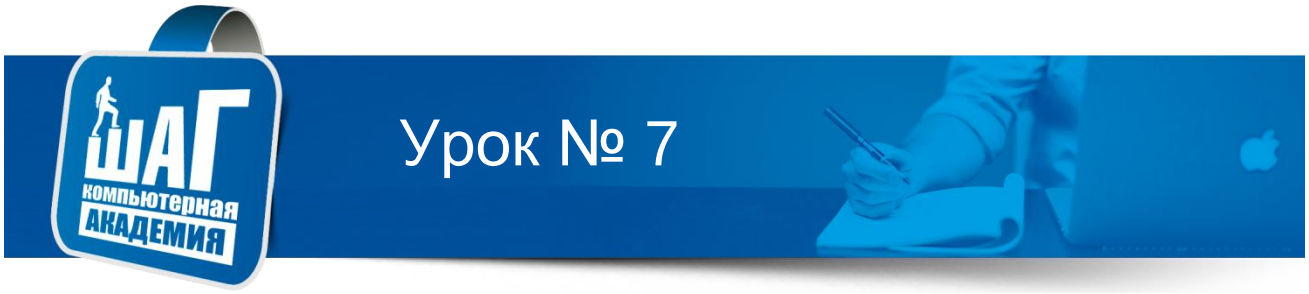

Курс: «Мануальное тестирование ПО»

Тема: Теория тестирования, часть 2 (процесс, компоненты).

План

1. Процесс тестирования.
2. Вечный круг тестирования.
3. Тестовые артефакты.
4. Техники тест дизайна.

1. Процесс тестирования.

Тестирование начинается не с того момента, когда вам дали рабочее приложение. Когда Вы узнали, что команда будет работать над проектом, можно считать, что вы уже приступили.
После получения спецификации или ТЗ, вы начинаете писать тест план, разрабатываете тест кейсы, оцениваете необходимость использования автоматизации.
Как только разработчики подготовили билд, вы должны провести дымовое тестирование, по результатам которого делается вывод о возможности и целесообразности дальнейшего тестирования:
В случае если Дымовое тестирование прошло не успешно(провалено), вы отправляете приложение на доработку.
Если же Дымовое тестирование прошло успешно, то вы переходите к следующему виду тестирования - регрессионное тестирование (Regression testing) и санитарное тестирование (Sanity testing).
Открыв багтрекинговую систему, вы должны перепроверить дефекты, которые разработчики исправили или не смогли воспроизвести. Дефекты, которые программисты не смогли воспроизвести для вас самые неприятные - это явное свидетельство того, что либо вы недостаточно хорошо локализовали дефект, не очень понятно описали шаги для воспроизведения, либо разработчик поленился воспроизвести ситуацию.
Закрыв все дефекты, вы переходите к основной работе - тестированию по тест кейсам и вы начинаете "исследовать" приложение.
Когда все, что было запланировано, пройдено, вы имеете результаты прогона тест

кейсов, баг репорты, вопросы к аналитикам и заметки на полях своих тетрадей. Основываясь на всем этом, вы составляете отчет по проведенному тестированию и отправляете его на проектную группу.
Подобный процесс проходит от версии к версии, и через какое-то время результаты тестирования сойдутся, с прописанными в плане тестирования критериями окончания тестирования. На этом основная работа, связанная с непосредственно с тестированием будет окончена.(Рис.1)

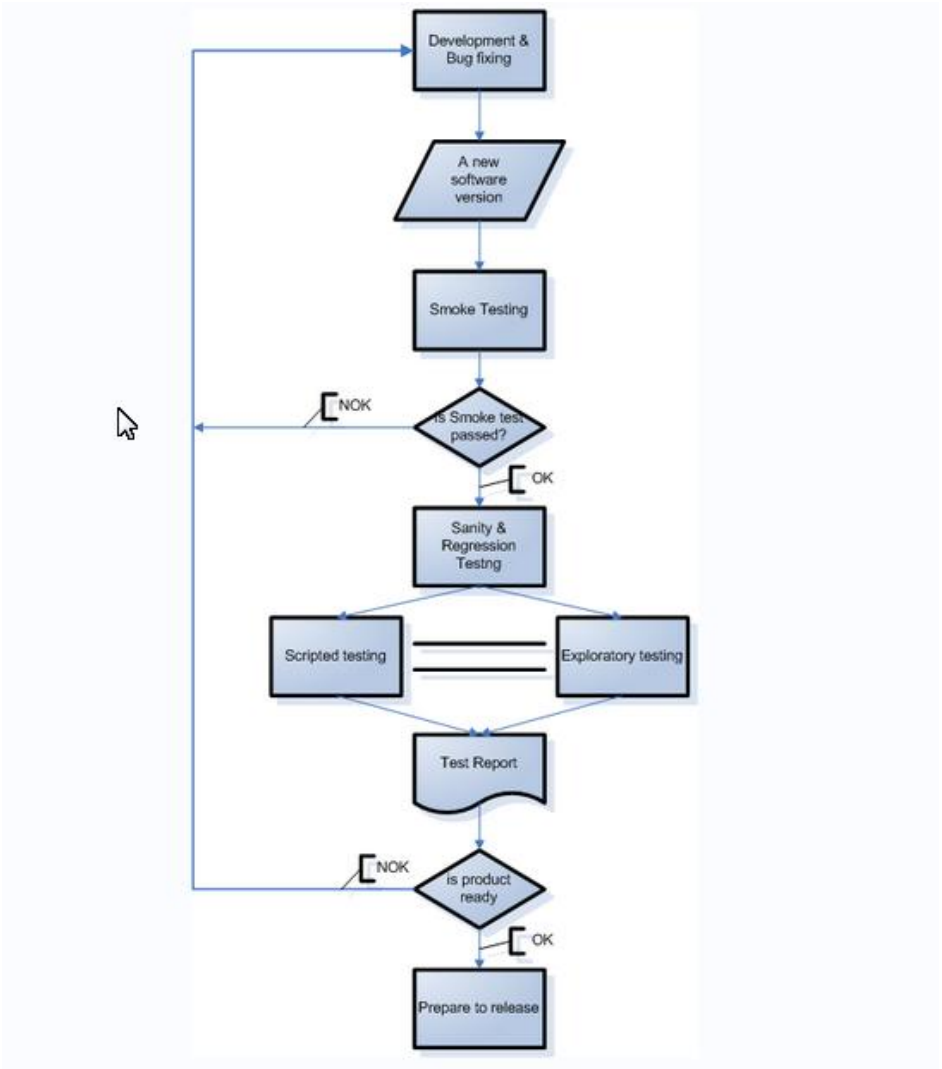

Рис. 1 Процесс тестирования**.**

2. **Вечный круг тестирования.**

Понять логику.

Когда мы получаем новый продукт, первая наша цель это понят логику продукта (для чего этот сайт или для чего это приложение) и разобраться в логике, во всех функциях, почему эти функции работают так, а не иначе.

Найти несоответствия.

На этом этапе мы открываем сайт или мобильное приложение(т.е. тестируемый продукт) и начинаем сравнивать с тем что мы ожидаем от продукта, после того как

прочли техническую документацию и поняли логику. Вы замечаете эти несоответствия(дефекты)

Донести информацию.

Далее оформляете баги и доносите их разработчикам, соответственно через багтрекинговую систему.

Перепроверить.

Разработчик дефекты исправляет и перенаправляет тестировщикам. После чего мы их перепроверяем.

Далее может показаться, что цикл на этом и закончился. Мы перепроверили баги и всё хорошо. На самом деле редко, когда проекты заканчиваются на одной итерации, когда что-то было разработано, протестировано и выпущено. Чаще всего продукт продолжает развиваться, т.е. пока мы тестируем и перепроверяем багги, в это время уже идёт разработка новых функций этого же продукта. И в итоге нам снова нужно понять логику, найти несоответствия, донести информацию и перепроверить. И так мы в принципе ходим по кругу, который называется «Вечный круг тестирования». Даже если один проект закончился, то следующий будет по такому же принципу. А в центре, у нас как у тестировщиков, должно быть стремление к совершенству. Это имеется ввиду совершенство Вас как профессионала тестировщика. Даже если нам кажется, что несоответствий в программе больше нет, мы должны всегда помнить что все багги найти невозможно, и они всегда и всё равно есть.

Вы должны думать: «Сейчас ещё раз посмотрю на те функции, которые уже проверял, скорее всего, найду какие-то несоответствия.» Или Вы можете поговорить с разработчиками, может, узнаете о новых функциональностях. Возможно, после такого разговора на ум придут новые тестовые сценарии. Всё это стимулирует Вас в совершенствовании как тестировщиков профессионалов и достижения более высокой квалификации.

3. Тестовые артефакты.

В соответствие с процессами или методологиями разработки ПО, во время проведения тестирования создается и используется определенное количество тестовых артефактов (документов). Наиболее распространенными тестовыми артефактами являются:

Спецификация программного обеспечения (Software Specification) - законченное описание поведения программы, которую требуется разработать.

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

Тестовый случай (**Test Case**) - это артефакт, описывающий совокупность шагов, конкретных условий и параметров, необходимых для проверки реализации

тестируемой функции или её части. Другими словами – это сценарий, по которому мы будем, что-либо тестировать.

Тестовый набор (Test suite) - это набор тест кейсов объединённых по какой-либо роли или функциональности.

Баг Репорты (Bug Reports) - это документы, описывающие ситуацию или последовательность действий, приведшую к некорректной работе объекта тестирования, с указанием причин и ожидаемого результата.

Чек лист (контрольный список) - список параметров, который необходимо проверить.

На этом уроке мы подробно поговорим о **чек листах.**

Чек лист является важным инструментом для тестирования веб-приложений и программных продуктов.

Чек лист - это документ, который описывает, какие функции должны быть проверены. Чеклисты тестирования могут иметь совершенно разные уровни детализации.

Чек лист тестирования состоит из двух основных частей:
- перечень функций конкретного продукта, выполнение которых должно быть проверено;
- список возможных ошибок, которые могут существовать в различных проектах.

Чек лист тестирования программного обеспечения используется для разделения задач по уровню квалификации, а также для поддержки отчетности и результатов тестирования.

Чек лист тестирования должен включать следующие пункты:

- подробный перечень проверок;
- статус проверки;
- результаты тестирования.

Чек лист создается на основе «Спецификации требований программного обеспечения». Определяя набор необходимых тестов, следует руководствоваться тремя основными правилами:

1. Чек лист должен охватывать весь функционал разрабатываемого продукта. Ни одно заявленное в спецификации требование не должно остаться без внимания.
2. Число тестов нужно минимизировать. Чем больше требований проверяется одним тестом – тем лучше.
3. Набор тестов должен не повторять требования, а проверять их.

Чек лист - список шагов или перечень функциональности который позволяет тестировщику убедиться в корректной работе приложения.
Чек лист не требует от нас детализации вводимых значений, и вообще детализации.
В какой-то мере, чек лист может считаться идеями для тест кейсов, его заголовками.

Обычно чек лист представляет собой таблицу из двух колонок:
Проверяемый фактор / Есть он у системы или нет
Например, мы хотим создать чек лист для чайника
В устройство можно влить воду ? да/нет
Из устройства можно вылить воду? да/нет
Если устройство , наполненное водой, включить , вода будет греться (устройство включено в сеть) ? да/нет
и т.д.
В интернете есть множество готовых чек листов.

4. Техники тест дизайна.

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

4.1 Верификация, валидация.

Верификация - это процесс оценки системы или её компонентов с целью определения удовлетворяют ли результаты текущего этапа разработки условиям, сформированным в начале этого этапа.

Валидация - это определение соответствия разрабатываемого ПО ожиданиям и потребностям пользователя, требованиям к системе.
Верификация и валидация являются одними из техник тест дизайна.

4.2 Positive\ negative testing.

Заключается в тестировании позитивных и негативных сценариев. Порядок тестирования такой: сначала проверяем позитивные сценарии, и только потом негативные.
Позитивный тестовый случай использует только корректные данные и проверяет, что программа работает так, как и полагается, при условии, что пользователь вносит корректные данные и не выходит за рамки предусмотренного сценария поведения.

4.3 Эквивалентное разделение (Equivalence Partitioning - EP), классы эквивалентности (equivalent classes-EC)

Например, у вас есть диапазон допустимых значений от 1 до 10, вы должны выбрать одно верное значение внутри интервала, скажем, 5, и одно неверное значение вне интервала - 0.

4.4 Анализ граничных Значений (Boundary Value Analysis – BVA)

Если взять пример выше, в качестве значений для позитивного тестирования

выберем минимальную и максимальную границы (1 и 10), и значения больше и меньше границ (0 и 11). Анализ Граничный значений может быть применен к полям, записям, файлам и другим параметрам, имеющим ограничения.
Чтобы удостовериться в правильности поведения программы при различных входных данных, в идеале следует протестировать все возможные значения для каждого элемента этих данных.

Например, пусть мы тестируем программу для отдела кадров, в ней есть поле "Возраст соискателя".
Пример взят из книги A Practitioner's guide to Sofware Test Design (Lee Copeland).

Требования по возрасту у нас будут такие:
0-13 лет - не нанимать
14-17 лет - можно нанимать на неполный день
18-54 года - можно нанимать на полный день
55-99 лет - не нанимать

Чтобы проверить все возможные разрешенные данные, нам нужно протестировать ввод чисел от 0 до 99. (Также ещё возможен ввод отрицательных чисел и нечисловых данных.) Необходимо ли тестировать все числа от 0 до 99? В случае, если программа анализирует каждое число по отдельности, вот таким образом, то видимо, да:

```
if (age == 13) hireStatus="NO";
if (age == 14) hireStatus="PART";
if (age == 15) hireStatus="PART";
if (age == 16) hireStatus="PART";
if (age == 17) hireStatus="PART";
if (age == 18) hireStatus="FULL";
```

Но , программы обычно пишут по-другому:

```
if (age >= 0 && age <=13)
        hireStatus="NO";
if (age >= 14 && age <=17)
        hireStatus="PART";
if (age >= 18 && age <=54)
        hireStatus="FULL";
if (age >= 55 && age <=99)
        hireStatus="NO";
```

Становится очевидным, что можно протестировать одно из чисел каждого диапазона. Например: 5, 15, 20, 60. А также граничные значения (первое и последнее значения из каждого диапазона): 0, 13, 14, 17, 18, 54, 55, 99.

Чтобы уменьшить количество тестируемых значений, производится
а) разбиение множества всех значений входной переменной на подмножества (классы эквивалентности), а затем
б) тестирование одного любого значения из каждого класса.
Все значения из каждого подмножества должны быть эквивалентны для наших тестов. Если тест проходит успешно для одного значения из класса эквивалентности, он должен проходить успешно для всех остальных. И наоборот, если тест не проходит для одного значения, он не должен проходить для всех

остальных.
В данном случае имеем 12 классов эквивалентности (каждое из 8 граничных значений по сути является отдельным классом).
Чтобы проверить правильность работы программы на всех разрешенных данных, нужно провести 12 тестов.

Запрещенные данные тестируются аналогично - можно выделить классы эквивалентности "дробное число от 0 до 99", "отрицательное число", "число больше 99", "набор букв", "пустая строка" и т.д.

Таким образом, метод классов эквивалентности можно разделить на три этапа:
1. Тестирование разрешенных значений
2. Тестирование граничных значений
3. Тестирование запрещенных значений

4.5 Причина/ Следствие (Cause/Effect - CE).

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

4.6 Предугадывание ошибки (Error Guessing - EG)

Это когда тест аналитик использует свои знания системы и способность к интерпретации спецификации на предмет того, чтобы "предугадать" при каких входных условиях система может выдать ошибку. Например, спецификация говорит: "пользователь должен ввести код". Тест аналитик, будет думать: "Что, если я не введу код?", "Что, если я введу неправильный код? ", и так далее. Это и есть предугадывание ошибки.

4.7 Исчерпывающее тестирование (Exhaustive Testing - ET).

Используется крайне редких случаях. В пределах этой техники вы должны проверить все возможные комбинации входных значений, и в результате, это должно найти все проблемы. На практике применение этого метода не представляется возможным, из-за огромного количества входных значений.

4.8 Попарное тестирование (Pairwise testing).

Метод классов эквивалентности применяется для тестирования каждого входного параметра по отдельности.
Пусть наша программа принимает на вход десяток параметров. Дефекты, возникающие при определенном сочетании всех десяти параметров, довольно редки. Взаимное влияние параметров, о котором пользователь не знает - это дефект интерфейса (интерфейс интуитивно не понятен).

Чаще всего будут встречаться ситуации, в которых один параметр влияет на один из оставшихся, т.е. самыми частыми будут дефекты, возникающие при определенном сочетании двух каких-то параметров.

Таким образом, можно упростить себе задачу и протестировать все возможные значения для каждой из пар параметров. Такой подход называется попарным тестированием (pairwise testing).

Пример. Пусть имеется 3 двоичных входных параметра (3 чекбокса). Количество всех возможных комбинаций - 2 в степени 3 = 8 , значит, нужно произвести 8 тестов. Давайте попробуем сэкономить, тестируя чекбоксы попарно.
Выпишем все комбинации для первого и второго чекбоксов:

1-й	2-й
0	0
0	1
1	0
1	1

Добавим третий столбец так, чтобы во втором и третьем столбце получились все 4 двоичные комбинации. Это можно сделать разными способами, мы сделаем так (на первый столбец можно не обращать внимания):

1-й	2-й	3-й
0	0	0
0	1	0
1	0	1
1	1	1

С помощью четырех наборов входных данных (четырех тестов) мы протестируем две пары параметров: первый со вторым и второй с третьим. Осталось протестировать пару "первый с третьим".
Выпишем отдельно 1 и 3 столбцы:

1-й	3-й
0	0
0	0
1	1
1	1

Как видно, мы имеем здесь две из четырех возможных комбинаций. Комбинации "01" и "10" здесь отсутствуют, а комбинации "00" и "11" присутствуют два раза.
Добавим еще 2 строки (еще два теста)

1-й	3-й
0	0
0	0
1	1
1	1
0	1
1	0

Вернем второй столбец на место:

1-й	2-й	3-й
0	0	0
0	1	0
1	0	1
1	1	1
0		1
1		0

Выходит, что последние два теста можно проходить при любых значениях второго параметра. Можно дописать для определенности нули в эти пустые места:

1-й	2-й	3-й
0	0	0
0	1	0
1	0	1
1	1	1
0	0	1
1	0	0

Получаем 6 тестов вместо 8 при полном переборе.

Можно ли сэкономить еще? Да, можно.
Вернемся к 1 шагу:

1-й	2-й
0	0
0	1
1	0
1	1

Давайте допишем третий столбец другим способом, поменяв порядок комбинаций:

1-й	2-й	3-й
0	0	1
0	1	0
1	0	0
1	1	1

Все комбинации для 1 и 2, а также для 2 и 3 параметра здесь есть.
Посмотрим теперь на комбинации 1 и 3 параметра

1-й	3-й
0	1
0	0
1	0
1	1

Что мы видим? Изменив порядок значений в третьем столбце, мы скомбинировали и 2-й с 3-м, и 1-й с 3-м параметры.
Итого имеем всего 4 строки, то есть 4 теста, эквивалентные первоначальным шести:

1-й	2-й	3-й
0	0	1
0	1	0
1	0	0
1	1	1

Полный перебор всех комбинаций в третьем столбце гарантированно даст минимальное количество тестов. Однако, судя по тому, что алгоритмы такой минимизации разрабатываются до сих пор, полный перебор неприемлем из-за большого времени исполнения. Существуют программы, дающие приемлемый результат в приемлемое время, например, программа PICT от Microsoft.

4.9 ADHOC testing.

Это тестирование без подробных спецификаций, сопроводительных документов, тест плана и т.д. Другими словами, тестирование в полном хаосе.
Преимущество такой техники в том, что нет необходимости в планировании и документации, наиболее важные ошибки находятся быстро, нет задержек со стартом проекта.

4.10 Дымовое (Smoke testing).

Понятие дымовое тестирование пошло из инженерной среды: "При вводе в эксплуатацию нового оборудования ("железа") считалось, что тестирование прошло удачно, если из установки не пошел дым."
В области же программного обеспечения, дымовое тестирование рассматривается как короткий цикл тестов, выполняемый для подтверждения того, что после сборки кода (нового или исправленного) устанавливаемое приложение, стартует и выполняет основные функции.