GeekBrains

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА по курсу «Ручное тестирование»

ТЕСТИРОВАНИЕ WEB-ПРИЛОЖЕНИЯ

Выполнил:

студентка курса «ручное тестирование»

Кудряшова Алена Николаевна

2022

# Содержание

Введение в тестирование 1

Чек-лист 2

Виды чек-листов

Преимущества и недостатки чек-листов

Создание чек-листа по ТЗ 3

Тестирование по чек-листу 4

Тест-кейс 5

Преимущества и недостатки тест-кейсов

Создание тест-кейса по ТЗ 6

Тестирование по тест-кейсу 7

Что такое техники тест дизайна ..................................8

Виды техник тест дизайна

Виды тестирования 9

Функциональное тестирование

Нефункциональное тестирование

Создание отчета о тестировании 10

Заключение 11

Список иллюстраций ii

Словарь терминов 12

Список использованных источников 13

### Введение в тестирование

Тестирование программного обеспечения — процесс исследования, испытания программного продукта, имеющий своей целью проверку соответствия между реальным поведением программы и её ожидаемым поведением на конечном наборе тестов, выбранных определённым образом (ISO/IEC TR 19759:2005).

В разное время и в различных источниках тестированию давались различные определения, в том числе:

* процесс выполнения программы с целью нахождения ошибок;
* интеллектуальная дисциплина, имеющая целью получение надежного программного обеспечения без излишних усилий на его проверку;
* техническое исследование программы для получения информации о её качестве с точки зрения определённого круга заинтересованных;
* проверка соответствия между реальным поведением программы и её ожидаемым поведением на конечном наборе тестов, выполненных определённым образом;
* процесс наблюдения за выполнением программы в специальных условиях и вынесения на этой основе оценки каких-либо аспектов её работы;
* процесс, имеющий целью выявление ситуаций, в которых поведение программы является неправильным, нежелательным или не соответствующим спецификации;
* процесс, содержащий в себе все активности жизненного цикла, как динамические, так и статические, касающиеся планирования, подготовки и оценки программного продукта и связанных с этим результатов работ с целью определить, что они соответствуют описанным требованиям, показать, что они подходят для заявленных целей и для определения дефектов.

ЧЕК-ЛИСТ

Чек-лист - список, содержащий ряд необходимых проверок для какой-либо работы.

Важность чек листов трудно переоценить. Каким бы опытным ни был сотрудник, в спешке он может легко забыть важную деталь.

В тестировании чек-лист — это список проверок для тестирования продукта. Чек-листы устроены предельно просто. Любой из них содержит перечень блоков, секций, страниц, других элементов, которые следует протестировать, например:

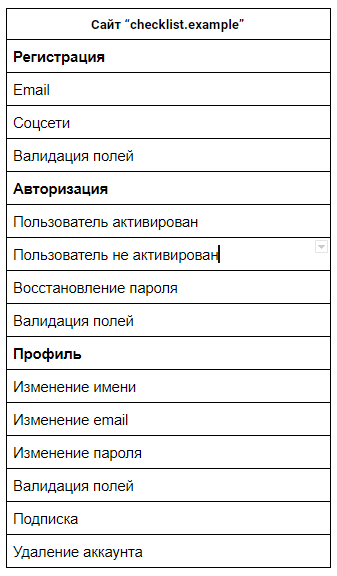


Рисунок 1 – Пример чек листа

Выполненные пункты отмечаются статусами, например: “Passed”, “Failed”, “Blocked”, “Skipped”, “Not run”. Эти статусы также могут иметь свой цвет:

C:\Users\user\Documents\GraduateWorkKudryashova\2.png

Рисунок 2 - Статусы

Преимущества использования чек-листов:

* улучшить представление о системе в целом, видеть статус ее готовности;
* понимать объем проделанной и предстоящей работы по тестированию;
* не повторяться в проверках и не упустить ничего важного в процессе тестирования.

ВИДЫ ЧЕК-ЛИСТОВ

Можно выделить два вида чек-листов: специальные и универсальные.

Специальные чек-листы создаются и используются для конкретных проектов, поэтому пункты такого чек-листа соответствуют специфики проекта. Тестировщик по специальному чек-листу проверяет возможность выполнить уникальное действие, предусмотренное требованиями. Вот примеры пунктов специального чек-листа:

* при наведении курсора на пункт меню “Товары”, должен меняться цвет на синий. Указатель должен менять форму на pointer;
* если пользователь открыл страницу “Ваша корзина” и в корзине присутствует хотя бы один товар, то должно показываться уведомление.

Такие чек-листы не подходят к использованию на других проектах.

Универсальные чек-листыподходят для тестирования проектов одного типа. Проверка по универсальному чек-листу не привязывается к графическим элементам или конкретной реализации, а проверяется сама возможность пользователя выполнить действие. Для универсального чек-листа составляется абстрактный список проверок. Пункты универсального чек-листа могут быть такими:

* пользователь может перейти в раздел “Товары”;
* оплата должна совершаться;
* товар должен добавляться в корзину;
* ссылки при наведении подчеркиваются;
* валидатор верстки показывает отсутствие ошибок и т.п.

Универсальные чек-листы можно использовать повторно на проектах одного типа. У многих агентств есть такие универсальные чек-листы, по ним определяется общий уровень качества продукта.

Чтобы составить работающий чек-лист, обратите внимание на эти рекомендации:

1. Один пункт = одна проверка. Минимальная полная операция проводимая тестировщиком при проверке — это один пункт чек-листа:
2. При составлении чек-листа нужно опираться на требования, чтобы не тестировать то, что не существенно.
3. Давайте пунктам чек-листа названия по форме, общей для всех членов команды, чтобы работа с чек-листом не вызывала неоднозначных толкований. Можно договориться использовать во всех пунктах только глаголы в инфинитиве или существительные: «проверить»/ «добавить»/ «отправить» либо «проверка»/«отправка»/«добавление».
4. Детализируйте чек-лист в зависимости от задачи.
5. Объединяйте чек-листы в матрицы, где можно отразить не только сами проверки, но и условия проверки (платформа, версия продукта, сотрудник и т.п.) и статус проверки. Матрицы — это компромисс между чек-листами и тест-кейсами. Их легче поддерживать, чем тест-кейсы, так как в такой таблице отсутствуют шаги (steps). В них одна строка = одна проверка:

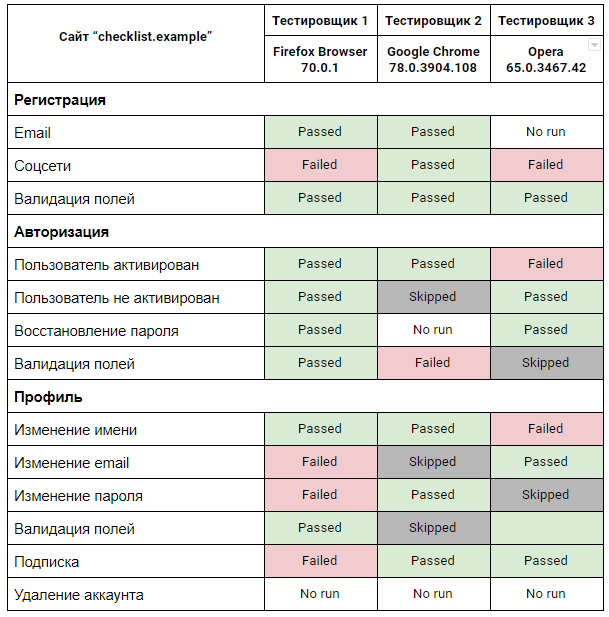


Рисунок 3 - Матрица

ПРЕИМУЩЕСТВА И НЕДОСТАТКИ ЧЕК-ЛИСТОВ

Преимущества:

* чек-лист легко читается;
* по чек-листу быстро тестировать: в тест-кейсе нужно отмечать статус каждого шага, в то время как в чек-листе достаточно одной строчки;
* чек-лист — источник результатов для отчёта: можно быстро посчитать сколько проверок выполнено, и в каком они статусе, узнать количество открытых репортов;
* в любой момент можно узнать статус — всегда есть то, что нужно проверить в первую очередь, можно упорядочить пункты чек-листа или изменить порядок, когда это требуется.

Недостатки:

* неопределенность тестового набора: каждый тестировщик выполняет пункт чек-листа по-своему;
* неопределенность тестовых данных;
* недостаточность детализации;
* сложнее обучить начинающих сотрудников: пункты чек-листа чаще абстрагируются от конкретных элементов интерфейса и описывают то, что нужно сделать;
* чек-лист менее эффективен для начинающих тестировщиков, лучше использовать тест-кейсы.

Чек-листы лучше применять на ранних этапах, когда софт быстро меняется, потому что тест-кейсы дорого поддерживать.

СОЗДАНИЕ ЧЕК-ЛИСТА ПО ТЗ

Создадим чек-лист на примере сайта <https://limelab.tech/foodbuzz>.

Специальные чек-листы создаются и используются для конкретных проектов, поэтому пункты такого чек-листа должны соответствовать ТЗ и независимо от реализации. Для примера берем раздел сайта «Футер» и читаем его описание в документации к сайту.

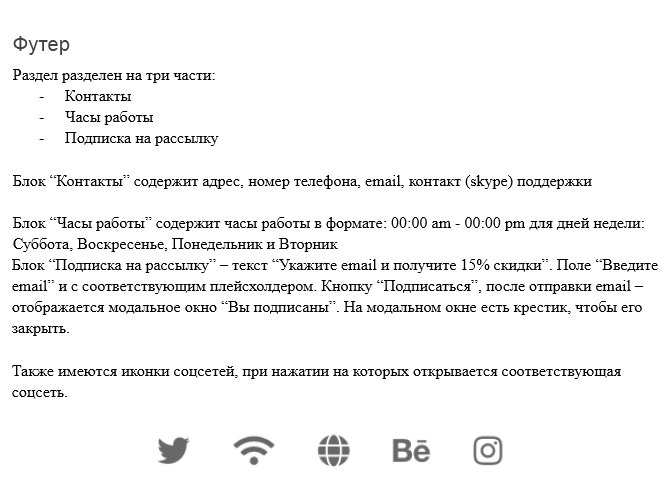


Рисунок 4 - Описание из ТЗ

Составляем чек-лист по принципу «Один пункт = одна проверка».

Условно разделим проверку на 4 части: контакты, часы работы, подписка на рассылку и иконки соцсетей. Каждую часть разделим на проверки.

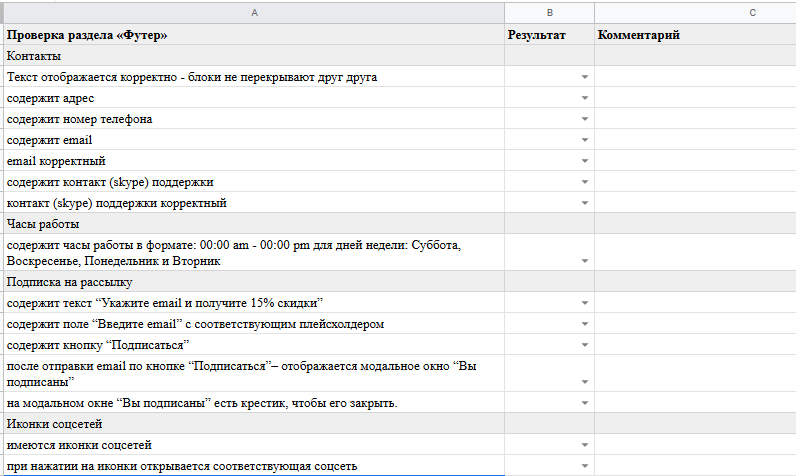


Рисунок 5 - Чек-лист раздела "Футер"

ТЕСТИРОВАНИЕ ПО ЧЕК-ЛИСТУ

Устанавливаем результат проверки для каждого пункта и пишем комментарии для статусов “Failed”, “Blocked”, “Skipped”.

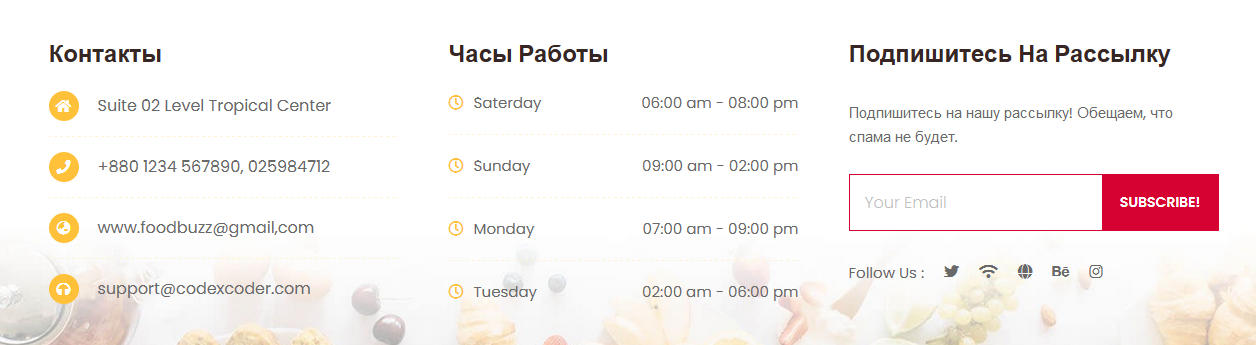


Рисунок 6 - Раздел "Футер"

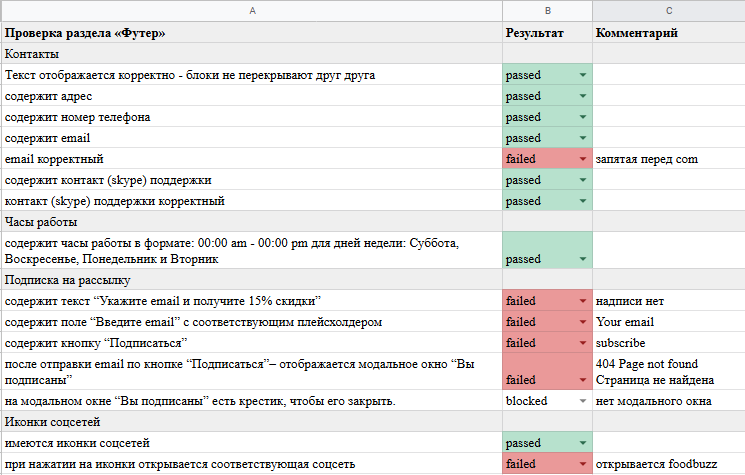


Рисунок 7 - Чек-лист для раздела "Футер"

ТЕСТ-КЕЙС

Тест-кейс — это проверка. "Выполни тест-кейс по вводу отрицательных значений" = проведи проверку такую-то и проверь, что результат будет такой-то.

Тест-кейс — это описание проверки работы системы, которое может выполнить любой человек из команды, будь то тестировщик, разработчик, аналитик или даже бизнес-заказчик.

Набор тест-кейсов называется тестовым набором (test suite).

Иногда этот набор некорректно называют тест-планом. Тест-план — это именно план: когда, что, зачем, какими ресурсами.

Стандартные атрибуты тест-кейса:

* Номер — уникальный идентификатор тест-кейса. Его удобно использовать для одинакового понимания, о какой проверке идет речь (например, дать ссылку в баге).
* Название — краткое описание сути проверки. Должно помещаться в твиттер и быть понятным! Кратко, но емко.
* Предварительные шаги — описание действий, которые необходимо выполнить, но прямого отношения к проверке они не имеют (например, зарегистрироваться в системе для проверки создания элемента). Если предварительных шагов нет, то секция не заполняется.
* Шаги — описание действий, необходимых для проверки (например, создание элемента).
* Ожидаемый результат (ОР) — сама проверка: что мы ожидаем получить после выполнения шагов ("Элемент создан").

ПРЕИМУЩЕСТВА И НЕДОСТАТКИ ТЕСТ-КЕЙСОВ

Преимущества: тест-кейсы можно доверить выполнять новичку или призванному на помощь коллеге из другого отдела, который ничего о проекте не знает. Дополнительных вопросов с его стороны будет по минимуму — все должно быть понятно.

Недостатки:

1. Очень много копипасты. Например при заполнении поля "ФИО". Тест-кейсы «ввести в поле только символы, только числа, строку нулевой длины и т. д.» будут очень похожи друг на друга, первые шаги одинаковые и будут копипаститься.
2. Сложно поддерживать. Представьте, что вкладку «Жильцы» переименовали в «Заказчики». Чтобы актуализировать тест-кейсы, надо внести изменения в сотни сценариев, что утомительно даже в режиме "Ctrl + C, Ctrl + V".
3. Неактуальное состояние. Тест-кейсы копипастятся друг от друга, и часто в них остаются неактуальные части из исходного кейса, которые забыли изменить.

СОЗДАНИЕ ТЕСТ-КЕЙСА ПО ТЗ

Возьмем сайт и раздел как для чек-листа.

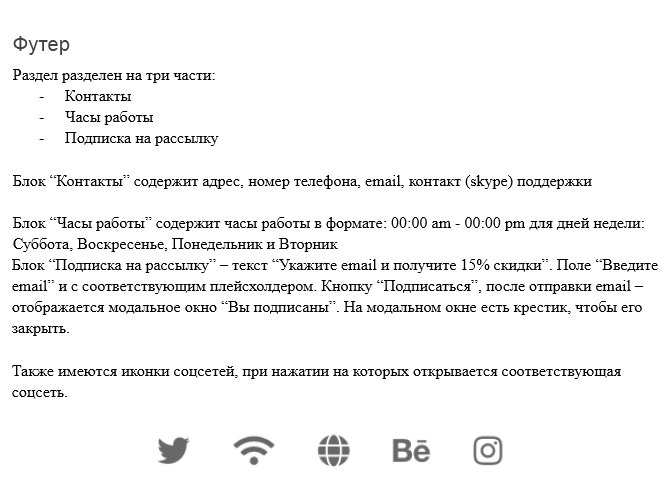


Рисунок 8 - Описание из ТЗ

Создадим тест-кейс используя стандартные атрибуты.

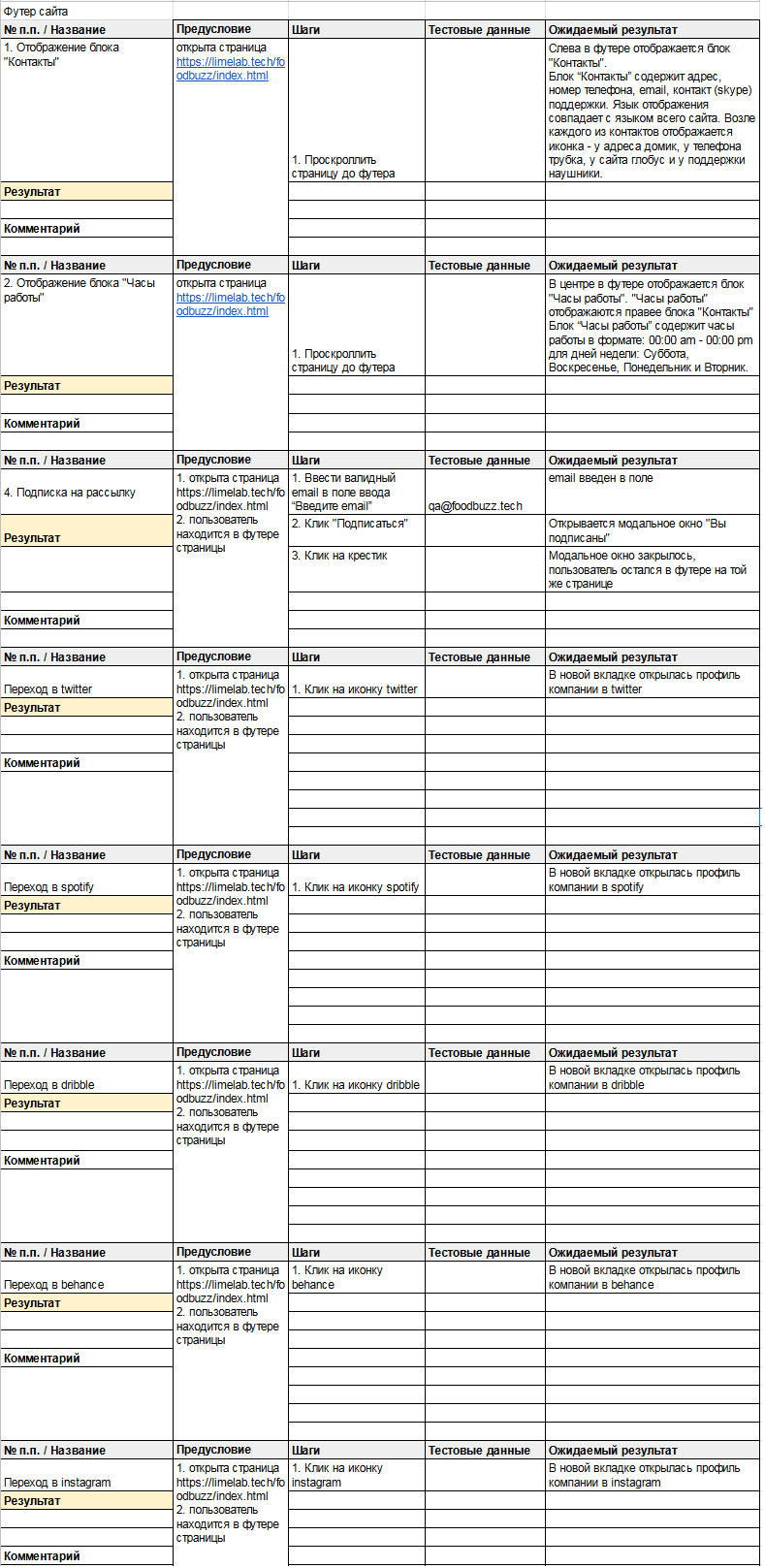


Рисунок 9 - Тест-кейс для раздела "Футер"

ТЕСТИРОВАНИЕ ПО ТЕСТ-КЕЙСУ

Выполняем проверку, используя предусловие и тестовые данные, выполняя действия из раздела «Шаги». Устанавливаем результат проверки для каждого пункта тест-кейса и пишем комментарии для статусов “Failed”, “Blocked”, “Skipped”.

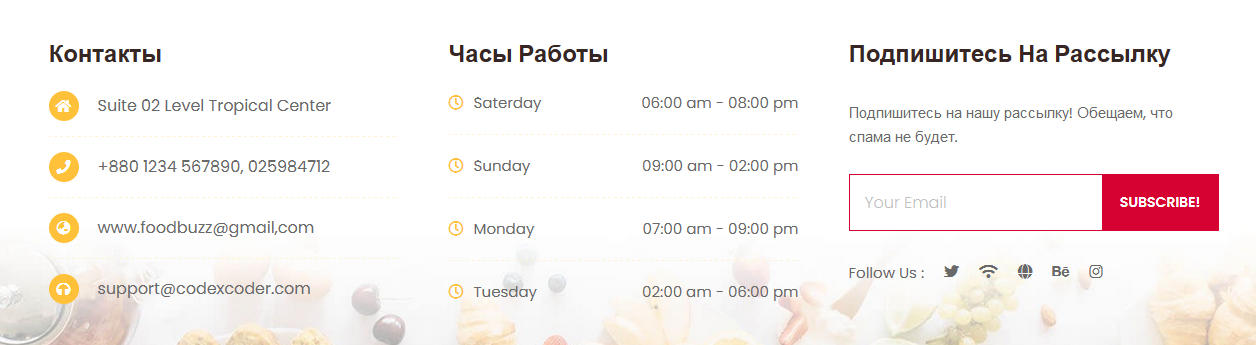


Рисунок 10 - раздел "Футер"

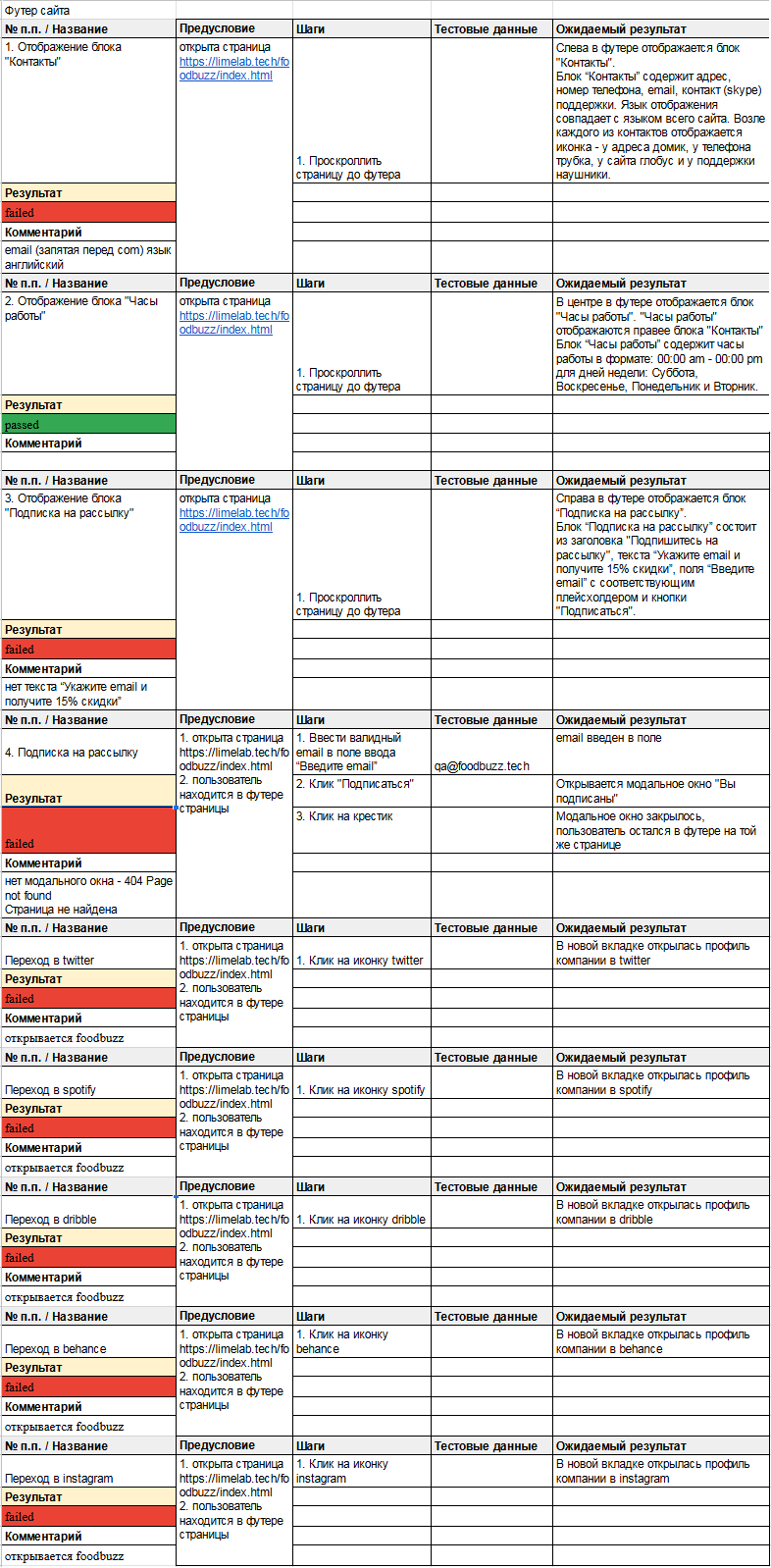


Рисунок 11 - Тест-кейс для раздела "Футер"

ЧТО ТАКОЕ ТЕХНИКИ ТЕСТ ДИЗАЙНА

Тест-дизайн — этап тестирования ПО, на котором проектируются и создаются тестовые случаи (тест-кейсы). Они соответствуют определённым ранее критериям качества и целям тестирования. Важно, что критерии качества и цели тестирования должны быть определены до начала написания тест-кейсов. От этих критериев и целей зависит, какими будут тест-кейсы, для каких модулей они будут описаны в первую очередь, проверка каких функций будет приоритетной.

Когда мы пишем тест-кейсы, одна из основных задач — создать оптимальное тестовое покрытие функциональности, то есть не допустить «слепых зон» в системе, которые не покрываются проверками.

Задачи тест-дизайна на проекте:

● максимально покрыть функциональность тестами;

● обнаружить серьёзные баги;

● сократить количество тестов, исключив непродуктивные;

● не пропустить важные тесты.

Нельзя провести исчерпывающее тестирование. Поэтому нужно применять разные техники, чтобы выполнить его эффективно и вовремя, избежав при этом проверки лишних кейсов. При этом вся функциональность должна быть покрыта тестами. Кроме того, нужно попытаться составить тесты так, чтобы с их помощью можно было обнаружить критичные дефекты. Нельзя выявить все баги и убедиться в их отсутствии, но усилия и внимание тестировщиков должны быть направлены на поиск самых серьёзных дефектов.

Если времени и специалистов мало, важно исключать из работы непродуктивные тесты, которые не обнаруживают ошибок. В этом тоже помогут техники тест-дизайна. Есть разные наборы техник тест-дизайна. И сочетания, и названия в разных источниках могут отличаться.

ВИДЫ ТЕХНИК ТЕСТ ДИЗАЙНА

Техники тест-дизайна:

1. Классы эквивалентности (эквивалентное разделение).

2. Граничные значения (анализ граничных значений, метод граничных

значений).

3. Попарное тестирование (тестовая комбинаторика, pairwise).

4. Тестирование состояний и переходов.

5. Таблицы принятия решений.

6. Исследовательское тестирование.

Класс эквивалентности (эквивалентное разделение).

Класс эквивалентности — набор данных, которые обрабатываются одинаковым образом и приводят к одному результату. Например, в требованиях есть условие для посещения онлайн-кинотеатра: «Возраст пользователя — от 16 лет и старше». Результат для пользователей, которые указывают возраст меньше 16 лет (не важно, 5 или 15), всегда должен быть одинаковым — сообщение «Извините, в связи с политикой сайта вы не можете пользоваться сервисом». Так же и со значениями от 16 и выше — не важно, какой возраст укажет пользователь (16, 23, 75, 99 лет), результат будет одинаковым:

«Добро пожаловать в наш кинотеатр. Желаем приятного просмотра!»

Если известно, что есть группа данных, использование которых приводит систему в одно и то же состояние, нет необходимости проверять каждое значение из этой группы отдельно. Исключения возможны, но мы не можем проверять все данные, так что приходится прибегать к подобным допущениям. Тестирование на основе классов эквивалентности (equivalence partitioning) — техника тест-дизайна на основе метода чёрного ящика: специалист не знает, как устроена система, и проходит все шаги тестов, используя только те инструменты,

которые доступны пользователю. Цель техники — обеспечить максимальную проверку всех требований тестами. Разделяя данные на классы эквивалентности и выбирая лишь несколько значений из каждого, можно существенно повысить эффективность и скорость тестирования,

разрабатывать и выполнять меньше тест-кейсов. Есть два признака, что данные в тесте относятся к одному классу эквивалентности:

1. Если один тест выявит ошибку, остальные, скорее всего, тоже это

сделают**.** Если в тестах используются значения из одного класса эквивалентности, и один из тестов выявляет ошибку, остальные тесты, построенные на данных из этого класса, тоже должны обнаружить эту ошибку. Например, если онлайн-кинотеатр позволяет пользователю в возрасте 14 лет зарегистрироваться на сайте, то, вероятнее всего, регистрация будет возможна и для пользователей, указавших возраст 5, 10, 12 лет. А по требованиям это ошибка.

2. Если один из тестов не выявит ошибку, остальные, скорее всего, тоже этого не сделают.Если пользователю, указавшему возраст 15 лет, было

отказано в регистрации на сайте, то нет смысла перебирать все значения от 0 до 15 лет. Вероятнее всего, они тоже обработаются корректно. Так как в тестировании нельзя быть уверенным в наличии или отсутствии ошибок, в описаниях часто встречаются комментарии «скорее всего», «с большой долей вероятности».

Рассмотрим несколько примеров определения классов эквивалентности. В требованиях о найме у HR-отдела есть условие, которое автоматически распределяет резюме кандидатов в разные категории:

|  |  |
| --- | --- |
| Возраст кандидата, лет | Статус резюме |
| 0-15 | Не нанимать, NO |
| 16-17 | Сокращённый рабочий день, максимум 4 часа, PART |
| 18-64 | Полный рабочий день, максимум 8 часов, FULL |
| 65-99 | Не нанимать, NO |

Мы рассмотрели данные, которые можно расположить на числовой прямой — классы эквивалентности этих данных будут линейными. Их можно разбить на диапазоны с точными границами начала и конца (от 0 до 15, от 16 до 18 и так далее).

Нелинейныеклассы эквивалентности — это набор неупорядоченных данных. У них нет границ, они являются частью множества данных. Пример — расширения файлов, операционные системы, группы пользователей с различными правами (пользователь, модератор, администратор) и так далее. В этом случае можно выделить только два класса эквивалентности:

* валидный — соответствует требованиям,
* невалидный — не соответствует требованиям или обрабатывается системой отличным от валидного класса образом.

Например, приложение обрабатывает только файлы в форматах MP3, APE, WAV. Остальные форматы файлов системой не поддерживаются. В этом случае невозможно выделить диапазоны и определить их границы. Можно выделить только валидный класс эквивалентности, то есть допустимые форматы файлов, и невалидный — все остальные форматы, которые система не поддерживает. Так как в валидном классе всего три значения, их можно проверить все, а из невалидного класса выбрать несколько вариантов.

Кроме чисел, на классы эквивалентности можно разбить:

* символы — они могут быть валидными (@ в адресе электронной почты) и невалидными (?, %,\*);
* длину строки — например, валидный класс от 1 до 30 знаков, невалидный — всё остальное (меньше 1 и больше 30);
* объём памяти, который необходим приложению для стабильной работы;
* разрешение экрана — всё, что меньше или больше заявленных требований к разрешению экрана, будет относиться к невалидным классам;
* версии операционных систем, библиотек — также определяются согласно требованиям. Например, приложение должно работать на ОС Windows 7, но поддержка Windows Nt уже не требуется.
* объём передаваемых данных — по требованиям. Например, если мощности сервера не позволяют обработать объём данных больше определённого значения.

Классы эквивалентности — одна из основных техник тест-дизайна. Именно с ней тестировщики и тест-дизайнеры работают чаще всего. Она сокращает число тестов (можно выбрать только несколько значений из класса эквивалентности), но к использованию нужно подходить внимательно: если неверно выделить класс эквивалентности, можно получить некорректные результаты тестирования и пропустить ошибку.

Граничные значения

Когда тестировщик работает с линейными классами эквивалентности (диапазонами значений), может потребоваться определить границы диапазона, чтобы точно отнести значение к конкретному классу эквивалентности. У каждого диапазона будет начальная и конечная граница — это места повышенного риска ошибок, так как разработчик может указать некорректный знак неравенства или задать ошибочную границу диапазона.

Граничное значение (border condition, boundary condition) — значение на границе классов эквивалентности.

Техника анализа граничных значений(boundary value testing) — проверка поведения продукта на граничных значениях входных данных. Граничные значения обязательно использовать при написании тестов, так как именно на границе классов эквивалентности чаще всего и обнаруживаются ошибки. Например, если в требованиях указано, что пользователь сайта должен быть старше 16 лет, тестировщику следует уточнить у аналитика, входит ли значение «16 лет» в валидный класс эквивалентности. А затем — проверить, действительно ли это реализовано в приложении. Может оказаться, что разработчик понял требования иначе и указал в коде, что сайтом могут пользоваться лица с 17 лет (**>16** вместо **>=16**).

Алгоритм определения граничных значений:

1. Выделить классы эквивалентности.
2. Определить граничные значения этих классов.
3. Определить, к какому классу будет относиться каждая граница.
4. Для каждой границы провести тесты: проверить значения до границы, на ней и сразу после неё.

Рассмотрим применение техники анализа граничных значений на знакомых примерах.

Требования

|  |  |
| --- | --- |
| Возраст кандидата, лет | Статус резюме |
| 0-15 | Не нанимать, NO |
| 16-17 | Сокращённый рабочий день, максимум 4 часа, PART |
| 18-64 | Полный рабочий день, максимум 8 часов, FULL |
| 65-99 | Не нанимать, NO |

1. Определяем граничные классы эквивалентности:

* 1-й класс эквивалентности — 0–15;
* 2-й класс эквивалентности — 16–17;
* 3-й класс эквивалентности — 18–64;
* 4-й класс эквивалентности — 65–99.

2. Выделяем граничные значения:



Рисунок 12 - Выделяем граничные значения

3. Определяем, к какому классу относится каждая граница:

* 1 класс эквивалентности — 0;
* 2 класс эквивалентности — 16;
* 3 класс эквивалентности — 18;
* 4 класс эквивалентности — 65.

4. Для каждой границы выделяем три значения:

* {-1, **0**, 1},
* {15, **16**, 17},
* {17, **18**, 19},
* {64, **65**, 66},
* {98, **99**, 100}.

5. Исключаем дубликаты (в нашем случае 17) и добавляем негативные проверки, например: {-36, 1001, FRED, %$#@}.

На основании этих данных можно проводить тестирование.

Попарное тестирование

Техники эквивалентного разбиения и анализа граничных значений — самые используемые в тестировании. На их основе формируется большинство проверок и тестов. Следующие техники тест-дизайна, которые мы рассмотрим в этом и следующем уроке, не так популярны. Их применение зависит от особенностей тестируемого проекта. Первая — попарное тестирование (pairwise). Рассмотрим несколько определений.

Попарное тестирование (pairwise testing) — техника формирования наборов тестовых данных, при которой каждое тестируемое значение каждого из проверяемых параметров хотя бы раз сочетается с каждым из тестируемых значений всех остальных проверяемых параметров.

Попарное тестирование — разработка тестов методом чёрного ящика, в которой тестовые сценарии разрабатываются таким образом, чтобы выполнить все возможные отдельные комбинации каждой пары входных параметров.

Попарное тестирование — техника тестирования, в которой вместо проверки всех возможных комбинаций значений всех параметров проверяются только комбинации значений каждой пары параметров.

Техника применяется на проектах, где много параметров и их значений.

Тестирование состояний и переходов

Тестирование на основе состояний и переходов(State-Transition Testing) используют для фиксирования требований и описания дизайна приложения.

В проекте может быть большой набор требований с описанием состояния системы и условий, при которых она в них переходит. Без визуального представления этих состояний трудно увидеть всю цепочку событий. А это может привести к дефектам архитектуры и дизайна приложения уже на уровне требований. Например, теперь в мессенджерах можно удалять сообщения как у отправителя, так и у получателя. То есть для состояния сообщения «Отправлено» или «Прочитано» должен быть предусмотрен переход в состояние «Удалено». Если он будет упущен при составлении требований, приложение получится неудобным для пользователей, вряд ли его станут часто запускать.

Чтобы избежать таких ошибок, можно использовать технику тест-дизайна «Тестирование состояний и переходов». Она позволяет составлять тестовые сценарии, основываясь на визуальном представлении состояний и переходов системы. Прежде чем рассматривать эту технику, познакомимся с основными понятиями, которые используются при составлении диаграмм переходов и состояний.



Точка входа — старт работы системы или приложения.

Переход (transition) — переход системы из одного состояния в другое. Происходит в результате действий пользователя или при определённых условиях.

Событие (event) — действие пользователя, которые он выполнил для перевода системы в другое состояние. Или действия самой системы, меняющие её состояние.

Действие (action) — реакция приложения на действия пользователя или самой системы (на событие).

Условия перехода (transition conditions) — условия, которые необходимы для перехода системы в другое состояние. Например, изменение даты для начисления процентов на вклад.

Состояние (state) — состояние системы до или после перехода в результате действий пользователя или при определённых условиях.

Точка выхода — успешное окончание полного цикла работы приложения, то есть выполнение всех переходов и состояний.

Роли пользователей (actors) — пользователи, которые могут по-разному влиять на систему в зависимости от уровня прав доступа (зарегистрированный пользователь, менеджер, администратор).

Таблицы принятия решений

Часто аналитики создают требования в виде сплошного текста с множеством условий вида «если …, то ...». Например, «если пользователь старше 16 лет, то доступ на сайт разрешён», «если пользователь авторизован в системе, то его личные данные в форме заказа должны быть заполнены автоматически». Тестирование таких требований и создание на их основе тест-кейсов трудоёмкое, нужно повышенное внимание. Для таких случаев можно использовать технику тест-дизайна «Таблицы принятия решений».

Таблицы принятия решений (таблицы решений) — способ компактно представить модели со сложной логикой. А ещё это техника тестирования чёрного ящика, которая применяется для систем со сложной логикой. Таблицы принятия решений используют, чтобы упорядочить и задокументировать сложную логику приложения, а также протестировать все комбинации условий и состояний.

Сущности, из которых состоят таблицы:

Условия (conditions) — короткое описание входных условий (данных), сформулированное в виде вопроса. Ответ — либо «да/нет», либо ограниченный набор значений. Например: «Пользователь авторизован в системе?», «Вид документа, предоставленный клиентом, — паспорт, водительские права, загранпаспорт?»

Действия (actions) — чёткое описание ожидаемого результата, действия системы.

Формулировка действия — утвердительное предложение. Одно предложение обязательно описывает только одно действие. Например: «Данные заполнены автоматически», «Сообщение об ошибке отображается на экране».

Значения (values) — значения, допустимые для входных данных, указанных в условии. Например: «да/нет», «паспорт, водительские права, загранпаспорт».

Правила (rules) — комбинации входных данных, которые отражены в таблице.

Исследовательское тестирование

Исследовательское тестирование не всегда относят к техникам тест-дизайна. Но по его результатам могут составлять тест-кейсы, поэтому рассмотрим его как технику. Исследовательское тестирование — это подход, когда тестировщик не использует тест-кейсы, а тестирует приложение по определённому сценарию, который часто составляется прямо во время проверки. Тестировщик проводит исследовательское тестирование приложения, в результате которого выявляются дефекты. Тот сценарий (тест), который выявил дефект, нужно задокументировать (создать тест-кейс), чтобы в дальнейшем проверять, что дефект исправлен и не появился вновь. Кроме того, стоит создать тест-кейсы (если их нет) и для проверки похожих сценариев, чтобы обнаружить другие подобные дефекты. В некоторых случаях проверки, проведённые при исследовательском тестировании, следует документировать (создавать тест-кейс), даже если они не обнаружили дефект. Это нужно, чтобы повторять проверки в будущем, в том числе при регрессионном тестировании. Таким образом, исследовательское тестирование как техника тест-дизайна позволяет дополнять наборы тест-кейсов новыми тестами, а также создавать актуальные тест-кейсы, которые выявляют дефекты. Исследовательское тестирование также используют как вспомогательный подход к тестированию по тест-кейсам. Оно помогает исключить эффект пестицида (когда тест-кейсы перестают выявлять дефекты) при частом использовании одних и тех же тест-кейсов.

Ещё случаи, когда исследовательское тестирование может быть эффективным:

1. Нужно быстро понять, насколько качественно выполнена новая функциональность: проверить, что в ней нет критических дефектов.
2. Нужно быстро изучить тестируемый продукт (например, новому тестировщику на проекте) и получить общую информацию о его основной функциональности.
3. Нужно проконтролировать работу других тестировщиков: проверить без использования тест-кейсов, что приложение работает (с позиции пользователя).
4. Недостаточно времени для составления тест-кейсов.
5. Отсутствуют требования, на основании которых можно составить тест-кейсы.
6. Тестируется небольшой проект, для которого не требуется структурированного подхода к тестированию.
7. В проекте произошли внезапные изменения, которые требуют быстрой проверки.

Важно понимать, что исследовательское тестирование — не хаотичное без документации и подготовки. Оно требует планирования и профессиональных навыков тестировщика. Есть решения, позволяющие сделать исследовательское тестирование более структурированным:

* использование чит-листов — списков базовых проверок, которые можно применять для тестирования однотипных приложений;
* сессионное тестирование — установка временного интервала для проведения исследовательского тестирования, например, сессии в 90 минут;
* парное тестирование — проверка одного блока или модуля двумя тестировщиками, один из которых может проводить тестирование, а второй — описывать найденные дефекты;
* тест-туры Джеймса Уиттакера — отдельная тема в исследовательском тестировании.

ВИДЫ ТЕСТИРОВАНИЯ

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

1. Функциональные
2. Нефункциональные

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

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

ФУНКЦИОНАЛЬНОЕ ТЕСТИРОВАНИЕ

Функциональное тестирование - это тестирование ПО, в процессе которого проверяется реализация функциональных требований, то есть проверка работы функциональностей, направленных на решение задач пользователя.

* 1. Три ящика тестирования:

Работая с программой, тестировщик обычно держит в уме её архитектурные компоненты и особенности их взаимодействия. Бывают ситуации, когда тестировщику ничего не известно об устройстве программы «под капотом». А иногда тестировщик видит код программы и пишет тесты, опираясь на него. В каждом случае у тестирования есть особенности, обусловленные уровнем знаний о внутреннем устройстве программы. В зависимости от этого уровня выделяются три вида тестирования: чёрного, белого и серого ящика.

* 1. Тестирование чёрного ящика (black box testing) — тестирование, основанное на анализе функциональной или нефункциональной спецификации системы без знания внутренней структуры. У тестировщика нет доступа к коду, он видит приложение как пользователь. Тестирование проводится через интерфейс приложения. Это ручное тестирование без знания, что находится «за кулисами» интерфейса. Тест-дизайн, основанный на технике чёрного ящика, это написание или отбор тест-кейсов на основе анализа документации без знания внутреннего устройства программы.
  2. Тестирование белого ящика (white box testing) основано на анализе внутренней структуры системы, на знании и понимании исходного кода. У тестировщика есть полный доступ к исходному коду.Для тестирования методом белого ящика нужно знать язык программирования, на котором написано приложение. Обычно этот вид тестирования применяют разработчики при написании юнит-тестов. Входные значения отбираются на основе кода, который будет их обрабатывать. Техника белого ящика применяется на разных уровнях тестирования, но главным образом для модульного тестирования компонентов.
  3. Тестирование серого ящика (gray box testing) — тестирование в условиях, когда часть внутренней структуры программы известна. Тестировщик работает не с кодом приложения, а с часть его внутренней структуры: проверяет запись в базе данных, лог-файлы, коды ответа от сервера. Для тестирования веб-приложений методом серого ящика тестировщик использует инструменты разработчика, например, Chrome DevTools. Техника серого ящика применяется на интеграционном уровне для проверки взаимодействия компонентов программы, например, API-интерфейса и базы данных.

1. Виды тестирования после изменений в коде

На протяжении всей разработки в код приложения вносят изменения: при добавлении новых функций и при исправлении дефектов. В результате тестировщик снова проводит тестирование той части приложения, которая уже была проверена, но подверглась изменениям. В зависимости от изменений выделяют регрессионное и повторное тестирование.

Регрессионное тестирование (regression testing) — тестирование уже проверенной функциональности после изменений в коде. Цель — убедиться, что эти изменения не добавили или не активизировали ошибки в изменённых областях.

Повторное или подтверждающее тестирование (re-testing/confirmation testing) — исполняются тестовые сценарии, выявившие ошибки во время последнего запуска. Цель — подтвердить, что ошибки исправили, и приложение работает в соответствии с требованиями. Повторное тестирование — обязательный этап. Тестировщик должен проверить, исправлен ли дефект, повторив сценарий, который выявил ошибку.

1. Статическое и динамическое тестирование

В зависимости от того, запускается код программы или нет, выделяют два виды тестирования — статическое и динамическое.

Статическое тестирование (static testing) — тестирование системы на уровне спецификации или реализации без исполнения кода. Так проводится тестирование:

* документации — требований, схем баз данных, тест-кейсов;
* кода приложения — проверка кода перед запуском специалистом, не участвовавшем в его написании или изменении, то есть аудит кода, или code review;
* параметров настройки среды приложения;
* подготовленных тестовых данных;
* прототипов пользовательского интерфейса.

Статическое тестирование начинается на ранних этапах жизненного цикла ПО и продолжается на протяжении всей разработки.

Динамическое тестирование (dynamic testing) — тестирование во время выполнения программного обеспечения, компонента или системы. Проверка реальное поведение ПО во время его работы.

Чтобы выполнить динамическое тестирование нужно, чтобы код программы запустился. Тестируется как система в целом, так и отдельные компоненты. Все виды функционального тестирования — динамические.

1. Позитивное и негативное тестирование

Позитивное тестирование (positive testing) — тестирование с использованием только корректных данных. Проверяет, правильно ли приложение выполняет вызываемые функции. Проводится, чтобы подтвердить работоспособность объекта тестирования. Тестировщик полностью следует требованиям и инструкции по работе с приложением. Например, при тестировании формы регистрации заполняет её корректными данными и нажимает кнопку «Зарегистрироваться».

Негативное тестирование (negative testing) направлено на исследование работы приложения в ситуациях, когда выполняют некорректные операции или используют данные, потенциально приводящие к ошибкам.

Негативное тестирование — это не попытка «сломать» систему, а проверка системы на правильность обработки некорректных действий пользователя.

1. Альфа- и бета-тестирование

Альфа-тестирование (alpha testing) — внутреннее пробное использование. Выполняется внутри организации-разработчика, иногда — с частичным привлечением пользователей. Альфа-тестирование проводится после модульного, интеграционного и системного тестирования, когда продукт уже частично готов к выпуску на рынок, но нужно его доработать. Представляет собой имитацию реального использования, но выполняется либо командой тестирования, либо другими сотрудниками компании-разработчика в тестовой среде. Например, на тестовых стендах, недоступных внешним пользователям. После внутреннего альфа-тестирования выпускают бета-версию продукта и передают её на внешнее (публичное) бета-тестирование.

Бета-тестирование (beta testing) выполняется вне организации с активным привлечением пользователей. Обычно представляет собой форму внешнего приёмочного тестирования. Продукт должен быть стабилен, но не исключено появление проблем и выявление недостатков. Поэтому сначала доступ открывают для небольшой группы лояльных пользователей, чтобы проверить работоспособность и получить обратную связь.

Часто при тестировании игр применяют ОБТ — открытое бета-тестирование. Привлекают либо всех желающих (например, заполнившие заявку), либо людей с опытом в играх такого типа. Обычно у компаний есть контакты тех, с кем они постоянно сотрудничают при проведении бета-тестирования.

Иногда бета-версия размещается в конкретной стране или регионе, чтобы собрать статистику или получить обратную связь прежде, чем полностью вывести продукт на рынок.

1. Тестирование на основе тест-кейсов и исследовательское тестирование

Тестирование на основе тест-кейсов (scripted testing, test case based testing) — формализованный подход, при котором тестирование проводится на основе заранее подготовленных тест-кейсов. Часто используется на проектах по разработке ПО, так как позволяет структурировать процесс тестирования и сделать его контролируемым.

Исследовательское тестирование (exploratory testing) — частично формализованный подход, при котором тестировщик работает с приложением по сценарию. В процессе сценарий дорабатывается для более полного исследования приложения.

Во время исследовательского тестирования неформальные (не созданные заранее) тестовые сценарии разрабатываются, выполняются, анализируются и оцениваются динамически. Результаты тестирования используют для изучения компонента или системы и последующей разработки тестовых сценариев для непокрытых областей.

Исследовательское тестирование проводится сессиями. Сессия — это выделенный промежуток времени, в котором тестировщик исследует программу, ориентируясь на поставленную цель. Например, требуется проверить все поля ввода на странице. Во время сессии ведётся протокол, а тестировщик фиксирует действия и результаты.

Исследовательское тестирование лучше всего подходит, когда документация недостаточная или вовсе отсутствует, в условиях сжатых сроков и как дополнение к другим, более формальным методам тестирования.

Свободное (интуитивное) тестирование (ad hoc testing) — неформальный подход, в котором не предполагается использование тест-кейсов, чек-листов и сценариев. Тестировщик полностью опирается на свой профессионализм и интуицию при спонтанном выполнении проверок.

Часто при таком подходе предполагается, что тестировщик плохо знаком с тестируемым приложением. Этот вид тестирования используется редко и только как дополнение к полностью или частично формализованному тестированию, когда для исследования некоторых функций приложения нет тест-кейсов, или они ещё не написаны.

НЕФУНКЦИОНАЛЬНОЕ ТЕСТИРОВАНИЕ

Нефункциональное тестирование (non-functional testing) — анализ свойств компонента или системы, не относящихся к ункциональности. То есть, проверяется, «как работает система».

Нефункциональное тестирование включает:

* 1. Тестирование производительности:
     1. нагрузочное тестирование,
     2. тестирование масштабируемости,
     3. объёмное тестирование,
     4. стрессовое тестирование.
  2. Тестирование безопасности.
  3. Инсталляционное тестирование.
  4. Тестирование интерфейса (GUI/UI-тестирование).
  5. Тестирование удобства использования.
  6. Тестирование локализации.
  7. Тестирование надёжности.

1. Тестирование производительности

Тестирование производительности (performance testing) помогает определить работоспособность, стабильность, потребление ресурсов в условиях разных сценариев использования и нагрузок.

Задача системы — обрабатывать нужное количество данных за установленное время. В случае превышения запланированных объёмов входных данных, система восстанавливается после отказа без потери данных. Пример: в требованиях указано, что система обрабатывает тысячу запросов пользователей в секунду без потери производительности. Чтобы проверить выполнение этого требования, тестировщик формирует тысячу запросов пользователей и направляет их на сервер.

* 1. Нагрузочное тестирование

Нагрузочное тестирование (load testing) — тип тестирования производительности, цель которого — оценить поведение системы при возрастающей нагрузке, а также определить нагрузку, которую может выдержать компонент или система. Нагрузка повышается, пока не будут достигнуты нужные характеристики. Затем отслеживается поведение на протяжении повышения загрузки системы. При этом:

* измеряют время выполнения операций при определённой интенсивности этих операций;
* определяют количество пользователей, одновременно работающих с приложением;
* определяют границы приемлемой производительности при увеличении нагрузки и при увеличении интенсивности выполнения операций.
  1. Тестирование масштабируемости

Тестирование масштабируемости (scalability testing) — тестирование для измерения возможностей вертикального и горизонтального масштабирования с точки зрения любой из нефункциональных возможностей: увеличение количества пользователей, рост количества транзакций, увеличение объёма данных.

Вертикальное масштабирование — это увеличение производительности каждого компонента системы для повышения общей производительности. Например, увеличивается объём оперативной памяти на сервере, чтобы он быстрее обрабатывал запросы. Это повысит производительность всей системы.

Горизонтальное масштабирование — разбиение системы на структурные компоненты и разнесение их по отдельным физическим машинам. А также увеличение количества серверов, параллельно выполняющих одну и ту же функцию. Например, увеличивается количество серверов, но каждый выполняет одну и ту же задачу: принимает одни и те же запросы и отвечает на них.

Если разработчики заранее не подумают, как они увеличат ресурсы при росте популярности, они потеряют значительную часть прибыли.

* 1. Объёмное тестирование

Объёмное тестирование (volume testing) — тестирование на больших объёмах данных. Например, тестируется поведение приложения при попытке загрузить в его базу данных нескольких файлов очень большого размера.

* 1. Стрессовое тестирование

Стрессовое тестирование (stress testing) оценивает систему на граничных значениях рабочих нагрузок, за их пределами или в состоянии ограниченных ресурсов — памяти или доступа к серверу. Например, стандартная нагрузка на сервер приложения — 1000 запросов в секунду. При стрессовом тестировании нужно проверить её поведение при увеличении нагрузки до 10 000 запросов в секунду. Если система не обработает такое количество запросов и отключится, при перезапуске все данные и настройки сохранятся.

# Список иллюстраций

Номер п/п Страница

1. Рисунок 13 – Пример чек листа 1
2. Рисунок 14 – Статусы 2
3. Рисунок 15 – Матрица 3

# Словарь терминов

Тести́рование програ́ммного обеспе́че́ния. процесс исследования, испытания программного продукта, имеющий своей целью проверку соответствия между реальным поведением программы и её ожидаемым поведением на конечном наборе тестов, выбранных определённым образом.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

* 1. <https://gb.ru/>
  2. <https://qualitica.ru/blog/chek-list/>
  3. <https://software-testing.ru/library/testing/testing-for-beginners/1991-test-case-writing>