимальное число для ввода 9999*. Нам известен формат данных, принимаемых программой, поэтому, исходя из этих условий, проектируем тесты только на проверку форматов данных.

Тесты с корректными данными:

– ввод в поля ввода однозначного положительного целого числа;

– ввод в поля ввода двузначного положительного целого числа;

– ввод в поля ввода трёхзначного положительного целого числа;

– ввод в поля ввода четырёхзначного положительного целого числа.

Тесты с некорректными данными:

– ввод в поля ввода дробных чисел;

– ввод в поля ввода букв латинского и кириллического алфавита;

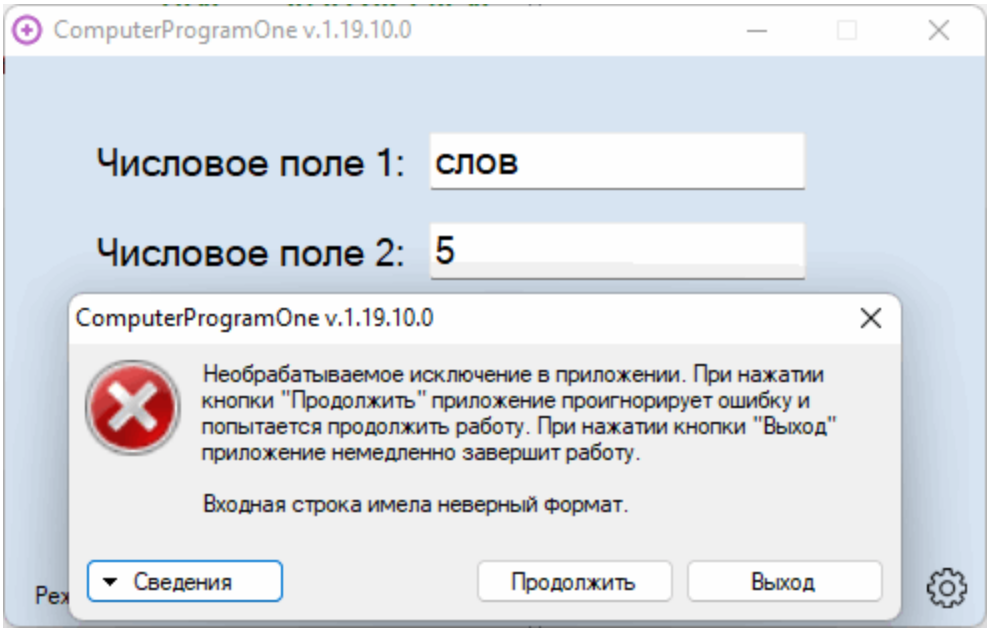
– ввод в поля ввода спецсимволов.

В итоге имеем семь тестов, на которые программа должна корректно реагировать: при вводе корректных данных допускать ввод и дальнейшее сложение; при вводе некорректных данных не завершать работу с ошибкой и не давать проводить операцию сложения введённых данных.

## Метод проектирования на основании случайных данных

*Проектирование на основании случайных данных – метод проектирования тестов, который предполагает случайный выбор входных данных для тестирования программы.*

Англоязычное именование метода «Random testing». Основной принцип этого метода заключается в проверке реакции программы на случайные сценарии, что помогает выявить неожиданные дефекты. В процессе проверки с использованием тестов, созданных с помощью данного метода, результаты выходных данных сравниваются с программными спецификациями для подтверждения успешного или неуспешного прохождения теста. Если спецификации отсутствуют, используются исключения языка: если тест вызывает исключение во время выполнения, это указывает на наличие ошибки в программе. Пример исключения, вызванного в программе вводом данных, которые программа не может обработать:




Тестами, созданными с помощью данного метода, можно проверить, как программа обрабатывает разнообразные варианты использования,

которые могли бы остаться незамеченными при тестировании с фиксированными данными.

Рассмотрим примеры применения тестов, основанных на этом методе:

Пример № 1. Редактор изображений. Пользователь случайным образом выбирает изображение в редакторе и применяет случайные изменения к его размеру и цвету. Такой подход помогает проверить, как редактор обрабатывает различные комбинации изменений.

Пример № 2. Электронная почта. Пользователь случайным образом создаёт новое электронное письмо, генерируя случайные темы и текст сообщения. Это может помочь проверить, как почтовый клиент обрабатывает различные варианты содержания письма.

Пример № 3. Онлайн-магазин. Пользователь случайным образом выбирает товары в интернет-магазине и добавляет их в корзину. Это может помочь проверить, как интернет-магазин обрабатывает различные комбинации товаров в процессе оформления заказа.

Пример № 4. Социальная сеть. Пользователь случайным образом выбирает несколько своих друзей и отправляет им приглашение для подключения к социальной сети. Такой способ тестирования может выявить, как система обрабатывает разные сценарии взаимодействия с друзьями.

Пример № 5. Календарь. Пользователь случайным образом создаёт события в календаре, выбирая случайные даты, время и описания. Такой подход помогает проверить, как календарь обрабатывает различные форматы данных и временные интервалы.

Пример № 6. Онлайн-калькулятор. Пользователь случайным образом вводит математические выражения в калькуляторе, проверяя, как программа обрабатывает различные операции и форматы ввода данных.

Пример № 7. Мессенджер. Пользователь случайным образом вставляет смайлики и эмодзи в чат, оценивая, как мессенджер обрабатывает разнообразные графические элементы в сообщениях.

Пример № 8. Банковское приложение. Пользователь случайным образом проводит операции в банковском приложении, такие как переводы, оплаты и запросы баланса. Такое тестирование помогает проверить, как приложение управляет различными финансовыми операциями.

Помните, мы с вами рассматривали один из принципов тестирования – парадокс пестицида, когда одни и те же тесты перестают быть эффективными для обнаружения дефектов в программе? Данный метод проектирования тестов нам помогает избегать парадокса пестицида.

Теперь рассмотрим подробнее пример проектирования тестов с помощью данного метода. Нам необходимо создать тесты, чтобы с их помощью проверять функционал сложения чисел в программе, которая складывает максимум четырёхзначные числа. Неопытный специалист при создании тест-кейса скорее всего напишет следующий тест-кейс:

| | | |
|---|---|---|
| | **Номер тест-кейса:** | 12 |
| | **Название:** | Сложение положительных целых чисел |
| | **Предусловия:** | Программа «Computer Program One» запущена |
| | | |
| | **Шаги** | **Ожидаемый результат** |
| 1 | Ввести с клавиатуры в поле «Числовое поле 1» число 22. | В поле «Числовое поле 1» отображается число 22. |
| 2 | Ввести с клавиатуры в поле «Числовое поле 2» число 33. | В поле «Числовое поле 2» отображается число 33. |
| 3 | Нажать на кнопку «Сложить» | В поле «Поле результата» отображается число 55. |

Однако опытный тестировщик, используя метод случайного тестирования, напишет следующий тест-кейс:

| | | |
|---|---|---|
| | **Номер тест-кейса:** | 12 |
| | **Название:** | Сложение положительных целых чисел |
| | **Предусловия:** | Программа «Computer Program One» запущена |
| | | |
| | **Шаги** | **Ожидаемый результат** |
| 1 | Ввести с клавиатуры в поле «Числовое поле 1» любое случайное положительное целое число, содержащее от 1-го до 4-х знаков. | В поле «Числовое поле 1» отображается введённое число. |
| 2 | Ввести с клавиатуры в поле «Числовое поле 2» любое случайное положительное целое число, содержащее от 1-го до 4-х знаков. | В поле «Числовое поле 2» отображается введённое число. |
| 3 | Нажать на кнопку «Сложить». | В поле «Поле результата» отображается сумма чисел «Числовое поле 1» + «Числовое поле 2». |

При тестировании программы по данному тест-кейсу специалисты по тестированию будут всегда вводить случайные числа и не будут повторяться. Аналогичным образом создаются тест-кейсы для тестирования программ других классов и предназначений. К примеру, для тестирования интернет-магазина в тест-кейсе пишут «добавить в корзину случайный товар из случайной категории товаров», а не «добавить в корзину телевизор марки 1105» и т. д.

## Метод дерева классификации

*Метод дерева классификации (классификационного дерева) – метод, используемый для проектирования тестов на основании различных условий и их комбинаций, описанных с помощью дерева классификации.*

Англоязычное именование метода «Classification tree method». Этот метод помогает определить, какие тесты следует провести, чтобы максимально охватить разнообразные сценарии работы программы.

Алгоритм проектирования тест-кейсов по данному методу:

1) Определить основные факторы, которые влияют на поведение программы.

2) Определить значения для каждого фактора.

3) Построить дерево классификации, где каждый узел представляет собой один из факторов, а ветви из узла – соответствующие значения.

4) Определить комбинации значений для конечных тест-кейсов.

Рассмотрим на примере интернет-магазина.

Определяем факторы, которые влияют на поведение программы:

– Категория товаров.

– Способ оплаты.

– Тип пользователя.

Определяем значения для каждого фактора:

1. Фактор: Категория товаров:

1.1. Значение: Электроника.

1.2. Значение: Одежда.

1.3. Значение: Книги.

2. Фактор: Способ оплаты:

2.1. Значение: Картой.

2.2. Значение: Наличными.

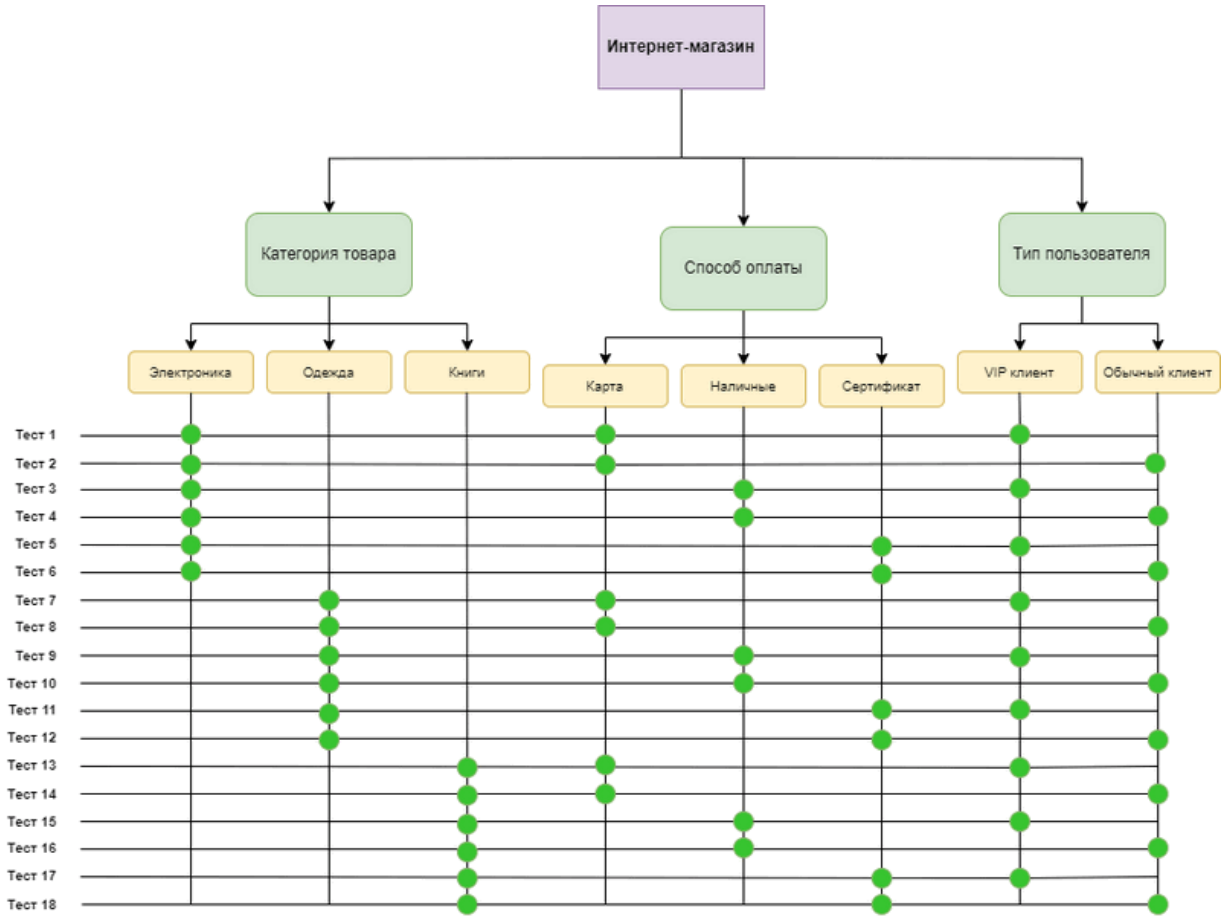
2.3. Значение: Сертификатом.

3. Фактор: Тип пользователя:

3.1. Значение: Обычный клиент.

3.2. Значение: VIP-клиент.

Строим дерево классификаций:




Самый верхний узел дерева – тестируемая программа (интернет-магазин). Следующий уровень узлов – факторы программы (категория товара, способ оплаты, тип пользователя). Следующий уровень узлов – значения факторов (электроника, одежда, книги и т. д.) или дополнительные факторы, вложенные в основной фактор. Какие дополнительные факторы могут быть? Пример:

Фактор 1 уровня: Тип пользователя:

1.1. Фактор 2 уровня: Зарегистрированный:

1.1.1. Значение: VIP-пользователь.

1.1.2. Значение: Обычный пользователь.

1.2. Фактор 2 уровня: Незарегистрированный:

1.2.1. Значение: VIP-пользователь.

1.2.2. Значение: Обычный пользователь.

С дополнительными факторами разобрались. Их вложенность зависит от сложности программы.

После того как диаграмма создана, требуется от каждого значения провести линию вниз. После этого – провести ряд горизонтальных линий. У нас получилась сетка линий под диаграммой. Для наглядности горизонтальные линии подписаны («Тест 1», «Тест 2» и т. д.). Смотрим на дерево классификаций. Теперь перейдём к рассмотрению тестов. «Тест 1» (первая горизонтальная линия):

1) Ставим маркер на пересечении горизонтальной линии и первой слева вертикальной линии под значением «Электроника». Этим мы обозначили, что надо проверить покупку электроники. Вертикальные линии напротив одежды и книг пропускаем, так как их мы не собираемся в данном тесте покупать.

2) Ставим маркер на пересечении горизонтальной линии и вертикальной линии под значением «Карта». Этим мы обозначаем, что будем покупать электронику, оплачивая её картой. Вертикальные линии напротив наличные и сертификат пропускаем, так как не собираемся платить наличными и сертификатом.

3) Ставим маркер на пересечении горизонтальной линии и вертикальной линии под значением «VIP-клиент». Этим мы обозначаем, что будем покупать электронику, оплаченную картой от имени VIP-клиента. Вертикальную линию напротив обычного клиента пропускаем, так как не собираемся в этом тесте от его имени совершать покупку.

Глянув на «Тест 1», мы видим, что, выполняя данный тест, нам необходимо войти в личный кабинет интернет-магазина под VIP-клиентом и купить товар из категории электроники, оплатив его картой. Аналогичным образом расставляем маркеры для остальных тестов. В итоге получилось 18 тестов. Теперь, имея эти данные, можно приступать к написанию чек-листа проверок или тест-кейсов, опираясь на данные дерева классификаций.

Сейчас попробуйте самостоятельно дописать тесты, продолжив вниз сетку дерева классификации, зная, что:

– совместно можно купить товары: из электроники и одежды; электроники и книг; одежды и книг; из всех имеющихся типов товаров;

– одновременно можно совершать оплату: картой и сертификатом; наличными и сертификатом.

Метод дерева классификации помогает систематизировать и оптимизировать проектирование тестов, покрывая максимальное количество возможных сценариев использования программы. Если программа имеет сложный и разветвлённый набор функций, то составление дерева классификации вручную сложное и трудоёмкое занятие или не представляется возможным. В этом случае, как и в методе попарного тестирования, используют специализированные инструменты для автоматического формирования дерева классификации.

# Метод причинно-следственного графа

Метод причинно-следственного графа[30] сложен в понимании, поэтому в процессе чтения данной главы будьте внимательны к деталям. Для понимания материала данной главы, возможно, стоит взять ручку и бумагу, чтобы в процессе чтения параллельно зарисовывать и записывать определённые моменты, повторяя за автором книги. Это позволит быстрее понять принципы данного метода. Я, в свою очередь, постараюсь просто и ёмко рассказать о данном методе проектирования тестов.

*Метод причинно-следственного графа – метод для создания тестов, основанных на анализе взаимосвязей между различными факторами в программе, использующий графическое моделирование причинно-следственных связей.*

Англоязычное именование метода «Cause-Effect graphing». Основная идея метода заключается в выявлении причинно-следственных связей между входными данными (причинами) и ожидаемым поведением программы (следствиями). Следствия также называют «эффектами», т. е., подав некие входные данные в программу, мы достигаем определённого эффекта (результата). Под входными данными тут подразумеваются определённые данные, переданные в программу, соблюдение определённых условий в программе или события, произошедшие в программе. Этот метод используется для проектирования тестов на основании спецификации требований.

Алгоритм проектирования тест-кейсов по данному методу:

1) С целью облегчения процесса проектирования тестов, спецификация требований разбивается на отдельные небольшие логические (функциональные) блоки.

2) Определяются входные условия (причины) и поведение программы (следствия).

3) Каждой причине и следствию присваивается свой идентификатор и составляется список причин и следствий.

4) Составляется причинно-следственный граф. Он делается от руки или в подходящей программе.

5) На основании причинно-следственного графа формируется таблица принятия решений.

6) На основании таблицы принятия решений создаются тест-кейсы.

Прежде чем приступать к практической части, рассмотрим базовые обозначения, которые используются для построения причинно-следственного графа. В ходе рассмотрения будем употреблять следующие понятия:

1) Причина (обозначается как «П») выполняется/не выполняется – это означает, что данное входное условие должно сработать или не сработать. К примеру, нажатие на кнопку «Сложения» – это причина. Причина – это событие, которое должно произойти в программе, или условие, которое должно выполниться в программе, чтобы в ней произошло определённое действие (наступило следствие).

2) Следствие (обозначается как «С») выполняется/не выполняется – это означает, что в программе должно произойти или не произойти определённое действие, достигнут определённый результат. К примеру, числа сложились, и программа вывела результат сложения – это следствие. Следствия – это то, что получим в программе после срабатывания определённых условий (причин).

Рассмотрим сокращённые записи сказанного, которые будем использовать далее:

1) П = 1 – причина выполняется. П = 0 – причина не выполняется.

2) С = 1 – следствие выполняется, С = 0 – следствие не выполняется.

Чтобы окончательно понять суть, рассмотрим сокращённую запись на примере. Есть требование к программе:

– если нажали на кнопку «Сложить», программа складывает числа и выводит результат, иначе программа не складывает числа и не выводит результат.

Вот так это будет выглядеть в причинно-следственной формулировке:

– если причина выполняется (нажали на кнопку «Сложить»), то следствие также выполняется (программа складывает числа и выводит результат), в противном случае следствие не выполняется (программа не складывает числа и не выводит результат).

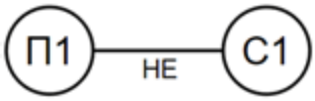
Так выглядит сокращённая причинно-следственная формулировка:

– если П = 1, то С = 1, иначе С = 0.

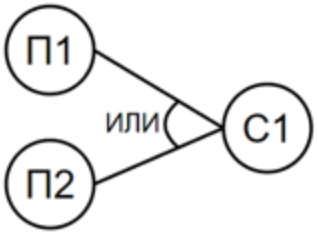
Надеюсь, принцип понятен. Далее будем оперировать сокращёнными записями. Кстати, не привязывайтесь к программе сложения, она указана для примера и, рассказывая далее о причинах и следствиях, под ними ничего конкретного не будет подразумеваться. Конкретику рассмотрим на примерах. Также хочу обратить ваше внимание на то, что причин и следствий может быть множество, и в этом случае они будут обозначаться как П1, П2, П3, С1, С2, С3 и т. д. Начинаем описание базовых визуальных обозначений, которые используются для построения графов и которые помогут нам их создавать.

Функция «тождество» – показывает, что если П1 = 1, то С1 = 1, если П = 0, то С = 0:

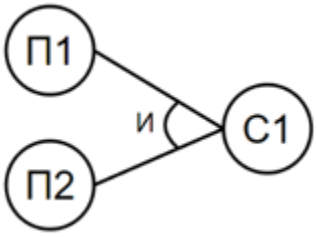

Функция «НЕ» – показывает, что если П1 = 1, то С1 = 0, если П = 0, то С = 1:



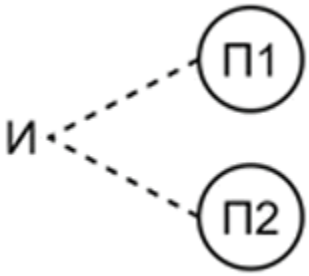

Функция «ИЛИ» – показывает, что если П1 = 1 или П2 = 1, то С1 = 1, если П1 = 1 и П2 = 1, то С1 = 1, если П1 = 0 и П2 = 0, то С1 = 0:



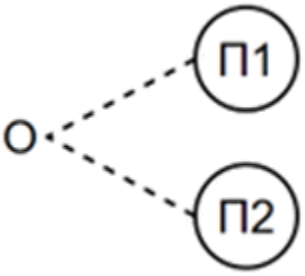

Функция «И» – показывает, что если П1 = 1 и П2 = 1, то С1 = 1, если П1 = 0 или П2 = 0, то С1 = 0:




Ограничение «исключение» – показывает, что П1 и П2 могут быть оба равны 0 или только один из них, может быть равен 1. Допустимые случаи: П1 = 0 и П2 = 0; П1 = 1 и П2 = 0; П1 = 0 и П2 = 1:

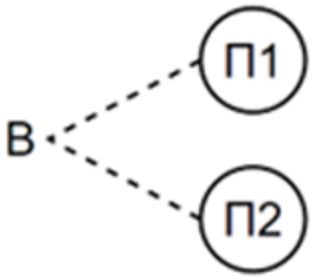



Ограничение «только один» – показывает, что П1 и П2 не могут быть одновременно равны 0 или 1. Один из них обязательно должен быть равен 1, а остальные равны 0. Допустимые случаи: П1 = 1 и П2 = 0; П1 = 0 и П2 = 1:

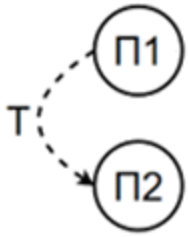



Ограничение «включение» – показывает, что П1 и П2 не могут быть одновременно равны 0. Минимум один из них обязательно должен

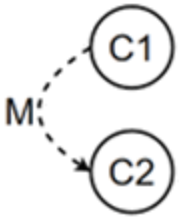
быть равен 1. В то же время они могут быть равны все 1 или несколько из них могут быть равны 1, если их больше двух элементов. Допустимые случаи: П1 = 1 и П2 = 0; П1 = 0 и П2 = 1; П1 = 1 и П2 = 1:




Ограничение «требуется» – показывает, что если П1 = 1, то и П2 должно быть равно 1. Нельзя чтобы П1 = 1, а П2 = 0:




Ограничение «маски» (скрытия) для следствий – показывает, что если С1 = 1, то С2 должно быть равно 0:




Все обозначения элементов разобрали. Пора приступать к рассмотрению примеров. Как я уже говорил, данный метод проектирования тестов не прост, поэтому будем рассматривать его на относительно простых примерах. За основу возьмём программу для

сложения чисел, которая нас сопровождает на протяжении всей книги. У данной программы есть спецификация требований. Предположим, что, изучив требования, мы логически разбили их на небольшие блоки, чтобы удобнее было проектировать тесты. Первый блок требований:

| Номер | Название | Требование |
|---|---|---|
| ТПИ-1.1 | Формы. | Программа имеет две формы:<br>- Основная форма, появляется при запуске программы.<br>- Форма предупреждения о закрытии программы, появляется при закрытии программы. |
| ТПИ-1.8 | Форма предупреждения о закрытии программы. Надписи. | На форме имеется надпись «Вы собираетесь выйти из программы. Продолжить?». |
| ТПИ-1.9 | Форма предупреждения о закрытии программы. Кнопки. | На форме имеются кнопки:<br>- Кнопка «Да».<br>- Кнопка «Нет».<br>- Кнопка закрытия формы (в виде крестика). |
| ТПИ-1.10 | Отображение программы в панели задач операционной системы. | При развёрнутом и свёрнутом состоянии программы в панели задач всегда отображается кнопка программы. |

Проанализировав требования, нам необходимо определить причины и следствия, не забывая их сразу нумеровать. Начнём со следствий, чтобы легче было понять.

Полный список следствий:

1) С1. Появление основной формы программы.

2) С2. Появление формы, предупреждающей о закрытии.

3) С3. Наличие надписи на форме предупреждения «Вы собираетесь выйти…»

4) С4. Наличие кнопок на форме предупреждения: «Да», «Нет», «Закрытие формы».

5) С5. Отображение кнопки программы на панели задач операционной системы при развёрнутой программе.

6) С6. Отображение кнопки программы на панели задач операционной системы при свёрнутой программе.

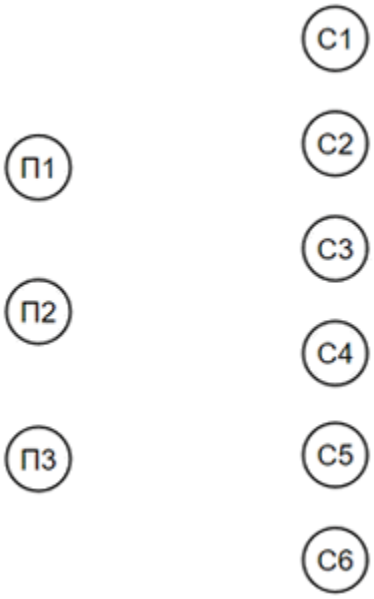
Полный список причин:

1) П1. Запустить программу.

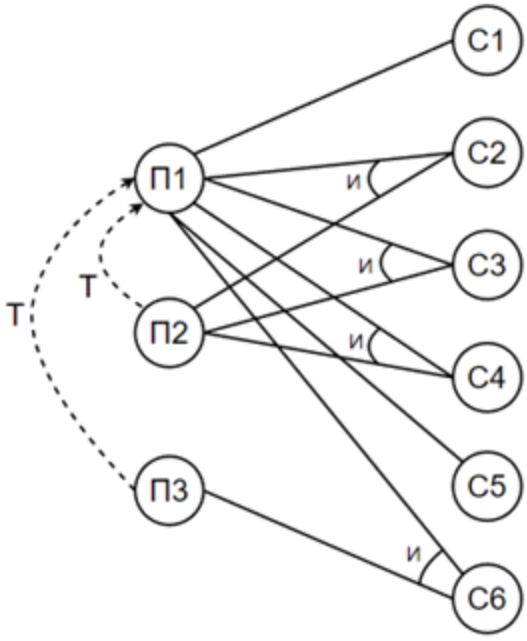
2) П2. Выйти из программы через кнопку закрытия программы.

3) П3. Свернуть программу.

Для всех следствий мы определили все возможные причины (условия для следствий). Теперь строим причинно-следственный граф. Для начала размещаем все причины и следствия без связей:




В данном случае разместили причины слева, а следствия справа. Далее создаём связи с учётом того, какое следствие от каких причин зависит, также требуется учесть все функции и ограничения. Получили следующий граф:




Рассмотрим все связи, которые отображены на причинно-следственном графе. Отталкиваться будем от следствий:

1) Проверка появления основной формы программы (С1). Для этого необходимо запустить программу (П1). Других условий нет, поэтому построена единственная связь от П1 до С1.

2) Проверка появления формы, предупреждающей о закрытии программы (С2). Для этого программа должна быть запущена (П1) и требуется выйти из программы через кнопку закрытия программы (П2), поэтому между связями П1 и П2 идущими к С2 указана функция «И». Также имеется ограничение (Т), указывающее, что для П2 обязательно должна отработать причина П1, т. е. чтобы выйти из программы, она должна быть запущена.

3) Проверка наличия надписи на форме предупреждения «Вы собираетесь выйти…» (С3). Для этого программа должна быть запущена (П1), и требуется выйти из программы через кнопку закрытия программы (П2), чтобы появилась форма, предупреждающая о закрытии, на которой имеется надпись, поэтому между связями П1 и П2, идущими к С3, указана функция «И». Также имеется ограничение (Т), указывающее, что для П2 обязательно должна отработать причина П1.

4) Проверка наличия кнопок на форме предупреждения (С4). Для этого программа должна быть запущена (П1), и требуется выйти из

программы через кнопку закрытия программы (П2), чтобы появилась форма, предупреждающая о закрытии, на которой имеются кнопки, поэтому между связями П1 и П2, идущими к С4, указана функция «И». Также имеется ограничение (Т), указывающее, что для П2 обязательно должна отработать причина П1.

5) Отображение кнопки программы на панели задач операционной системы при развёрнутой программе (С5). Для этого необходимо запустить программу (П1). Других условий нет, поэтому одна связь от П1 до С5.

6) Отображение кнопки программы на панели задач операционной системы при свёрнутой программе (С6). Для этого программа должна быть запущена (П1), и необходимо свернуть программу (П3), поэтому между связями П1 и П3, идущими к С6, указана функция «И». Также имеется ограничение (Т) указывающее, что для П3 обязательно должна отработать причина П1, т. е. чтобы свернуть программу, она должна быть запущена.

После составления причинно-следственного графа формируется таблица принятия решений:

| номера столбцов тестов >> | **1** | **2** | **3** | **4** | **5** | **6** |
|---|---|---|---|---|---|---|
| **Причина (условие)** | | | | | | |
| П1. Запустить программу. | 1 | 1 | 1 | 1 | 1 | 1 |
| П2. Выйти из программы через кнопку закрытия программы. | 0 | 1 | 1 | 1 | 0 | 0 |
| П3. Свернуть программу. | 0 | 0 | 0 | 0 | 0 | 1 |
| | | | | | | |
| **Следствие (действие)** | | | | | | |
| С1. Появление основной формы программы. | 1 | 0 | 0 | 0 | 0 | 0 |
| С2. Появление формы, предупреждающей о закрытии. | 0 | 1 | 0 | 0 | 0 | 0 |
| С3. Наличие надписи на форме предупреждения «Вы собираетесь выйти…». | 0 | 0 | 1 | 0 | 0 | 0 |
| С4. Наличие кнопок на форме предупреждения: «Да», «Нет», «Закрытие формы». | 0 | 0 | 0 | 1 | 0 | 0 |
| С5. Отображение кнопки программы на панели задач операционной системы при развёрнутой программе. | 0 | 0 | 0 | 0 | 1 | 0 |
| С6. Отображение кнопки программы на панели задач операционной системы при свёрнутой программе. | 0 | 0 | 0 | 0 | 0 | 1 |

Перед собой мы видим таблицу, похожую на ту, которую составляли при рассмотрении метода проектирования тестов по таблице решений. Есть небольшая особенность заполнения нумерованных столбцов. Их удобнее заполнять, отталкиваясь от следствий. Сейчас рассмотрим с вами заполнение нескольких столбцов.

Столбец № 1. Первое следствие (действие) – это появление основной формы программы (С1). Ставим напротив этого следствия 1, т. е. в данном случае проверять будем, что оно выполняется. Чтобы это следствие выполнилось, необходимо, чтобы запустили программу (П1), соответственно, ставим напротив этой причины 1. Остальные данные в столбце заполняем нулями, так как другие следствия не проверяем, и для следствия С1 не требуется выполнения других причин.

Столбец № 6. В данном случае будет проверяться следствие С6 – отображение кнопки программы на панели задач, когда программа свёрнута. Чтобы это следствие проверить, надо свернуть программу, а значит напротив причины П3 устанавливаем 1 в этом столбце, а также необходимо обязательное условие, чтобы программа была запущена (вспомните на графе ограничение Т), а это означает, что напротив причины П1 также устанавливаем 1 в этом столбце. Напротив других следствий и причин устанавливаем 0.

По аналогии заполняются данные и в других нумерованных столбцах. По итогу у нас получится 6 нумерованных столбцов.

На основании полученной таблицы создаются тест-кейсы. Рассмотрим пример тест-кейса по нумерованному столбцу 6:

| | | |
|---|---|---|
| | **Номер тест-кейса:** | 86 |
| | **Название:** | Отображение кнопки программы на панели задач операционной системы при свёрнутой программе. |
| | **Предусловия:** | |
| | | |
| | **Шаги** | **Ожидаемый результат** |
| 1 | Запустить программу «Computer Program One» с помощью ярлыка, располагающегося на рабочем столе компьютера. | Программа запустилась. Отображается основная форма программы. |
| 2 | Навести курсор компьютерной мыши на кнопку сворачивания окна программы и нажать левую кнопку мыши. | Программа свернулась в панель задач операционной системы. Кнопка программы отображается на панели задач. |

Обратите внимание, что в шаге 1 тест-кейса учтена причина П1, в шаге 2 учтена причина П3, а заголовок учёл следствие С6. Таким образом составляются остальные тест-кейсы, что является последним шагом методики причинно-следственного графа.

Хочу сделать акцент: рассмотренная таблица принятия решений может быть записана в сокращённом варианте. Однако его трудно читать, так как в этом случае надо много данных держать в голове или постоянно обращаться к составленному списку причин и следствий и причинно-следственному графу, но вы должны о данном варианте знать:

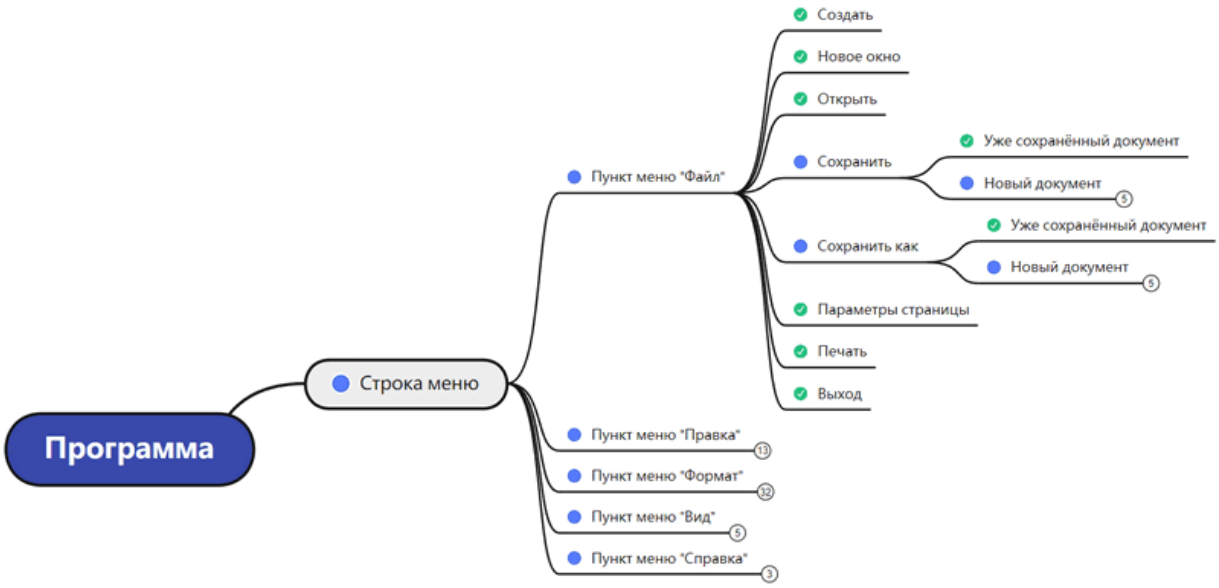
| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | | | | | | |
| П1 | 1 | 1 | 1 | 1 | 1 | 1 |
| П2 | 0 | 1 | 1 | 1 | 0 | 0 |
| П3 | 0 | 0 | 0 | 0 | 0 | 1 |
| | | | | | | |
| С1 | 1 | 0 | 0 | 0 | 0 | 0 |
| С2 | 0 | 1 | 0 | 0 | 0 | 0 |
| С3 | 0 | 0 | 1 | 0 | 0 | 0 |
| С4 | 0 | 0 | 0 | 1 | 0 | 0 |
| С5 | 0 | 0 | 0 | 0 | 1 | 0 |
| С6 | 0 | 0 | 0 | 0 | 0 | 1 |

Мы с вами пошагово рассмотрели метод причинно-следственного графа для проектирования тестов. Обратите внимание, что мы охватили малую часть спецификации требований программы. Для создания тестов на основании всей спецификации требований необходимо всю спецификацию требований разбить на небольшие части и по отдельности на каждую часть создать тест-кейсы через прорисовку графов. Таким образом мы упростим процесс создания тестов. Если спецификацию требований не разбивать на мелкие составляющие, то выявить все причины и следствия и попробовать создать один большой граф будет проблематично. Мы получим нечитаемый граф, из которого также проблематично будет составить таблицу принятия решений, а соответственно и тест-кейсы.

## Метод проектирования на основании модели

*Проектирование на основании модели – это подход к проектированию тестов, предполагающий формализацию поведения программы в некой абстрактной модели с целью дальнейшей генерации тестов по модели, по которым уже тестируется реально разработанная программа.*

Данный метод также называют «тестирование на основании модели». Англоязычное именование метода «Model based testing». Под моделью подразумевается графическое описание поведения программы. В качестве модели могут выступать функциональные карты, схема переходов, дерево классификаций, причинно-следственные графы, UML-диаграммы и прочее. Т. е. в качестве модели может использоваться любое графическое и схематическое представление функциональности и поведения программы. Пример одного из вариантов модели программы в виде функциональной карты:




Схему переходов вы уже видели в главе, посвящённой методу проектирования тестов по таблице переходов, дерево классификаций описано в главе, посвящённой методу проектирования тестов по дереву классификаций, причинно-следственный граф показан в главе,

посвящённой методу проектирования тестов по причинно-следственному графу.

Принцип создания тестов по данному методу:

1) Изучение базиса тестирования. Изучается вся доступная документация по разрабатываемой программе.

2) Создание модели. Разрабатывается абстрактная модель программы, которая представляет её ключевые аспекты, состояния, переходы и взаимодействия с окружающей средой.

3) Создание тестов. На основании подготовленной модели создаются тесты. Они представляют собой последовательности действий и входных данных, направленные на проверку различных состояний системы и её функциональности.

После того как программа будет разработана и передана в тестирование, проводится тестирование программы с использованием разработанных тестов, и результаты сравниваются с ожидаемыми результатами.

Данный метод проектирования тестов напоминает некоторые из ранее рассмотренных методов проектирования тестов. Однако он не так формализован. К примеру, в методе причинно-следственного графа, по сути, создаётся модель – причинно-следственный граф, и на основании графа (модели) создаются тесты. В методе проектирования тестов на основании модели в отличии от метода причинно-следственного графа всё намного проще. Однако суть такая же, как и у других похожих методов – создать модель и на её основании создать проверки (тесты). Примеры разбирать не будем, так как при описании других методов рассматривались примеры, когда на базе определённой модели программы создавались тесты.

# Методы белого ящика

Методы проектирования тестов из категории белого ящика основываются на знании внутренней структуры объекта тестирования – исходного кода программы. Методы белого ящика используются на уровне компонентного и интеграционного компонентного тестирования, когда пишутся юнит-тесты.

Главы, посвящённые методам белого ящика, будут понятны специалистам, которые знают какой-либо язык программирования, специалистам, которые находятся в процессе изучения какого-либо языка программирования, или специалистам, которые не сталкивались с языками программирования, но вопреки всему хотят разобраться в том, что здесь написано. Не переживайте, если в процессе чтения вы осознаете, что ничего не понимаете в описании методов белого ящика. Пропустите эти главы и продолжайте чтение с тех, что посвящены методам, основанным на опыте. В будущем, когда станете опытнее, сможете вернуться к пропущенным главам.

Рассказывая о методах проектирования тестов методами белого ящика постараюсь приводить простые и понятные большинству читателей примеры. Здесь не будут строиться сложные структуры юнит-тестов, так как наша цель – не попытка научить писать юнит-тесты. Цель – показать принципы проектирования тестов рассматриваемыми методами, на основании которых вы сможете проектировать тесты. Неопытным специалистам, которые собираются начать писать юнит-тесты, просьба учесть эту информацию и не воспринимать примеры, как готовые решения, которые можно скопировать и применить в своих проектах. Они дают простое понимание принципов применения описываемых методов, но не дают понимания основ написания юнит-тестов.

## Метод тестирования операторов

*Тестирование операторов – метод проектирования тестов, при котором проверяется, что каждый оператор в коде исполняется*

*хотя бы один раз.*

Англоязычное именование метода «Statement testing». Операторы – это те части кода, которые выполняют конкретные действия, такие как присваивание значений, выполнение условных операций и циклов. В методе тестирования операторов должно проверяться каждое утверждение (оператор) в программе по отдельности. То есть мы «прощупываем» каждый отдельный оператор программы с помощью тестов. Это и есть суть данного метода.

**Пример № 1**

Рассмотрим тестирование функции, которая складывает все элементы одного массива и возвращает результат сложения:

JS app.js > ...

```
// Функция, которая возвращает сумму элементов массива
function sumArrayElements(arr) {
    // Оператор 1: Инициализация переменной для хранения суммы
    let sum = 0;

    // Оператор 2: Цикл для перебора элементов массива
    for (let i = 0; i < arr.length; i++) {
        // Оператор 3: Сложение текущего элемента с общей суммой
        sum += arr[i];
    }

    // Оператор 4: Возврат общей суммы
    return sum;
}

// Тест 1: Случай, когда массив содержит положительные числа
let testArray1 = [1, 2, 3, 4, 5];
console.log("Результат теста 1:", sumArrayElements(testArray1)); // 15

// Тест 2: Случай, когда массив содержит отрицательные числа
let testArray2 = [-1, -2, -3, -4, -5];
console.log("Результат теста 2:", sumArrayElements(testArray2)); // -15

// Тест 3: Случай, когда массив содержит положительные и отрицательные числа
let testArray3 = [-1, 2, -3, 4, 5];
console.log("Результат теста 3:", sumArrayElements(testArray3)); // 7

// Тест 4: Случай, когда массив пуст
let testArray4 = [];
console.log("Результат теста 4:", sumArrayElements(testArray4)); // 0
```

В этом примере:

– оператор 1 инициализирует переменную sum для хранения суммы элементов;

– оператор 2 представляет собой цикл, который перебирает элементы массива;

– оператор 3 складывает текущий элемент массива с общей суммой;

– оператор 4 возвращает общую сумму.

**Пример № 2**

Рассмотрим тестирование функции, которая осуществляет сложение двух чисел и возврат результата сложения:

JS app.js > ...

```
// Функция для сложения двух чисел
function add(x, y) {
    // Оператор 1: присваивание значения x переменной a
    let a = x;

    // Оператор 2: присваивание значения y переменной b
    let b = y;

    // Оператор 3: сложение переменных a и b и присваивание результата переменной result
    let result = a + b;

    // Оператор 4: возврат результата из функции
    return result;
}

// Тест 1: Проверка, что функция корректно складывает положительные числа
console.log("Результат теста 1:", add(2, 3)); // 5

// Тест 2: Проверка, что функция корректно складывает отрицательные числа
console.log("Результат теста 2:", add(-5, -7)); // -12

// Тест 3: Проверка, что функция корректно складывает положительное и отрицательное
число
console.log("Результат теста 3:", add(10, -3)); // 7
```

В этом примере:

– оператор 1 присваивает значение x переменной a;

– оператор 2 присваивает значение y переменной b;

– оператор 3 складывает a и b и присваивает результат переменной result;

– оператор 4 возвращает результат из функции.

Тестирование операторов выполняется с использованием нескольких тестов, каждый из которых проверяет разные аспекты функции сложения. Такие тесты помогают убедиться, что каждый оператор выполняется правильно, и функция работает корректно в различных сценариях.

# Метод тестирования решений

В программировании решение – это место в коде программы, где она принимает решение о том, какую логику выполнить в зависимости от какого-то условия.

*Тестирование решений – это метод проектирования тестов, который подразумевает создание тестов, проверяющих, что программные решения принимаются правильно в зависимости от различных сценариев.*

Англоязычное именование метода «Decision testing». Главная цель тестов, основанных на данном методе – убедиться, что при различных входных данных программа принимает правильные решения. Это важно, потому что ошибки в решениях могут привести к неправильной работе программы. Если она принимает неправильное решение при определённых условиях, это может повлечь нежелательные последствия.

**Пример № 1**

Рассмотрим тестирование функции, которая проверяет число на чётность или нечётность и возвращает результат проверки:

JS app.js > ...

```
// Функция проверки четности числа
function checkNumber(number) {
    if (number % 2 === 0) {
        return "Чётное";
    } else {
        return "Нечётное";
    }
}

// Тест 1: Случай, когда число чётное положительное
console.log("Результат теста 1:", checkNumber(8)); // Чётное

// Тест 1: Случай, когда число чётное отрицательное
console.log("Результат теста 2:", checkNumber(-8)); // Чётное

// Тест 3: Случай, когда число нечётное положительное
console.log("Результат теста 3:", checkNumber(15)); // Нечётное

// Тест 4: Случай, когда число нечётное отрицательное
console.log("Результат теста 4:", checkNumber(-15)); // Нечётное
```

В этом примере:

– функция checkNumber принимает значение;

– решение основано на проверке наличия или отсутствия остатка от деления;

– различные тесты передают различные числа для проверки решения.

**Пример № 2**

Рассмотрим тестирование функции, которая проверяет число передано в функцию или не число:

JS app.js > ...

```
// Функция проверки типа данных
function checkDataType(value) {
    if (typeof value === 'number') {
        return "Число";
    } else {
        return "Не число";
    }
}

// Тест 1: Случай, когда значение - число
console.log("Результат теста 1:", checkDataType(42)); // Число

// Тест 2: Случай, когда значение - строка
console.log("Результат теста 2:", checkDataType("Hello")); // Не число

// Тест 3: Случай, когда значение - логическое значение
console.log("Результат теста 3:", checkDataType(true)); // Не число

// Тест 4: Случай, когда значение - массив
console.log("Результат теста 4:", checkDataType([1, 2])); // Не число
```

В этом примере:

– функция checkDataType принимает значение;

– решение основано на проверке типа данных с использованием typeof;

– различные тесты подразумевают разные типы входных данных для проверки определённого решения.

## Метод тестирования потоков данных

*Тестирование потоков данных – это метод проектирования тестов, которые фокусируются на проверке того, как данные перемещаются и изменяются внутри программы.*

Англоязычное именование метода «Data flow testing». Суть метода заключается в исследовании путей, по которым данные перемещаются от момента ввода в программу до вывода результата. Основной упор делается на тестирование различных потоков данных и проверке их

корректного перемещения через различные компоненты программы. Поток данных – это путь, по которому данные проходят через программу.

Цель тестов, спроектированных по этому методу – убедиться, что в управлении данными внутри программы нет потерь данных, их искажения или некорректной обработки. Тесты помогают выявлять проблемы в алгоритмах обработки данных, а также обнаружить возможные утечки, перепутывания или искажения информации в процессе выполнения программы. Этот метод особенно полезен в сложных системах, где множество компонентов взаимодействуют друг с другом, и правильный поток данных является критическим для функциональности программы.

**Пример № 1**

Допустим, у нас есть простой калькулятор, который складывает и вычитает числа. В нём есть две переменные для вводимых чисел, кнопки для операции (сложение или вычитание), и переменная, в которой хранится результата.

Ввод данных:

– вводим первое число 5 и помещаем его в переменную А;

– вводим второе число 3 и помещаем его в переменную Б.

Хранение данных:

– переменная А содержит первое число – 5;

– переменная Б содержит второе число – 3.

Процесс обработки данных:

– выбираем операцию сложения;

– программа берет число из переменной А, число из переменной Б и складывает их;

– результат помещается в переменную Г.

Вывод результата:

– программа берет число из переменной Г и выводит результат – 8.

При тестировании потоков данных необходимо убедиться, что на каждом этапе все происходит так, как задумано:

Тест № 1. Ввод данных. Подаём на вход числа 5 и 3. Убеждаемся, что они правильно попали в переменные А и Б.

Тест № 2. Хранение данных. Проверяем, что переменные А и Б правильно содержат введённые числа 5 и 3.

Тест № 3. Процесс обработки данных. Выбираем операцию сложения и проверяем, что результат 8 правильно появился в переменной Г.

Тест № 4. Вывод результата. Убеждаемся, что программа правильно выводит результат 8, который хранился в переменной Г.

**Пример № 2**

Предположим, у нас есть простая система авторизации в программе. Пользователь вводит свой логин и пароль, после чего система проверяет их правильность. Если логин и пароль верны, то пользователь успешно авторизуется, в противном случае он получает сообщение об ошибке.

Ввод данных:

– пользователь вводит логин и пароль, и они помещаются в переменные.

Хранение данных:

– система хранит логин и пароль в переменных.

Процесс обработки данных:

– система проверяет, совпадают ли логин и пароль с данными в базе.

Вывод результата:

– если данные верны, пользователь авторизуется;

– в противном случае, система выводит сообщение об ошибке.

Тестирование:

Тест № 1. Ввод данных. Подаём на вход корректные логин и пароль. Убеждаемся, что они правильно попали в переменные.

Тест № 2. Хранение данных. Проверяем, что переменные правильно содержат введённые логин и пароль.

Тест № 3. Процесс обработки данных. Проверка совпадения данных с данными в базе данных.

Тест № 4. Вывод результата. Убеждаемся, что программа авторизует пользователя.

Аналогичные тесты пишем для ввода некорректных логина и пароля и убеждаемся, что система выводит сообщение об ошибке.

Примеры не прописывал кодом, так как тесты, описываемые тем форматом, который задан в данной книге, будут неестественно и непонятно выглядеть, если не описать их готовую структуру в виде юнит-тестов. Поэтому в данном методе обойдёмся без примера кода. Однако надеюсь и без него суть вам понятна.

# Метод тестирования ветвлений

В программировании ветвление – это возможность изменять поток выполнения программы в зависимости от условий. Например, если что-то истинно, делаем одну вещь, а если ложно – другую.

*Тестирование ветвлений – это метод проектирования тестов, который подразумевает создание тестов, которые будут проверять, что все ветвления в программе работают правильно.*

Англоязычное именование метода «Branch testing». Тесты, написанные с помощью данного метода, проверяют, что истинные и ложные условия обрабатываются правильно, когда в коде есть «если – то – иначе» (if – else) или другие условные операторы. Этот метод фокусируется на тестировании каждой ветви (ветки) в программе, независимо от условий внутри ветвей, чтобы гарантировать, что каждая ветвь программы была выполнена хотя бы один раз в процессе тестирования. Если в коде есть ветвление A и B, тестирование ветвлений покрывает оба случая (выполнение A и B), независимо от конкретных значений внутри условий.

**Пример № 1**

Рассмотрим тестирование функции программы, которая проверяет, какую оценку ученик получил на контрольной, и хвалит ученика или даёт совет. Если оценка 4 или 5, сообщается, что ученик молодец. Если 3, 2 или 1, сообщается, что нужно больше заниматься. В программе есть ветвление. Одна ветка с 4 и 5. Другая ветка с 1, 2 и 3. При тестировании ветвлений нас интересует каждая ветка по отдельности:

JS app.js > ...

```
// Функция проверки оценки
function checkGrade(grade) {

    // Первая ветка: на 4 или 5
    if (grade === 4 || grade === 5) {
        return "Молодец!";
    }

    // Вторая ветка: на 3, 2 или 1
    if (grade === 3 || grade === 2 || grade === 1) {
        return "Нужно больше учиться";
    }

}

// Тест 1: Для первой ветки
console.log("Результат теста 1:", checkGrade(5)); // Молодец

// Тест 2: Для первой ветки
console.log("Результат теста 2:", checkGrade(2)); // Нужно больше
учиться
```

В первом тесте мы подаём на вход 5 и смотрим правильный результат для первой ветки программы. Во втором тесте подаём 2 и проверяем работу второй ветки.

**Пример № 2**

Рассмотрим тестирование функции, которая проверяет, сдал студент экзамен или нет. Если баллов больше или равно 60, то считаем, что студент сдал экзамен:

JS app.js > ...

```
// Функция, которая проверяет, сдал ли студент экзамен
function checkExam(score) {

    if (score >= 60) {
        return "Вы сдали экзамен!";
    } else {
        return "Вы не сдали экзамен.";
    }

}

// Тест 1: Для первой ветки, когда баллов больше 60
console.log("Результат теста 1:", checkExam(75)); // Вы сдали экзамен

// Тест 3: Для второй ветки, когда баллов меньше 60
console.log("Результат теста 3:", checkExam(45)); // Вы не сдали экзамен
```

Функция checkExam принимает баллы студента в качестве аргумента. Внутри функции есть ветвление с использованием условного оператора if – else. Если баллы больше или равны 60, выполняется код внутри блока if, иначе выполняется код внутри блока else. Тестирование ветвлений проводится с помощью двух вызовов функции, чтобы гарантировать, что каждая ветвь программы была выполнена хотя бы один раз в процессе тестирования.

## Метод тестирования условий ветвлений

В программировании условия ветвлений подобны решениям, которые программа принимает на основании текущей ситуации. Они позволяют программе приспосабливаться к разным сценариям и принимать разные решения в зависимости от условий.

*Тестирование условий ветвлений – это метод проектирования тестов, который подразумевает создание тестов, проверяющих каждое возможное условие в программе, чтобы убедиться, что они работают правильно.*

Англоязычное именование метода «Branch condition testing». Этот метод фокусируется на проверке всех условий внутри ветвлений, чтобы убедиться, что каждое условие влияет на принятие решения, чтобы гарантировать, что каждое условие внутри ветвления было протестировано как истинное, так и ложное, с целью охватить все возможные ветки выполнения кода. Если в коде есть ветвление с условиями A и B, тестирование условий ветвлений проверит оба случая (истинное и ложное) для каждого условия (A и B).

**Пример № 1**

Рассмотрим тестирование функции, которую ранее рассматривали. Функция проверяет, какую оценку ученик получил на контрольной, и хвалит ученика или даёт совет. Если оценка 4 или 5, сообщается, что ученик молодец. Если 3, 2 или 1, сообщается, что нужно больше заниматься:

JS app.js > ...

```
// Функция проверки оценки
function checkGrade(grade) {

    // Первая ветка: на 4 или 5
    if (grade === 4 || grade === 5) {
        return "Молодец!";
    }

    // Вторая ветка: на 3, 2 или 1
    if (grade === 3 || grade === 2 || grade === 1) {
        return "Нужно больше учиться";
    }

}

// Тест 1: Для первого условия первой ветки
console.log("Результат теста 1:", checkGrade(4)); // Молодец

// Тест 2: Для второго условия первой ветки
console.log("Результат теста 2:", checkGrade(5)); // Молодец

// Тест 3: Для первого условия второй ветки
console.log("Результат теста 3:", checkGrade(3)); // Нужно больше учиться

// Тест 4: Для второго условия второй ветки
console.log("Результат теста 4:", checkGrade(2)); // Нужно больше учиться

// Тест 5: Для третьего условия второй ветки
console.log("Результат теста 5:", checkGrade(1)); // Нужно больше учиться
```

В программе есть два ветвления, и у каждого из них есть несколько условий. Одна ветка с 4 и 5 имеет два условия. Другая ветка с 1, 2 и 3 имеет три условия.

Если при проектировании тестов методом тестирования ветвлений проверялась работа каждого ветвления и для этого достаточно было двух тестов, то при проектировании тестов методом тестирования условий ветвлений проверяются все прописанные в коде условия каждого ветвления, и у нас вместо двух тестов создаётся пять тестов.

**Пример № 2**

Рассмотрим тестирование функции, которая начисляет бонусы в зависимости от суммы покупки – amount и наличия у покупателя

подписки – isSubscribed:

```
JS app.js > ...
1   // Функция начисления бонусов за покупку
2   function addBonuses(amount, isSubscribed) {
3
4       if (amount > 1000) {
5           if (isSubscribed) {
6               return "Бонусы начислены!";
7           }
8       }
9
10      return "Бонусы не начисляются";
11  }
12
13  // Тест 1: Для первого условия amount > 1000, нет подписики
14  console.log("Результат теста 1:", addBonuses(2000, false)); // Бонусы не начисляются
15
16  // Тест 2: Для второго условия amount < 1000, есть подписка
17  console.log("Результат теста 2:", addBonuses(500, true)); // Бонусы не начисляются
18
19  // Тест 3: Для третьего условия amount > 1000, есть подписка
20  console.log("Результат теста 3:", addBonuses(2500, true)); // Бонусы начислены
```

Условия начисления бонусов:

– сделана покупка на сумму больше 1 000 денежных единиц;

– есть подписка на email-рассылку.

Тесты проверяют:

Тест № 1. Покупка на сумму больше 1 000 денежных единиц, нет подписки. Бонусы не должны начисляться.

Тест № 2. Покупка на сумму меньше 1 000 денежных единиц, есть подписка. Бонусы не должны начисляться.

Тест № 3. Покупка на сумму больше 1 000 денежных единиц, есть подписка. Бонусы должны начисляться.

В тестах проверяем условия по отдельности и в совокупности.

## Метод тестирования комбинаций условий ветвлений

Комбинация – это объединение или совмещение различных сущностей или элементов, создающее новое целое.

*Тестирование комбинаций условий ветвлений – это метод проектирования тестов, с помощью которого проектируются тесты, направленные на проверку различных комбинаций истинных и ложных значений условий внутри ветвления в программном коде.*

Англоязычное именование метода «Branch condition combination testing». Этот метод проектирования тестов направлен на создание тестов, которые проверяют различные комбинации истинных и ложных значений условий внутри ветвления, чтобы гарантировать, что все возможные комбинации истинных и ложных значений каждого условия внутри ветвления были протестированы. Это важно для уверенности в правильной работе программы в различных сценариях, когда несколько условий взаимодействуют друг с другом. Если в коде есть ветвление с условиями A и B, тестирование комбинаций условий ветвлений проверит все возможные комбинации: A = true, B = true; A = true, B = false; A = false, B = true; A = false, B = false.

**Пример № 1**

Представим, что нам надо составить условия для программирования роботизированного помощника по дому. Он должен выполнять разные задачи в зависимости от условий. У нас получилось три условия:

Условие № 1:

– если у помощника есть сливочное масло, он может готовить;

– если у помощника нет масла, он должен отправиться в магазин и купить его.

Условие № 2:

– если владелец помощника хочет омлет, помощник должен разогреть сковороду;

– если владелец хочет суп, помощник должен закипятить воду в кастрюле.

Условие № 3:

– если помощнику не нравится готовить, он будет отдыхать;

– если у помощника есть настроение готовить, он будет готовить.

Составляя тесты по методу, который мы рассматриваем, у нас получатся тесты, которые проверяют, как помощник будет вести себя, когда условия объединены:

Тест № 1. У помощника нет масла, заказали омлет, и у него хорошее настроение.

Тест № 2. У помощника нет масла, заказали суп, и у него нет настроения готовить.

Тест № 3. У помощника нет масла, заказали суп, и у него хорошее настроение.

И так далее.

Таким образом проверяем разные комбинации условий (1, 2, 3) вместе. Это важно, поскольку мы хотим быть уверенными, что помощник может правильно реагировать на разные сценарии одновременно. Подобное тестирование помогает убедиться, что наш помощник справляется с разными ситуациями, когда условия объединены. Теперь рассмотрим, как реализовать тесты в коде:

JS app.js > ...

```
function make(isButter, isOmelette, isMood) {
    // isButter - есть масло? isOmelette - - готовить омлет? isMood - есть настроение?
    if (!isMood) {
        return 'Сегодня готовите сами';
    } else if (!isButter) {
        return 'Помощник идёт в магазин';
    } else if (isOmelette) {
        return 'Приготовить омлет';
    } else {
        return 'Приготовить суп';
    }
}

// Тест 1
console.log("Результат теста 1:", make(true, true, true)); // Приготовить омлет
// Тест 2
console.log("Результат теста 2:", make(true, true, false)); // Сегодня готовите сами
// Тест 3
console.log("Результат теста 3:", make(true, false, false)); // Сегодня готовите сами
// Тест 4
console.log("Результат теста 4:", make(false, false, false)); // Сегодня готовите сами
// Тест 5
console.log("Результат теста 5:", make(false, false, true)); // Помощник идёт в магазин
// Тест 6
console.log("Результат теста 6:", make(false, true, true)); // Помощник идёт в магазин
// Тест 7
console.log("Результат теста 7:", make(false, true, false)); // Сегодня готовите сами
// Тест 8
console.log("Результат теста 8:", make(true, false, true)); // Приготовить суп
```

**Пример № 2**

Рассмотрим тестирование функции, которая возвращает информацию, как нам сегодня одеваться в зависимости от того, холодно сегодня и идёт ли дождь:

```
function getDressed(isCold, isRainy) {
  // isCold - холодно сегодня? isRainy - дождь идёт?
  if (isCold && isRainy) {
    return "Возьми зонт и надень тёплую куртку";
  } else if (isCold) {
    return "Надень тёплую куртку";
  } else if (isRainy) {
    return "Возьми зонт";
  } else {
    return "Надевай, что угодно";
  }
}
// Тест 1
console.log("Результат теста 1:", getDressed(true, true)); // Возьми зонт и надень тёплую куртку
// Тест 2
console.log("Результат теста 2:", getDressed(true, false)); // Надень теплую куртку
// Тест 3
console.log("Результат теста 3:", getDressed(false, true)); // Возьми зонт
// Тест 4
console.log("Результат теста 4:", getDressed(false, false)); // Надевай, что угодно
```

В этом примере:

Тест № 1. Проверка, что при холоде и дожде программа предлагает «Возьми зонт и надень тёплую куртку».

Тест № 2. Проверка, что при холоде без дождя программа предлагает «Надень тёплую куртку».

Тест № 3. Проверка, что при отсутствии холода и если идёт дождь на улице, программа предлагает «Возьми зонт».

Тест № 4. Проверка, что при отсутствии холода и дождя программа предлагает «Надевай, что угодно».

После запуска этих тестов, если они завершатся с результатами, которые будут равны ожидаемым, мы можем быть уверены, что наш код корректно реагирует на различные комбинации условий внутри ветвления функции getDressed.

# Метод модифицированного покрытия условий и решений

*Модифицированное покрытие условий и решений – это метод, с помощью которого проектируются тесты, гарантирующие, что каждое условие внутри ветвления было протестировано как истинное, так и ложное, и что каждое решение было выполнено хотя бы один раз.*

Англоязычное именование метода «Modified Condition Decision Coverage Testing». Перейдём сразу к рассмотрению данного метода на примерах.

**Пример № 1**

Рассмотрим тестирование функции, которая определяет, предоставляется скидка покупателю или нет.

Условия:

– товар со скидкой – isSale;

– сумма покупки от 1 500 денежных единиц – amount;

– доставка в регионе продавца – isLocal.

В тестах требуется проверить:

– каждое условие в отдельности как истинное и ложное;

– каждое решение целиком как истинное и ложное.

JS app.js > ...

```
// Функция для проверки скидки
function getDiscount(isSale, amount, isLocal) {
    // 1-е условие
    let condition1 = isSale;
    // 2-е условие
    let condition2 = amount > 1500;
    // 3-е условие
    let condition3 = isLocal;
    // Решение
    if (condition1 && condition2 && condition3) {
        return "Скида предоставляется";
    }

    return "Скидка не предоставляется";
}

// Тест 1: Проверка 1-го условия как True
console.log(getDiscount(true, 600, false));
// Тест 2: Проверка 1-го условия как False
console.log(getDiscount(false, 2000, true));
// Тест 3: Проверка 2-го условия как True
console.log(getDiscount(false, 2000, false));
// Тест 4: Проверка 2-го условия как False
console.log(getDiscount(true, 1000, true));
// Тест 5: Проверка 3-го условия как True
console.log(getDiscount(false, 1000, true));
// Тест 6: Проверка 3-го условия как False
console.log(getDiscount(true, 2000, false));
// Тесты 7 - 8: Проверка всего решения как True и False
console.log(getDiscount(true, 2000, true));
console.log(getDiscount(false, 500, false));
```

В данном примере:
Тест № 1. Первое условие передаётся как True, остальные – False.

Тест № 2. Первое условие передаётся как False, остальные – True.

Тест № 3. Второе условие передаётся как True, остальные – False.

Тест № 4. Второе условие передаётся как False, остальные – True.

…

Тест № 7. Проверяем решение целиком. Все условия передаются как True.

Тест № 8. Проверяем решение целиком. Все условия передаются как False.

**Пример № 2**

Рассмотрим тестирование функции, которая определяет, предоставляется скидка покупателю или нет, а также начисляются ему бонусы или нет.

Условия для скидки:

– сумма покупки больше 10 000 денежных единиц – amount;

– покупатель имеет статус «Золотой» – isGold.

Условия для начисления бонусов:

– товар со скидкой – isSale.

В тестах требуется проверить:

– каждое условие в отдельности как истинное и ложное;

– каждое решение целиком как истинное и ложное.

JS app.js > ...

```
function checkDiscount(amount, isGold, isSale) {
    // 1-е условие
    let condition1 = amount > 4000;
    // 2-е условие
    let condition2 = isGold;
    // 3-е условие
    let condition3 = isSale;
    // Решение 1
    if (condition1 && condition2) {
        return "Скидка предоставляется.";
    }
    // Решение 2
    if (condition3) {
        return "Бонусы начисляются.";
    }

    return "Скидки нет. Бонусов нет.";
}

console.log(checkDiscount(7000, false, false)); // Тест 1: Проверка условия 1 на True
console.log(checkDiscount(3000, true, true)); // Тест 2: Проверка условия 1 на False
console.log(checkDiscount(3000, true, false)); // Тест 3: Проверка условия 2 на True
console.log(checkDiscount(7000, false, true)); // Тест 4: Проверка условия 2 на False
console.log(checkDiscount(3000, false, true)); // Тест 5: Проверка условия 3 на True
console.log(checkDiscount(7000, true, false)); // Тест 6: Проверка условия 3 на False
console.log(checkDiscount(7000, true, false)); // Тест 7: Проверка решения 1 на True
console.log(checkDiscount(3000, false, false)); // Тест 8: Проверка решения 1 на False
console.log(checkDiscount(3000, false, true)); // Тест 9: Проверка решения 2 на True
console.log(checkDiscount(7000, true, false)); // Тест 10: Проверка решения 2 на False
```

Как видим, каждое условие и каждое решение проверены как истинное и ложное.

## Примеры интеграционных компонентных тестов

Опытные специалисты, которые занимаются написанием юнит-тестов для интеграционного компонентного тестирования, знают, что это такое и как это делается. Есть специалисты, которые только начинают постигать эту науку – им и посвящена данная глава.

Вспоминаем, что такое интеграционное компонентное тестирование. Это тестирование взаимодействия между компонентами внутри одной программы, которое фокусируется на проверке взаимодействий и интерфейсов между интегрированными компонентами. Данное тестирование проводится после тестирования отдельных компонентов.

Рассматривая проектирование тестов методами белого ящика, мы приводили код примеров тестов, которые тестировали отдельные компоненты программы – функции. В программе есть функции, которые взаимодействуют между собой, и их взаимодействие требуется проверять, поэтому после того, как написаны тесты на отдельные компоненты, необходимо написать тесты, проверяющие взаимодействие компонентов между собой.

Для понимания разберём пример, чтобы иметь представление, о чём идёт речь. В программе имеются две функции, которые взаимодействуют между собой. Функция сложения вызывает функцию логирования в процессе своей работы.

app.js > ...

```
1   // Функция сложения
2   function calculate(a, b, operation) {
3
4       let result;
5
6       if (operation === '+') {
7           result = a + b;
8       }
9
10      // Логирование
11      logger(`Выполнена операция ${operation} с результатом ${result}`);
12
13      return result;
14  }
15
16  // Функция логирования
17  function logger(message) {
18      console.log(`LOG: ${message}`)
19  }
20
21  // Тест 1 (интеграционный компонентный)
22  calculate(3, 5, '+'); // LOG: Выполнена операция + с результатом 8
23
24  // Тест 2 (обычный)
25  console.log('Результат теста 2:', calculate(10, 7, '+')); // 3
```

В данном примере:

Тест № 1. Проверяем, что при сложении отрабатывает функция логирования, которую вызывает функция сложения. Проверяем не возврат результата сложения, а результат логирования. В этом тесте проверяется интеграция.

Тест № 2. Проверяем функцию сложения, т. е. возврат результата сложения. Здесь интеграция не проверяется.

# Методы, основанные на опыте

Методы проектирования тестов из данной категории основываются на умении, интуиции и опыте специалиста по тестированию. Они могут пригодиться при создании тестов, которые не получить, применяя подходы из других методов.

## Метод предположения об ошибках

*Предположение об ошибках – метод проектирования тестов, в котором опыт специалиста по тестированию и его интуиция играют ключевую роль в предвидении возможных дефектов в тестируемом компоненте или программе, а также в создании тестов для выявления этих дефектов.*

Англоязычное именование метода «Error guessing». Тестировщик использует свой опыт и интуицию для предположения, где в программе могут возникнуть дефекты. На основании этого он создаёт тесты, фокусируясь на тех участках программы, которые, по его мнению, наиболее подвержены дефектам. Это может быть основано на предыдущем опыте, знании особенностей программы или интуитивном понимании её работы. Данный метод обычно применяется дополнительно с другими методами проектирования тестов и позволяет выявить проблемы, которые могли бы быть упущены более формальными методами, рассмотренными ранее. Этот метод основан на субъективных оценках, однако может быть эффективным в выявлении дефектов, особенно когда специалист по тестированию обладает широким опытом и пониманием тестируемой программы. Разберём с вами примеры применения данного метода:

Пример № 1. Специалист по тестированию предполагает, что ошибка может возникнуть на сайте при вводе специальных символов в поле ввода имени пользователя. В этом случае он пишет тест-кейс на проверку ввода в поле имени пользователя строку с использованием

спецсимволов (например, @#$%^), чтобы проверить, как программа реагирует на данную ситуацию.

Пример № 2. Предполагаем, что программа может некорректно обработать отправку заполняемой формы с пустыми полями. Пишем тест-кейс на проверку отправки формы с пустыми полями.

Пример № 3. Предполагаем, что программа может неправильно обрабатывать случаи, когда пароли не совпадают. Пишем тест-кейс на проверку ввода разных паролей при создании аккаунта.

Пример № 4. Предполагаем, что могут возникнуть дефекты из-за несовместимости сайта с разными браузерами. Пишем тест-кейс на проверку ключевой функциональности сайта в различных браузерах: Chrome, Firefox, Safari и т. д.

Пример № 5. Предполагаем, что могут возникнуть дефекты при изменении параметров в URL. Пишем тест-кейс на проверку изменения параметров в URL, например, добавить дополнительные значения.

Пример № 6. Предполагаем, что программа может некорректно обрабатывать электронные адреса почты с не валидными символами. Пишем тест-кейс на проверку ввода в поле электронного адреса почты строки с использованием не валидных символов.

Пример № 7. Предполагаем, что в программе возможны проблемы с загрузкой изображений неверного формата. Пишем тест-кейс на проверку загрузки изображений с форматом изображений, отличным от поддерживаемых программой.

Пример № 8. Предполагаем, что в программе могут возникнуть проблемы при обработке больших объёмов данных. Пишем тест-кейс для загрузки в программу большого количества данных.

Приведённые примеры отражают сценарии, которые могли бы быть предположены опытным специалистом по тестированию как потенциальные источники дефектов, на которые стоит обратить внимание и подготовить тесты. Примеры хорошо показывают суть данного метода проектирования тестов.

# Метод исследовательского тестирования

Рассмотрим с вами, что такое исследовательское тестирование.

*Исследовательское тестирование – это неформальный и интуитивный подход к тестированию, при котором специалист импровизирует и экспериментирует в процессе тестирования программы без заранее спроектированных тестов.*

Англоязычное именование метода «Exploratory testing». Приведу простейший пример использования исследовательского тестирования. Специалисту по тестированию передают программу, но нет ни документации, ни знаний о программе. Специалист начинает тестировать программу, исследуя её в процессе тестирования. При этом, исследуя программу, он пытается определить логику её работы, особенности и потенциальные дефекты. Он опирается в первую очередь на собственную креативность, опыт, догадки и интуицию. То есть это, по сути, «творческое» тестирование без предварительных ограничений.

Появляется вопрос «Как проектируются тесты, если уже идёт тестирование программы, а из всего, что мы ранее узнали, следует, что тесты должны проектироваться до начала проведения тестирования, ведь тестирование проводится по тестам?». Тесты в данном случае проектируются во время проведения тестирования методом исследовательского тестирования.

*Проектирование тестов методом исследовательского тестирования – представляет собой неформальный подход к проектированию тестов, в рамках которого специалист создаёт тесты в процессе их выполнения.*

При проведении исследовательского тестирования тест-кейсы разрабатываются, выполняются, анализируются динамически (на ходу) в процессе выполнения тестов. Таким образом проводится тестирование и сразу создаются тесты. Полученная в ходе

тестирования и создания тестов информация используется для формирования новых и улучшенных тестов.

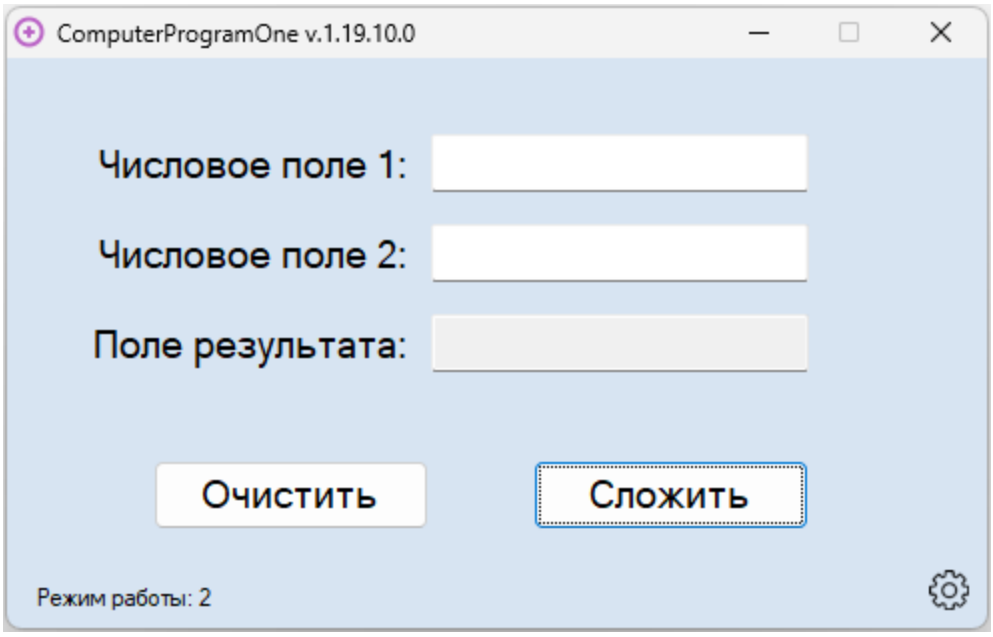
Наиболее эффективно применять метод исследовательского тестирования для проектирования тестов в следующих случаях:

– отсутствуют или плохо формализованы требования и спецификации к программе;

– время на проведение тестирования ограничено;

– как дополнение к более структурированным методам проектирования тестов.

Рассмотрим пример. Есть программа для сложения чисел:




У нас нет спецификации требований, нет сценариев использования программы, нет какой-либо документации. Однако необходимо проверить программу и создать тест-кейсы. Что в этом случае делать? Исследовать программу в процессе проведения исследовательского тестирования! Специалист начинает проводить тестирование и составлять тест-кейсы проводя определённые действия:

1) Ввести в первое поле ввода число 2, ввести во второе поле ввода число 3, нажать на кнопку «Сложить», в ответ в поле отображения результата отобразилось число 5. Специалист на основании полученных знаний создаёт тест-кейс с названием «Сложение однозначных положительных целых чисел».

2) Каждое поле для ввода данных может обрабатывать данные определённой длины. Тестировщик пытается ввести в первое поле ввода максимальное количество символов. Ввелось максимум четырёхзначное число. Он вводит во второе поле ввода максимальное количество символов. Ввелось максимум четырёхзначное число. Если оба поля принимают максимум четырёхзначное число, это скорее всего правило. На основании полученных данных создаётся два тест-кейса на проверку ввода максимального количества символов в каждое поле.

3) В оба поля для ввода данных вводятся буквы, и программа их не отображает – игнорирует ввод. Вводятся спецсимволы, и программа игнорирует ввод. В ходе исследования специалист определяет, что вводить можно только числа и ничего другого. На основании полученных данных создаётся тест-кейс на проверку ввода только чисел в поля ввода.

Постепенно изучая программу и выявляя закономерности и правила, специалист по тестированию проектирует тесты и одновременно с этим тестирует программу.

Мы с вами рассмотрели методы проектирования тестов. Методов проектирования тестов множество, и скорее всего их количество будет расти, так как желающие изобрести что-то своё уникальное будут всегда. Какие-то методы будут удобными в работе, а какие-то останутся в виде теории, не удобной в практическом применении. Множество новых методов будут основываться на существующих методах, какие-то из них окажутся сложны в использовании. Например, вспомните метод причинно-следственного графа, который имеет в своей основе элементы метода проектирования по таблице решений и метода проектирования на основании модели, а также дополнительные техники.

В какой момент и какие методы применять? Здесь нет универсального ответа. Методы применяются в тот момент и там, где они помогают специалистам в работе. Приведу пример. Когда я работал в одной из компаний ко мне обратился сотрудник за помощью. Он тестировал информационную систему, имеющую сложную бизнес-логику, и ему необходимо было разработать тесты, чтобы протестировать требуемый функционал, а опыта было недостаточно. В итоге, проанализировав информационную систему и бизнес-процессы, которые требовалось протестировать, выбрали метод попарного

тестирования и метод проектирования по таблице решений. И благодаря им удалось создать все необходимые тесты. В тот момент эти два метода подходили лучше всего.

Не обладая знаниями и достаточным опытом, специалисту на первых порах придётся периодически перебирать все методы, чтобы найти тот, который поможет в определённый момент времени. Постепенно специалист начнёт безошибочно выбирать и применять подходящий в конкретной ситуации метод проектирования тестов.

При рассмотрении методов проектирования тестов приводились примеры тестов для позитивного тестирования. Однако вы должны помнить: с помощью рассмотренных методов проектируются тесты как для позитивного, так и для негативного тестирования.

# Заключение

Дорогой читатель! Мы с вами вместе совершили захватывающее путешествие, на протяжении которого осваивали основные принципы тестирования программного обеспечения. Вы познакомились с разнообразными видами, уровнями и подходами к тестированию программ. Теперь вы знаете отличия между чек-листами и тест-кейсами, видами тестирования, методами проектирования тестов и многое другое. Однако это лишь база. Тестирование – область обширная, быстро развивающаяся и позволяющая постоянно расширять свой кругозор. Поэтому смело идите дальше по этому увлекательному пути! С каждой новой протестированной программой вы будете становиться всё более опытным специалистом в данной сфере.

Пусть эта книга станет для вас фундаментом, на базе которого вы построите собственное здание знаний и практических навыков в тестировании. Конечная цель у нас одна – повышать качество выпускаемых программ и способствовать выпуску стабильных и бесперебойно работающих информационных систем!

Желаю вам успехов и новых открытий! Я в свою очередь сажусь за написание продолжения данной книги, где мне предстоит многое вам рассказать из того, что ещё не было сказано мной.

# Благодарность

Хочу поблагодарить всех, кто помог мне в подготовке и выпуске этой книги.

Огромная благодарность моей любимой жене Алёне. Её неустанная поддержка, терпение и помощь сыграли ключевую роль для завершения этого проекта. Спасибо тебе, родная!

Я признателен сообществу профессионалов сферы тестирования, которое позволило накопить огромный опыт и знания в сфере тестирования, что привело к написанию данной книги.

Спасибо редактору за то, что нашла время в своём плотном графике, чтобы сделать текст более читабельным и понятным, а также помогла избавить текст от различных опечаток и стилистических погрешностей. Это очень ценно.

Спасибо Дмитру Дьяку за создание яркого и привлекательного дизайна обложки книги. Он прекрасно передаёт тематику книги.

Я благодарен всем сотрудникам ООО «ЛитРес», которые принимали участие в процессе подготовки и выпуска книги.

И конечно же, спасибо моим читателям! Я старался вложить в книгу свой многолетний практический опыт в тестировании программного обеспечения и надеюсь, она будет вам полезна. Спасибо, что выбрали мою книгу!

# Авторские права

# Об авторе

Автор книги – Захаров Виктор Владимирович. Родился 11 мая 1979 года. За годы обучения продемонстрировал выдающиеся способности и получил техническое образование, заложившее крепкий фундамент для его будущей карьеры. Обладая образованием, полученным в школе (11 классов), техникуме и институте, он с самого начала своей карьеры проявил активное стремление к знаниям и новациям.

За более чем 17 лет работы в сфере информационных технологий автор прошёл путь от специалиста технической поддержки до высококвалифицированного руководителя. В процессе своего профессионального пути автор работал инженером-конструктором, ведущим инженером-технологом, специалистом по тестированию, начальником отдела тестирования, руководителем подразделения качества программного обеспечения.

Продуктивный период, посвящённый руководству, составляет 13 лет, в течение которых автор разрабатывал и применял уникальные методики для контроля эффективности процессов подразделений, которыми он управлял.

Глубокие знания теории тестирования и многолетний практический опыт в области тестирования программного обеспечения позволяют

автору успешно выстраивать процессы тестирования и эффективно управлять подразделениями. Его экспертность в технической сфере, в сочетании с богатым опытом работы, формируют уникальный образ профессионала, способного достигать выдающихся результатов и поставленных целей.

Накопленные за долгие годы опыт и знания легли в основу данной книги, цель которой – передать читателям бесценный опыт автора в сфере тестирования. Эта книга не только результат многолетнего опыта работы автора в сфере информационных технологий, но и его вклад в развитие сферы тестирования программного обеспечения.

Для связи с автором книги воспользуйтесь формой обратной связи на сайте www.victorz.ru.

ЗАХАРОВ ВИКТОР

# ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ. ОСНОВЫ

# Примечания

## 1

ISTQB – международная квалификационная комиссия по тестированию программного обеспечения.

Вернуться

## 2

Программист – специалист, занимающийся программированием (написанием кода программы), то есть созданием компьютерных программ.

Вернуться

## 3

Аналитик – это специалист, который занимается изучением информации и делает выводы на основании этой информации. Он анализирует данные, чтобы понять, как должна работать программа в конкретных ситуациях. Не полный перечень занятий аналитика: общение с пользователями; выявление требований пользователей к программе, которую будут разрабатывать; документирование требований и их согласование; постановка задач программистам; демонстрация готовой программы пользователям.

Вернуться

## 4

Программа «Computer Program One» для Windows размещена для скачивания по ссылке https://victorz.ru/books/book-1

Вернуться

# 5

База данных (БД) – это хранимый набор данных, который каким-либо образом структурирован.

Вернуться

# 6

Устройства ввода-вывода – это устройства, которые позволяют нам говорить, вводить информацию в компьютер и получать информацию от компьютера. Клавиатура и мышь – это устройства ввода, потому что мы с их помощью вводим информацию в компьютер. Экран и принтер – устройства вывода, потому что выводят информацию из компьютера.

Вернуться

# 7

Программная библиотека – это набор готовых частей программного кода, которые могут использоваться для создания других программ.

Вернуться

# 8

Приоритизация – это процесс определения того, что является важным и должно быть выполнено первым, а что менее важно и может быть выполнено позже.

Вернуться

# 9

Задача, заявка на изменение (ЗнИ) – это запрос, который предлагает внести изменения в программу. Он может быть создан как внутри организации, так и внешними пользователями. Он фиксируется и оценивается командой, которая разрабатывает программу. Запрос

проходит процесс оценки и утверждения, прежде чем быть включённым в планы на реализацию.

Вернуться

# 10

Заказчик – лицо, заинтересованное в разработке программы для своих нужд или для дальнейшей продажи программы пользователям. Заказчиками могут быть физические или юридические лица.

Вернуться

# 11

Артефакт – это результат какой-либо деятельности. Им может быть любая документация, написанный код и прочее.

Вернуться

# 12

Объект тестирования – компонент или программа, которые должны быть протестированы.

Вернуться

# 13

Ознакомиться с планом тестирования можно по ссылке https://victorz.ru/books/book-1

Вернуться

# 14

Ознакомиться с отчётом о тестировании можно по ссылке https://victorz.ru/books/book-1

Вернуться

# 15

Пользовательская история (user story) – высокоуровневое пользовательское или бизнес-требование. Обычно состоит из одного или нескольких предложений на разговорном или формальном языке, описывающих функциональность, необходимую пользователю, любые нефункциональные требования.

Вернуться

# 16

Ознакомиться с отчётом IBM за октябрь 2008 года можно по ссылке https://victorz.ru/books/book-1

Вернуться

# 17

Итерация – повторение какого-либо действия.

Вернуться

# 18

Сервер – это компьютер или устройство, которые выполняют специализированные задачи, такие как хранение данных, обработка запросов, обеспечение доступа к файлам или ресурсам, обмен информацией и т. д.

Вернуться

# 19

Code Review (обзор кода) – это процесс, когда программисты в команде вместе смотрят и обсуждают код, написанный одним из участников. Вместе они просматривают строки кода, чтобы

удостовериться, что код написан правильно, не содержит ошибок и соответствует стандартам написания кода команды.

Вернуться

# 20

Юнит-тестирование – это тестирование отдельных модулей или компонентов программного кода. При создании программы код делят на логические блоки: функции, классы, методы. Это и есть юниты. Для каждого юнита пишут специальные тесты – юнит-тесты. В тестах проверяют поведение юнита в разных ситуациях. Цель юнит-тестирования – проверить, что каждая отдельная часть кода (юнит) работает как задумано.

Вернуться

# 21

Бизнес-процесс в ПО – это последовательность действий внутри программы, которая приводит к получению нужного результата или выполнению какой-то задачи. Это алгоритм действий внутри программы для получения результата, важного для пользователя.

Вернуться

# 22

Опытная эксплуатация – это один из этапов внедрения программы, следующий сразу после приёмочного тестирования. В опытной эксплуатации программа запускается в работу с реальными данными в условиях, максимально приближенных к промышленным, но программа работает с ограниченным числом пользователей. Фиксируются любые сбои, ошибки, отклонения в работе. По итогам опытной эксплуатации принимается окончательное решение о запуске программы в промышленную эксплуатацию.

Вернуться

# 23

Интегральная характеристика – это обобщённый показатель качества программы, который определяется совокупностью различных факторов и отдельных свойств этой программы: надёжность; удобство использования; скорость работы и т. д. Они позволяют целостно оценить сложные объекты типа программ.

Вернуться

# 24

Регрессия в программировании – это ошибка, дефект или ухудшение работы функциональности программы, которая ранее работала корректно. Регрессии обычно возникают после изменения или дополнения кода программы при выпуске очередных обновлений. Когда внесение нового кода приводит к поломкам в других, не менявшихся модулях.

Вернуться

# 25

Аппаратная платформа – это совокупность аппаратных средств, на которых устанавливается и функционирует программное обеспечение. Аппаратное средство – это физическое устройство, которое составляет аппаратную часть компьютера или другой цифровой системы. Любые комплектующие, узлы и периферийные устройства, которые можно потрогать руками – это и есть аппаратные средства компьютера, в отличие от программного обеспечения.

Вернуться

# 26

Транзакция – это процесс, в рамках которого одна программа отправляет запрос или команду другой программе для выполнения

определённой операции или передачи данных.

Вернуться

# 27

Модель данных – это способ структурированного представления и описания данных, с которыми работает программа или информационная система. Проще говоря, это схема того, как устроена и как соотносится вся информация внутри системы.

Вернуться

# 28

Ознакомиться со стандартом ISO/IEC/IEEE 29119-4 и ISTQB Foundation Level Syllabus можно по ссылке https://victorz.ru/books/book-1

Вернуться

# 29

Сценарий использования (use case) – это формальное описание последовательности взаимодействия между пользователем и программой, которое позволяет пользователю достичь некоторой цели.

Вернуться

# 30

Граф – это способ визуального представления некоторой системы или процесса в виде сети, состоящей из узлов и связей между ними.

Вернуться